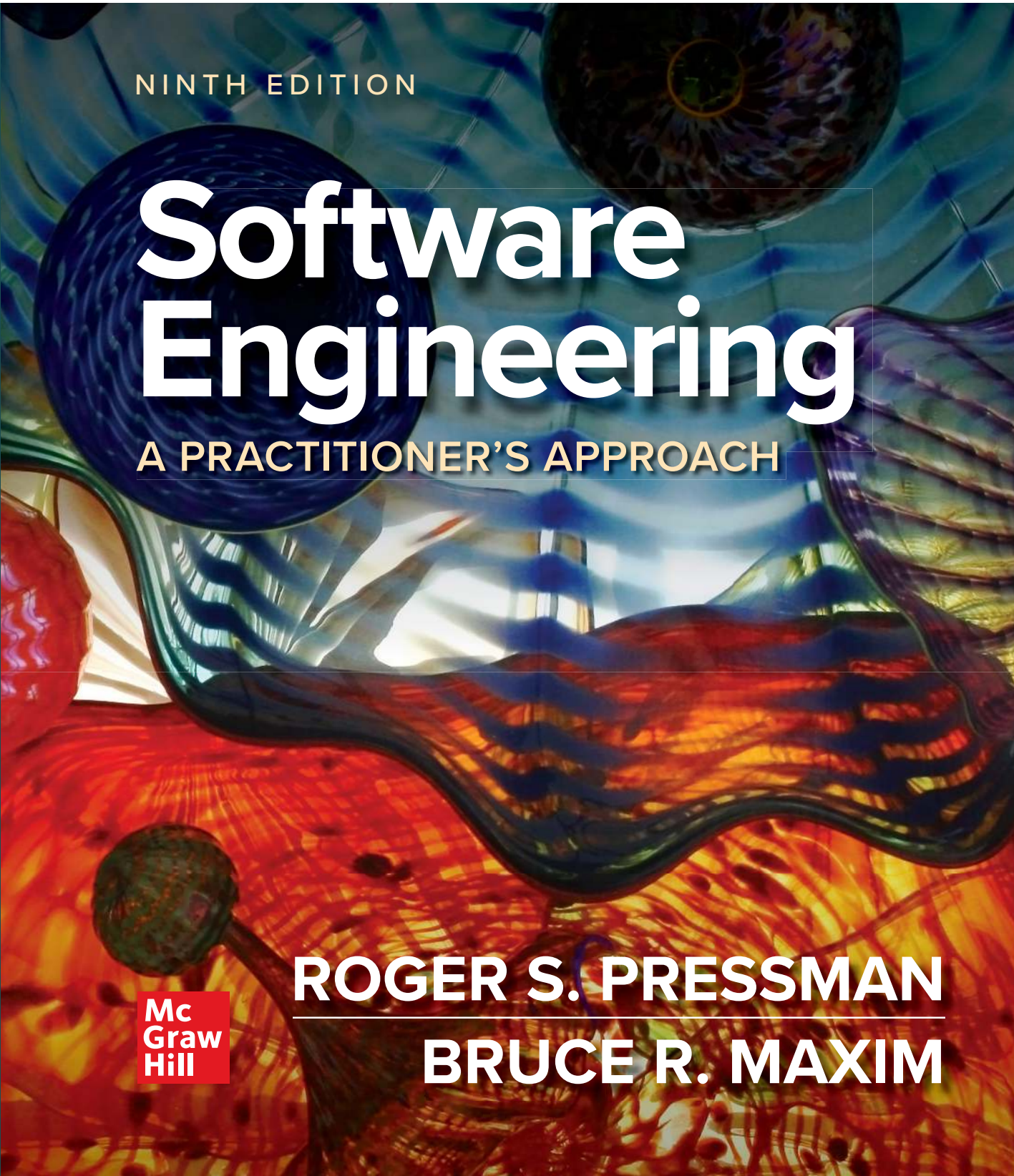
This International Student Edition is for use outside of the U.S.



NINTH EDITION

Software Engineering

A PRACTITIONER'S APPROACH

ROGER S. PRESSMAN

BRUCE R. MAXIM

Mc Graw Hill

Software Engineering

A PRACTITIONER'S APPROACH

Software Engineering

A PRACTITIONER'S APPROACH

NINTH EDITION

Roger S. Pressman, Ph.D.
Bruce R. Maxim, Ph.D.

Mc Graw Hill



SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright © 2020 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw-Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 24 23 22 21 20 19

ISBN 978-1-260-54800-6
MHID 1-260-54800-7

Cover Image: ©*R.L. Hausdorf/Shutterstock*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw-Hill Education, and McGraw-Hill Education does not guarantee the accuracy of the information presented at these sites.

mheducation.com/highered

To Barbara, Matt, Mike,
Shiri, Adam, Lily,
and Maya.

—Roger S. Pressman

To my family who
support me in all
that I do.

—Bruce R. Maxim

About the Authors



Courtesy of Roger Pressman

Roger S. Pressman is an internationally recognized consultant and author in software engineering. For almost five decades, he has worked as a software engineer, a manager, a professor, an author, a consultant, and an entrepreneur.

Dr. Pressman was president of R. S. Pressman & Associates, Inc., a consulting firm that specialized in helping companies establish effective software engineering strategies. Over the years he developed a set of techniques and tools that improved software engineering practice. He is also the founder and chief technology officer of EVANNEX®, an automotive aftermarket company that specializes in the design and manufacture of accessories for the Tesla line of electric vehicles.

Dr. Pressman is the author of ten books, including two novels, and many technical and management papers. He has been on the editorial boards of *IEEE Software* and *The Cutter IT Journal* and was editor of the "Manager" column in *IEEE Software.*

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He has presented tutorials at the International Conference on Software Engineering and at many other industry meetings. He has been a member of the ACM, IEEE, Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.



Michigan Creative/
UM-Dearborn

Bruce R. Maxim has worked as a software engineer, project manager, professor, author, and consultant for more than thirty years. His research interests include software engineering, user experience design, serious game development, artificial intelligence, and engineering education.

Dr. Maxim is professor of computer and information science and collegiate professor of engineering at the University of Michigan–Dearborn. He established the GAME Lab in the College of Engineering and Computer Science. He has published papers on computer algorithm animation, game development, and engineering education. He is coauthor of a best-selling introductory computer science text and two edited collections of software engineering research papers. Dr. Maxim has supervised several hundred industry-based software development projects as part of his work at the University of Michigan–Dearborn.

Dr. Maxim's professional experience includes managing research information systems at a medical school, directing instructional computing for a medical campus, and working as a statistical programmer. Dr. Maxim served as the chief technology officer for a game development company.

Dr. Maxim was the recipient of several distinguished teaching awards, a distinguished community service award, and a distinguished faculty governance award. He is a member of Sigma Xi, Upsilon Pi Epsilon, Pi Mu Epsilon, Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers, and International Game Developers Association.

Contents at a Glance

CHAPTER 1 Software and Software Engineering 1

PART ONE THE SOFTWARE PROCESS 19

CHAPTER 2 Process Models 20
CHAPTER 3 Agility and Process 37
CHAPTER 4 Recommended Process Model 54
CHAPTER 5 Human Aspects of Software Engineering 74

PART TWO MODELING 83

CHAPTER 6 Principles That Guide Practice 84
CHAPTER 7 Understanding Requirements 102
CHAPTER 8 Requirements Modeling—A Recommended Approach 126
CHAPTER 9 Design Concepts 156
CHAPTER 10 Architectural Design—A Recommended Approach 181
CHAPTER 11 Component-Level Design 206
CHAPTER 12 User Experience Design 233
CHAPTER 13 Design for Mobility 264
CHAPTER 14 Pattern-Based Design 289

PART THREE QUALITY AND SECURITY 309

CHAPTER 15 Quality Concepts 310
CHAPTER 16 Reviews—A Recommended Approach 325
CHAPTER 17 Software Quality Assurance 339
CHAPTER 18 Software Security Engineering 356
CHAPTER 19 Software Testing—Component Level 372
CHAPTER 20 Software Testing—Integration Level 395
CHAPTER 21 Software Testing—Specialized Testing for Mobility 412
CHAPTER 22 Software Configuration Management 437
CHAPTER 23 Software Metrics and Analytics 460

PART FOUR MANAGING SOFTWARE PROJECTS 489

CHAPTER 24 Project Management Concepts 490
CHAPTER 25 Creating a Viable Software Plan 504
CHAPTER 26 Risk Management 532
CHAPTER 27 A Strategy for Software Support 549

PART FIVE ADVANCED TOPICS 567

CHAPTER 28 Software Process Improvement 568
CHAPTER 29 Emerging Trends in Software Engineering 583
CHAPTER 30 Concluding Comments 602

APPENDIX 1 An Introduction to UML 611
APPENDIX 2 Data Science for Software Engineers 629
REFERENCES 639
INDEX 659

Table of Contents

Preface *xxvii*

CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING 1

1.1 The Nature of Software 4
- 1.1.1 Defining Software 5
- 1.1.2 Software Application Domains 7
- 1.1.3 Legacy Software 8

1.2 Defining the Discipline 8

1.3 The Software Process 9
- 1.3.1 The Process Framework 10
- 1.3.2 Umbrella Activities 11
- 1.3.3 Process Adaptation 11

1.4 Software Engineering Practice 12
- 1.4.1 The Essence of Practice 12
- 1.4.2 General Principles 14

1.5 How It All Starts 15

1.6 Summary 17

PART ONE THE SOFTWARE PROCESS 19

CHAPTER 2 PROCESS MODELS 20

2.1 A Generic Process Model 21

2.2 Defining a Framework Activity 23

2.3 Identifying a Task Set 23

2.4 Process Assessment and Improvement 24

2.5 Prescriptive Process Models 25
- 2.5.1 The Waterfall Model 25
- 2.5.2 Prototyping Process Model 26
- 2.5.3 Evolutionary Process Model 29
- 2.5.4 Unified Process Model 31

2.6 Product and Process 33

2.7 Summary 35

CHAPTER 3 AGILITY AND PROCESS 37

3.1 What Is Agility? 38
3.2 Agility and the Cost of Change 39
3.3 What Is an Agile Process? 40
3.3.1 Agility Principles 40
3.3.2 The Politics of Agile Development 41
3.4 Scrum 42
3.4.1 Scrum Teams and Artifacts 43
3.4.2 Sprint Planning Meeting 44
3.4.3 Daily Scrum Meeting 44
3.4.4 Sprint Review Meeting 45
3.4.5 Sprint Retrospective 45
3.5 Other Agile Frameworks 46
3.5.1 The XP Framework 46
3.5.2 Kanban 48
3.5.3 DevOps 50
3.6 Summary 51

CHAPTER 4 RECOMMENDED PROCESS MODEL 54

4.1 Requirements Definition 57
4.2 Preliminary Architectural Design 59
4.3 Resource Estimation 60
4.4 First Prototype Construction 61
4.5 Prototype Evaluation 64
4.6 Go, No-Go Decision 65
4.7 Prototype Evolution 67
4.7.1 New Prototype Scope 67
4.7.2 Constructing New Prototypes 68
4.7.3 Testing New Prototypes 68
4.8 Prototype Release 68
4.9 Maintain Release Software 69
4.10 Summary 72

CHAPTER 5 HUMAN ASPECTS OF SOFTWARE ENGINEERING 74

5.1 Characteristics of a Software Engineer 75
5.2 The Psychology of Software Engineering 75

5.3 The Software Team 76
5.4 Team Structures 78
5.5 The Impact of Social Media 79
5.6 Global Teams 80
5.7 Summary 81

PART TWO MODELING 83

CHAPTER 6 PRINCIPLES THAT GUIDE PRACTICE 84

6.1 Core Principles 85
6.1.1 Principles That Guide Process 85
6.1.2 Principles That Guide Practice 86
6.2 Principles That Guide Each Framework Activity 88
6.2.1 Communication Principles 88
6.2.2 Planning Principles 91
6.2.3 Modeling Principles 92
6.2.4 Construction Principles 95
6.2.5 Deployment Principles 98
6.3 Summary 100

CHAPTER 7 UNDERSTANDING REQUIREMENTS 102

7.1 Requirements Engineering 103
7.1.1 Inception 104
7.1.2 Elicitation 104
7.1.3 Elaboration 104
7.1.4 Negotiation 105
7.1.5 Specification 105
7.1.6 Validation 105
7.1.7 Requirements Management 106
7.2 Establishing the Groundwork 107
7.2.1 Identifying Stakeholders 107
7.2.2 Recognizing Multiple Viewpoints 107
7.2.3 Working Toward Collaboration 108
7.2.4 Asking the First Questions 108
7.2.5 Nonfunctional Requirements 109
7.2.6 Traceability 109

7.3 Requirements Gathering 110
7.3.1 Collaborative Requirements Gathering 110
7.3.2 Usage Scenarios 113
7.3.3 Elicitation Work Products 114
7.4 Developing Use Cases 114
7.5 Building the Analysis Model 118
7.5.1 Elements of the Analysis Model 119
7.5.2 Analysis Patterns 122
7.6 Negotiating Requirements 122
7.7 Requirements Monitoring 123
7.8 Validating Requirements 123
7.9 Summary 124

CHAPTER 8 REQUIREMENTS MODELING—A RECOMMENDED APPROACH 126

8.1 Requirements Analysis 127
8.1.1 Overall Objectives and Philosophy 128
8.1.2 Analysis Rules of Thumb 128
8.1.3 Requirements Modeling Principles 129
8.2 Scenario-Based Modeling 130
8.2.1 Actors and User Profiles 131
8.2.2 Creating Use Cases 131
8.2.3 Documenting Use Cases 135
8.3 Class-Based Modeling 137
8.3.1 Identifying Analysis Classes 137
8.3.2 Defining Attributes and Operations 140
8.3.3 UML Class Models 141
8.3.4 Class-Responsibility-Collaborator Modeling 144
8.4 Functional Modeling 146
8.4.1 A Procedural View 146
8.4.2 UML Sequence Diagrams 148
8.5 Behavioral Modeling 149
8.5.1 Identifying Events with the Use Case 149
8.5.2 UML State Diagrams 150
8.5.3 UML Activity Diagrams 151
8.6 Summary 154

CHAPTER 9 DESIGN CONCEPTS 156

9.1 Design Within the Context of Software Engineering 157
9.2 The Design Process 159
9.2.1 Software Quality Guidelines and Attributes 160
9.2.2 The Evolution of Software Design 161
9.3 Design Concepts 163
9.3.1 Abstraction 163
9.3.2 Architecture 163
9.3.3 Patterns 164
9.3.4 Separation of Concerns 165
9.3.5 Modularity 165
9.3.6 Information Hiding 166
9.3.7 Functional Independence 167
9.3.8 Stepwise Refinement 167
9.3.9 Refactoring 168
9.3.10 Design Classes 169
9.4 The Design Model 171
9.4.1 Design Modeling Principles 173
9.4.2 Data Design Elements 174
9.4.3 Architectural Design Elements 175
9.4.4 Interface Design Elements 175
9.4.5 Component-Level Design Elements 176
9.4.6 Deployment-Level Design Elements 177
9.5 Summary 178

CHAPTER 10 ARCHITECTURAL DESIGN— A RECOMMENDED APPROACH 181

10.1 Software Architecture 182
10.1.1 What Is Architecture? 182
10.1.2 Why Is Architecture Important? 183
10.1.3 Architectural Descriptions 183
10.1.4 Architectural Decisions 184
10.2 Agility and Architecture 185
10.3 Architectural Styles 186
10.3.1 A Brief Taxonomy of Architectural Styles 187
10.3.2 Architectural Patterns 192
10.3.3 Organization and Refinement 193

10.4 Architectural Considerations 193
10.5 Architectural Decisions 195
10.6 Architectural Design 196
10.6.1 Representing the System in Context 196
10.6.2 Defining Archetypes 197
10.6.3 Refining the Architecture into Components 198
10.6.4 Describing Instantiations of the System 200
10.7 Assessing Alternative Architectural Designs 201
10.7.1 Architectural Reviews 202
10.7.2 Pattern-Based Architecture Review 203
10.7.3 Architecture Conformance Checking 204
10.8 Summary 204

CHAPTER 11 COMPONENT-LEVEL DESIGN 206

11.1 What Is a Component? 207
11.1.1 An Object-Oriented View 207
11.1.2 The Traditional View 209
11.1.3 A Process-Related View 211
11.2 Designing Class-Based Components 212
11.2.1 Basic Design Principles 212
11.2.2 Component-Level Design Guidelines 215
11.2.3 Cohesion 216
11.2.4 Coupling 218
11.3 Conducting Component-Level Design 219
11.4 Specialized Component-Level Design 225
11.4.1 Component-Level Design for WebApps 226
11.4.2 Component-Level Design for Mobile Apps 226
11.4.3 Designing Traditional Components 227
11.4.4 Component-Based Development 228
11.5 Component Refactoring 230
11.6 Summary 231

CHAPTER 12 USER EXPERIENCE DESIGN 233

12.1 User Experience Design Elements 234
12.1.1 Information Architecture 235
12.1.2 User Interaction Design 236
12.1.3 Usability Engineering 236
12.1.4 Visual Design 237

12.2 The Golden Rules 238
12.2.1 Place the User in Control 238
12.2.2 Reduce the User's Memory Load 239
12.2.3 Make the Interface Consistent 240
12.3 User Interface Analysis and Design 241
12.3.1 Interface Analysis and Design Models 241
12.3.2 The Process 242
12.4 User Experience Analysis 243
12.4.1 User Research 244
12.4.2 User Modeling 245
12.4.3 Task Analysis 247
12.4.4 Work Environment Analysis 248
12.5 User Experience Design 249
12.6 User Interface Design 250
12.6.1 Applying Interface Design Steps 251
12.6.2 User Interface Design Patterns 252
12.7 Design Evaluation 253
12.7.1 Prototype Review 253
12.7.2 User Testing 255
12.8 Usability and Accessibility 255
12.8.1 Usability Guidelines 257
12.8.2 Accessibility Guidelines 259
12.9 Conventional Software UX and Mobility 261
12.10 Summary 261

CHAPTER 13 DESIGN FOR MOBILITY 264

13.1 The Challenges 265
13.1.1 Development Considerations 265
13.1.2 Technical Considerations 266
13.2 Mobile Development Life Cycle 268
13.2.1 User Interface Design 270
13.2.2 Lessons Learned 271
13.3 Mobile Architectures 273
13.4 Context-Aware Apps 274
13.5 Web Design Pyramid 275
13.5.1 WebApp Interface Design 275
13.5.2 Aesthetic Design 277
13.5.3 Content Design 277
13.5.4 Architecture Design 278
13.5.5 Navigation Design 280

13.6 Component-Level Design 282
13.7 Mobility and Design Quality 282
13.8 Mobility Design Best Practices 285
13.9 Summary 287

CHAPTER 14 PATTERN-BASED DESIGN 289

14.1 Design Patterns 290
14.1.1 Kinds of Patterns 291
14.1.2 Frameworks 293
14.1.3 Describing a Pattern 293
14.1.4 Machine Learning and Pattern Discovery 294
14.2 Pattern-Based Software Design 295
14.2.1 Pattern-Based Design in Context 295
14.2.2 Thinking in Patterns 296
14,2.3 Design Tasks 297
14.2.4 Building a Pattern-Organizing Table 298
14.2.5 Common Design Mistakes 298
14.3 Architectural Patterns 299
14.4 Component-Level Design Patterns 300
14.5 Anti-Patterns 302
14.6 User Interface Design Patterns 304
14.7 Mobility Design Patterns 305
14.8 Summary 306

PART THREE QUALITY AND SECURITY 309

CHAPTER 15 QUALITY CONCEPTS 310

15.1 What Is Quality? 311
15.2 Software Quality 312
15.2.1 Quality Factors 312
15.2.2 Qualitative Quality Assessment 314
15.2.3 Quantitative Quality Assessment 315
15.3 The Software Quality Dilemma 315
15.3.1 “Good Enough” Software 316
15.3.2 The Cost of Quality 317
15.3.3 Risks 319
15.3.4 Negligence and Liability 320

15.3.5 Quality and Security 320
15.3.6 The Impact of Management Actions 321
15.4 Achieving Software Quality 321
15.4.1 Software Engineering Methods 322
15.4.2 Project Management Techniques 322
15.4.3 Machine Learning and Defect Prediction 322
15.4.4 Quality Control 322
15.4.5 Quality Assurance 323
15.5 Summary 323

CHAPTER 16 REVIEWS—A RECOMMENDED APPROACH 325

16.1 Cost Impact of Software Defects 326
16.2 Defect Amplification and Removal 327
16.3 Review Metrics and Their Use 327
16.4 Criteria for Types of Reviews 330
16.5 Informal Reviews 331
16.6 Formal Technical Reviews 332
16.6.1 The Review Meeting 332
16.6.2 Review Reporting and Record Keeping 333
16.6.3 Review Guidelines 334
16.7 Postmortem Evaluations 336
16.8 Agile Reviews 336
16.9 Summary 337

CHAPTER 17 SOFTWARE QUALITY ASSURANCE 339

17.1 Background Issues 341
17.2 Elements of Software Quality Assurance 341
17.3 SQA Processes and Product Characteristics 343
17.4 SQA Tasks, Goals, and Metrics 343
17.4.1 SQA Tasks 343
17.4.2 Goals, Attributes, and Metrics 345
17.5 Formal Approaches to SQA 347
17.6 Statistical Software Quality Assurance 347

17.6.1 A Generic Example 347
17.6.2 Six Sigma for Software Engineering 349
17.7 Software Reliability 350
17.7.1 Measures of Reliability and Availability 350
17.7.2 Use of AI to Model Reliability 351
17.7.3 Software Safety 352
17.8 The ISO 9000 Quality Standards 353
17.9 The SQA Plan 354
17.10 Summary 355

CHAPTER 18 SOFTWARE SECURITY ENGINEERING 356

18.1 Why Software Security Information Is Important 357
18.2 Security Life-Cycle Models 357
18.3 Secure Development Life-Cycle Activities 359
18.4 Security Requirements Engineering 360
18.4.1 SQUARE 360
18.4.2 The SQUARE Process 360
18.5 Misuse and Abuse Cases and Attack Patterns 363
18.6 Security Risk Analysis 364
18.7 Threat Modeling, Prioritization, and Mitigation 365
18.8 Attack Surface 366
18.9 Secure Coding 367
18.10 Measurement 368
18.11 Security Process Improvement and Maturity Models 370
18.12 Summary 370

CHAPTER 19 SOFTWARE TESTING—COMPONENT LEVEL 372

19.1 A Strategic Approach to Software Testing 373
19.1.1 Verification and Validation 373
19.1.2 Organizing for Software Testing 374
19.1.3 The Big Picture 375
19.1.4 Criteria for “Done” 377

19.2 Planning and Recordkeeping 378
19.2.1 Role of Scaffolding 379
19.2.2 Cost-Effective Testing 379
19.3 Test-Case Design 381
19.3.1 Requirements and Use Cases 382
19.3.2 Traceability 383
19.4 White-Box Testing 383
19.4.1 Basis Path Testing 384
19.4.2 Control Structure Testing 386
19.5 Black-Box Testing 388
19.5.1 Interface Testing 388
19.5.2 Equivalence Partitioning 389
19.5.3 Boundary Value Analysis 389
19.6 Object-Oriented Testing 390
19.6.1 Class Testing 390
19.6.2 Behavioral Testing 392
19.7 Summary 393

CHAPTER 20 SOFTWARE TESTING—INTEGRATION LEVEL 395

20.1 Software Testing Fundamentals 396
20.1.1 Black-Box Testing 397
20.1.2 White-Box Testing 397
20.2 Integration Testing 398
20.2.1 Top-Down Integration 398
20.2.2 Bottom-Up Integration 399
20.2.3 Continuous Integration 400
20.2.4 Integration Test Work Products 402
20.3 Artificial Intelligence and Regression Testing 402
20.4 Integration Testing in the OO Context 404
20.4.1 Fault-Based Test-Case Design 405
20.4.2 Scenario-Based Test-Case Design 406
20.5 Validation Testing 407
20.6 Testing Patterns 409
20.7 Summary 409

CHAPTER 21 SOFTWARE TESTING—SPECIALIZED TESTING FOR MOBILITY 412

21.1 Mobile Testing Guidelines 413
21.2 The Testing Strategies 414
21.3 User Experience Testing Issues 415
21.3.1 Gesture Testing 415
21.3.2 Virtual Keyboard Input 416
21.3.3 Voice Input and Recognition 416
21.3.4 Alerts and Extraordinary Conditions 417
21.4 Web Application Testing 418
21.5 Web Testing Strategies 418
21.5.1 Content Testing 420
21.5.2 Interface Testing 421
21.5.3 Navigation Testing 421
21.6 Internationalization 423
21.7 Security Testing 423
21.8 Performance Testing 424
21.9 Real-Time Testing 426
21.10 Testing AI Systems 428
21.10.1 Static and Dynamic Testing 429
21.10.2 Model-Based Testing 429
21.11 Testing Virtual Environments 430
21.11.1 Usability Testing 430
21.11.2 Accessibility Testing 433
21.11.3 Playability Testing 433
21.12 Testing Documentation and Help Facilities 434
21.13 Summary 435

CHAPTER 22 SOFTWARE CONFIGURATION MANAGEMENT 437

22.1 Software Configuration Management 438
22.1.1 An SCM Scenario 439
22.1.2 Elements of a Configuration Management System 440
22.1.3 Baselines 441

22.1.4 Software Configuration Items 441
22.1.5 Management of Dependencies and Changes 442
22.2 The SCM Repository 443
22.2.1 General Features and Content 444
22.2.2 SCM Features 444
22.3 Version Control Systems 445
22.4 Continuous Integration 446
22.5 The Change Management Process 447
22.5.1 Change Control 448
22.5.2 Impact Management 451
22.5.3 Configuration Audit 452
22.5.4 Status Reporting 452
22.6 Mobility and Agile Change Management 453
22.6.1 e-Change Control 453
22.6.2 Content Management 455
22.6.3 Integration and Publishing 455
22.6.4 Version Control 457
22.6.5 Auditing and Reporting 458
22.7 Summary 458

CHAPTER 23 SOFTWARE METRICS AND ANALYTICS 460

23.1 Software Measurement 461
23.1.1 Measures, Metrics, and Indicators 461
23.1.2 Attributes of Effective Software Metrics 462
23.2 Software Analytics 462
23.3 Product Metrics 463
23.3.1 Metrics for the Requirements Model 464
23.3.2 Design Metrics for Conventional Software 466
23.3.3 Design Metrics for Object-Oriented Software 468
23.3.4 User Interface Design Metrics 471
23.3.5 Metrics for Source Code 473
23.4 Metrics for Testing 474
23.5 Metrics for Maintenance 476
23.6 Process and Project Metrics 476

23.7 Software Measurement 479
23.8 Metrics for Software Quality 482
23.9 Establishing Software Metrics Programs 485
23.10 Summary 487

PART FOUR MANAGING SOFTWARE PROJECTS 489

CHAPTER 24 PROJECT MANAGEMENT CONCEPTS 490

24.1 The Management Spectrum 491
24.1.1 The People 491
24.1.2 The Product 491
24.1.3 The Process 492
24.1.4 The Project 492
24.2 People 493
24.2.1 The Stakeholders 493
24.2.2 Team Leaders 493
24.2.3 The Software Team 494
24.2.4 Coordination and Communications Issues 496
24.3 Product 497
24.3.1 Software Scope 497
24.3.2 Problem Decomposition 497
24.4 Process 498
24.4.1 Melding the Product and the Process 498
24.4.2 Process Decomposition 498
24.5 Project 500
24.6 The W^5HH Principle 501
24.7 Critical Practices 502
24.8 Summary 502

CHAPTER 25 CREATING A VIABLE SOFTWARE PLAN 504

25.1 Comments on Estimation 505
25.2 The Project Planning Process 506

25.3 Software Scope and Feasibility 507
25.4 Resources 507
25.4.1 Human Resources 508
25.4.2 Reusable Software Resources 509
25.4.3 Environmental Resources 509
25.5 Data Analytics and Software Project Estimation 509
25.6 Decomposition and Estimation Techniques 511
25.6.1 Software Sizing 511
25.6.2 Problem-Based Estimation 512
25.6.3 An Example of LOC-Based Estimation 512
25.6.4 An Example of FP-Based Estimation 514
25.6.5 An Example of Process-Based Estimation 515
25.6.6 An Example of Estimation Using Use Case Points 517
25.6.7 Reconciling Estimates 518
25.6.8 Estimation for Agile Development 519
25.7 Project Scheduling 520
25.7.1 Basic Principles 521
25.7.2 The Relationship Between People and Effort 522
25.8 Defining a Project Task Set 523
25.8.1 A Task Set Example 524
25.8.2 Refinement of Major Tasks 524
25.9 Defining a Task Network 525
25.10 Scheduling 226
25.10.1 Time-Line Charts 526
25.10.2 Tracking the Schedule 528
25.11 Summary 530

CHAPTER 26 RISK MANAGEMENT 532

26.1 Reactive Versus Proactive Risk Strategies 533
26.2 Software Risks 534
26.3 Risk Identification 535
26.3.1 Assessing Overall Project Risk 536
26.3.2 Risk Components and Drivers 537
26.4 Risk Projection 538

26.4.1 Developing a Risk Table 538
26.4.2 Assessing Risk Impact 540
26.5 Risk Refinement 542
26.6 Risk Mitigation, Monitoring, and Management 543
26.7 The RMMM Plan 546
26.8 Summary 547

CHAPTER 27 A STRATEGY FOR SOFTWARE SUPPORT 549

27.1 Software Support 550
27.2 Software Maintenance 552
27.2.1 Maintenance Types 553
27.2.2 Maintenance Tasks 554
27.2.3 Reverse Engineering 555
27.3 Proactive Software Support 557
27.3.1 Use of Software Analytics 558
27.3.2 Role of Social Media 559
27.3.3 Cost of Support 559
27.4 Refactoring 560
27.4.1 Data Refactoring 561
24.4.2 Code Refactoring 561
27.4.3 Architecture Refactoring 561
27.5 Software Evolution 562
27.5.1 Inventory Analysis 563
27.5.2 Document Restructuring 564
27.5.3 Reverse Engineering 564
27.5.4 Code Refactoring 564
27.7.5 Data Refactoring 564
27.5.6 Forward Engineering 565
27.6 Summary 565

PART FIVE ADVANCED TOPICS 567

CHAPTER 28 SOFTWARE PROCESS IMPROVEMENT 568

28.1 What Is SPI? 569
28.1.1 Approaches to SPI 569
28.1.2 Maturity Models 570
28.1.3 Is SPI for Everyone? 571

28.2 The SPI Process 571
28.2.1 Assessment and GAP Analysis 572
28.2.2 Education and Training 573
28.2.3 Selection and Justification 573
28.2.4 Installation/Migration 574
28.2.5 Evaluation 575
28.2.6 Risk Management for SPI 575
28.3 The CMMI 576
28.4 Other SPI Frameworks 579
28.4.1 SPICE 579
28.4.2 TickIT Plus 579
28.5 SPI Return on Investment 580
28.6 SPI Trends 580
28.7 Summary 581

CHAPTER 29 EMERGING TRENDS IN SOFTWARE ENGINEERING 583

29.1 Technology Evolution 584
29.2 Software Engineering as a Discipline 585
29.3 Observing Software Engineering Trends 586
29.4 Identifying “Soft Trends” 587
29.4.1 Managing Complexity 588
29.4.2 Open-World Software 589
29.4.3 Emergent Requirements 590
29.4.4 The Talent Mix 591
29.4.5 Software Building Blocks 591
29.4.6 Changing Perceptions of “Value” 592
29.4.7 Open Source 592
29.5 Technology Directions 593
29.5.1 Process Trends 593
29.5.2 The Grand Challenge 594
29.5.3 Collaborative Development 595
29.5.4 Requirements Engineering 596
29.5.5 Model-Driven Software Development 596
29.5.6 Search-Based Software Engineering 597
29.5.7 Test-Driven Development 598
29.6 Tools-Related Trends 599
29.7 Summary 600

CHAPTER 30 CONCLUDING COMMENTS 602

30.1 The Importance of Software—Revisited 603
30.2 People and the Way They Build Systems 603
30.3 Knowledge Discovery 605
30.4 The Long View 606
30.5 The Software Engineer's Responsibility 607
30.6 A Final Comment from RSP 609

APPENDIX 1 An Introduction to UML 611
APPENDIX 2 Data Science for Software Engineers 629
REFERENCES 639
INDEX 659

PREFACE

When computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

It has been nearly four decades since the first edition of this book was written. During that time, software engineering has evolved from an obscure idea practiced by a relatively small number of zealots to a legitimate engineering discipline. Today, it is recognized as a subject worthy of serious research, conscientious study, and tumultuous debate. Throughout the industry, software engineer has replaced programmer or coder as the job title of preference. Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry segments.

Although managers and practitioners alike recognize the need for a more disciplined approach to software, they continue to debate the manner in which discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service today's most advanced technologies. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers, and bad things happen. In addition, debate, and controversy about the true nature of the software engineering approach continue. The status of software engineering is a study in contrasts. Attitudes have changed, progress has been made, but much remains to be done before the discipline reaches full maturity.

NEW TO THE NINTH EDITION

The ninth edition of *Software Engineering: A Practitioner's Approach* is intended to serve as a guide to a maturing engineering discipline. The ninth edition, like the eight editions that preceded it, is intended for both students and practitioners, retaining its appeal as a guide for the industry professional and a comprehensive introduction to the student at the upper-level undergraduate or first-year graduate level.

The ninth edition is considerably more than a simple update. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. In addition, we have further enhanced the popular "support system" for the book, providing a comprehensive set of student, instructor, and professional resources to complement the content of the book.

Readers of the past few editions of *Software Engineering: A Practitioner's Approach* will note that the ninth edition has actually been reduced in page length. Our goal was concision, making the book stronger from a pedagogical viewpoint and less daunting for the reader who desires to journey through the entire book. An anecdote, attributed to Blaise Pascal, the famous mathematician and physicist, goes like this: In writing a overly long letter to a friend, Pascal ended with this sentence. "I wanted to write you a shorter letter, but I didn't have the time." As we worked on concision for the ninth edition, we came to appreciate Pascal's words.

The 30 chapters of the ninth edition are organized into five parts. This organization better compartmentalizes topics and assists instructors who may not have the time to complete the entire book in one term.

Part 1, *The Software Process,* presents a variety of different views of software process, considering several important process models and frameworks that allow us to address the debate between prescriptive and agile process philosophies. Part 2, *Modeling,* presents analysis and design methods with an emphasis on object-oriented techniques and UML modeling. Pattern-based design and design for mobility computing applications are also considered. The coverage of user experience design has been expanded in this section. Part 3, *Quality and Security,* presents the concepts, procedures, techniques, and methods that enable a software team to assess software quality, review software engineering work products, conduct SQA procedures, and apply an effective testing strategy and tactics. In addition, we present software security practices that can be inserted into incremental software development models. Part 4, *Managing Software Projects,* presents topics that are relevant to those who plan, manage, and control a software development project. Part 5, *Advanced Topics,* considers software process improvement and software engineering trends. Boxed features are included throughout the book to present the trials and tribulations of a (fictional) software team and to provide supplementary materials about methods and tools that are relevant to chapter topics.

The five-part organization of the ninth edition enables an instructor to "cluster" topics based on available time and student need. An entire one-term course can be built around one or more of the five parts. A software engineering survey course would select chapters from all five parts. A software engineering course that emphasizes analysis and design would select topics from Parts 1 and 2. A testing-oriented software engineering course would select topics from Parts 1 and 3, with a brief foray into Part 2. A "management course" would stress Parts 1 and 4. By organizing the ninth edition in this way,

we have attempted to provide an instructor with a number of teaching options. In every case the content of the ninth edition is complemented by the following elements of the *SEPA, 9/e Support System.*

Additional Resources

A wide variety of resources can be accessed through the instructor website, including an extensive online learning center encompassing problem solutions, a variety of Web-based resources with software engineering checklists, an evolving collection of "tiny tools," and comprehensive case studies. *Professional Resources* provide several hundred categorized web references which allow students to explore software engineering in greater detail, along with a reference library with links to several hundred downloadable references providing an in-depth source of advanced software engineering information. Additionally, a complete online *Instructor's Guide* and supplementary teaching materials along with several hundred PowerPoint slides that may be used for lectures are included.

The *Instructor's Guide for Software Engineering: A Practitioner's Approach* presents suggestions for conducting various types of software engineering courses, recommendations for a variety of software projects to be conducted in conjunction with a course, solutions to selected problems, and a number of useful teaching aids.

When coupled with its online support system, the ninth edition of *Software Engineering: A Practitioner's Approach* provides flexibility and depth of content that cannot be achieved by a textbook alone.

Bruce Maxim has taken the lead in developing new content for the ninth edition of *Software Engineering: A Practitioner's Approach,* while Roger Pressman has served as editor-in-chief as well as providing contributions in select circumstances.

Acknowledgments Special thanks go to Nancy Mead from Software Engineering Institute at Carnegie Mellon University who wrote the chapter on software security engineering; Tim Lethbridge of the University of Ottawa who assisted us in the development of UML and OCL examples and developed the case study that accompanies this book; Dale Skrien of Colby College who developed the UML tutorial in Appendix 1; William Grosky of the University of Michigan–Dearborn who developed the overview of data science in Appendix 2 with his student Terry Ruas; and our Australian colleague Margaret Kellow for updating the pedagogical Web materials that accompany this book. In addition, we would like to thank Austin Krauss for providing insight into software development in the video game industry from his perspective as a senior software engineer.

Special Thanks BRM: I am grateful to have had the opportunity to work with Roger on the ninth edition of this book. During the time I have been working on this book, my son, Benjamin, has become a software engineering manager and my daughter, Katherine, used her art background to create the figures that appear in the book chapters. I am quite pleased to see the adults they have become and enjoy my time with their children

(Isla, Emma, and Thelma). I am very grateful to my wife, Norma, for her enthusiastic support as I filled my free time working on this book.

RSP: As the editions of this book have evolved, my sons, Mathew and Michael, have grown from boys to men. Their maturity, character, and success in the real world have been an inspiration to me. After many years of following our own professional paths, the three of us now work together in a business that we founded in 2012. Nothing has filled me with more pride. Both of my sons now have children of their own, Maya and Lily, who start still another generation. Finally, to my wife, Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of "the book."

Bruce R. Maxim
Roger S. Pressman

Design element: Quick Look icon magnifying glass: © Roger Pressman

Affordability & Outcomes = Academic Freedom!

You deserve choice, flexibility and control. You know what's best for your students and selecting the course materials that will help them succeed should be in your hands.

That's why providing you with a wide range of options that lower costs and drive better outcomes is our highest priority.

Mc Graw Hill connect®

Students—study more efficiently, retain more and achieve better outcomes. Instructors—focus on what you love—teaching.



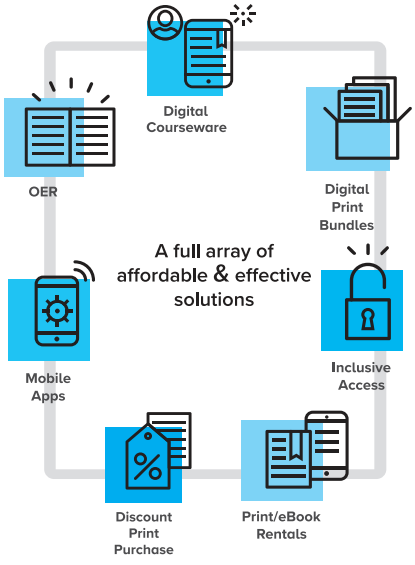
Laptop: McGraw-Hill Education

They'll thank you for it.

Study resources in Connect help your students be better prepared in less time. You can transform your class time from dull definitions to dynamic discussion. Hear from your peers about the benefits of Connect at **www.mheducation.com/highered/connect/smartbook**

Make it simple, make it affordable.

Connect makes it easy with seamless integration using any of the major Learning Management Systems—Blackboard®, Canvas, and D2L, among others—to let you organize your course in one convenient location. Give your students access to digital materials at a discount with our inclusive access program. Ask your McGraw-Hill representative for more information.



Learning for everyone.

McGraw-Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services office and ask them to email accessibility@mheducation.com, or visit **www.mheducation.com/about/accessibility.html** for more information.

Learn more at: www.mheducation.com/realvalue

Rent It

Affordable print and digital rental options through our partnerships with leading textbook distributors including Amazon, Barnes & Noble, Chegg, Follett, and more.

Go Digital

A full and flexible range of affordable digital solutions ranging from Connect, ALEKS, inclusive access, mobile apps, OER and more.

Get Print

Students who purchase digital materials can get a loose-leaf print version at a significantly reduced rate to meet their individual preferences and budget.

CHAPTER

1 SOFTWARE AND SOFTWARE ENGINEERING

As he finished showing me the latest build of one of the world's most popular first-person shooter video games, the young developer laughed.

"You're not a gamer, are you?" he asked.

I smiled. "How'd you guess?"

The young man was dressed in shorts and a tee shirt. His leg bounced up and down like a piston, burning the nervous energy that seemed to be commonplace among his co-workers.

KEY CONCEPTS

application domains 7
failure curves 5
framework activities 10
general principles 14
legacy software 8
principles 14
problem solving 12
SafeHome 16
software,
  definition 5
  nature of 4
  process 9
  questions about 4
software engineering,
  definition 3
  layers 9
  practice 12
umbrella activities 11
wear 5

QUICK LOOK

What is it? Computer software is a work product that software professionals build and then support over many years. These work products include programs that execute within computers of any size and architecture. Software engineering encompasses a process, a collection of methods (practice), and an array of tools that allow professionals to build high-quality computer software.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it. Software engineers apply the software engineering process.

Why is it important? Software engineering is important because it enables us to build complex systems in a timely manner and with high quality. It imposes discipline to work that can become quite chaotic, but it also allows the people who build computer software to adapt their approach in a manner that best suits their needs.

What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product.

What is the work product? From the software engineer's point of view, the work product is the set of programs, content (data), and other work products that support computer software. But from the user's point of view, the work product is a tool or product that somehow makes the user's world better.

How do I ensure that I've done it right? Read the remainder of this book, select those ideas that are applicable to the software that you build, and apply them to your work.

"Because if you were," he said, "you'd be a lot more excited. You've gotten a peek at our next generation product and that's something that our customers would kill for . . . no pun intended."

We sat in a development area at one of the most successful game developers on the planet. Over the years, earlier generations of the game he demoed sold over 50 million copies and generated billions of dollars in revenue.

"So, when will this version be on the market?" I asked.

He shrugged. "In about five months, and we've still got a lot of work to do."

He had responsibility for game play and artificial intelligence functionality in an application that encompassed more than three million lines of code.

"Do you guys use any software engineering techniques?" I asked, half-expecting that he'd laugh and shake his head.

He paused and thought for a moment. Then he slowly nodded. "We adapt them to our needs, but sure, we use them."

"Where?" I asked, probing.

"Our problem is often translating the requirements the creatives give us."

"The creatives?" I interrupted.

"You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game."

"And after the requirements are established?"

He shrugged. "We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends."

"You know my book?" I was honestly surprised.

"Sure, used it in school. There's a lot there."

"I've talked to some of your buddies here, and they're more skeptical about the stuff in my book."

He frowned. "Look, we're not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques."

"And how will your subset change as the years pass?"

He paused as if to ponder the future. "Games will become bigger and more complex, that's for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don't, we're dead."

*****

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Sixty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the media); that software would be the driving force behind the personal computer revolution; that software applications would be purchased by

consumers using their mobile devices; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than all industrial-era companies; or that a vast software-driven network would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and support high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering.*

To build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- Software has become deeply embedded in virtually every aspect of our lives. The number of people who have an interest in the features and functions provided by a specific application[1] has grown dramatically. *A concerted effort should be made to understand the problem before a software solution is developed.*
- The information technology requirements demanded by individuals, businesses, and governments grow increasingly complex with each passing year. Large teams of people now create computer programs. Sophisticated software that was once implemented in a predictable, self-contained computing environment is now embedded inside everything from consumer electronics to medical devices to autonomous vehicles. *Design has become a pivotal activity.*
- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic consequences. *Software should exhibit high quality.*
- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time in use increase, demands for adaptation and enhancement will also grow. *Software should be maintainable.*

These simple realities lead to one conclusion: *Software in all its forms and across all its application domains should be engineered.* And that leads us to the topic of this book—*software engineering.*

1 We will call these people "stakeholders" later in this book.

1.1 The Nature of Software

Today, software takes on a dual role. It is a product, and the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, by a network of computers that are accessible by local hardware. Whether it resides within a mobile device, on the desktop, in the cloud, or within a mainframe computer or autonomous machine, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as an augmented-reality representation derived from data acquired from dozens of independent sources and then overlaid on the real world. As the vehicle used to deliver a product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information.* It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet); and provides the means for acquiring information in all its forms. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

The role of computer software has undergone significant change over the last 60 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build and protect complex systems.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone programmer are the same questions that are asked when modern computer-based systems are built:[2]

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

2 In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin asking 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

These, and many other questions, are a manifestation of the concern about software and how it is developed—a concern that has led to the adoption of software engineering practice.

1.1.1 Defining Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:

> Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information; and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered. But a more formal definition probably won't measurably improve your understanding. To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has one fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected, and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust,

FIGURE 1.1

Failure curve for hardware

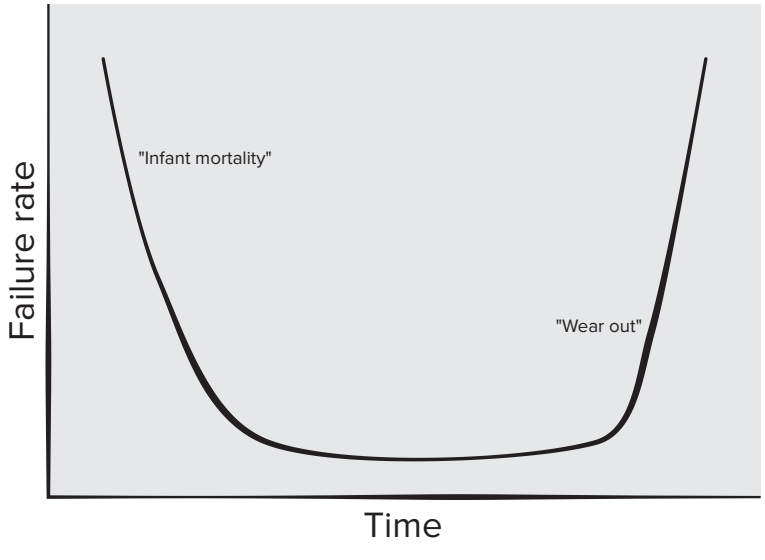
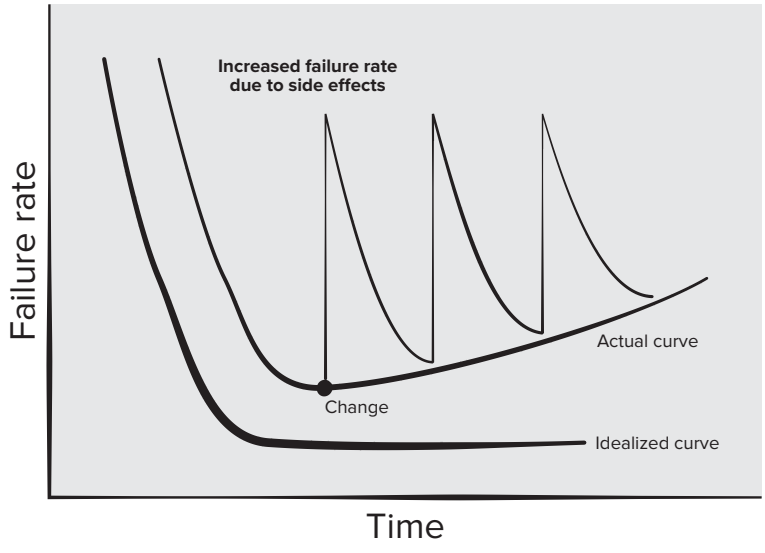


FIGURE 1.2

Failure curves for software



vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does *deteriorate!*

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,[3] software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

3 In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by a variety of different stakeholders.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software. A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,[4] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

Application software. Stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.

Engineering/scientific software. A broad array of "number-crunching" or data science programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, from computer-aided design to consumer spending habits, and from genetic analysis to meteorology.

Embedded software. Resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software. Composed of reusable components and designed to provide specific capabilities for use by many different customers. It may focus on a limited and esoteric marketplace (e.g., inventory control products) or attempt to address the mass consumer market.

Web/mobile applications. This network-centric software category spans a wide array of applications and encompasses browser-based apps, cloud computing, service-based computing, and software that resides on mobile devices.

Artificial intelligence software. Makes use of heuristics[5] to solve complex problems that are not amenable to regular computation or straightforward analysis. Applications within this area include robotics, decision-making systems, pattern recognition (image and voice), machine learning, theorem proving, and game playing.

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.

---

4 Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

5 The use of heuristics is an approach to problem solving that employs a practical method or "rule of thumb" not guaranteed to be perfect, but sufficient for the task at hand.

1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

These changes may create an additional side effect that is often present in legacy software—*poor quality*.[6] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, and a poorly managed change history. The list can be quite long. And yet, these systems often support "core functions and are indispensable to the business." What to do?

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it work with other more modern systems or databases.
- The software must be re-architected to make it viable within an evolving computing environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable in the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution; that is, the notion that software systems change continually, new software systems can be built from the old ones, and . . . all must interact and cooperate with each other" [Day99].

1.2 Defining the Discipline

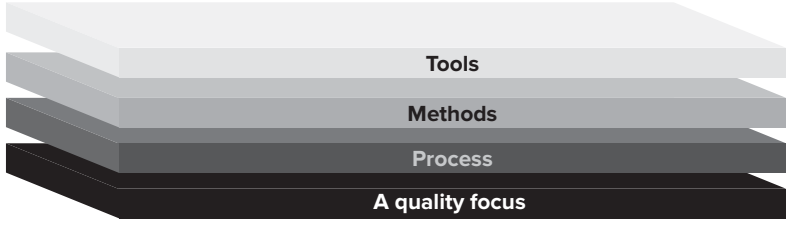
The IEEE [IEE17] has developed the following definition for software engineering:

> Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

6 In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

FIGURE 1.3

Software engineering layers



And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. You may have heard of total quality management (TQM) or Six Sigma, and similar philosophies[7] that foster a culture of continuous process improvement. It is this culture that ultimately leads to more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

1.3 THE SOFTWARE PROCESS

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which

7 Quality management and related approaches are discussed throughout Part Three of this book.

software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

1.3.1 The Process Framework

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders).[8] The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

---

8 A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, and so forth. Rob Thomsett jokes that, "a stakeholder is a person holding a large and sharp stake. . . . If you don't look after your stakeholders, you know where the stake will end up."

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs; the creation of Web applications; and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

1.3.2 Umbrella Activities

Software engineering process framework activities are complemented by a number of *umbrella activities.* In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

Software project tracking and control. Allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management. Assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance. Defines and conducts the activities required to ensure software quality.

Technical reviews. Assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement. Defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management. Manages the effects of change throughout the software process.

Reusability management. Defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production. Encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Each of these umbrella activities is discussed in detail later in this book.

1.3.3 Process Adaptation

Previously in this section, we noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational

culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are:

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor with which the process is described
- Degree to which the customer and other stakeholders are involved with the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

In Part One of this book, we examine software process in considerable detail.

1.4 SOFTWARE ENGINEERING PRACTICE

In Section 1.3, we introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.[9]

1.4.1 The Essence of Practice

In the classic book *How to Solve It,* written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

Understand the Problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and

9 You should revisit relevant sections within this chapter as we discuss specific software engineering methods and umbrella activities later in this book.

then think, *Oh yeah, I understand, let's get on with solving this thing*. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the Solution. Now you understand the problem (or so you think), and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry Out the Plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the Result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

1.4.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., creating a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:[10]

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is no, don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is not to say that features should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the simpler ones. Simple does not mean "quick and dirty." It often takes a lot of thought and work over multiple iterations to simplify the design. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume*

Always specify, design, document, and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

---

10 Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

The Fifth Principle: *Be Open to the Future*

In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this, systems must be ready to adapt to these and other changes. Systems that do this successfully have been designed this way from the start. *Never design yourself into a corner.* Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[11]

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[12] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh Principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results.* When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

1.5 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a "legacy system" to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

11 This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

12 Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than building targeted software. In some cases, the cost differential cannot be justified.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

SAFEHOME[13]



How a Project Starts

The scene: Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive vice president, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11n wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-businesspeople with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off the shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs and tablets have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer app. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as $30 to $40 million in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* product line. The engineering effort will succeed only if *SafeHome* software succeeds. The market will accept the product only if the software embedded within it properly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

13 *SafeHome* will be used throughout this book to illustrate the inner workings of project teams as they build a software product. The company, the project, and the people are purely fictitious, but the situations and problems are real.

1.6 SUMMARY

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 60 years, software has evolved from a specialized problem-solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem-solving activity that follows a set of core principles. As you learn more about software engineering, you'll begin to understand why these principles should be considered when beginning any software project.

PROBLEMS AND POINTS TO PONDER

1.1. Provide at least five additional examples of how the law of unintended consequences applies to computer software.

1.2. Provide a number of examples (both positive and negative) that indicate the impact of software on our society.

1.3. Develop your own answers to the five questions asked at the beginning of Section 1.1. Discuss them with your fellow students.

1.4. Many modern applications change frequently—before they are presented to the end user and then after the first version has been put into use. Suggest a few ways to build software to stop deterioration due to change.

1.5. Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.

1.6. As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm, either economic or human.

1.7. Describe a process framework in your own words. When we say that framework activities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.

1.8. Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities?

Design element: Quick Look icon magnifying glass: © Roger Pressman

PART

One

THE SOFTWARE PROCESS

In this part of *Software Engineering: A Practitioner's Approach,* you'll learn about the process that provides a framework for software engineering practice. These questions are addressed in the chapters that follow:

- What is a software process?
- What are the generic framework activities that are present in every software process?
- How are processes modeled, and what are process patterns?
- What are the prescriptive process models, and what are their strengths and weaknesses?
- Why is *agility* a watchword in modern software engineering work?
- What is agile software development, and how does it differ from more traditional process models?

Once these questions are answered, you'll be better prepared to understand the context in which software engineering practice is applied.

CHAPTER

2 PROCESS MODELS

Building computer software is an iterative social learning process, and the outcome, something that Baetjer [Bae98] would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

But what exactly is a software process from a technical point of view? Within the context of this book, we define a *software process* as a framework for the activities, actions, and tasks required to build high-quality software. Is "process" synonymous with "software engineering"? The answer is yes and no. A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

More important, software engineering is performed by creative, knowledgeable people who should adapt a mature software process so that it is appropriate for the products that they build and the demands of their marketplace.

KEY CONCEPTS

evolutionary process model 29
generic process model 21
process assessment 24
process flow 24
process improvement 24
process patterns 24
prototyping 26
spiral model 29
task set 21
Unified Process 31
waterfall model 25



QUICK LOOK

What is it? When you work to build a product or system, it's important to follow a series of predictable steps (a road map) that helps you deliver a high-quality product on time. This road map is called a "software process."

Who does it? Software engineers adapt a process to their needs and then follow it. The people who have requested the software also have a role to play in the process of defining, building, and testing it.

Why is it important? A process provides stability, control, and organization to an activity so that it does not become chaotic. However, a modern software engineering process must be "agile." It must include only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? The process that you adopt depends on the software that you're building. A process might be appropriate for creating software for an aircraft avionics system but may not work well for the creation of a mobile app or video game.

What is the work product? The work products are the programs, documents, and data produced by the engineering activities and tasks included in the process.

How do I ensure that I've done it right? The quality, timeliness, and long-term viability of the product built are the best indicators of the success of the process used.

2.1 A Generic Process Model

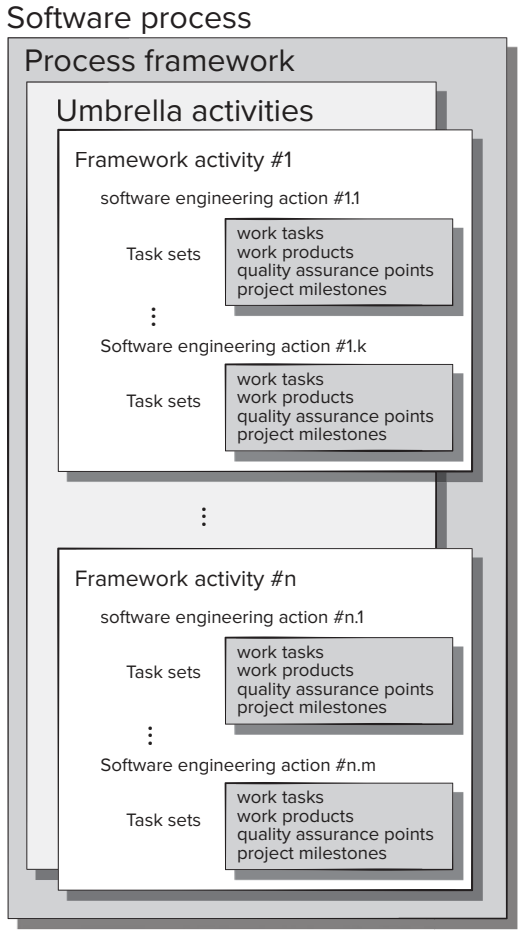
In Chapter 1, a process was defined as a collection of activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 2.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As we discussed in Chapter 1, a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control,

FIGURE 2.1

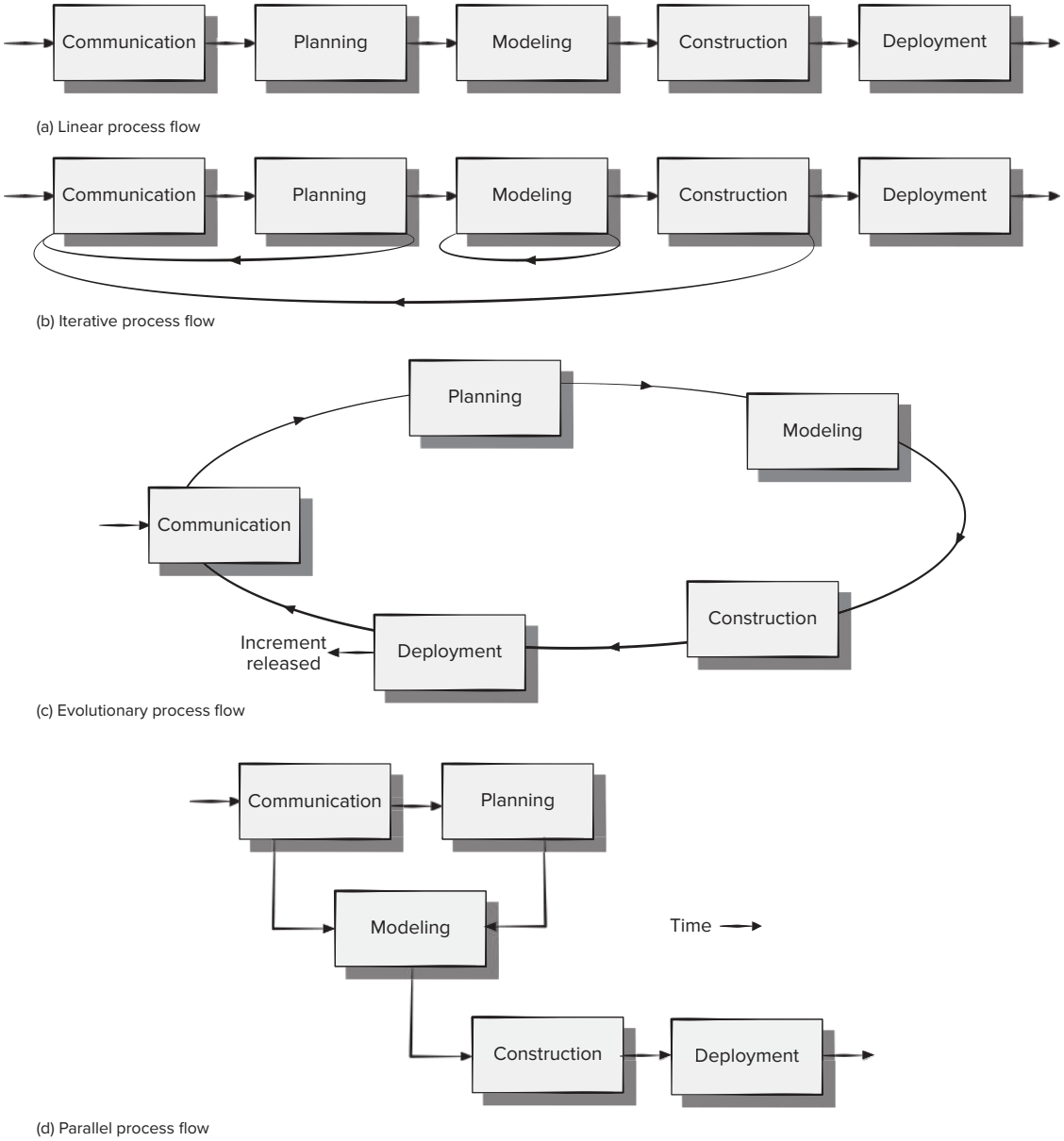
A software process framework



risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

You should note that one important aspect of the software process has not been discussed yet. This aspect—called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time. It is illustrated in Figure 2.2.

FIGURE 2.2 **Process flow**



A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 2.2b). An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c). A *parallel process flow* (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

2.2 DEFINING A FRAMEWORK ACTIVITY

Although we have described five framework activities and provided a basic definition of each in Chapter 1, a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the **communication** activity might encompass little more than a phone call or e-mail with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and develop notes.
3. Organize notes into a brief written statement of requirements.
4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with lots of stakeholders, each with a different set of (sometimes conflicting) requirements, the communication activity might have six distinct actions: *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions might have many work tasks and in some cases a number of distinct work products.

2.3 IDENTIFYING A TASK SET

Referring again to Figure 2.1, each software engineering action (e.g., *elicitation,* an action associated with the **communication** activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. Different projects demand different task sets. You should choose a task set that best accommodates the

TASK SET



A task set defines the actual work that needs to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the **communication** activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve "requirements gathering," but they are quite different in their depth and level of formality. The software team chooses the task set that allows it to achieve the goal for each action and still maintain quality and agility.

needs of the project and the characteristics of your team. This implies that a software engineering action should be adapted to the specific needs of the software project and the characteristics of the project team.

2.4 PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapter 15). Process patterns must be coupled with solid software engineering practice (Part Two of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for successful software engineering.[1]

The current thinking among most engineers is that software processes and activities should be assessed using numeric measures or software analytics (metrics). The

1 The SEI's CMMI-DEV [CMM07] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

progress you have made in a journey toward an effective software process will define the degree to which you can measure improvement in a meaningful way. The use of software process metrics to assess process quality is introduced in Chapter 17. A more detailed discussion of process assessment and improvement methods is presented in Chapter 28.

2.5 PRESCRIPTIVE PROCESS MODELS

Prescriptive process models define a predefined set of process elements and a predictable process work flow. Prescriptive process models[2] strive for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we provide an overview of the prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapter 1, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner. In Chapters 3 and 4 we will discuss software engineering practices that strive to accommodate the changes that are inevitable during the development of many software projects.

2.5.1 The Waterfall Model

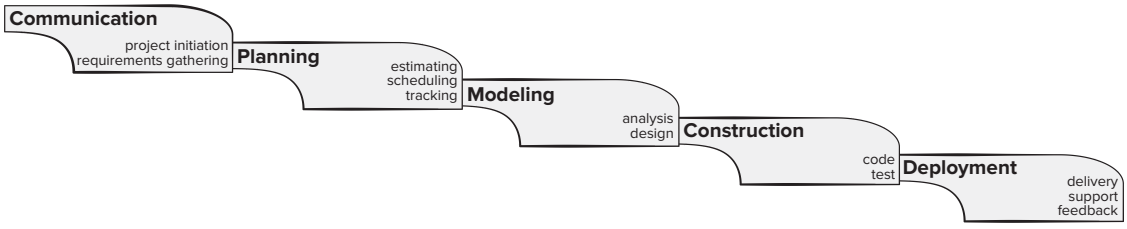
There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software because it needs to accommodate changes to mandated government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The *waterfall model,* sometimes called the *linear sequential model*, suggests a systematic, sequential approach[3] to software development that begins with customer

2 Prescriptive process models are sometimes referred to as "traditional" process models.

3 Although the original waterfall model proposed by Winston Royce [Roy70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

FIGURE 2.3 The waterfall model



specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).

The waterfall model is the oldest paradigm for software engineering. However, over the past five decades, criticism of this process model has caused even ardent supporters to question its efficacy. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential work flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly at the beginning of most projects.
3. The customer must have patience because a working version of the program(s) will not be available until late in the project time span.
4. Major blunders may not be detected until the working program is reviewed.

Today, software work is fast paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work.

2.5.2 Prototyping Process Model

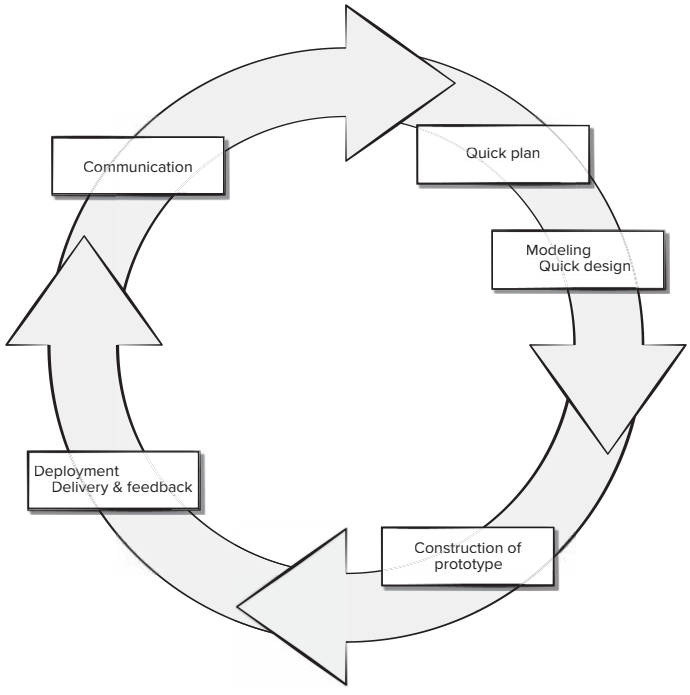
Often, a customer defines a set of general objectives for software but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

For example, a fitness app developed using incremental prototypes might deliver the basic user interface screens needed to sync a mobile phone with the fitness device

FIGURE 2.4

The prototyping paradigm



and display the current data; the ability to set goals and store the fitness device data on the cloud might be included in the second prototype, creating and modifying the user interface screens based on customers feedback; and a third prototype might include social media integration to allow users to set fitness goals and share progress toward them with a set of friends.

The prototyping paradigm (Figure 2.4) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software. They may be unaware that the prototype architecture (program structure) is also evolving. This means that the developers may not have considered the overall software quality or long-term maintainability.
2. As a software engineer, you may be tempted to make implementation compromises to get a prototype working quickly. If you are not careful, these less-than-ideal choices have now become an integral part of the evolving system.

SafeHome



Selecting a Process Model, Part 1

The scene: Meeting room for the software engineering group at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

The conversation:

Lee: So let's recapitulate. I've spent some time discussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply define the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

Doug: Seems like we've been pretty disorganized in our approach to software in the past.

Ed: I don't know, Doug, we always got product out the door.

Doug: True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

Jamie: Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particularly if we have a very tight time line.

Doug (smiling): I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

Jamie (with a frown): My job is to build computer programs, not push paper around.

Doug: Give it a chance before you go negative on me. Here's what I mean. (Doug proceeds to describe the process framework described in Chapter 1 and the prescriptive process models presented to this point.)

Doug: So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

Vinod: Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

Doug: I agree.

Ed: That prototyping approach seems okay. A lot like what we do here anyway.

Vinod: That's a problem. I'm worried that it doesn't provide us with enough structure.

Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built in part to serve as a mechanism for defining requirements. It is often desirable to design a prototype so it can be evolved into the final product. The reality is developers may need to discard (at least in part) a prototype to better meet the customer's evolving needs.

2.5.3 Evolutionary Process Model

Software, like all complex systems, evolves over time. Business and product requirements often change as development proceeds, making a straight-line path to an end product unrealistic. Tight market deadlines may make completion of a comprehensive software product impossible. It might be possible to create a limited version of a product to meet competitive or business pressure and release a refined version once all system features are better understood. In a situation like this you need a process model that has been explicitly designed to accommodate a product that grows and changes.

Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.[4] Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.5. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 26) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

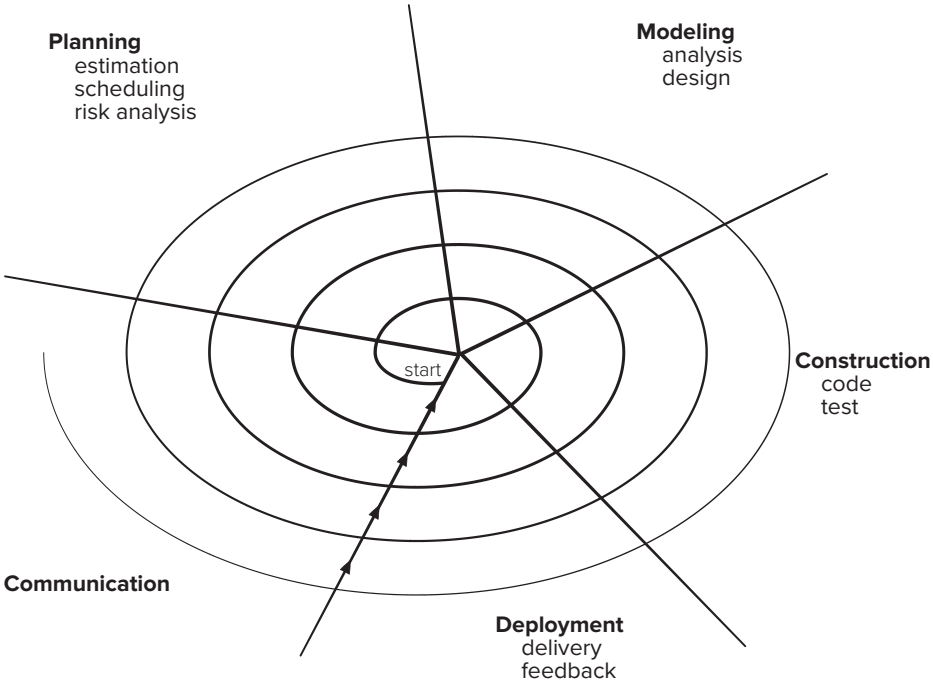
The first circuit around the spiral (beginning at the inside streamline nearest the center, as shown in Figure 2.5) might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. The spiral model

4 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98] and [Boe01a].

FIGURE 2.5

A typical spiral model



is a realistic approach to the development of large-scale systems and software. It uses prototyping as a risk reduction mechanism. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

We have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer-user satisfaction. In many cases, time to market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.[5]

The intent of evolutionary models is to develop high-quality software[6] in an iterative or incremental manner. However, it is possible to use an evolutionary process to

5 It is important to note, however, that being the first to reach a market is no guarantee of success. In fact, many very successful software products have been second or even third to reach the market (learning from the mistakes of their predecessors).

6 In this context, software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Part Two of this book.

SAFEHOME



Selecting a Process Model, Part 2

The scene: Meeting room for the software engineering group at CPI Corporation, a company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Vinod and Jamie, members of the software engineering team.

The conversation: (Doug describes evolutionary process options.)

Jamie: Now I see something I like. An incremental approach makes sense, and I really like the flow of that spiral model thing. That's keepin' it real.

Vinod: I agree. We deliver an increment, learn from customer feedback, re-plan, and then deliver another increment. It also fits into the nature of the product. We can have something on the market fast and then add functionality with each version, er, increment.

Lee: Wait a minute. Did you say that we regenerate the plan with each tour around the spiral, Doug? That's not so great; we need one plan, one schedule, and we've got to stick to it.

Doug: That's old-school thinking, Lee. Like the guys said, we've got to keep it real. I submit that it's better to tweak the plan as we learn more and as changes are requested. It's way more realistic. What's the point of a plan if it doesn't reflect reality?

Lee (frowning): I suppose so, but . . . senior management's not going to like this . . . they want a fixed plan.

Doug (smiling): Then you'll have to reeducate them, buddy.

emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

2.5.4 Unified Process Model

In some ways the Unified Process (UP) [Jac99] is an attempt to draw on the best features and characteristics of traditional software process models but characterize them in a way that implements many of the best principles of agile software development (Chapter 3). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case).[7] It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [Jac99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

7 A *use case* (Chapter 7) is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

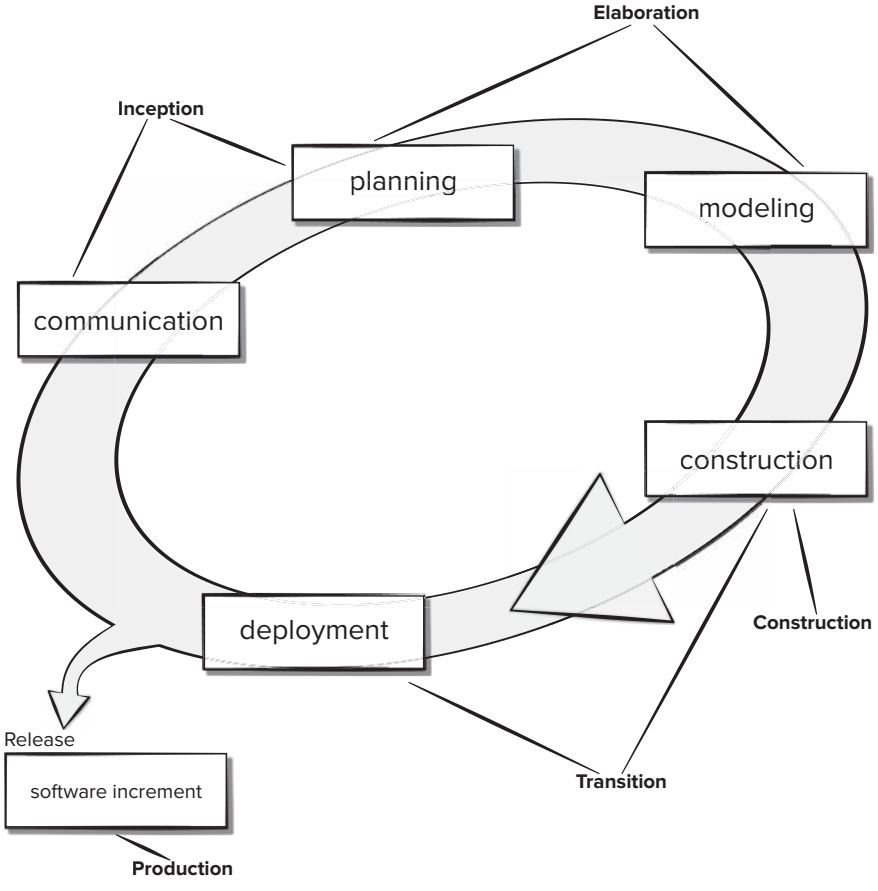
UML, the *unified modeling language,* was developed to support their work. UML contains a robust notation for the modeling and development of object-oriented systems and has became a de facto industry standard for modeling software of all types. UML is used throughout Part Two of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial and a list of recommended books for those who are unfamiliar with basic UML notation and modeling rules.

Figure 2.6 depicts the "phases" of the Unified Process and relates them to the generic activities that were discussed in Section 2.1.

The *inception phase* of the UP is where customer communication and planning takes place. Fundamental business requirements are described through a set of preliminary use cases (Chapter 7) that describe which features and functions each major class of users desires that will become realized in the software architecture. Planning identifies resources, assesses major risks, and defines a preliminary schedule for the software increments.

FIGURE 2.6

The Unified Process



The *elaboration phase* encompasses the planning and modeling activities of the generic process model (Figure 2.6). Elaboration refines and expands the preliminary use cases and creates an architectural baseline that includes five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model.[8] Modifications to the plan are often made at this time.

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests[9] are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software and supporting documentation is given to end users for beta testing, and user feedback reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production phase* of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

2.6 PRODUCT AND PROCESS

Some of the strengths and weaknesses of the process models we have discussed are summarized in Table 2.1. In previous editions of this book we have discussed many others. The reality is that no process is perfect for every project. Usually the software team adapts one or more of the process models discussed in 2.5 or the agile process models discussed in Chapter 3 to meet their needs for the project at hand.

8 It is important to note that the architectural baseline is not a prototype in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

9 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 19 through 21).

TABLE 2.1
Comparing process models

Waterfall pros	It is easy to understand and plan. It works for well-understood small projects. Analysis and testing are straightfoward.
Waterfall cons	It does not accommodate change well. Testing occurs late in the process. Customer approval is at the end.
Prototyping pros	There is a reduced impact of requirement changes. The customer is involved early and often. It works well for small projects. There is reduced likelihood of product rejection.
Prototyping cons	Customer involvement may cause delays. There may be a temptation to "ship" a prototype. Work is lost in a throwaway prototype. It is hard to plan and manage.
Spiral pros	There is continuous customer involvement. Development risks are managed. It is suitable for large, complex projects. It works well for extensible products.
Spiral cons	Risk analysis failures can doom the project. The project may be hard to manage. It requires an expert development team.
Unified Process pros	Quality documentation is emphasized. There is continuous customer involvement. It accommodates requirements changes. It works well for maintenance projects.
Unified Process cons	Use cases are not always precise. It has tricky software increment integration. Overlapping phases can cause problems. It requires an expert development team.

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless comments on the duality of product and process:

> About every ten years give or take five, the software community redefines "the problem" by shifting its focus from product issues to process issues. . . .
>
> While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve "the problem" for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.
>
> . . . You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product.
>
> All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or

> appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.
>
> Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. . . .

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. As a creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

2.7 SUMMARY

A generic process model for software engineering encompasses a set of framework and umbrella activities, actions, and work tasks. Each of a variety of process models can be described by a different process flow—a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

Prescriptive process models have been applied for many years in an effort to bring order and structure to software development. Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

Sequential process models, such as the waterfall model, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply across all software engineering activities—from concept development to long-term system maintenance.

The Unified Process is a "use case driven, architecture-centric, iterative and incremental" software process designed as a framework for UML methods and tools.

PROBLEMS AND POINTS TO PONDER

2.1. Baetjer [Bae98] notes: "The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]." List five questions that (1) designers should ask users, (2) users should ask designers, (3) users should ask themselves about the software product that is to be built, and (4) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

2.2. Discuss the differences among the various process flows described in Section 2.1. Identify the types of problems that might be applicable to each of the generic flows described.

2.3. Try to develop a set of actions for the communication activity. Select one action, and define a task set for it.

2.4. A common problem during communication occurs when you encounter two stakeholders who have conflicting ideas about what the software should be. That is, they have mutually conflicting requirements. Develop a process pattern that addresses this problem and suggest an effective approach to it.

2.5. Provide three examples of software projects that would be amenable to the waterfall model. Be specific.

2.6. Provide three examples of software projects that would be amenable to the prototyping model. Be specific.

2.7. As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?

2.8. Is it possible to combine process models? If so, provide an example.

2.9. What are the advantages and disadvantages of developing software in which quality is "good enough"? That is, what happens when we emphasize development speed over product quality?

2.10. It is possible to prove that a software component and even an entire program is correct? So why doesn't everyone do this?

2.11. Are the Unified Process and UML the same thing? Explain your answer.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

3 AGILITY AND PROCESS

In 2001, a group of noted software developers, writers, and consultants [Bec01] signed the "Manifesto for Agile Software Development" in which they argued in favor of "individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan."

KEY CONCEPTS

acceptance tests 48
Agile Alliance 40
agile process 40
agility 38
agility principles 40
cost of change 39
DevOps 50
Extreme Programming (XP) 46
Kanban 48
pair programming 48
politics of agile development 41
project velocity 47
refactoring 48
Scrum 42

QUICK LOOK

What is it? Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged).

Who does it? Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

Why is it important? Modern business environments that spawn computer-based systems and software products are fast paced and ever changing. Agile software engineering represents a reasonable alternative to conventional software engineering. It has been demonstrated to deliver successful systems quickly.

What are the steps? Agile development might best be termed "software engineering lite." The basic framework activities—communication, planning, modeling, construction, and deployment—remain. But they morph into a minimal task set that pushes the project team toward construction and delivery.

What is the work product? The most important work product is an operational "software increment" that is delivered to the customer on the appropriate commitment date. The most important documents created are the use stories and their associated test cases.

How do I ensure that I've done it right? If the agile team agrees that the process works, and the team produces deliverable software increments that satisfy the customer, you've done it right.

The underlying ideas that guide agile development led to the development of agile[1] methods designed to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it may not be applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a mobile application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.

Fluidity implies change, and change is expensive—particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change through the software process.

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 2 have a major failing: *they forget the frailties of the people who build computer software.* Software engineers are not robots. They exhibit great variation in working styles and significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows "tolerance" for the people who do software engineering work.

3.1 What Is Agility?

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] argues that the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.

But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, and between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and deemphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an

1 Agile methods are sometimes referred to as *light methods* or *lean methods.*

agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

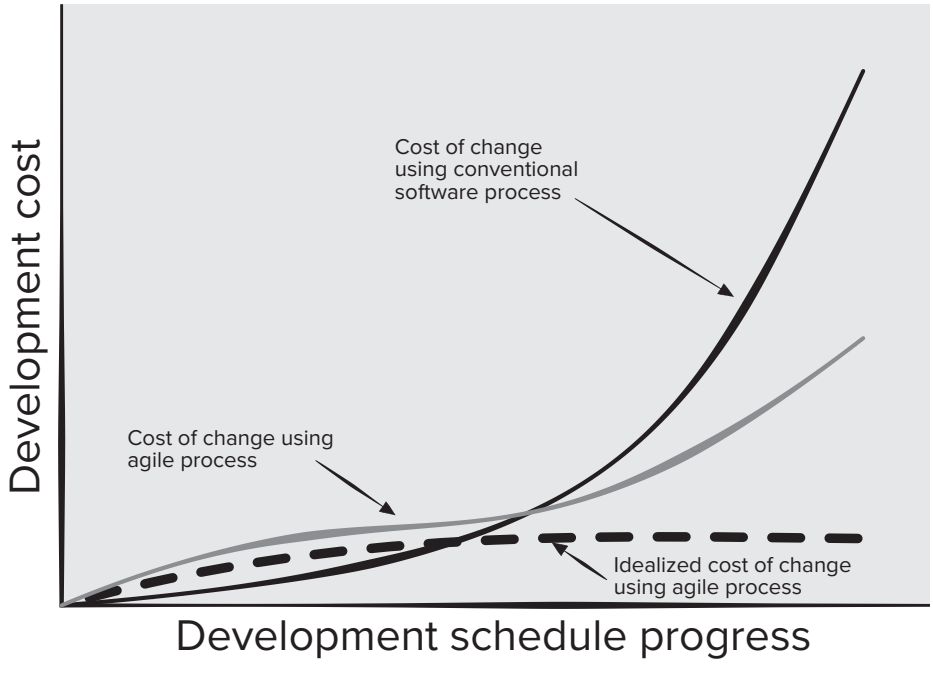
3.2 AGILITY AND THE COST OF CHANGE

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 3.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and effort required to ensure that the change is made without unintended side effects are nontrivial.

Proponents of agility (e.g., [Bec99], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing

FIGURE 3.1

Change costs as a function of time in development



a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed briefly in Section 3.5.1 and in more detail in Chapter 20), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

3.3 What Is an Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability*? The answer, as we have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable.*

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally.* To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

3.3.1 Agility Principles

The Agile Alliance [Agi17][2] defines 12 principles for those software organizations that want to achieve agility. These principles are summarized in the paragraphs that follow.

Customer satisfaction is achieved by providing value through software that is delivered to the customer as rapidly as possible. To achieve this, agile developers

2 The Agile Alliance home page contains much useful information: https://www.agilealliance.org/.

recognize that requirements will change. They deliver software increments frequently and work together with all stakeholders so that feedback on their deliveries is rapid and meaningful.

An agile team is populated by motivated individuals, who communicate face-to face and work in an environment that is conducive to high quality software development. The team follows a process that encourages technical excellence and good design, emphasizing simplicity—"the art of maximized the amount of work not done" [Agi17]. Working software that meets customer needs is their primary goal, and the pace and direction of the team's work must be "sustainable," enabling them to work effectively for long periods of time.

An agile team is a "self-organizing team"—one that can develop well-structured architectures that lead to solid designs and customer satisfaction. Part of the team culture is to consider its work introspectively, always with the intent of improving the manner in which it addresses its primary goal.

Not every agile process model applies characteristics described in this section with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more agile principles. However, these principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

3.3.2 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"): "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making.

No one is against agility. The real question is: What is the best way to achieve it? Keep in mind that working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability. How do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

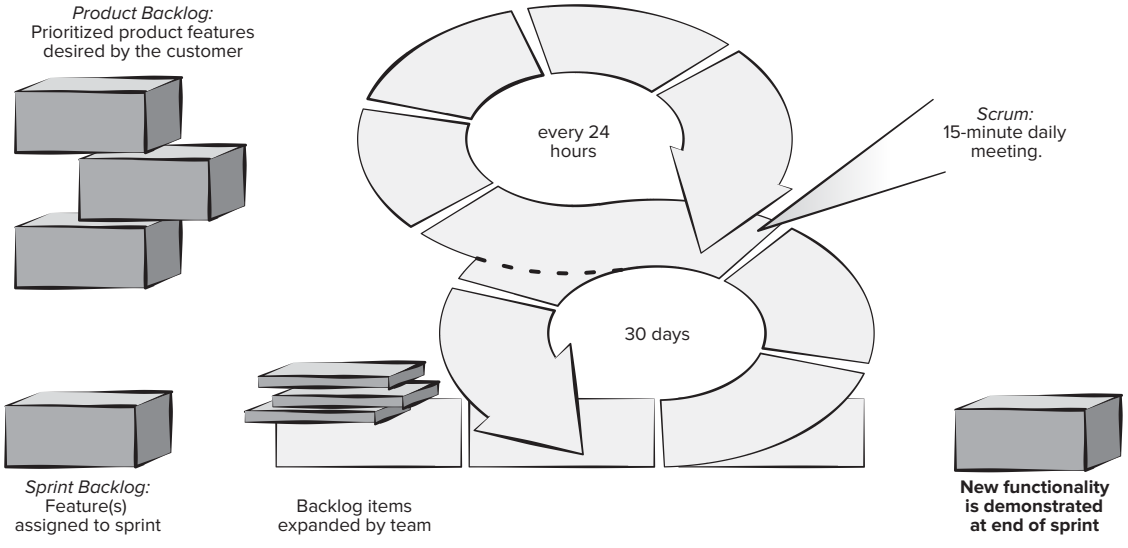
There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed framework models (Sections 3.4 and 3.5), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. The bottom line is there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

3.4 SCRUM

Scrum (the name is derived from an activity that occurs during a rugby match)[3] is a very popular agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Further development on the Scrum methods was performed by Schwaber and Beedle [Sch01b].

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks take place in a relatively short time-boxed[4] period called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on size of the product and its complexity) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure 3.2. Much of our description of the Scrum framework appears in Fowler and Sutherland [Fow16].[5]

FIGURE 3.2 Scrum process flow



3 A group of players forms around the ball, and the teammates work together (sometimes violently!) to move the ball downfield.

4 A *time-box* is a project management term (see Part Four of this book) that indicates a period of time that has been allocated to accomplish some task.

5 The Scrum Guide is available at: https://www.Scrum.org/resources/what-is-Scrum.

SAFEHOME



Considering Agile Software Development

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation: (A knock on the door, Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure Jamie, what's up?

Jamie: We've been thinking about our process discussion yesterday . . . you know, what process we're going to choose for this new *SafeHome* project.

Doug: And?

Vinod: I was talking to a friend at another company, and he was telling me about Scrum. It's an agile process model . . . heard of it?

Doug: Yeah, some good, some bad.

Jamie: Well, it sounds pretty good to us. Lets you develop software really fast, uses something called sprints to deliver software increments when the team decides the product is done . . . it's pretty cool, I think.

Doug: It does have a lot of really good ideas. I like the sprint concept, the emphasis on early test case creation, and the idea that the process owner should be part of the team.

Jamie: Huh? You mean that marketing will work on the project team with us?

Doug (nodding): They're stakeholders but not really the product owner. That would be Marg.

Jamie: Good. She will filter the changes marketing will want to send every 5 minutes.

Vinod: Even so, my friend said that there are ways to "embrace" changes during an agile project.

Doug: So you guys think we should use Scrum?

Jamie: It's definitely worth considering.

Doug: I agree. And even if we choose an incremental model as our approach, there's no reason why we can't incorporate much of what Scrum has to offer.

Vinod: Doug, before you said "some good, some bad." What was the bad?

Doug: The thing I don't like is the way Scrum downplays analysis and design . . . sort of says that writing code is where the action is . . .

(The team members look at one another and smile.)

Doug: So you agree with the Scrum approach?

Jamie (speaking for both): It can be adapted to fit our needs. Besides, writing code is what we do, Boss!

Doug (laughing): True, but I'd like to see you spend a little less time coding and then recoding and a little more time analyzing what needs to be done and designing a solution that works.

Vinod: Maybe we can have it both ways, agility with a little discipline.

Doug: I think we can, Vinod. In fact, I'm sure of it.

3.4.1 Scrum Teams and Artifacts

The Scrum team is a self-organizing interdisciplinary team consisting of a *product owner*, a *Scrum master,* and a small (three to six people) *development team.* The principle Scrum artifacts are the *product backlog*, the *sprint backlog*, and the code *increment*. Development proceeds by breaking the project into a series of incremental prototype development periods 2 to 4 weeks in length called *sprints*.

The product backlog is a prioritized list of product requirements or features that provide business value for the customer. Items can be added to the backlog at any time with the approval of the product owner and the consent of the development team. The product owner orders the items in the product backlog to meet the most important goals of all stakeholders. The product backlog is never complete while the product is evolving to meet stakeholder needs. The product owner is the only person who decides whether to end a sprint prematurely or extend the sprint if the increment is not accepted.

The sprint backlog is the subset of product backlog items selected by the product team to be completed as the code increment during the current active sprint. The increment is the union of all product backlog items completed in previous sprints and all backlog items to be completed in the current sprints. The development team creates a plan for delivering a software increment containing the selected features intended to meet an important goal as negotiated with the product owner in the current sprint. Most sprints are time-boxed to be completed in 3 to 4 weeks. How the development team completes the increment is left up to the team to decide. The development team also decides when the increment is done and ready to demonstrate to the product owner. No new features can be added to the sprint backlog unless the sprint is cancelled and restarted.

The Scrum master serves as facilitator to all members of the Scrum team. She runs the daily Scrum meeting and is responsible for removing obstacles identified by team members during the meeting. She coaches the development team members to help each other complete sprint tasks when they have time available. She helps the product owner find techniques for managing the product backlog items and helps ensure that backlog items are stated in clear and concise terms.

3.4.2 Sprint Planning Meeting

Prior to beginning, any development team works with the product owner and all other stakeholders to develop the items in the product backlog. Techniques for gathering these requirements are discussed in Chapter 7. The product owner and the development team rank the items in the product backlog by the importance of the owner's business needs and the complexity of the software engineering tasks (programming and testing) required to complete each of them. Sometimes this results in the identification of missing features needed to deliver the required functionality to the end users.

Prior to starting each sprint, the product owner states her development goal for the increment to be completed in the upcoming sprint. The Scrum master and the development team select the items to move from to the sprint backlog. The development team determines what can be delivered in the increment within the constraints of the time-box allocated for the sprint and, with the Scrum master, what work will be needed to deliver the increment. The development team decides which roles are needed and how they will need to be filled.

3.4.3 Daily Scrum Meeting

The daily Scrum meeting is a 15-minute event scheduled at the start of each workday to allow team members to synchronize their activities and make plans for the next

24 hours. The Scrum master and the development team always attend the daily Scrum. Some teams allow the product owner to attend occasionally.

Three key questions are asked and answered by all team members:

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

The Scrum master leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. It is the Scrum master's task to clear obstacles presented before the next Scrum meeting if possible. These are not problem-solving meetings, those occur off-line and only involve the affected parties. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

Some teams use these meetings to declare sprint backlog items complete or done. When the team considers all sprint backlog items complete, the team may decide to schedule a demo and review of the completed increment with the product owner.

3.4.4 Sprint Review Meeting

The sprint review occurs at the end of the sprint when the development team has judged the increment complete. The sprint review is often time-boxed as a 4-hour meeting for a 4-week sprint. The Scrum master, the development team, the product owner, and selected stakeholders attend this review. The primary activity is a *demo* of the software increment completed during the sprint. It is important to note that the demo may not contain all planned functionality, but rather those functions that were to be delivered within the time-box defined for the sprint.

The product owner may accept the increment as complete or not. If it is not accepted, the product owner and the stakeholders provide feedback to allow a new round of sprint planning to take place. This is the time when new features may be added or removed from the product backlog. The new features may affect the nature of the increment developed in the next sprint.

3.4.5 Sprint Retrospective

Ideally, before beginning another sprint planning meeting, the Scrum master will schedule a 3-hour meeting (for a 4-week sprint) with the development team called a *sprint retrospective*. During this meeting the team discusses:

- What went well in the sprint
- What could be improved
- What the team will commit to improving in the next sprint

The Scrum master leads the meeting and encourages the team to improve its development practices to become more effective for the next sprint. The team plans ways to improve product quality by adapting its definition of "done." At the end of this meeting, the team should have a good idea about the improvements needed in the next sprint and be ready to plan the increment at the next sprint planning meeting.

3.5 Other Agile Frameworks

The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process frameworks—each contending for acceptance within the software development community—the agile movement has followed the same historical path.[6]

As we noted in the last section, one of the most widely used of all agile frameworks is Scrum. But many other agile frameworks have been proposed and are in use across the industry. In this section, we present a brief overview of three popular agile methods: Extreme Programming (XP), Kanban, and DevOps.

3.5.1 The XP Framework

In this section we provide a brief overview of *Extreme Programming* (XP), one of the most widely used approaches to agile software development. Kent Beck [Bec04a] wrote the seminal work on XP.

Extreme Programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.3 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. The key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity (also called the *planning game*) begins with a requirements activity called *listening*. Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *user story* (described in Chapter 7) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.[7] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. It is important to note that new stories can be written at any time.

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

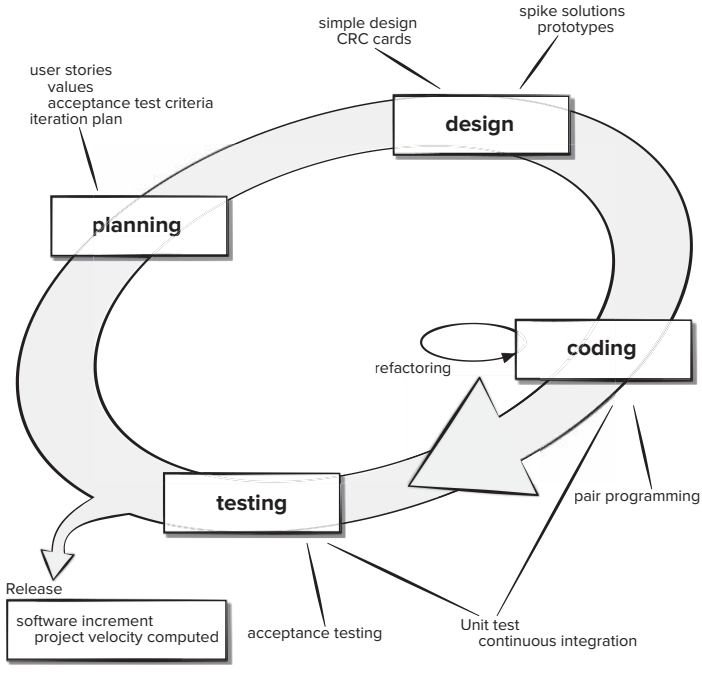
---

6 This is not a bad thing. Before one or more models or methods are accepted as a *de facto* standard, all must contend for the hearts and minds of software engineers. The "winners" evolve into best practice, while the "losers" either disappear or merge with the winning models.

7 The value of a story may also be dependent on the presence of another story.

FIGURE 3.3

The Extreme Programming process



After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the number of customer stories implemented during the first release. Project velocity can then be used to help estimate delivery dates and schedule for subsequent releases. The XP team modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[8]

XP encourages the use of CRC cards (Chapter 8) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes[9] that are relevant to the current software increment. CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the

8 These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

9 Object-oriented classes are discussed throughout Part Two of this book.

design. A central notion in XP is that design occurs both before *and after* coding commences. Refactoring—modifying/optimizing the code in a way that does not change the external behavior of the software [Fow00]—means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

Coding. After user stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).[10] Once the unit test[11] has been created, the developer is better able to focus on what must be implemented to pass the test. Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity (and one of the most talked-about aspects of XP) is *pair programming.* XP recommends that two people work together at one computer to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created).[12]

As pair programmers complete their work, the code they develop is integrated with the work of others. This "continuous integration" strategy helps uncover compatibility and interfacing errors early.

Testing. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages implementing a regression testing strategy (Chapter 20) whenever code is modified (which is often, given the XP refactoring philosophy). XP *acceptance tests,* also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. They are derived from user stories that have been implemented as part of a software release.

3.5.2 Kanban

The *Kanban* method [And16] is a lean methodology that describes methods for improving any process or workflow. Kanban is focused on change management and service delivery. Change management defines the process through which a requested change is integrated into a software-based system. Service delivery is encouraged by focusing on understanding customer needs and expectations. The team members manage the work and are given the freedom to organize themselves to complete it. Policies evolve as needed to improve outcomes.

---

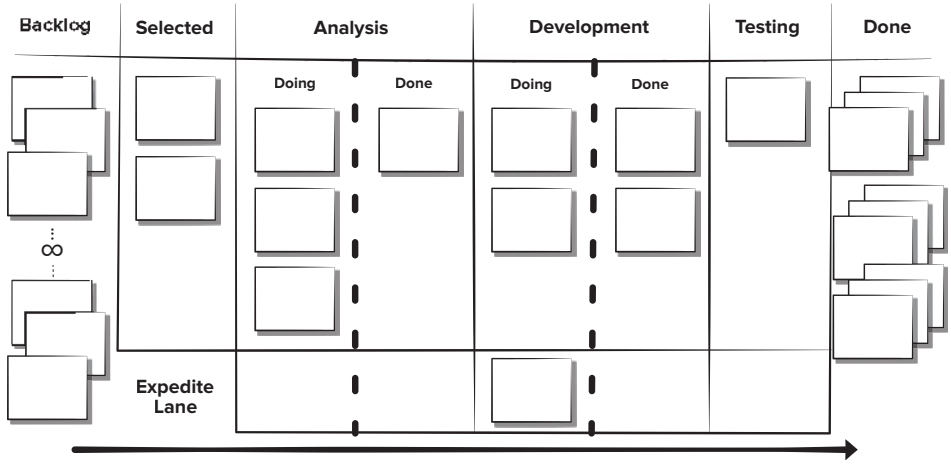
10 This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

11 Unit testing, discussed in detail in Chapter 20, focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.

12 Pair programming has become so widespread throughout the software community that *The Wall Street Journal* [Wal12] ran a front-page story about the subject.

FIGURE 3.4

Kanban board



Kanban originated at Toyota as a set of industrial engineering practices and was adapted to software development by David Anderson [And16]. Kanban itself depends on six core practices:

1. Visualizing workflow using a Kanban board (an example is shown in Figure 3.4). The Kanban board is organized into columns representing the development stage for each element of software functionality. The cards on the board might contain single user stories or recently discovered defects on sticky notes and the team would advance them from "to do," to "doing," and "done" as the project progresses.
2. Limiting the amount of *work in progress* (WIP) at any given time. Developers are encouraged to complete their current task before starting another. This reduces lead time, improves work quality, and increases the team's ability to deliver software functionality frequently to their stakeholders.
3. Managing workflow to reduce waste by understanding the current value flow, analyzing places where it is stalled, defining changes, and then implementing the changes.
4. Making process policies explicit (e.g., write down your reasons for selecting items to work on and the criteria used to define "done").
5. Focusing on continuous improvement by creating feedback loops where changes are introduced based on process data and the effects of the change on the process are measured after the changes are made.[13]
6. Make process changes collaboratively and involve all team members and other stakeholders as needed.

13 The use of process metrics is discussed in Chapter 23.

The team meetings for Kanban are like those in the Scrum framework. If Kanban is being introduced to an existing project, not all items will start in the backlog column. Developers need to place their cards in the team process column by asking themselves: Where are they now? Where did they come from? Where are they going?

The basis of the daily Kanban standup meeting is a task called "walking the board." Leadership of this meeting rotates daily. The team members identify any items missing from the board that are being worked on and add them to the board. The team tries to advance any items they can to "done." The goal is to advance the high business value items first. The team looks at the flow and tries to identify any impediments to completion by looking at workload and risks.

During the weekly retrospective meeting, process measurements are examined. The team considers where process improvements may be needed and proposes changes to be implemented. Kanban can easily be combined with other agile development practices to add a little more process discipline.

3.5.3 DevOps

DevOps was created by Patrick DeBois [Kim16a] to combine Development and Operations. DevOps attempts to apply agile and lean development principles across the entire software supply chain. Figure 3.5 presents an overview of the DevOps workflow. The DevOps approach involves several stages that loop continuously until the desired product exists:

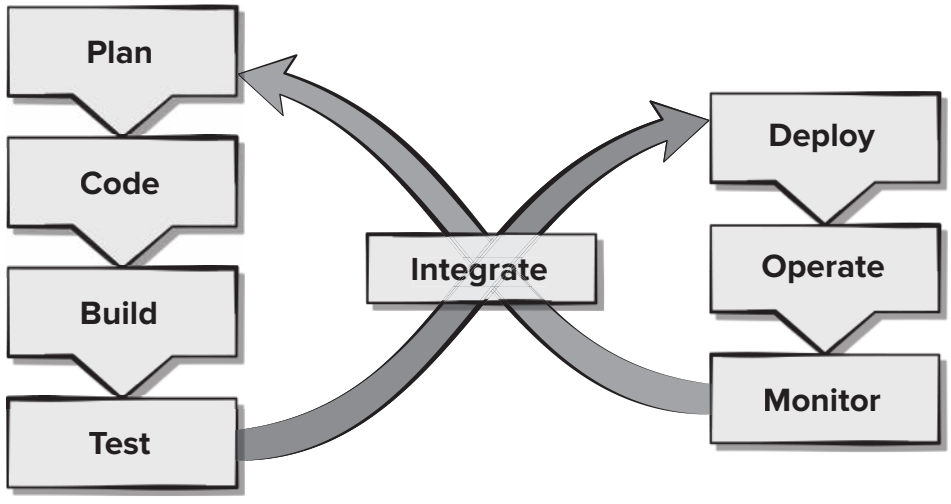
- **Continuous development.** Software deliverables are broken down and developed in multiple sprints with the increments delivered to the quality assurance[14] members of the development team for testing
- **Continuous testing.** Automated testing tools[15] are used to help team members test multiple code increments at the same time to ensure they are free of defects prior to integration.
- **Continuous integration.** The code pieces with new functionality are added to the existing code and to the run-time environment and then checked to ensure there are no errors after deployment.
- **Continuous deployment.** At this stage the integrated code is deployed (installed) to the production environment, which might include multiple sites globally that need to be prepared to receive the new functionality.
- **Continuous monitoring.** Operations staff who are members of the development team help to improve software quality by monitoring its performance in the production environment and proactively looking for possible problems before users find them.

DevOps enhances customers' experiences by reacting quickly to changes in their needs or desires. This can increase brand loyalty and increase market share. Lean approaches like DevOps can provide organizations with increased capacity to innovate

14 The quality assurance is discussed in Chapter 17.

15 Automated testing tools are discussed in Chapter 19.

FIGURE 3.5
DevOps



by reducing rework and allowing shifts to higher business value activities. Products do not make money until consumers have access to them, and DevOps can provide faster deployment time to production platforms [Sha17].

3.6 SUMMARY

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile—to define maneuverable, adaptive, lean processes that can accommodate the needs of modern business.

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform, communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity, and an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Some of the strengths and weaknesses of the agile methods we discussed are summarized in Table 3.1. In previous editions of this book we have discussed many others. The reality is that no agile method is perfect for every project. Agile developers work on self-directed teams and are empowered to create their own process models.

Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, changing requirements, and business criticality. There is no reason why a Scrum team could not adopt the use of a Kanban chart to help organize its daily planning meeting.

TABLE 3.1
Comparing agile techniques

Scrum pros	The product owner sets priorities. The team owns decision making. Documentation is lightweight. It supports frequent updating.
Scrum cons	It is difficult to control the cost of changes. It may not be suitable for large teams. It requires expert team members.
XP pros	It emphasizes customer involvement. It establishes rational plans and schedules. There is high developer commitment to the project. There is reduced likelihood of product rejection.
XP cons	There is temptation to "ship" a prototype. It requires frequent meetings about increasing costs. It may allow for excessive changes. There is a dependence on highly skilled team members.
Kanban pros	It has lower budget and time requirements. It allows for early product delivery. Process policies are written down. There is continuous process improvement.
Kanban cons	Team collaboration skills determine success. Poor business analysis can doom the project. Flexibility can cause developers to lose focus. Developer reluctance to use measurement.
DevOps pros	There is reduced time to code deployment. The team has developers and operations staff. The team has end-to-end project ownership. There is proactive monitoring of deployed product.
DevOps cons	There is pressure to work on both old and new code. There is heavy reliance on automated tools to be effective. Deployment may affect the production environment. It requires an expert development team.

Extreme programming (XP) is organized around four framework activities—**planning, design, coding,** and **testing**—XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and then prioritized by stakeholders. There is nothing preventing them from using DevOps techniques to decrease their time to deployment.

PROBLEMS AND POINTS TO PONDER

3.1. Read the "Manifesto of Agile Software Development" [Bec01] noted at the beginning of this chapter. Can you think of a situation in which one or more of the four "values" could get a software team into trouble?

3.2. Describe agility (for software projects) in your own words.

3.3. Why does an iterative process make it easier to manage change? Is every agile process discussed in this chapter iterative? Is it possible to complete a project in just one iteration and still be agile? Explain your answers.

3.4. Try to come up with one more "agility principle" that would help a software engineering team become even more maneuverable.

3.5. Why do requirements change so much? After all, don't people know what they want?

3.6. Most agile process models recommend face-to-face communication. Yet today, members of a software team and their customers may be geographically separated from one another. Do you think this implies that geographical separation is something to avoid? Can you think of ways to overcome this problem?

3.7. Write a user story that describes the "favorite places" or "favorites" feature available on most Web browsers.

3.8. Describe the XP concepts of refactoring and pair programming in your own words.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

4 RECOMMENDED PROCESS MODEL

In Chapters 2 and 3, we provided brief descriptions of several software process models and software engineering frameworks. Every project is different, and every team is different. There is no single software engineering framework that is appropriate for every software product. In this chapter, we'll share our thoughts on using an adaptable process that can be tailored to fit the needs of software developers working on many types of products.

KEY CONCEPTS

defining requirements 57
estimating resources 60
evaluating prototype. 64
go, no-go decision. 65
maintain. 69
preliminary architectural design 59
prototype construction 61
prototype evolution. 67
release candidate 68
risk assessment. 66
scope definition. 67
test and evaluate. 63

QUICK LOOK

What is it? Every software product needs a "road map" or "generic software process" of some kind. It doesn't have to be complete before you start, but you need to know where you're headed before you begin. Any road map or generic process should be based on best industry practices.

Who does it? Software engineers and their product stakeholders collaborate to adapt a generic software process model to meet the needs of team and then either follow it directly, or more likely, adapt as needed. Every software team should be disciplined but flexible and self-empowered when needed.

Why is it important? Software development can easily become chaotic without the control and organization offered by a defined process. As we stated in Chapter 3, a modern software engineering approach must be "agile" and embrace changes that are needed to satisfy the stakeholders' requirements. It is important not to be too focused on documents and rituals. The process should only include those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? Even if a generic process must be adapted to meet the needs of the specific products being built, you need to be sure that all stakeholders have a role to play in defining, building, and testing the evolving software. There is likely to be substantial overlap among the basic framework activities (communication, planning, modeling, construction, and deployment). *Design a little, build a little, test a little, repeat* is a better approach than creating rigid project plans and documents for most software projects.

What is the work product? From the point of view of a software team, the work products are working program increments, useful documents, and data that are produced by the process activities.

How do I ensure that I've done it right? The timeliness, levels of stakeholder satisfaction, overall quality, and long-term viability of the product increments built are the best indicators that your process is working.

A paper by Rajagoplan [Raj14] reviews the general weaknesses of prescriptive software life-cycle approaches (e.g., the waterfall model) and contains several suggestions that should be considered when organizing a modern software development project.

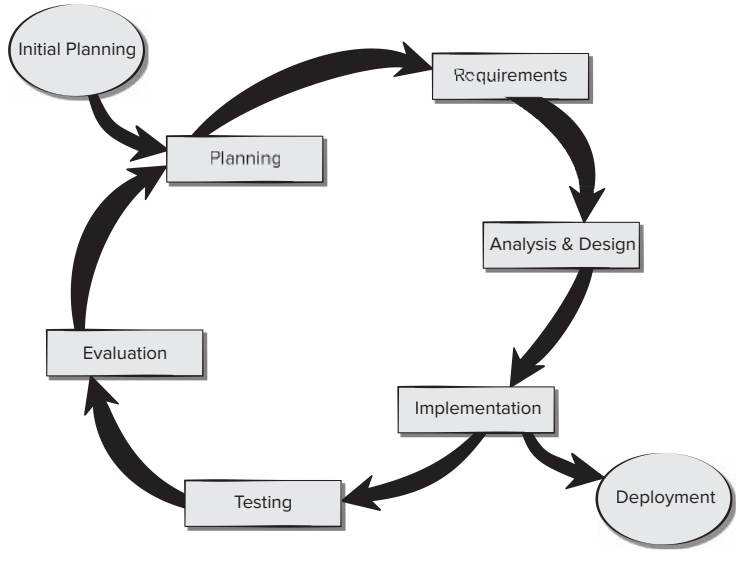
1. It is risky to use a linear process model without ample feedback.
2. It is never possible nor desirable to plan big up-front requirements gathering.
3. Up-front requirements gathering may not reduce costs or prevent time slippage.
4. Appropriate project management is integral to software development.
5. Documents should evolve with the software and should not delay the start of construction.
6. Involve stakeholders early and frequently in the development process.
7. Testers need to become involved in the process prior to software construction.

In Section 2.6, we listed the pros and cons of several prescriptive process models. The waterfall model is not amenable to changes that may need to be introduced once developers start coding. Stakeholder feedback is therefore limited to the beginning and end of the project. Part of the reason for this is the waterfall model suggests that all analysis and design work products be completed before any programming or testing occurs. This makes it hard to adapt to projects with evolving requirements.

One temptation is to switch to an incremental model (Figure 4.1) like the prototyping model or Scrum. Incremental process models involve customers early and often and therefore reduce the risk of creating product that is not accepted by the customers. There is a temptation to encourage lots of changes as stakeholders view each prototype and realize that functions and features they now realize they need are missing. Often, developers do not plan for prototype evolution and create throwaway prototypes.

FIGURE 4.1

Incremental model for prototype development



Recall that the goal of software engineering is to reduce unnecessary effort, so prototypes need to be designed with reuse in mind. Incremental models do provide a better basis for creating an adaptable process if changes can be managed wisely.

In Section 3.5, we discussed the pros and cons of several agile process models other than Scrum. Agile process models are very good at accommodating the uncertain knowledge about the stakeholders' real needs and problems. Key characteristics of agile process models are:

- Prototypes created are designed to be extended in future software increments.
- Stakeholders are involved throughout the development process.
- Documentation requirements are lightweight, and documentation should evolve along with the software.
- Testing is planned and executed early.

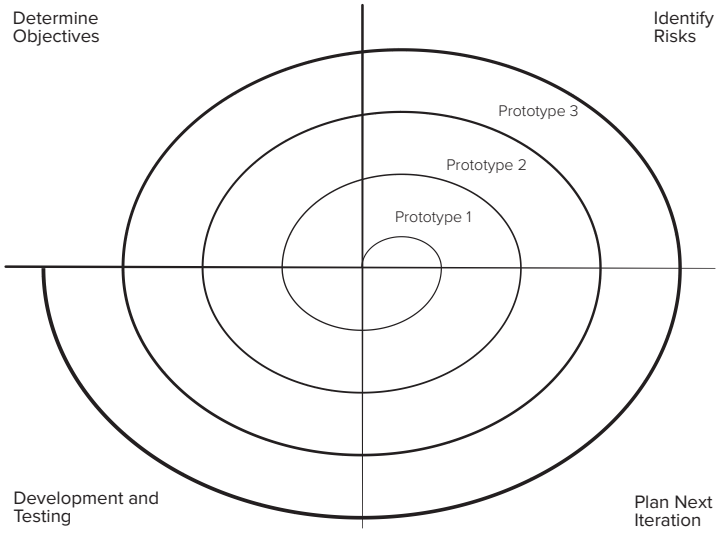
Scrum and Kanban extend these characteristics. Scrum is sometimes criticized for requiring too many meetings. But daily meetings make it hard for developers to stray too far from building products that stakeholders find useful. Kanban (Section 3.5.2) provides a good lightweight tracking system for managing the status and priorities of user stories.

Both Scrum and Kanban allow for controlled introduction of new requirements (user stories). Agile teams are small by design and may not be suitable for projects requiring large numbers of developers, unless the project can be partitioned into small and independently assignable components. Still, agile process models offer many good features that can be incorporated into an adaptable process model.

The spiral model (Figure 4.2) can be thought of as an evolutionary prototyping model with a risk assessment element. The spiral model relies on moderate stakeholder

FIGURE 4.2

Spiral model for prototype development



involvement and was designed for large teams and large projects. Its goal is to create extensible prototypes each time the process is iterated. Early testing is essential. Documentation evolves with the creation of each new prototype. The spiral model is somewhat unique in that formal risk assessment is built in and used as the basis for deciding whether to invest the resources needed to create the next prototype. Some people argue that it may be hard to manage a project using the spiral model, because the project scope may not be known at the start of the project. This is typical of most incremental process models. The spiral is a good basis for building an adaptable process model.

How do agile process models compare to evolutionary models? We've summarized some of the key characteristics in a sidebar.

Characteristics of Agile Models

Agile

1. Not suitable for large high-risk or mission critical projects.
2. Minimal rules and minimal documentation
3. Continuous involvement of testers
4. Easy to accommodate product changes
5. Depends heavily on stakeholder interaction
6. Easy to manage
7. Early delivery of partial solutions
8. Informal risk management
9. Built-in continuous process improvement

Spiral

1. Not suitable for small, low-risk projects
2. Several steps required, along with documentation done up front
3. Early involvement of testers (might be done by outside team)
4. Hard to accommodate product changes until prototype completed
5. Continuous stakeholder involvement in planning and risk assessment
6. Requires formal project management and coordination
7. Project end not always obvious
8. Good risk management
9. Process improvement handled at end of project

Creative, knowledgeable people perform software engineering. They adapt software processes to make them appropriate for the products that they build and to meet the demands of the marketplace. We think that using a spiral-like approach that has agility built into every cycle is a good place to start for many software projects. Developers learn many things as they proceed in the development process. That's why it is important for developers to be able to adapt their process as quickly as practical to accommodate this new knowledge.

4.1 Requirements Definition

Every software project begins with the team trying to understand the problem to be solved and determining what outcomes are important to the stakeholders. This includes understanding the business needs motivating the project and the technical issues which constrain it. This process is called *requirements engineering* and will be discussed in

more detail in Chapter 7. Teams that fail to spend a reasonable amount of time on this task will find that their project contains expensive rework, cost overruns, poor product quality, late delivery times, dissatisfied customers, and poor team morale. Requirements engineering cannot be neglected, nor can it be allowed to iterate endlessly before proceeding to product construction.

It's reasonable to ask what best practices should be followed to achieve thorough and agile requirements engineering. Scott Ambler [Amb12] suggests several best practices for agile requirements definition:

1. Encourage active stakeholder participation by matching their availability and valuing their input.
2. Use simple models (e.g., Post-it notes, fast sketches, user stories) to reduce barriers to participation.
3. Take time to explain your requirement representation techniques before using them.
4. Adopt stakeholder terminology, and avoid technical jargon whenever possible.
5. Use a breadth-first approach to get the big picture of the project done before getting bogged down in details.
6. Allow the development team to refine (with stakeholder input) requirement details "just in time" as user stories are scheduled to be implemented.
7. Treat the list of features to be implemented like a prioritized list, and implement the most important user stories first.
8. Collaborate closely with your stakeholders and only document requirements at a level that is useful to all when creating the next prototype.
9. Question the need to maintain models and documents that will not be referred to in the future.
10. Make sure you have management support to ensure stakeholder and resource availability during requirements definition.

It is important to recognize two realities: (1) it is impossible for stakeholders to describe an entire system before seeing the working software, and (2) it is difficult for stakeholders to describe quality requirements needed for the software before seeing it in action. Developers must recognize that requirements will be added and refined as the software increments are created. Capturing stakeholders' descriptions about what the system needs to do in their own words in a user story is a good place to begin.

If you can get stakeholders to define acceptance criteria for each user story, your team is off to a great start. It is likely that stakeholders will need to see a user story coded and running to know whether it has been implemented correctly or not. Therefore, requirements definition needs to be done iteratively and include the development of prototypes for stakeholder review.

Prototypes are tangible realizations of project plans that can be easily referenced by stakeholders when trying to describe desired changes. Stakeholders are motivated to discuss requirements changes in more concrete terms, which improves communication. It's important to recognize that prototypes allow developers to focus on short-term goals by only focusing on users' visible behaviors. It will be important to review

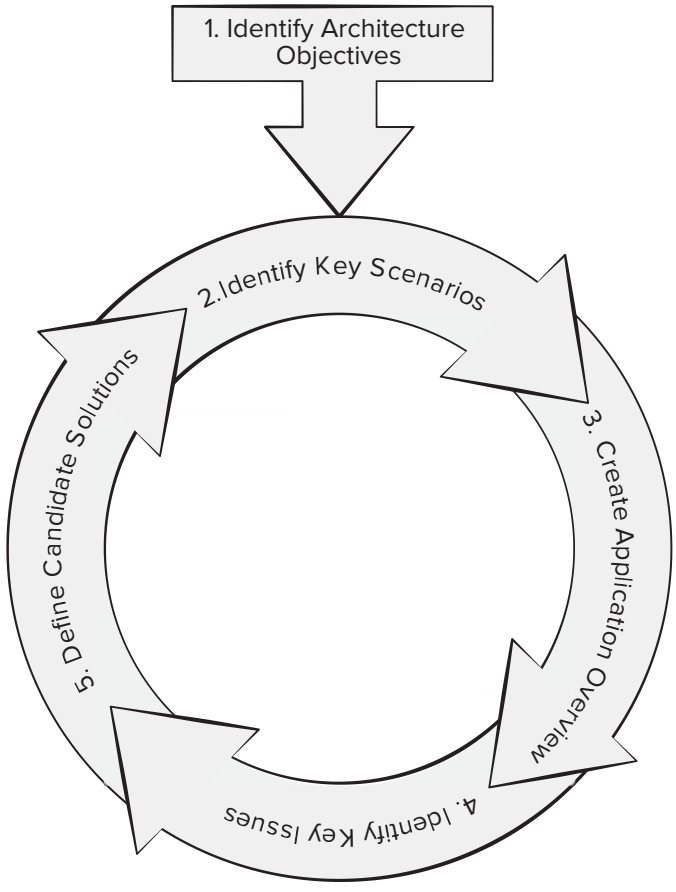
prototypes with an eye to the quality. Developers need to be aware that using prototypes may increase the volatility of the requirements if stakeholders are not focused on getting things right the first time. There is also the risk that creating prototypes before the software architectural requirements are well understood may result in prototypes that must be discarded, wasting time and resources [Kap15].

4.2 Preliminary Architectural Design

The decisions required to develop a solid architectural design are discussed in Chapter 10, but preliminary design decisions must often be made as requirements are defined. As shown in Figure 4.3, at some point in time, architectural decisions will need to be allocated to product increments. According to Bellomo and her colleagues [Bel14], early understanding of requirements and architecture choices is key to managing the development of large or complex software products.

FIGURE 4.3

Architectural design for prototype development



Requirements can be used to inform architecture design. Exploring the architecture as the prototype is developed facilitates the process of detailing the requirements. It is best to conduct these activities concurrently to achieve the right balance. There are four key elements to agile architectural design:

1. Focus on key quality attributes, and incorporate them into prototypes as they are constructed.
2. When planning prototypes, keep in mind that successful software products combine customer-visible features and the infrastructure to enable them.
3. Recognize that an agile architecture enables code maintainability and evolvability if sufficient attention is paid to architectural decisions and related quality issues.
4. Continuously managing and synchronizing dependencies between the functional and architectural requirements is needed to ensure the evolving architectural foundation will be ready just in time for future increments.

Software architecture decision making is critical to the success of a software system. The architecture of a software system determines its qualities and impacts the system throughout its life cycle. Dasanayake et al. [Das15] found that software architects are prone to making errors when their decisions are made under levels of uncertainty. Architects make fewer bad decisions if they can reduce this uncertainty through better architectural knowledge management. Despite the fact that agile approaches discourage heavy documentation, failing to record design decisions and their rationale early in the design process makes it hard to revisit them when creating future prototypes. Documenting the right things can assist with process improvement activities. Documenting your lessons learned is one of the reasons that retrospectives should be conducted after evaluating the delivered prototype and before beginning the next program increment. Reuse of previously successful solutions to architectural problems is also helpful and will be discussed in Chapter 14.

4.3 Resource Estimation

One of the more controversial aspects of using spiral or agile prototyping is estimating the time it will take to complete a project when it cannot be defined completely. It is important to understand before you begin whether you have a reasonable chance of delivering software products on time and with acceptable costs before you agree to take on the project. Early estimates run the risk of being incorrect because the project scope is not well defined and is likely to change once development starts. Estimates made when the project is almost finished do not provide any project management guidance. The trick is to estimate the software development time early based on what is known at the time and revise your estimates on a regular basis as requirements are added or after software increments are delivered. We discuss methods of estimating project scope in Chapter 25.

Let's examine how an experienced software project manager might estimate a project using the agile spiral model we have proposed. The estimates produced by this

method would need to be adjusted for the number of developers and the number of user stories that can be completed simultaneously.

1. Use historic data (Chapter 23), and work as a team to develop an estimate of how many days it will take to complete each of the user stories known at the start of the project.
2. Loosely organize the user stories into the sets that will make up each sprint[1] (Section 3.4) planned to complete a prototype.
3. Sum the number of days to complete each sprint to provide an estimate for the duration of the total project.
4. Revise the estimate as requirements are added to the project or prototypes are delivered and accepted by the stakeholders.

Keep in mind that doubling the number of developers almost never cuts the development time in half.

Rosa and Wallshein [Ros17] found that knowing initial software requirements at the start of a project provides an adequate but not always accurate estimate of project completion times. To get more accurate estimates, it is also important to know the type of project and the experience of the team. We will describe more detailed estimation techniques (e.g., function points or use case points) in Part Four of this book.

4.4 First Prototype Construction

In Section 2.5.2 we described the creation of prototypes as a means of helping the stakeholders move from statements of general objectives and user stories to the level of detail that developers will need to implement this functionality. Developers may use the first prototype to prove that their initial architectural design is a feasible approach to delivering the required functionality while satisfying the customer's performance constraints. To create an operational prototype suggests that requirements engineering, software design, and construction all proceed in parallel. This process is shown in Figure 4.1. This section describes steps that will be used to create the first prototypes. Details of best practices for software design and construction appear later in this book.

Your first task is to identify the features and functions that are most important to the stakeholders. These will help define the objectives for the first prototype. If the stakeholders and developers have created a prioritized list of user stories, it should be easy to confirm which are the most important.

Next, decide how much time will be allowed to create the first prototype. Some teams may choose a fixed time, such as a 4-week sprint, to deliver each prototype. In this case, the developers will look at their time and resource estimate and determine which of the high-priority user stories can be finished in 4 weeks. The team would then confirm with the stakeholders that the selected user stories are the best ones to include in the first prototype. An alternative approach would be to have the stakeholders

1 Sprint was described (Section 3.4) as a time period in which a subset of the system user stories will be delivered to the product owner.

and developers jointly choose a small number of high-priority user stories to include in the first prototype and use their time and resource estimates to develop the schedule to complete the first prototype.

The engineers working at National Instruments published a white paper that outlines their process for creation of a first functional prototype [Nat15]. These steps can be applied to a variety of software projects:

1. Transition from paper prototype to software design
2. Prototype a user interface
3. Create a virtual prototype
4. Add input and output to your prototype
5. Engineer your algorithms
6. Test your prototype
7. Prototype with deployment in mind

Referring to these seven steps, creating a *paper prototype* for a system is very inexpensive and can be done early in the development process. Customers and stakeholders are not usually experienced developers. Nontechnical users can often recognize what they like or do not like about a user interface very quickly once they see it sketched out. Communications between people are often filled with misunderstandings. People forget to tell each other what they really need to know or assume that everyone has the same understanding. Creating a paper prototype and reviewing it with the customer before doing any programming can help avoid wasted time building the wrong prototype. We will talk about several diagrams that can be used to model a system in Chapter 8.

Creating a *prototype user interface* as part of the first functional prototype is a wise idea. Many systems are implemented on the Web or as mobile applications and rely heavily on touch user interfaces. Computer games and virtual reality applications require a great deal of communication with end users to operate correctly. If customers find a software product easy to learn and use, they are more likely to use it.

Many misunderstandings between developers and stakeholders can be alleviated by beginning with a paper prototype of the user interface. Sometimes stakeholders need to see the basics of the user interface in action to be able to explain what they really like and dislike about it. It is less expensive to throw away an early user interface design than to finish the prototype and try to put a new user interface on top of it. Designing user interfaces that provide good user experiences is discussed in Chapter 12.

Adding input and output to your user interface prototype provides an easy way to begin testing the evolving prototype. Testing software component interfaces should be accomplished before testing the code that makes up the component's algorithms. To test the algorithms themselves, developers often use a "test frame" to ensure their implemented algorithms are working as intended. Creating a separate test frame and throwing it away is often a poor use of project resources. If properly designed, the user interface can serve as the test frame for component algorithms, thereby eliminating the effort required to build separate test frames.

Engineering your algorithms refers to the process of transforming your ideas and sketches into programming language code. You need to consider both the functional

requirements stated in the user story and the performance constraints (both explicit and implicit) when designing the necessary algorithms. This is the point where additional support functionality is likely to be identified and added to the project scope, if it does not already exist in a code library.

Testing your prototype demonstrates required functionality and identifies yet undiscovered defects before demonstrating it to the customer. Sometimes it is wise to involve the customer in the testing process before the prototype is finished to avoid developing the wrong functionality. The best time to create test cases is during requirements gathering or when use cases have been selected for implementation. Testing strategies and tactics are discussed in Chapters 19 through 21.

Prototyping with deployment in mind is very important because it helps you to avoid taking shortcuts that lead to creating software that will be hard to maintain in the future. This is not saying that every line of code will make it to the final software product. Like many creative tasks, developing a prototype is iterative. Drafts and revisions are to be expected.

As prototype development occurs, you should carefully consider the software architectural choices you make. It is relatively inexpensive to change a few lines of code if you catch errors before deployment. It is very expensive to change the architecture of a software application once it has been released to end users around the globe.

SafeHome



Room Designer
Considering First Prototype

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation: (A knock on the door. Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure, Jamie, what's up?

Jamie: We've been thinking about the scope of this *SafeHome* room design tool.

Doug: And?

Vinod: There's a lot of work that needs to be done on the back end of the project before people can start dropping alarm sensors and trying furniture layouts.

Jamie: We don't want to work on the back end for months and then have the project cancelled when the marketing folks decide they hate the product.

Doug: Have you tried to work out a paper prototype and review it with the marketing group?

Vinod: Well, no. We thought it was important to get a working computer prototype quickly and did not want to take the time to do one.

Doug: My experience is that people need to see something before they know whether they like it or not.

Jamie: Maybe we should step back and create a paper prototype of the user interface and get them to work with it and see if they like the concept at all.

Vinod: I suppose it wouldn't be too hard to program an executable user interface using the game engine we were considering using for the virtual reality version of the app.

Doug: Sounds like a plan. Try that approach, and see if you have the confidence you need to start evolving your prototype.

4.5 Prototype Evaluation

Testing is conducted by the developers as the prototype is being built and becomes an important part of prototype evaluation. Testing demonstrates that prototype components are operational, but it's unlikely that test cases will have found all the defects. In the spiral model, the results of evaluation allow the stakeholders and developers to assess whether it is desirable to continue development and create the next prototype. Part of this decision is based on user and stakeholder satisfaction, and part is derived from an assessment of the risks of cost overruns and failure to deliver a working product when the project is finished. Dam and Siang [Dam17] suggest several best-practice tips for gathering feedback on your prototype.

1. Provide scaffolding when asking for prototype feedback.
2. Test your prototype on the right people.
3. Ask the right questions.
4. Be neutral when presenting alternatives to users.
5. Adapt while testing.
6. Allow the user to contribute ideas.

Providing scaffolding is a mechanism for allowing the user to offer feedback that is not confrontational. Users are often reluctant to tell developers that they hate the product they are using. To avoid this, it is often easier to ask the user to provide feedback using a framework such as "I like, I wish, What if" as a means of providing open and honest feedback. *I like* statements encourage users to provide positive feedback on the prototype. *I wish* statements prompt users to share ideas about how the prototype can be improved. These statements can provide negative feedback and constructive criticism. *What if* statements encourage users to suggest ideas for your team to explore when creating prototypes in future iterations.

Getting the right people to evaluate the prototype is essential to reduce the risk of developing the wrong product. Having development team members do all the testing is not wise because they are not likely to be the representative of the intended user population. It's important to have the right mix of users (e.g., novice, typical, and advanced) to give you feedback on the prototype.

Asking the right questions implies that all stakeholders agree on prototype objectives. As a developer, it's important to keep an open mind and do your best to convince users that their feedback is valuable. Feedback drives the prototyping process as you plan for future product development activities. In addition to general feedback, try to ask specific questions about any new features included in the prototype.

Be neutral when presenting alternatives allows the software team to avoid making users feel they are being "sold" on one way to do things. If you want honest feedback, let the users know that you have not already made up your mind that there is only one right way to do things. *Egoless programming* is a development philosophy that focuses on producing the best product the team can create for the intended users. Although it is not desirable to create throwaway prototypes, egoless programming suggests that things that are not working need to be fixed or

discarded. So try not to become too attached to your ideas when creating early prototypes.

Adapt while testing means that you need a flexible mind-set while users are working with the prototype. This might mean altering your test plan or making quick changes to the prototype and then restarting the testing. The goal is to get the best feedback you can from users, including direct observation of them as they interact with the prototype. The important thing is that you get the feedback you need to help decide whether to build the next prototype or not.

Allow users to contribute ideas means what it says. Make sure you have a way of recording their suggestions and questions (electronically or otherwise). We will discuss additional ways of conducting user testing in Chapter 12.

SafeHome



Room Designer
Evaluating First Prototype

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation: (A knock on the door. Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure, Jamie, what's up?

Jamie: We completed the evaluation of the *SafeHome* room design tool working with our marketing stakeholders.

Doug: How did things go?

Vinod: We mostly focused on the user interface that will allow users to place alarm sensors in the room.

Jamie: I am glad we let them review a paper prototype before we created the PC prototype.

Doug: Why is that?

Vinod: We made some changes, and the marketing people liked the new design better, so that's the design we used when we started programming it.

Doug: Good. What's the next step?

Jamie: We completed the risk analysis and since we did not pick up any new user stories, we think it is reasonable to work on creating the next incremental prototype since we are still on time and within budget.

Vinod: So, if you agree, we'll get the developers and stakeholders together and begin planning the next software increment.

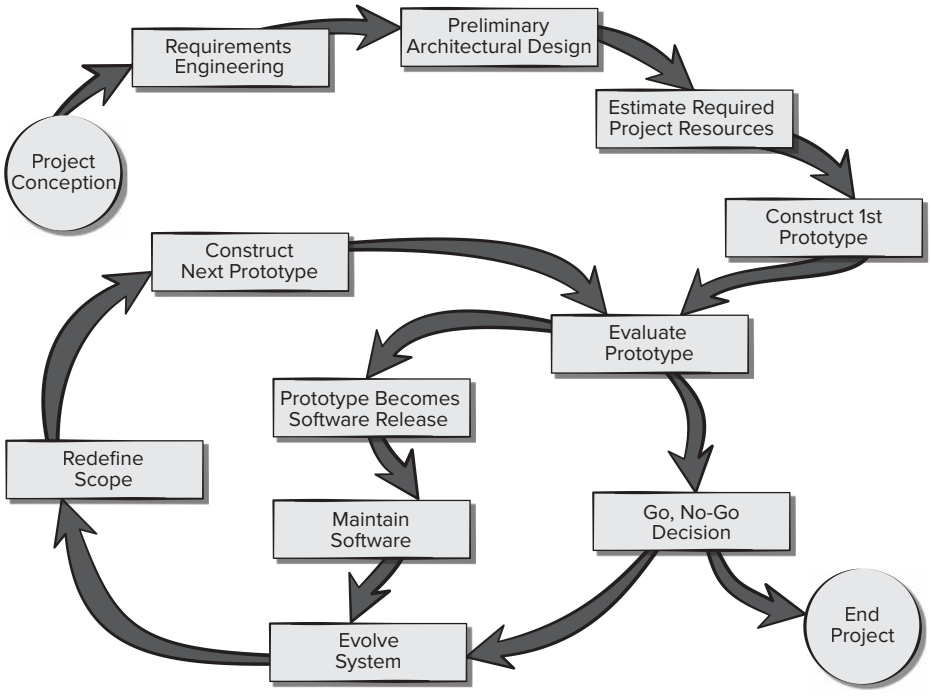
Doug: I agree. Keep me in the loop and try to keep the development time on the next prototype to 6 weeks or less.

4.6 Go, No-Go Decision

After the prototype is evaluated, project stakeholders decide whether to continue development of the software product. If you refer to Figure 4.4, a slightly different decision based on the prototype evaluation might be to release the prototype to the

FIGURE 4.4

Recommended software process model



end users and begin the maintenance process. Sections 4.8 and 4.9 discuss this decision in more detail. We discussed the use of risk assessment in the spiral model as part of the prototype assessment process in Section 2.5.3. The first circuit around the spiral might be used to solidify project requirements. But it really should be more. In the method we are proposing here, each trip around the spiral develops a meaningful increment of the final software product. You can work with the project user story or feature backlog to identify an important subset of the final product to include in the first prototype and repeat this for each subsequent prototype.

A pass through the planning region follows the evaluation process. Revised cost estimates and schedule changes are proposed based on what was discovered when evaluating the current software prototype. This may involve adding new user stories or features to the project backlog as the prototype is evaluated. The risk of exceeding the budget and missing the project delivery date is assessed by comparing new cost and time estimates against old ones. The risk of failing to satisfy user expectations is also considered and discussed with the stakeholders and sometimes senior management before deciding to create another prototype.

The goal of the risk assessment process is to get the commitment of all stakeholders and company management to provide the resources needed to create the next prototype. If the commitment is not there because the risk of project failure is too great, then the project can be terminated. In the Scrum framework (Section 3.4), the go, no-go decision might be made during the Scrum retrospective

meeting held between the prototype demonstration and the new sprint planning meeting. In all cases, the development team lays out the case for the product owners and lets them decide whether to continue product development or not. A more detailed discussion of software risk assessment methods is presented in Chapter 26.

4.7 PROTOTYPE EVOLUTION

Once a prototype has been developed and reviewed by the development team and other stakeholders, it's time to consider the development of the next prototype. The first step is to collect all feedback and data from the evaluation of the current prototype. The developers and stakeholders then begin negotiations to plan the creation of another prototype. Once the desired features for the new prototype have been agreed upon, consideration is given to any known time and budget constraints as well as the technical feasibility of implementing the prototype. If the development risks are deemed to be acceptable, the work continues.

The evolutionary prototyping process model is used to accommodate changes that inevitably occur as software is developed. Each prototype should be designed to allow for future changes to avoid throwing it away and creating the next prototype from scratch. This suggests favoring both well-understood and important features when setting the goals for each prototype. As always, the customer's needs should be given great importance in this process.

4.7.1 New Prototype Scope

The process of determining the scope of a new prototype is like the process of determining the scope of the initial prototype. Developers would either: (1) select features to develop within the time allocated to a sprint, or (2) allocate sufficient time to implement the features needed to satisfy the goals set by the developers with stakeholder input. Either approach requires the developers to maintain a prioritized list of features or user stories. The priorities used to order the list should be determined by the goals set for the prototype by the stakeholders and developers.

In XP (Section 3.5.1), the stakeholders and developers work together to group the most important user stories into a prototype that will become the next release of the software and determine its completion date. In Kanban (Section 3.5.2), developers and stakeholders make use of a board that allows them to focus on the completion status of each user story. This is a visual reference that can be used to assist developers using any incremental prototype process model to plan and monitor the progress of the software development. Stakeholders can easily see the feature backlog and help the developers order it to identify the most useful stories needed in the next prototype. It is probably easier to estimate the time required to complete the selected user stories than to find the user stories that need to fit into a fixed time block. But the advice to keep prototype development time to 4 to 6 weeks should be followed to ensure adequate stakeholder involvement and feedback.

4.7.2 Constructing New Prototypes

A user story should contain both a description of how the customer plans to interact with the system to achieve a specific goal and a description of what the customer's definition of acceptance is. The development team's task is to create additional software components to implement the user stories selected for inclusion in the new prototype along with the necessary test cases. Developers need to continue communication with all stakeholders as they create the new prototype.

What makes this new prototype trickier to build is that software components created to implement new features in the evolving prototype need to work with the components used to implement the features included in the previous prototype. It gets even trickier if developers need to remove components or modify those that were included in the previous prototype because requirements have changed. Strategies for managing these types of software changes are discussed in Chapter 22.

It is important for the developers to make design decisions that will make the software prototype more easily extensible in the future. Developers need to document design decisions in a way that will make it easier to understand the software when making the next prototype. The goal is to be agile in both development and documentation. Developers need to resist the temptation to overdesign the software to accommodate features that may or may not be included in the final product. They also should limit their documentation to that which they need to refer to during development or when changes need to be made in the future.

4.7.3 Testing New Prototypes

Testing the new prototype should be relatively straightforward if the development team created test cases as part of the design process before the programming was completed. Each user story should have had acceptance criteria attached to it as it was created. These acceptance statements should guide the creation of the test cases intended to help verify that the prototype meets customer needs. The prototype will need to be tested for defects and performance issues as well.

One additional testing concern for evolutionary prototypes is to ensure that adding new features does not accidentally break features that were working correctly in the previous prototype. *Regression testing* is the process of verifying that software that was previously developed and tested still performs the same way after it has been changed. It is important to use your testing time wisely and make use of the test cases that are designed to detect defects in the components most likely to be affected by the new features. Regression testing is discussed in more detail in Chapter 20.

4.8 Prototype Release

When an evolutionary prototyping process is applied, it can be difficult for developers to know when a product is finished and ready for release to the customers. Software developers do not want to release a buggy software product to the end users and have them decide the software has poor quality. A prototype being considered as a release candidate must be subjected to user acceptance testing in addition to functional and nonfunctional (performance) testing that would have been conducted during prototype construction.

User acceptance tests are based on the agreed-upon acceptance criteria that were recorded as each user story was created and added to the product backlog. This allows

user representatives to verify that the software behaves as expected and collect suggestions for future improvements. David Nielsen [Nie10] has several suggestions for conducting prototype testing in industrial settings.

When testing a release candidate, functional and nonfunctional tests should be repeated using the test cases that were developed during the construction phases of the incremental prototypes. Additional nonfunctional tests should be created to verify that the performance of the prototype is consistent with the agreed-upon benchmarks for the final product. Typical performance benchmarks may deal with system response time, data capacity, or usability. One of the most important nonfunctional requirements to verify is ensuring that the release candidate will run in all planned run-time environments and on all targeted devices. The process should be focused on testing limited to the acceptance criteria established before the prototype was created. Testing cannot prove a software product is bug free, only that the test cases ran correctly.

User feedback during acceptance testing should be organized by user-visible functions as portrayed via the user interface. Developers should examine the device in question and make changes to the user interface screen if implementing these changes will not delay the release of the prototype. If changes are made, they need to be verified in a second round of testing before moving on. You should not plan for more than two iterations of user acceptance testing.

It is important, even for projects using agile process models, to use an issue tracking or bug reporting system (e.g., Bugzilla[2] or Jira[3]) to capture the testing results. This allows developers to record test failures and makes it easier to identify the test cases that will need to be run again to verify that a repair properly corrects the problem that was uncovered. In each case the developers need to assess whether the changes can be made to the software without causing a cost overrun or late product delivery. The implications of not fixing a problem need to be documented and shared with both the customer and senior managers who may decide to cancel the project rather than committing the resources needed to deliver the final project.

The issues and lessons learned from creating the release candidate should be documented and considered by the developers and stakeholders as part of the project postmortem. This information should be considered before deciding to undertake future development of a software product following its release to the user community. The lessons learned from the current product can help developers make better cost and time estimates for similar projects in the future.

Techniques for conducting user acceptance testing are discussed in Chapters 12 and 20. A more detailed discussion on software quality assurance is presented in Chapter 17.

4.9 Maintain Release Software

Maintenance is defined as the activities needed to keep software operational after it has been accepted and delivered (released) in the end-user environment. Maintenance will continue for the life of the software product. Some software engineers believe that the majority of the money spent on a software product will be spent on maintenance activities. *Corrective maintenance* is the reactive modification of software to repair problems

2 https://www.bugzilla.org/.

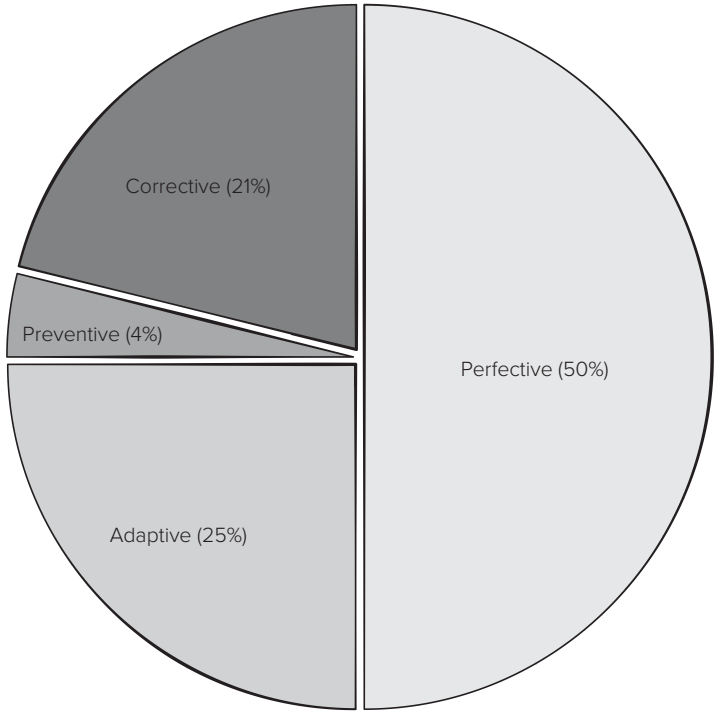
3 https://www.atlassian.com/software/jira.

discovered after the software has been delivered to a customer's end user. *Adaptive maintenance* is reactive modification of software after delivery to keep the software usable in a changing end-user environment. *Perfective maintenance* is the proactive modification of the software after delivery to provide new user features, better program code structure, or improved documentation. *Preventive maintenance* is the proactive modification of the software after delivery to detect and correct product faults before they are discovered by users in the field [SWEBOK[4]]. Proactive maintenance can be scheduled and planned for. Reactive maintenance is often described as *firefighting* because it cannot be planned for and must be attended immediately for software systems that are critical to the success of the end-users' activities. Figure 4.5 shows that only 21 percent of the developers' time is typically spent on corrective maintenance.

For an agile evolutionary process model like the one described in this chapter, developers release working partial solutions with the creation of each incremental prototype. Much of the engineering work done is preventive or perfective maintenance as new features are added to the evolving software system. It is tempting to think that maintenance is handled simply by planning to make another trip around the spiral. But software problems cannot always be anticipated, so repairs may need to be made quickly and developers may be tempted to cut corners when trying to repair the broken

FIGURE 4.5

Distribution of maintenance effort



4 SWEBOK refers to the Software Engineering Body of Knowledge, which can be accessed using the following link: https://www.computer.org/web/swebok/v3.

software. Developers may not want to spend time on risk assessment or planning. Yet, developers cannot afford to make changes to software without considering the possibility that the changes required to fix one problem will cause new problems to other portions of the program.

It is important to understand the program code before making changes to it. If developers have documented the code, it will be easier to understand if other people need to do the maintenance work. If the software is designed to be extended, maintenance may also be accomplished more easily, other than emergency defect repairs. It is essential to test the modified software carefully to ensure software changes have their intended effect and do not break the software in other places.

The task of creating software products that are easy to support and easy to maintain requires careful and thoughtful engineering. A more detailed discussion on the task of maintaining and supporting software after delivery is presented in Chapter 27.

Recommended Software Process Steps

1. Requirements engineering
   - Gather user stories from all stakeholders.
   - Have stakeholders describe acceptance criteria user stories.
2. Preliminary architectural design
   - Make use of paper prototypes and models.
   - Assess alternatives using nonfunctional requirements.
   - Document architecture design decisions.
3. Estimate required project resources
   - Use historic data to estimate time to complete each user story.
   - Organize the user stories into sprints.
   - Determine the number of sprints needed to complete the product.
   - Revise the time estimates as use stories are added or deleted.
4. Construct first prototype
   - Select subset of user stories most important to stakeholders.
   - Create paper prototype as part of the design process.
   - Design a user interface prototype with inputs and outputs.
   - Engineer the algorithms needed for first prototypes.
   - Prototype with deployment in mind.
5. Evaluate prototype
   - Create test cases while prototype is being designed.
   - Test prototype using appropriate users.
   - Capture stakeholder feedback for use in revision process.
6. Go, no-go decision
   - Determine the quality of the current prototype.
   - Revise time and cost estimates for completing development.
   - Determine the risk of failing to meet stakeholder expectations.
   - Get commitment to continue development.
7. Evolve system
   - Define new prototype scope.
   - Construct new prototype.
   - Evaluate new prototype and include regression testing.
   - Assess risks associated with continuing evolution.
8. Release prototype
   - Perform acceptance testing.
   - Document defects identified.
   - Share quality risks with management.
9. Maintain software
   - Understand code before making changes.
   - Test software after making changes.
   - Document changes.
   - Communicate known defects and risks to all stakeholders.

4.10 Summary

Every project is unique, and every development team is made up of unique individuals. Every software project needs a road map, and the process of developing software requires a predictable set of basic tasks (communication, planning, modeling, construction, and deployment). However, these tasks should not be performed in isolation and may need to be adapted to meet the needs of each new project. In this chapter, we suggested the use of a highly interactive, incremental prototyping process. We think this is better than producing rigid product plans and large documents prior to doing any programming. Requirements change. Stakeholder input and feedback should occur early and often in the development process to ensure the delivery of useful product.

We suggest the use of an evolutionary process model that emphasizes frequent stakeholder involvement in the creation and evaluation of incremental software prototypes. Limiting requirements engineering artifacts to the set of minimal useful documents and models allows the early production of prototypes and test cases. Planning to create evolutionary prototypes reduces the time lost repeating the work needed to create throwaway prototypes. Making use of paper prototypes early in the design process can also help to avoid programming products that do not satisfy customer expectations. Getting the architectural design right before beginning actual development is also important to avoiding schedule slippage and cost overruns.

Planning is important but should be done expeditiously to avoid delaying the start of development. Developers should have a general idea about how long a project will take to complete, but they need to recognize that they are not likely to know all the project requirements until the software products are delivered. Developers would be wise to avoid detailed planning that extends beyond planning the current prototype. The developers and stakeholders should adopt a process for adding features to be implemented in future prototypes and to assess the impact of these changes on the project schedule and budget.

Risk assessment and acceptance testing are an important part of the prototype assessment process. Having an agile philosophy about managing requirements and adding new features to the final product is important as well. The biggest challenges developers have with evolutionary process models is managing scope creep while delivering a product that meets customer expectations and doing all this while delivering the product on time and within budget. That's what makes software engineering so challenging and rewarding.

Problems and Points to Ponder

4.1. How does the Extreme Programming (XP) model differ from the spiral model in its treatment of incremental prototypes?

4.2. Write the acceptance criteria for the user story that describe the use of the "favorite places" or "favorites" feature found on most Web browsers that you wrote for Problem 3.7 in Chapter 3.

4.3. How would you create a preliminary architectural design for the first prototype for a mobile app that lets you create and save a shopping list on your device?

4.4. Where would you get the historic date needed to estimate the development time for the user stories in a prototype before it is written?

4.5. Create a series of sketches representing the key screens for a paper prototype for the shopping list app you created in Problem 4.3.

4.6. How can you test the viability of the paper prototype you created for Problem 4.5?

4.7. What data points are needed to make the go, no-go decision during the assessment of an evolutionary prototype?

4.8. What is the difference between reactive and proactive maintenance?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

5 HUMAN ASPECTS OF SOFTWARE ENGINEERING

In a special issue of *IEEE Software*, the guest editors [deS09] make the following observation:

> Software engineering has an abundance of techniques, tools, and methods designed to improve both the software development process and the final product. However, software isn't simply a product of the appropriate technical solutions applied in appropriate technical ways. Software is developed by people, used by people, and supports interaction among people. As such, human characteristics, behavior, and cooperation are central to practical software development.
>
> Without a team of skilled and motivated people, success is unlikely.

KEY CONCEPTS

agile team 78
global teams 80
jelled team 76
psychology, software engineering 75
social media 79
team attributes 77
team structures 78
team toxicity 77
traits, software engineer 75

QUICK LOOK

What is it? At the end of the day, *people* build computer software. The human aspects of software engineering often have as much to do with the success of a project as the latest and greatest technology.

Who does it? Individuals and teams do software engineering work. In some cases, one person has much of the responsibility, but in most industry-grade software efforts, a team of people does the work.

Why is it important? A software team will be successful only if the dynamics of the team are right. It is essential for software engineers on a team to play well with their colleagues and with other product stakeholders.

What are the steps? First, you need to try to emulate personal characteristics of successful software engineers. Next, you should appreciate the complex psychology of software engineering work so that you can navigate your way through a project without peril. Then, you need to understand the structure and dynamics of software teams. Finally, you should appreciate the impact of social media, the cloud, and other collaborative tools.

What is the work product? Better insight into the people, the process, and the product.

How do I ensure that I've done it right? Spend the time to observe how successful software engineers do their work, and tune your approach to take advantage of the strengths they project.

5.1 CHARACTERISTICS OF A SOFTWARE ENGINEER

So you want to be a software engineer? Obviously, you need to master the technical stuff, learn the skills required to understand the problem, design an effective solution, build the software, and test it in an effort to develop the highest-quality products possible. You need to manage change, communicate with stakeholders, and use appropriate tools as needed. We will discuss these things at great length later in this book.

But there are other things that are equally important—the human aspects that will make you effective as a software engineer. Erasmus [Era09] identifies seven traits that are present when a software engineer exhibits "superprofessional" behavior.

An effective software engineer has a sense of *individual responsibility*. This implies a drive to deliver on her promises to peers, stakeholders, and her management. It implies that she will do what needs to be done, when it needs to be done in an overriding effort to achieve a successful outcome.

An effective software engineer has an *acute awareness* of the needs of other team members, the stakeholders requesting changes to an existing software solution, and the managers who have overall control of the project. He observes the environment in which people work and adapts his behavior to take both into account.

An effective software engineer is *brutally honest*. If she sees a flawed design, she points out the flaws in a constructive but honest manner. If asked to distort facts about schedules, features, performance, or other product or project characteristics, she opts to be realistic and truthful.

An effective software engineer exhibits *resilience under pressure.* Software engineering is always on the edge of chaos. Pressure comes in many forms—changes in requirements and priorities, demanding stakeholders, and overbearing managers. An effective software engineer manages pressure so that his performance does not suffer.

An effective software engineer has a *heightened sense of fairness.* She gladly shares credit with her colleagues. She tries to avoid conflicts of interest and never acts to sabotage the work of others.

An effective software engineer exhibits *attention to detail.* This does not imply an obsession with perfection. He carefully considers the broader criteria (e.g., performance, cost, quality) that have been established for the product and the project in making his daily technical decisions.

Finally, an effective software engineer is pragmatic. She recognizes that software engineering is not a religion in which dogmatic rules must be followed, but rather a discipline that can be adapted based on the circumstances at hand.

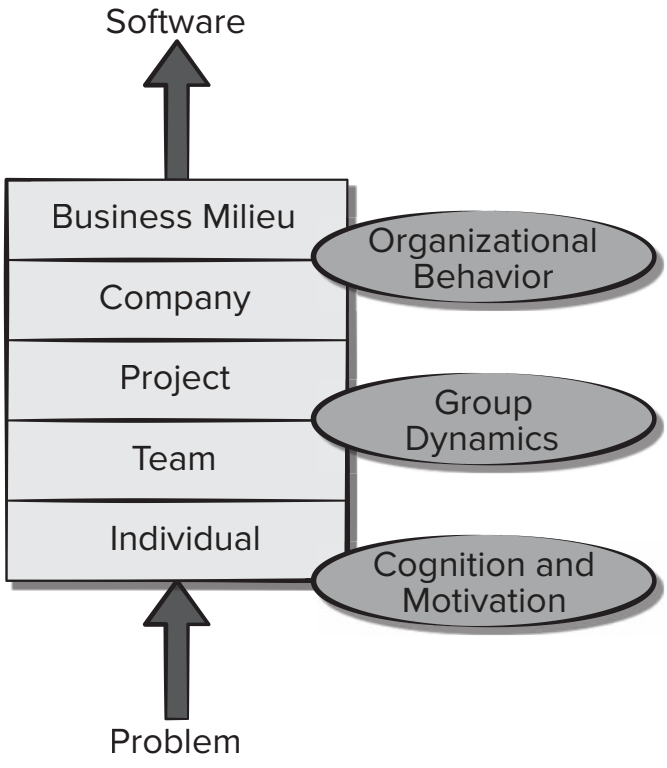
5.2 THE PSYCHOLOGY OF SOFTWARE ENGINEERING

In a seminal paper on the psychology of software engineering, Bill Curtis and Diane Walz [Cur90] suggest a layered behavioral model for software development (Figure 5.1). At an individual level, software engineering psychology focuses on recognition of the problem to be solved, the problem-solving skills required to solve it, and the motivation to complete the solution within the constraints established by outer layers in the model. At the team and project levels, group dynamics becomes the

FIGURE 5.1

A layered behavioral model for software engineering (adapted from [Cur90])

Source: Adapted from Curtis, Bill, and Walz, Diane, "The Psychology of Programming in the Large: Team and Organizational Behavior," Psychology of Programming, Academic Press, 1990.



dominating factor. Here, team structure and social factors govern success. Group communication, collaboration, and coordination are as important as the skills of an individual team member. At the outer layers, organizational behavior governs the actions of the company and its response to the business milieu.

5.3 THE SOFTWARE TEAM

In their classic book *Peopleware,* Tom DeMarco and Tim Lister [DeM98] discuss the cohesiveness of a software team:

> We tend to use the word *team* loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't behave like teams. They may not have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success. . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

There is no foolproof method for creating a jelled team. But there are attributes that are normally found in effective software teams.[1] Miguel Carrasco [Car08] suggests that an effective software team must establish a *sense of purpose*. An effective team must also inculcate a *sense of involvement* that allows every member to feel that his skill set and contributions are valued.

An effective team should foster a *sense of trust.* Software engineers on the team should trust the skills and competence of their peers and their managers. The team should encourage a *sense of improvement,* by periodically reflecting on its approach to software engineering and looking for ways to improve their work.

The most effective software teams are diverse in the sense that they combine a variety of different skill sets. Highly skilled technologists are complemented by members who may have less technical background but are more empathetic to the needs of stakeholders.

But not all teams are effective and not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

To avoid a frenzied work environment, the team should have access to all information required to do the job. Major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built and the people doing the work and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[2] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some work

1 Bruce Tuckman observes that successful teams go through four phases (forming, storming, norming, and performing) on their way to becoming productive (http://en.wikipedia.org/wiki/Tuckman%27s_stages_of_group_development).

2 Technical reviews are discussed in detail in Chapter 16.

hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last-minute deadline. Recognition of human differences, along with other guidelines presented in this section, provide a higher likelihood of creating teams that jell.

5.4 TEAM STRUCTURES

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes a number of project factors that should be considered when planning the structure of software engineering teams: (1) difficulty of the problem to be solved, (2) "size" of the resultant program(s) in lines of code or function points,[3] (3) time that the team will stay together (team lifetime), (4) degree to which the problem can be modularized, (5) required quality and reliability of the system to be built, (6) rigidity of the delivery date, and (7) degree of sociability (communication) required for the project.

Over the past decade, agile software development (Chapter 3) has been suggested as an antidote to many of the problems that have plagued software project work. The agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity.

A small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in Section 5.3 and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency, coupled with group collaboration as critical success factors for the team. Channeling creative energy into a high-performance team must be a central goal of a software engineering organization. Cockburn and Highsmith [Coc01a] suggest that "good" software people can work within the framework of any software process, and weak performers will struggle regardless. The bottom line, they contend, is that "people trump process" but that even good people can be hampered by an ill-defined process and poor resource support. We agree.

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing.* A self-organizing team does not necessarily maintain a single team structure. The team makes the changes needed to its structure to respond to the changes in the development environment or changes in the evolving engineering problem solution.

Communication between team members and all project stakeholders is essential. Agile teams often have customer representatives as team members. This fosters respect among the developers and stakeholders, as well as providing avenues for timely and frequent feedback on the evolving products.

3 Lines of code (LOC) and function points are measures of the size of a computer program and are discussed in Chapter 24.

SAFEHOME



Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team), Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold on to that for a moment. I'd like to talk about how we're going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and the fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

5.5 THE IMPACT OF SOCIAL MEDIA

E-mail, texting, and videoconferencing have become ubiquitous activities in software engineering work. But these communication mechanisms are really nothing more than modern substitutes or supplements for the face-to-face contact. Social media is different.

Begel [Beg10] and his colleagues address the growth and application of social media in software engineering when they write:

> The social processes around software development are . . . highly dependent on engineers' abilities to find and connect with individuals who share similar goals and complementary skills, to harmonize each team member's communication and teaming preferences, to collaborate and coordinate during the entire software lifecycle, and advocate for their product's success in the marketplace.

In some ways, this "connection" can be as important as face-to-face communication. The value of social media grows as team size increases and is magnified further when the team is geographically dispersed.

Social networking tools (e.g., Facebook, LinkedIn, Slack, Twitter) allow degrees-of-separation connections among software developers and related technologists. This allows "friends" on a social networking site to learn about friends of friends who may have knowledge or expertise related to the application domain or problem to be solved. Specialized private networks built on the social networking paradigm can be used within an organization.

It is very important to note that privacy and security issues should not be overlooked when using social media for software engineering work. Much of the work performed by software engineers may be proprietary to their employer and disclosure could be very harmful. For that reason, the distinct benefits of social media must be weighed against the threat of uncontrolled disclosure of private information.

5.6 Global Teams

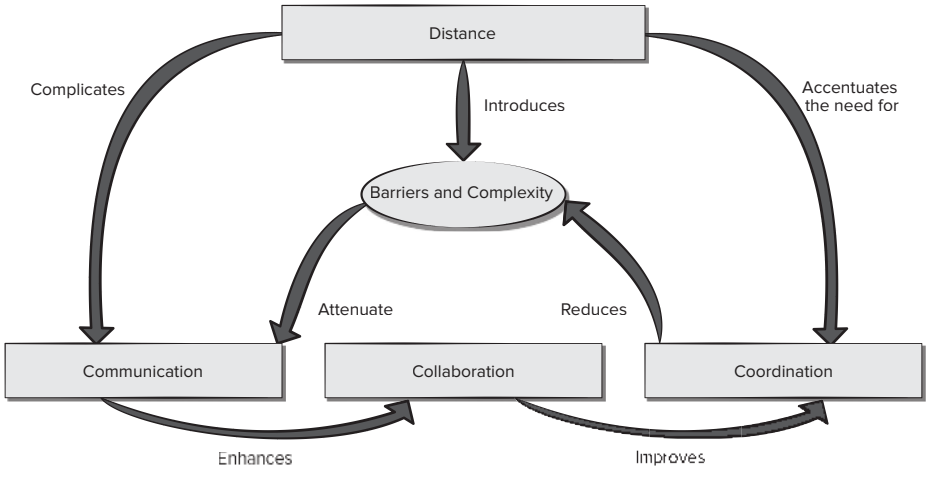
In the software domain, globalization implies more than the transfer of goods and services across international boundaries. For the past few decades, an increasing number of major software products have been built by software teams that are often located in different countries. These global software development (GSD) teams have unique challenges that include coordination, collaboration, communication, and specialized decision making. Approaches to coordination, collaboration, and communication are influenced by the team structure that has been established. Decision making on all software teams is complicated by four factors [Gar10a]:

- Complexity of the problem.
- Uncertainty and risk associated with the decision.
- The law of unintended consequences (i.e., a work-associated decision has an unintended effect on another project objective).
- Different views of the problem that lead to different conclusions about the way forward.

For a GSD team, the challenges associated with coordination, collaboration, and communication can have a profound effect on decision making. Figure 5.2 illustrates the impact of distance on the challenges that face a GSD team. Distance complicates

FIGURE 5.2

Factors affecting a GSD team



communication, but, at the same time, accentuates the need for coordination. Distance also introduces barriers and complexity that can be driven by cultural differences. Barriers and complexity attenuate communication (i.e., the signal-to-noise ratio decreases). The problems inherent in this dynamic can result in a project that becomes unstable.

5.7 SUMMARY

A successful software engineer must have technical skills. But, in addition, he must take responsibility for his commitments, be aware of the needs of his peers, be honest in his assessment of the product and the project, be resilient under pressure, treat his peers fairly, and exhibit attention to detail.

The psychology of software engineering includes individual cognition and motivation, the group dynamics of a software team, and the organizational behavior of the company. A successful ("jelled") software team is more productive and motivated than average. To be effective, a software team must have a sense of purpose, a sense of involvement, a sense of trust, and a sense of improvement. In addition, the team must avoid "toxicity" that is characterized by a frenzied and frustrating work atmosphere, an inappropriate software process, an unclear definition of roles on the software team, and continuous exposure to failure.

Agile teams subscribe to the agile philosophy and generally have more autonomy than more conventional software teams with rigid member roles and external management control. Agile teams emphasize communication, simplicity, feedback, courage, and respect.

Social media tools are becoming an integral part of many software projects, providing services that enhance communication and collaboration for a software team. Social media and electronic communication are particularly useful for global software development where geographic separation can precipitate barriers to successful software engineering.

PROBLEMS AND POINTS TO PONDER

5.1. Based on your personal observation of people who are excellent software developers, name three personality traits that appear to be common among them.

5.2. How can you be "brutally honest" and still not be perceived (by others) as insulting or aggressive?

5.3. How does a software team construct "artificial boundaries" that reduce their ability to communicate with others?

5.4. Write a scenario in which the *SafeHome* team members make use of one or more forms of social media as part of their software project.

5.5. Referring to Figure 5.2, why does distance complicate communication? Why does distance accentuate the need for coordination? What types of barriers and complexities are introduced by distance?

Design element: Quick Look icon magnifying glass: © Roger Pressman

PART

Two

MODELING

In this part of *Software Engineering: A Practitioner's Approach,* you'll learn about the principles, concepts, and methods that are used to create high-quality requirements and design models. These questions are addressed in the chapters that follow:

- What concepts and principles guide software engineering practice?
- What is requirements engineering, and what are the underlying concepts that lead to good requirements analysis?
- How is the requirements model created, and what are its elements?
- What are the elements of a good design?
- How does architectural design establish a framework for all other design actions, and what models are used?
- How do we design high-quality software components?
- What concepts, models, and methods are applied as the user experience is designed?
- What is pattern-based design?
- What specialized strategies and methods are used to design mobile apps?

Once these questions are answered, you'll be better prepared to apply software engineering practice.

CHAPTER

6 Principles That Guide Practice

Software engineers are often depicted as working long hours by themselves to meet impossible deadlines without connecting to other people. This is a dark image of software engineering practice to be sure. Yet, as we discussed in the previous chapters, most software engineers work on teams and frequently interact with their stakeholders. If you search for surveys of technical professionals on the Internet, you will see software engineers listed among those experiencing the greatest satisfaction from their jobs.

Key Concepts

coding principles 96
communication principles 88
core principles 85
deployment principles 98
modeling principles 92
planning principles 91
practice 85
process 85
testing principles 96

Quick Look

What is it? Software engineering practice is a broad array of principles, concepts, methods, and tools that you must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted.

Who does it? Practitioners (software engineers) and their managers conduct a variety of software engineering tasks.

Why is it important? Software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Software practice provides you with the details you'll need to drive along the road. It tells you where the bridges, the roadblocks, and the forks are located. It helps you understand the concepts and principles that must be understood and followed to drive safely and rapidly. It instructs you on how to drive, where to slow down, and where to speed up. In the context of software engineering, practice is what you do day in and day out as software evolves from an idea to a reality.

What are the steps? Four elements of practice apply regardless of the process model that is chosen. They are principles, concepts, methods, and tools. Tools support the application of methods.

What is the work product? Practice encompasses the technical activities that produce all work products that are defined by the software process model that has been chosen.

How do I ensure that I've done it right? First, have a firm understanding of the principles that apply to the work (e.g., design) that you're doing at the moment. Then, be certain that you've chosen an appropriate method for the work, be sure that you understand how to apply the method, use automated tools when they're appropriate for the task, and be adamant about the need for techniques to ensure the quality of work products that are produced. You also need to be agile enough to make changes to your plans and methods as needed.

People who create computer software practice the art or craft or discipline[1] that is software engineering. But what is software engineering "practice"? In a generic sense, *practice* is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how-to's to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

Various aspects of software engineering practice will be examined throughout the remainder of this book. In this chapter, our focus is on principles and concepts that guide software engineering practice in general.

6.1 Core Principles

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs the framework and umbrella activities and produces a set of software engineering work products. At the practice level, core principles establish a collection of values and rules that can guide you in analyzing a problem, designing a solution, implementing and testing the solution, and ultimately deploying the software so stakeholders can use it.

6.1.1 Principles That Guide Process

In Part One of this book we discussed the importance of the software process and described several process models that have been proposed for software engineering work. Every project is unique, and every team is unique. That means you must adapt your process to best fit your needs. Regardless of the process model your team adopts, it contains elements of the generic process framework we described in Chapter 1. The following set of core principles can be applied to this framework and to every software process. A simplified view of this framework is shown in Figure 6.1.

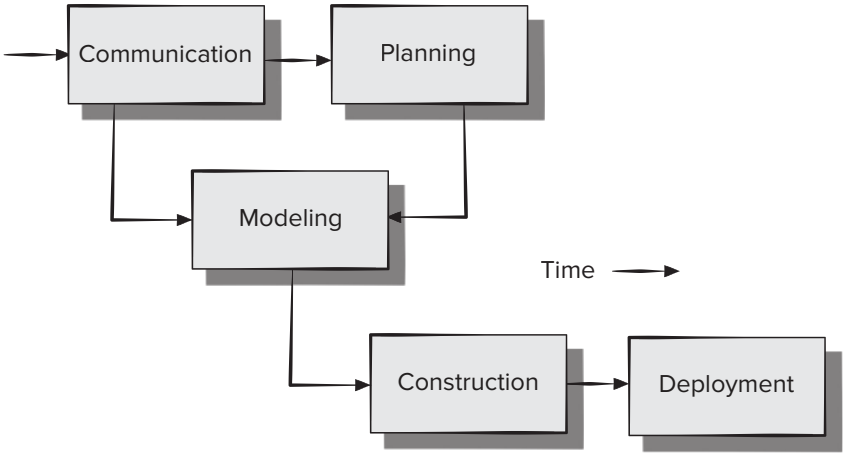
Principle 1. ***Be agile.*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.

Principle 2. ***Focus on quality at every step.*** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle 3. ***Be ready to adapt.*** Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

1 Some writers argue for one of these terms to the exclusion of the others. In reality, software engineering is all three.

FIGURE 6.1
Simplified process framework



Principle 4. ***Build an effective team.*** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.[2]

Principle 5. ***Establish mechanisms for communication and coordination.*** Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues, and they must be addressed.

Principle 6. ***Manage change.*** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

Principle 7. ***Assess risk.*** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans. Some of these contingency plans will form the basis for security engineering tasks (Chapter 18).

Principle 8. ***Create work products that provide value for others.*** Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. Be sure that the work product imparts the necessary information without ambiguity or omission.

Part Four of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

6.1.2 Principles That Guide Practice

Software engineering practice has a single overriding goal: *to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders*. To achieve this goal, you should adopt a set of core principles that guide your technical work. These principles have merit regardless of the analysis

2 The characteristics of effective software teams were discussed in Chapter 5.

and design methods that you apply, the construction techniques (e.g., programming language, automated tools) that you use, or the verification and validation approach that you choose. The following set of core principles is fundamental to the practice of software engineering:

Principle 1. ***Divide and conquer.*** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoCs). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*).

Principle 2. ***Understand the use of abstraction.*** At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. When we use the abstraction *spreadsheet*, it is assumed that you understand what a spreadsheet is, the general structure of content that a spreadsheet presents, and the typical functions that can be applied to it. In software engineering practice, you use many different levels of abstraction, each imparting or implying meaning that must be communicated. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a *column* or the *SUM* function).

Principle 3. ***Strive for consistency.*** Whether it's creating an analysis model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a mobile app. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons help to create a highly effective user experience.

Principle 4. ***Focus on the transfer of information.*** Software is about information transfer—from a database to an end user, from a legacy system to a WebApp, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another—the list is almost endless. In every case, information flows across an interface, and this means there are opportunities for errors, omissions, or ambiguity. The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.

Principle 5. ***Build software that exhibits effective modularity.*** Separation of concerns (Principle 1) establishes a philosophy for software. *Modularity* provides a mechanism for realizing the philosophy. Any complex system can be divided into modules (components), but good software engineering practice demands more. Modularity must be *effective*. That is, each module should focus exclusively on one well-constrained aspect of the system. Additionally, modules should be interconnected in a relatively simple manner to other modules, to data sources, and to other environmental aspects.

Principle 6. ***Look for patterns.*** Software engineers use patterns as a means of cataloging and reusing solutions to problems they have encountered in the past. The use of these design patterns can be applied to wider systems engineering and systems integration problems, by allowing components in complex systems to evolve independently. Patterns will be discussed further in Chapter 14.

Principle 7. ***When possible, represent the problem and its solution from several different perspectives.*** When a problem and its solution are examined from different

perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered. The unified modeling language (UML) provides a means of describing a problem solution from multiple viewpoints, as described in Appendix 1.

Principle 8. ***Remember that someone will maintain the software.*** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

These principles are not all you'll need to build high-quality software, but they do establish a foundation for every software engineering method discussed in this book.

6.2 Principles That Guide Each Framework Activity

In the sections that follow, we consider principles that have a strong bearing on the success of each generic framework activity defined as part of the software process. In many cases, the principles that are discussed for each of the framework activities are a refinement of the principles presented in Section 6.1. They are simply core principles stated at a lower level of abstraction.

6.2.1 Communication Principles

Before customer requirements can be analyzed, modeled, or specified, they must be gathered through the communication activity. A customer has a problem that may be amenable to a computer-based solution. You respond to the customer's request for help. Communication has begun. But the road from communication to understanding is often full of potholes.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that you will confront. There are many ways to communicate, but it's important to recognize that not all are equal in richness or effectiveness (Figure 6.2). In this context, we discuss communication principles as they apply to customer communication. However, many of the principles apply equally to all forms of communication that occur within a software project.

INFO



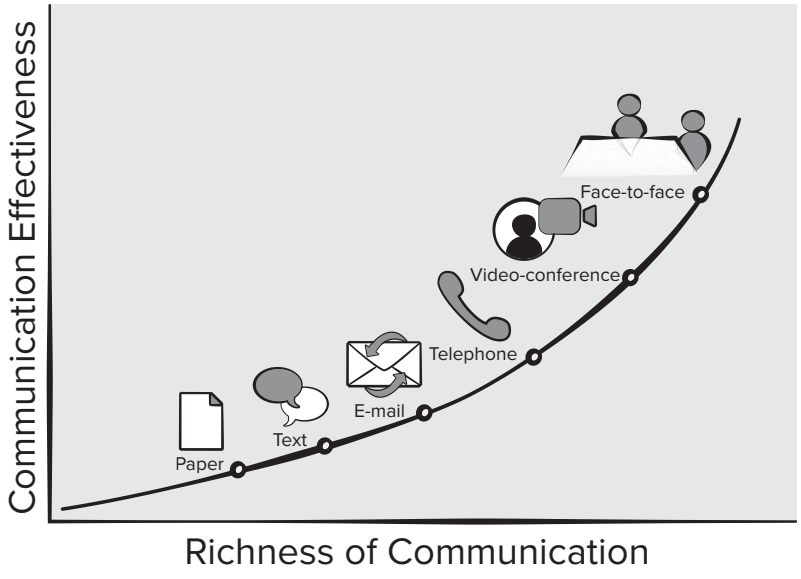
The Difference Between Customers and End Users

A *customer* is the person or group who (1) originally requested the software to be built, (2) defines overall business objectives for the software, (3) provides basic product requirements, and (4) coordinates funding for the project. In a product or system business, the customer is often the marketing group. In an information technology (IT) environment, the customer might be a business component or department.

An *end user* is the person or group who (1) will actually use the software that is built to achieve some business purpose and (2) will define operational details of the software so the business purpose can be achieved. In some cases, the customer and the end user may be one and the same, but for many projects that is not the case.

FIGURE 6.2

Effectiveness of communication modes



Principle 1. ***Listen.*** Before communicating, be sure you understand the point of view of the other party, know a bit about his or her needs, and then listen. Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, and avoid constant interruptions. *Never* become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2. ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3. ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure that other principles are followed.

Principle 4. ***Face-to-face communication is best.*** However, this communication usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

Principle 5. ***Take notes and document decisions.*** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle 6. ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle 7. ***Stay focused; modularize your discussion.*** The more people are involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved (however, see Principle 9).

Principle 8. ***If something is unclear, draw a picture.*** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

Principle 9. ***(a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified right now, move on.*** Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle 10. ***Negotiation is not a contest or a game. It works best when both parties win.*** There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

SAFEHOME



Communication Mistakes

The scene: Software engineering team workspace

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member.

The conversation:

Ed: What have you heard about this *SafeHome* project?

Vinod: The kickoff meeting is scheduled for next week.

Jamie: I've already done a little bit of investigation, but it didn't go well.

Ed: What do you mean?

Jamie: Well, I gave Lisa Perez a call. She's the marketing honcho on this thing.

Vinod: And . . . ?

Jamie: I wanted her to tell me about *SafeHome* features and functions . . . that sort of thing. Instead, she began asking me questions about security systems, surveillance systems . . . I'm no expert.

Vinod: What does that tell you?

(Jamie shrugs)

Vinod: That marketing will need us to act as consultants and that we'd better do some homework on this product area before our kickoff meeting. Doug said that he wanted us to "collaborate" with our customer, so we'd better learn how to do that.

Ed: Probably would have been better to stop by her office. Phone calls just don't work as well for this sort of thing.

Jamie: You're both right. We've got to get our act together or our early communications will be a struggle.

Vinod: I saw Doug reading a book on "requirements engineering." I'll bet that lists some principles of good communication. I'm going to borrow it from him.

Jamie: Good idea . . . then you can teach us.

Vinod (smiling): Yeah, right.

6.2.2 Planning Principles

The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.

Try as we might, it's impossible to predict exactly how a software project will evolve. There is no easy way to determine what unforeseen technical problems will be encountered, what important information will remain undiscovered until late in the project, what misunderstandings will occur, or what business issues will change. And yet, a good software team must plan its approach. Often planning is iterative (Figure 6.3).

There are different planning philosophies.[3] Some people are "minimalists," arguing that change often obviates the need for a detailed plan. Others are "traditionalists," arguing that the plan provides an effective road map and the more detail it has, the less likely the team will become lost.

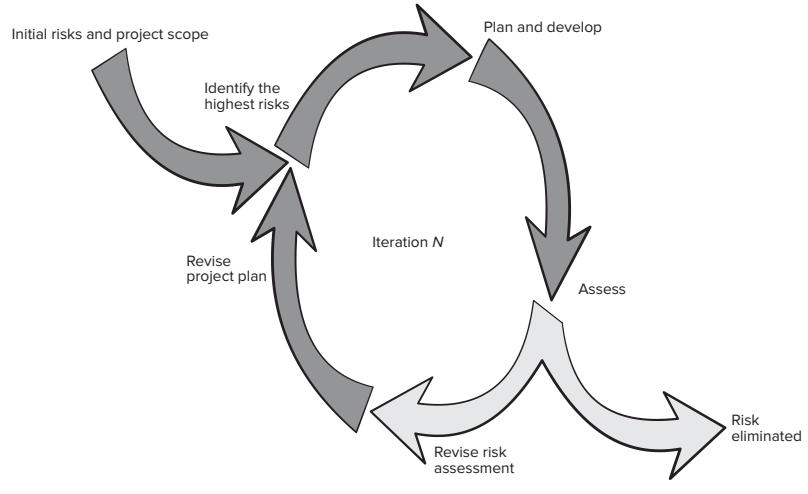
What to do? On many projects, overplanning is time consuming and fruitless (too many things change), but underplanning is a recipe for chaos. Like most things in life, planning should be agile and conducted in moderation, enough to provide useful guidance for the team—no more, no less. Regardless of the rigor with which planning is conducted, the following principles always apply:

Principle 1. ***Understand the scope of the project.*** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

Principle 2. ***Involve stakeholders in the planning activity.*** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

FIGURE 6.3

Iterative planning



3 A detailed discussion of software project planning and management is presented in Part Four of this book.

Principle 3. ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it is very likely that things will change. The plan will need to be adjusted. Iterative, incremental process models include time for revising plans after the delivery of each software increment based on feedback received from users.

Principle 4. ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle 5. ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary. The project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

Principle 6. ***Be realistic.*** People don't work 100 percent of every day. Changes will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

Principle 7. ***Adjust granularity as you define the plan.*** The term *granularity* refers to the detail with which some element of planning is represented or conducted. A *high-granularity* plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A *low-granularity* plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Activities that won't occur for many months do not require high granularity (too much can change).

Principle 8. ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality. If technical reviews[4] are to be conducted, they should be scheduled. If pair programming (Chapter 3) is to be used during construction, it should be explicitly defined within the plan.

Principle 9. ***Describe how you intend to accommodate change.*** Uncontrolled change can obviate even the best planning. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle 10. ***Track the plan frequently, and make adjustments as required.*** Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress daily, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

To be most effective, everyone on the software team should participate in the planning activity. Only then will team members "sign up" to the plan.

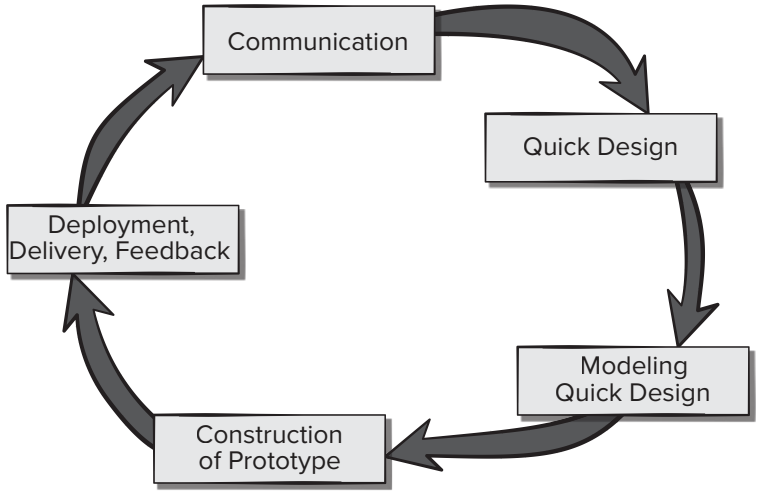
6.2.3 Modeling Principles

We create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a

4 Technical reviews are discussed in Chapter 16.

FIGURE 6.4

Role of software modeling



three-dimensional (3D) model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that users desire, and the behavior of the system as the transformation is taking place. Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level. Figure 6.4 shows how modeling may be used in agile software design.

In software engineering work, two classes of models can be created: requirements models and design models. *Requirements models* (also called *analysis models*) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain (Chapter 8). *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail (Chapters 9 through 12).

In their book on agile modeling, Scott Ambler and Ron Jeffries [Amb02] define a set of modeling principles[5] that are intended for those who use an agile process model (Chapter 3) but are appropriate for all software engineers who perform modeling action and tasks:

Principle 1. ***The primary goal of the software team is to build software, not create models.*** Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

5 The principles noted in this section have been abbreviated and rephrased for the purposes of this book.

Principle 2. ***Travel light—don't create more models than you need.*** Every model that is created must be kept up to date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

Principle 3. ***Strive to produce the simplest model that will describe the problem or the software.*** Don't overbuild the software [Amb02]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. ***Build models in a way that makes them amenable to change.*** Assume that your models will change, but in making this assumption don't get sloppy. The problem with this attitude is that you may not create a reasonably complete requirements model, which means you'll create a design (design model) that will invariably miss important functions and features.

Principle 5. ***Be able to state an explicit purpose for each model that is created.*** Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

Principle 6. ***Adapt the models you develop to the system at hand.*** It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software for adaptive cruise control in an automobile.

Principle 7. ***Try to build useful models, but forget about building perfect models.*** When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make a model that is complete and internally consistent may not be worth the benefits of these properties. Iterating endlessly to make a model "perfect" does not serve the need for agility.

Principle 8. ***Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.*** Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

Principle 9. ***If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.*** If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

Principle 10. ***Get feedback as soon as you can.*** The intent of any model is to communicate information. It should stand on its own. Assume that you won't be there to explain the model. Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

6.2.4 Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user.

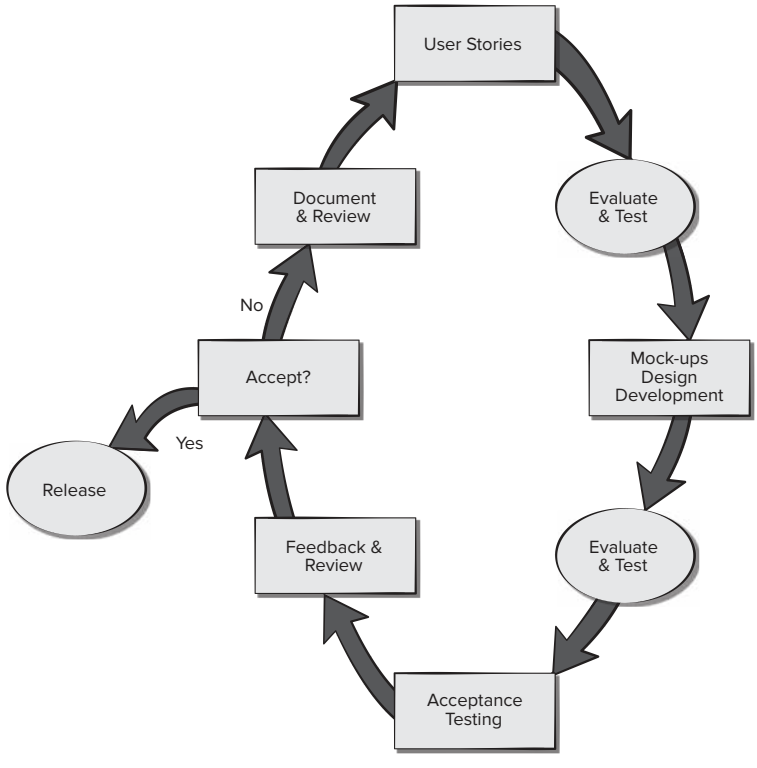
In modern software engineering work, coding may be: (1) the direct creation of programming language source code, (2) the automatic generation of source code using an intermediate design like representation of the component to be built, or (3) the automatic generation of executable code using a fourth-generation programming language (e.g., Unreal4 *Blueprints*).[6]

The initial focus of testing is at the component level, often called *unit testing*. Other levels of testing include (1) *integration testing* (conducted as the system is constructed), (2) *validation testing* that assesses whether requirements have been met for the complete system (or software increment), and (3) *acceptance testing* that is conducted by the customer in an effort to exercise all required features and functions. Figure 6.5 shows where testing and test case design is placed in agile processes.

Testing is considered in detail in Chapters 19 through 21. The following set of fundamental principles and concepts are applicable to coding and testing.

FIGURE 6.5

Testing in agile processes



6 *Blueprints* is a visual scripting tool created by Epic Games (https://docs.unrealengine.com/latest/INT/Engine/Blueprints/).

Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. There are some fundamental principles that might be followed:

Preparation Principles: Before you write one line of code, be sure you:

Principle 1. ***Understand the problem you're trying to solve.***

Principle 2. ***Understand basic design principles and concepts.***

Principle 3. ***Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.***

Principle 4. ***Select a programming environment that provides tools that will make your work easier.***

Principle 5. ***Create a set of unit tests that will be applied once the component you code is completed.***

Coding Principles: As you begin writing code, be sure you:

Principle 6. ***Constrain your algorithms by following structured programming [Boh00] practice.***

Principle 7. ***Consider the use of pair programming.***

Principle 8. ***Select data structures that will meet the needs of the design.***

Principle 9. ***Understand the software architecture and create interfaces that are consistent with it.***

Validation Principles: After you've completed your first coding pass, be sure you:

Principle 10. ***Conduct a code walkthrough when appropriate.***

Principle 11. ***Perform unit tests and correct errors you've uncovered.***

Principle 12. ***Refactor the code to improve its quality.***

Testing Principles. In a classic book on software testing, Glen Myers [Mye79] states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint for some software developers. They move counter to the commonly held view that a successful test is one in which no errors are found. Your objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

As a secondary benefit, testing demonstrates that software functions appear to be working according to specification and that behavioral and performance requirements appear to have been met. In addition, the data collected as testing is conducted provide a good indication of software reliability and some indication of software quality.

FIGURE 6.6

Testing is never complete



Testing cannot show the absence of errors and defects; it can show only that software errors and defects are present (Figure 6.6). It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

Davis [Dav95b] suggests a set of testing principles[7] that have been adapted for use in this book. In addition, Everett and Meyer [Eve09] suggest additional principles:

Principle 1. ***All tests should be traceable to customer requirements.***[8] The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. ***Tests should be planned long before testing begins.*** Test planning (Chapter 19) can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

Principle 3. ***The Pareto principle applies to software testing.*** In this context, the Pareto principle implies that 80 percent of all errors uncovered during testing will

7 Only a small subset of Davis's testing principles are noted here. For more information, see [Dav95b].

8 This principle refers to *functional tests*, that is, tests that focus on requirements. *Structural tests* (tests that focus on architectural or logical detail) may not address specific requirements directly.

likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. ***Testing should begin "in the small" and progress toward testing "in the large."*** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts to looking for errors in integrated clusters of components and ultimately in the entire system.

Principle 5. ***Exhaustive testing is not possible.*** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Principle 6. ***Apply to each module in the system a testing effort commensurate with its expected fault density.*** These are often the newest modules or the ones that are least understood by the developers.

Principle 7. ***Static testing techniques can yield high results.*** More than 85 percent of software defects originate in the software documentation (requirements, specifications, code walk-throughs, and user manuals) [Jon91]. There may be value in testing the system documentation.

Principle 8. ***Track defects and look for patterns in defects uncovered by testing.*** The total defects uncovered is a good indicator of software quality. The types of defects uncovered can be a good measure of software stability. Patterns of defects found over time can forecast numbers of expected defects.

Principle 9. ***Include test cases that demonstrate software is behaving correctly.*** As software components are being maintained or adapted, unexpected interactions cause unintended side effects in other components. It is important to have a set of regression test cases (Chapter 19) ready to check system behavior after changes are made to a software product.

6.2.5 Deployment Principles

As we noted in Part One of this book, the deployment activity encompasses three actions: delivery, support, and feedback. Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but several times as software moves toward completion. Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features. Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date. Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment. Typical deployment actions are illustrated in Figure 6.7.

The delivery of a software increment represents an important milestone for any software project. Some key principles should be followed as the team prepares to deliver an increment:

Principle 1. ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment

FIGURE 6.7

Deployment actions



occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [Kar94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time period provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2. ***A complete delivery package should be assembled and tested.*** All executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in all possible computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements).

Principle 3. ***A support regimen must be established before the software is delivered.*** An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate record-keeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle 4. ***Appropriate instructional materials must be provided to end users.*** The software team delivers more than the software itself. Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided,

and when necessary, a "what's different about this software increment" description should be published.[9]

Principle 5. ***Buggy software should be fixed first, delivered later.*** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

6.3 SUMMARY

Software engineering practice encompasses principles, concepts, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different. Yet, a set of generic principles applies to the process and to the practice of each framework activity regardless of the project or the product.

A set of core principles helps in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it navigates through the software process. At the practice level, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles provide guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modeling principles serve as a foundation for the methods and notation that are used to create representations of the software. Modeling encompasses both analysis and design, describing representations of the software that progressively become more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested. Coding principles define generic actions that should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: Testing is a process of executing a program with the intent of finding an error.

Deployment occurs as each software increment is presented to the customer and encompasses delivery, support, and feedback. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support

9 During the communication activity, the software team should determine what types of help materials users want.

information for the software. Support demands advance preparation. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.

PROBLEMS AND POINTS TO PONDER

6.1. Because a focus on quality demands resources and time, is it possible to be agile and still maintain a quality focus?

6.2. Of the eight core principles that guide process (discussed in Section 6.1.1), which do you believe is most important?

6.3. Describe the concept of *separation of concerns* in your own words.

6.4. Why is it necessary to "move on"?

6.5. Do some research on "negotiation" for the communication activity, and prepare a set of guidelines that focus solely on negotiation.

6.6. Why are models important in software engineering work? Are they always necessary? Are there qualifiers to your answer about necessity?

6.7. What is a successful test?

6.8. Why is feedback important to the software team?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

7 UNDERSTANDING REQUIREMENTS

Understanding problem requirements is among the most difficult tasks facing a software engineer. When you first think about it, developing a clear understanding of the requirements doesn't seem that hard. After all, doesn't the customer know what is required? Shouldn't the end users have a good understanding of the features and functions that they need? Surprisingly, in many instances, the answer to these questions is no. And even if customers and end users can state their needs explicitly, those needs will change throughout the project.

KEY CONCEPTS

analysis model 118
analysis patterns 122
collaboration 108
elaboration 104
elicitation 104
inception 104
negotiation 105
requirements engineering 103
requirements gathering 110
requirements management 106
requirements monitoring 123
specification 105
stakeholders 107
use cases 113
validating requirements 123
validation 105
viewpoints 107
work products 114

QUICK LOOK

What is it? Before you begin any technical work, it's a good idea to create a set of requirements for the engineering tasks. By establishing a set of requirements, you'll gain an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software.

Who does it? Software engineers and other project stakeholders (managers, customers, and end users) all participate in requirements engineering.

Why is it important? To understand what the customer wants before you begin to design and build a computer-based system. Building an elegant computer program that solves the wrong problem helps no one.

What are the steps? Requirements engineering begins with inception (a task that defines the scope and nature of the problem to be solved). It moves onward to elicitation (a task that helps stakeholders define what is required), and then elaboration (where basic requirements are refined and modified). As stakeholders define the problem, negotiation occurs (what are the priorities, what is essential, when is it required?). Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

What is the work product? Requirements engineering provides all parties with a written understanding of the problem. The work products may include: usage scenarios, function and feature lists, and requirements models.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that everyone is on the same page. A word of warning: Even after all parties agree, things will change, and they will continue to change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, one of us [RSP] wrote:

> It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.
>
> All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a disorganized manner, and we spend far too little time verifying what we do record. We allow change to control us, rather than establishing mechanisms to control change. In short, we fail to establish a solid foundation for the system or software. Each of these problems is challenging. When they are combined, the outlook is daunting for even the most experienced managers and practitioners. But solutions do exist.

It's reasonable to argue that the techniques we'll discuss in this chapter are not a true "solution" to the challenges just noted. But they do provide a solid approach for addressing these challenges.

7.1 REQUIREMENTS ENGINEERING

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program, and that all else is secondary. What makes these arguments seductive is that they contain elements of truth. But each argument is flawed and can lead to a failed software project.

Requirements engineering is the term for the broad spectrum of tasks and techniques that lead to an understanding of requirements. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs. It must be adapted to the needs of the process, the project, the product, and the people doing the work. It is important to realize that each of these tasks is done iteratively as the project team and the stakeholders continue to share information about their respective concerns.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins with the project stakeholders (e.g., managers, customers, and end users), where business needs are defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it begins with a broader system definition,

where software is but one component of the larger system domain. But regardless of the starting point, the journey across the bridge takes you high above the project, allowing you to examine the context of the software work to be performed; the specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resulting design. Requirements engineering encompasses seven tasks with sometimes muddy boundaries: *inception, elicitation, elaboration, negotiation, specification, validation,* and *management.* It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project. Expect to do a bit of design during requirements work and a bit of requirements work during design.

7.1.1 Inception

How does a software project get started? In general, most projects begin with an identified business need or when a potential new market or service is discovered. At project inception, you establish a basic understanding of the problem, the people who want a solution, and the nature of the solution that is desired. Communication between all stakeholders and the software team needs to be established during this task to begin an effective collaboration.

7.1.2 Elicitation

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

An important part of elicitation is to understand the business goals [Cle10]. A *goal* is a long-term aim that a system or product must achieve. Goals may deal with either functional or nonfunctional (e.g., reliability, security, usability) concerns [Lam09].

Goals are often a good way to explain requirements to stakeholders and, once established, can be used to manage conflicts among stakeholders. Goals should be specified precisely and serve as the basis for requirements elaboration, verification and validation, conflict management, negotiation, explanation, and evolution.

Your job is to engage stakeholders and to encourage them to share their goals honestly. Once the goals are captured, you establish a prioritization mechanism and create a design rationale for a potential architecture (that meets stakeholder goals).

Agility is an important aspect of requirements engineering. The intent of elicitation is to transfer ideas from stakeholders to the software team smoothly and without delay. It is highly likely that new requirements will continue to emerge as iterative product development occurs.

7.1.3 Elaboration

The elaboration task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information (Chapter 8). Elaboration is driven by the creation and refinement of user scenarios obtained during elicitation. These scenarios describe how the end users (and other actors) will interact with the system. Each user scenario is parsed to extract *analysis classes*—business domain

entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified. The relationships and collaboration between classes are identified. Elaboration is a good thing, but you need to know when to stop. The key is to describe the problem in a way that establishes a firm base for design and then move on. Do not obsess over unnecessary details.

7.1.4 Negotiation

It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

These conflicts need to be reconciled through the process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. There should be no winner and no loser in an effective negotiation. Both sides win, because a "deal" that both can live with is solidified. You should use an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts. In this way, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

7.1.5 Specification

In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. The formality and format of a specification varies with the size and the complexity of the software to be built. For large systems, a written document, combining natural language descriptions and graphical models, may be the best approach. A template for a formal software requirements specification document can be downloaded from: https://web.cs.dal.ca/~hawkey/3130/srs_template-ieee.doc. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

7.1.6 Validation

The work products produced during requirements engineering are assessed for quality during a validation step. A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the *technical review* (Chapter 16). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in

content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

To illustrate some of the problems that occur during requirements validation, consider two seemingly innocuous requirements:

- The software should be user friendly.
- The probability of a successful unauthorized database intrusion should be less than 0.0001.

The first requirement is too vague for developers to test or assess. What exactly does "user friendly" mean? To validate it, it must be quantified or qualified in some manner.

The second requirement has a quantitative element ("less than 0.0001"), but intrusion testing will be difficult and time consuming. Is this level of security even warranted for the application? Can other complementary requirements associated with security (e.g., password protection, specialized handshaking) replace the quantitative requirement noted?

7.1.7 Requirements Management

Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 22.

INFO



Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

1. Are requirements stated clearly? Can they be misinterpreted?
2. Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
3. Is the requirement bounded in quantitative terms?
4. What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
5. Does the requirement violate any system domain constraints?
6. Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
7. Is the requirement traceable to any system model that has been created?
8. Is the requirement traceable to overall system and product objectives?
9. Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
10. Has an index for the specification been created?
11. Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

7.2 Establishing the Groundwork

In an ideal setting, stakeholders and software engineers work together on the same team. In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may reside in different cities or countries, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer. None of these things are desirable, but all are common, and you are often forced to work within the constraints imposed by this situation.

In the sections that follow, we discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

7.2.1 Identifying Stakeholders

Sommerville and Sawyer [Som97] define a *stakeholder* as "anyone who benefits in a direct or indirect way from the system which is being developed." We have already identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 7.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

7.2.2 Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

Several things can make it hard to elicit requirements for software that satisfies its users: project goals are unclear, stakeholders' priorities differ, people have unspoken assumptions, stakeholders interpret meanings differently, and requirements are stated in a way that makes them difficult to verify [Ale11]. The goal of effective requirements engineering is to eliminate or at least reduce these problems.

7.2.3 Working Toward Collaboration

If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, we have noted that customers (and other stakeholders) should collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

Collaboration does not necessarily mean that requirements are "defined by committee." In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

INFO



Using "Planning Poker"

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points*. All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of each (from his viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

7.2.4 Asking the First Questions

Questions asked at the inception of the project should be "context free" [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders and the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?

- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful elicitation. But a question-and-answer (Q&A) meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 7.3.

7.2.5 Nonfunctional Requirements

A *nonfunctional requirement* (NFR) can be described as a quality attribute, a performance attribute, a security attribute, or a general constraint on a system. These are often not easy for stakeholders to articulate. Chung [Chu09] suggests that there is a lopsided emphasis on functionality of the software, yet the software may not be useful or usable without the necessary nonfunctional characteristics.

It is possible to define a two-phase approach [Hne11] that can assist a software team and other stakeholders in identifying nonfunctional requirements. During the first phase, a set of software engineering guidelines is established for the system to be built. These include guidelines for best practice, but also address architectural style (Chapter 10) and the use of design patterns (Chapter 14). A list of NFRs (e.g., requirements that address usability, testability, security, or maintainability) is then developed. A simple table lists NFRs as *column labels* and software engineering guidelines as *row labels*. A relationship matrix compares each guideline to all others, helping the team to assess whether each pair of guidelines is *complementary*, *overlapping*, *conflicting*, or *independent*.

In the second phase, the team prioritizes each nonfunctional requirement by creating a homogeneous set of nonfunctional requirements using a set of decision rules that establish which guidelines to implement and which to reject.

7.2.6 Traceability

Traceability is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases). A *traceability matrix* allows a requirements engineer to represent the relationship between requirements and other software engineering work products. Rows of the traceability

matrix are labeled using requirement names, and columns can be labeled with the name of a software engineering work product (e.g., a design element or a test case). A matrix cell is marked to indicate the presence of a link between the two.

The traceability matrices can support a variety of engineering development activities. They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used. Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

As the number of requirements and the number of work products grows, it becomes increasingly difficult to keep the traceability matrix up to date. Nonetheless, it is important to create some means for tracking the impact and evolution of the product requirements [Got11].

7.3 REQUIREMENTS GATHERING

Requirements gathering combines elements of problem solving, elaboration, negotiation, and specification. To encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements [Zah90].

7.3.1 Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- Meetings (either real or virtual) are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be worksheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

A one- or two-page "product request" is generated during inception (Section 7.2). A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross section of users. If only one user defines all requirements, acceptance risk is high (meaning there may be several other stakeholders who will not accept the product). The product request is distributed to all attendees before the meeting date.

As an example, consider an excerpt from a product request written by a marketing person involved in the *SafeHome* project. This person writes the following narrative about the home security function that is to be part of *SafeHome:*

> Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with "alarm systems," so this would be an easy sell. We might also consider using voice control of the system using some technology like Alexa.
>
> The home security function would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, carbon monoxide levels, and others. It'll use our wireless sensors to detect each situation, can be programmed by the homeowner, and will automatically contact a monitoring agency and the owner's cell phone when a situation is detected.

In reality, others would contribute to this narrative during the requirements gathering meeting and considerably more information would be available. But even with additional information, ambiguity is present, omissions are likely to exist, and errors might occur. For now, the preceding "functional description" will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy, security) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a tablet, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone using a wireless router, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within 1 second, and an event priority scheme should be implemented).

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on a group forum or at an internal website or posed in a social networking environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted, and additions can be made. At this stage, critique and debate are strictly prohibited. Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the

discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product or system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for entries on the lists or by creating a use case (Section 7.4) that involves the object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be:

> The control panel is a wall-mounted unit that is approximately 230 × 130 mm in size. The control panel has wireless connectivity to sensors and a tablet. User interaction occurs through a keypad containing 12 keys. A 75 × 75 mm OLED color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.

SAFEHOME



Case Study Example
Conducting a Requirements Gathering Meeting

The scene: A meeting room. The first requirements gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

Many stakeholder concerns (e.g., accuracy, data accessibility, security) are the basis for nonfunctional system requirements (Section 7.2). As stakeholders enunciate these concerns, software engineers must consider them within the context of the system to be built. The questions that must be answered [Lag10] are:

- Can we build the system?
- Will this development process allow us to beat our competitors to market?
- Do adequate resources exist to build and maintain the proposed system?
- Will the system performance meet the needs of our customers?

7.3.2 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until you understand how the features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* [Jac92], provide a description of how the system will be used. Use cases are discussed in greater detail in Section 7.4.

SAFEHOME



Developing a Preliminary User Scenario

The scene: A meeting room, continuing the first requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user ID and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

7.3.3 Elicitation Work Products

The work products produced during requirements elicitation will vary depending on the size of the system or product to be built. For large systems, the work products may include: (1) a statement of need and feasibility; (2) a bounded statement of scope for the system or product; (3) a list of customers, users, and other stakeholders who participated in requirements elicitation; (4) a description of the system's technical environment; (5) a list of requirements (preferably organized by function) and the domain constraints that apply to each; and (6) a set of usage scenarios that provide insight into the use of the system or product under different operating conditions. Each of these work products is reviewed by all people who have participated in requirements elicitation.

7.4 Developing Use Cases

A use case tells a stylized story about how an end user (playing one of several possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text (a *user story*), an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user's point of view.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors will represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play several different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a user who interacts with the program that allows experimenting with alarm sensor configuration

in a virtual building. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: placement mode, testing mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: editor, tester, monitor, and troubleshooter. In some cases, the user can play all the roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests questions that should be answered by a use case:

1. Who is the primary actor, the secondary actor(s)?
2. What are the actor's goals?
3. What preconditions should exist before the story begins?
4. What main tasks or functions are performed by the actor?
5. What exceptions might be considered as the story is described?
6. What variations in the actor's interaction are possible?
7. What system information will the actor acquire, produce, or change?
8. Will the actor have to inform the system about changes in the external environment?
9. What information does the actor desire from the system?
10. Does the actor wish to be informed about unexpected changes?

Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in different ways using either the alarm control panel, a tablet, or a cell phone.

The homeowner:

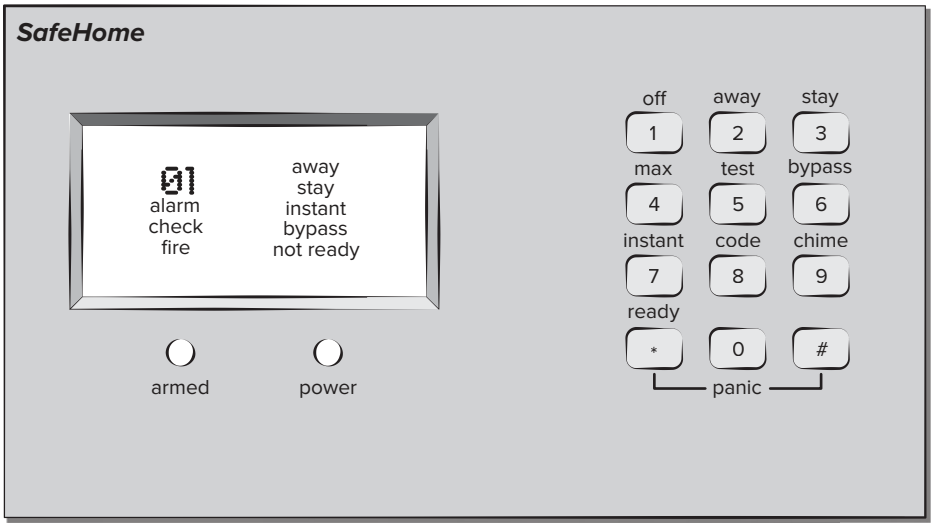
1. Enters a password to allow all other interactions
2. Inquires about the status of a security zone
3. Inquires about the status of a sensor
4. Presses the panic button in an emergency
5. Activates and deactivates the security system

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:

1. The homeowner observes the *SafeHome* control panel (Figure 7.1) to determine if the system is ready for input. If the system is not ready, a *not ready*

FIGURE 7.1

***SafeHome* control panel**



message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. (A *not ready* message implies that a sensor is open, i.e., that a door or window is open.)

2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in "stay" or "away" (see Figure 7.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level user story that describes the interaction between the actor and the system.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.
Trigger:	The homeowner decides to "set" the system, that is, to turn on the alarm functions.

Scenario:

1. Homeowner observes control panel.
2. Homeowner enters password.
3. Homeowner selects "stay" or "away."
4. Homeowner observes red alarm light to indicate that *SafeHome* has been armed.

Exceptions:

1. Control panel is *not ready:* Homeowner checks all sensors to determine which are open and then closes them.
2. Password is incorrect (control panel beeps once): Homeowner reenters correct password.
3. Password not recognized: Monitoring and response subsystem must be contacted to reprogram password.
4. *Stay* is selected: Control panel beeps twice, and a *stay* light is lit; perimeter sensors are activated.
5. *Away* is selected: Control panel beeps three times, and an *away* light is lit; all sensors are activated.

Priority:	Essential, must be implemented
When available:	First increment
Frequency of use:	Many times per day
Channel to actor:	Via control panel interface
Secondary actors:	Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

Use cases for other **homeowner** interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem. Use cases are often written informally as user stories. However, using the template shown here helps to ensure that you've addressed all key issues. This is very important for systems where user safety or security is a stakeholder concern.

SafeHome



Developing a High-Level Use Case Diagram

The scene: A meeting room, continuing the requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 7.2.)

Jamie: I'm just beginning to learn UML notation. So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

Facilitator: Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

Doug: Is that legal in UML?

Facilitator: Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

Vinod: Okay, so we have use case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

Facilitator: Probably, but that can wait until we've considered other *SafeHome* functions.

Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

Facilitator: Oh really. Tell me what we've missed.

(The meeting continues.)

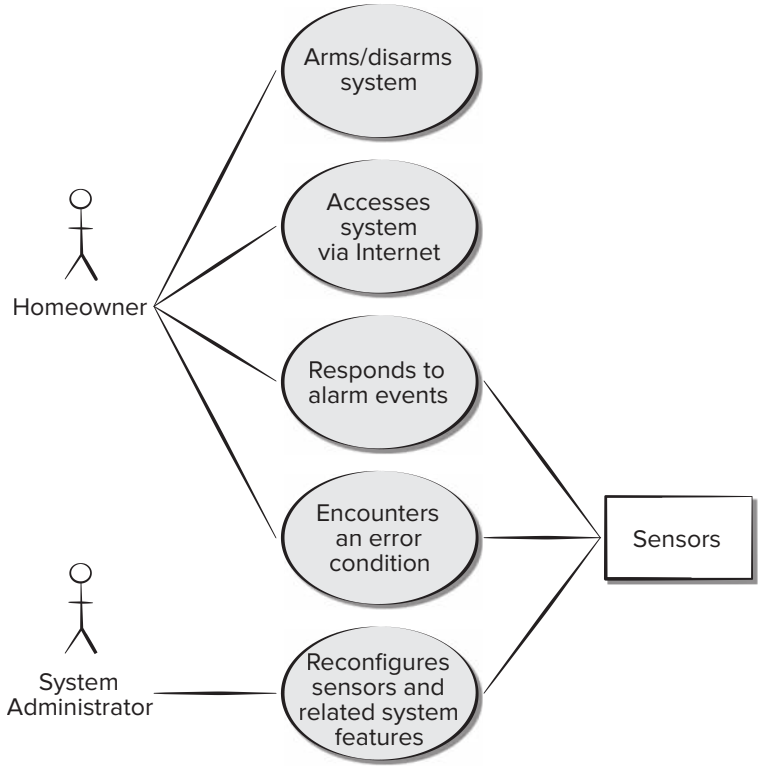
7.5 Building the Analysis Model

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and as stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

As the analysis model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system. If your team finds that it does not use certain elements of the analysis model as the project moves to design and construction, those elements should not be created in the future and should not be maintained as the requirements change in the current project. The analysis model and the methods that are used to build it are presented in detail in Chapter 8. We present a brief overview in the sections that follow.

FIGURE 7.2

UML use case diagram for *SafeHome* home security function



7.5.1 Elements of the Analysis Model

There are many ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use several different modes of representation to depict the analysis model. Using different modes of representation forces you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity. It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used. A set of generic elements common to most analysis models is introduced in this chapter.

Scenario-Based Elements. Scenario-based elements of the requirements model are often the first part of the model that is developed. They describe the system from the user's point of view. For example, basic user stories (Section 7.4) and their corresponding use case diagrams (Figure 7.2) may evolve into more elaborate template-based use cases (Section 7.4). As such, they serve as input for the creation of other modeling elements. It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used.

Class-Based Elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 7.3).

Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify, enable*) that can be applied to modify these attributes. Other analysis modeling elements depict how classes collaborate with one another and the relationships and interactions among classes. One way to isolate classes is to look for descriptive nouns in a use case script. At least some of the nouns will be candidate classes. The verbs found in the use case script may be considered candidate methods for these classes. These and other techniques are discussed in more detail in Chapter 8.

Behavioral Elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activations) are taken when events occur. External stimuli (events) cause transitions between states.

FIGURE 7.3
Class diagram for sensor

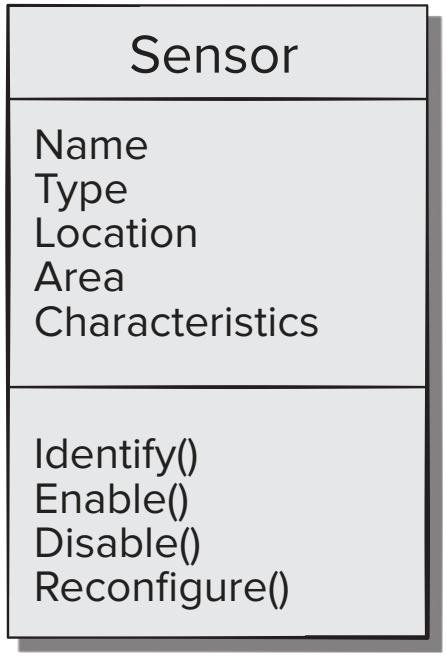
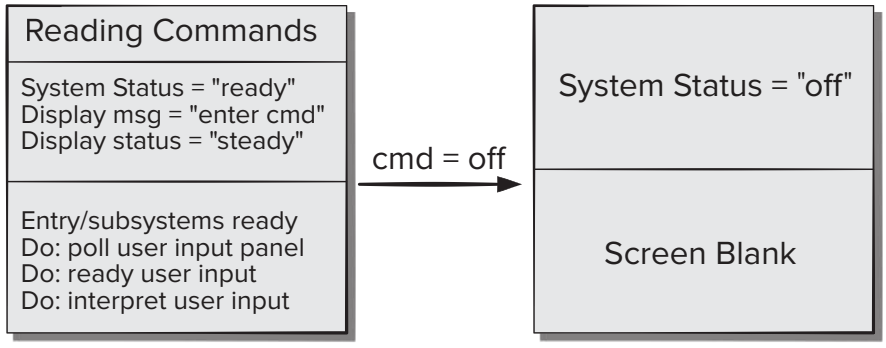


FIGURE 7.4

UML state diagram notation



To illustrate the use of a state diagram, consider software embedded within the *SafeHome* control panel that is responsible for reading user input. An example of UML state diagram notation is shown in Figure 7.4. Further discussion of behavioral modeling is presented in Chapter 8.

SAFEHOME



Preliminary Behavioral Modeling

The scene: A meeting room, continuing the requirements meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've just about finished talking about *SafeHome* home security functionality. But before we do, I want to discuss the behavior of the function.

Marketing person: I don't understand what you mean by behavior.

Ed (smiling): That's when you give the product a "timeout" if it misbehaves.

Facilitator: Not exactly. Let me explain.

(The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

Marketing person: This seems a little technical. I'm not sure I can help here.

Facilitator: Sure you can. What behavior do you observe from the user's point of view?

Marketing person: Uh . . . well, the system will be *monitoring* the sensors. It'll be *reading commands* from the homeowner. It'll be *displaying* its status.

Facilitator: See, you can do it.

Jamie: It'll also be *polling* the PC to determine if there is any input from it, for example, Internet-based access or configuration information.

Vinod: Yeah, in fact, *configuring the system* is a state in its own right.

Doug: You guys are rolling. Let's give this a bit more thought . . . is there a way to diagram this stuff?

Facilitator: There is, but let's postpone that until after the meeting.

7.5.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects notices that certain problems reoccur across all projects within a specific application domain. These *analysis patterns* [Fow97] suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and reuse them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01] that is discussed in more detail in Chapter 14. Examples of analysis patterns and further discussion of this topic are presented in Chapter 8.

7.6 NEGOTIATING REQUIREMENTS

In an ideal world, the requirements engineering tasks (inception, elicitation, and elaboration) determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. You may have to enter into *negotiations* with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time to market. The intent of these negotiations is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a "win-win" result. That is, stakeholders win by getting the system or product that satisfies most their needs, and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Fricker [Fri10] and his colleagues suggest replacing the traditional handoff of requirements specifications to software teams with a bidirectional communication process called handshaking. Handshaking might be one way to accomplish a win-win result. In *handshaking*, the software team proposes solutions to requirements, describes their impact, and communicates their intentions to customer representatives. The customer representatives review the proposed solutions, focusing on missing features and seeking clarification of novel requirements. Requirements are determined to be *good enough* if the customers accept the proposed solution. Handshaking tends to improve identification, analysis, and selection of variants and promotes win-win negotiation.

SAFEHOME



The Start of a Negotiation

The scene: Lisa Perez's office, after the first requirements gathering meeting.

The players: Doug Miller, software engineering manager, and Lisa Perez, marketing manager.

The conversation:

Lisa: So, I hear the first meeting went really well.

Doug: Actually, it did. You sent some good people to the meeting . . . they really contributed.

Lisa (smiling): Yeah, they actually told me they got into it, and it wasn't a "propeller head activity."

Doug (laughing): I'll be sure to take off my techie beanie the next time I visit . . . Look, Lisa, I think we may have a problem with getting all of the functionality for the home security system out by the dates your management is talking about. It's early, I know, but I've already been doing a little back-of-the-envelope planning and . . .

Lisa (frowning): We've got to have it by that date, Doug. What functionality are you talking about?

Doug: I figure we can get full home security functionality out by the drop-dead date, but we'll have to delay Internet access 'til the second release.

Lisa: Doug, it's the Internet access that gives *SafeHome* "gee whiz" appeal. We're going to build our entire marketing campaign around it. We've gotta have it!

Doug: I understand your situation, I really do. The problem is that in order to give you Internet access, we'll have to have a fully secure website up and running. That takes time and people. We'll also have to build a lot of additional functionality into the first release . . . I don't think we can do it with the resources we've got.

Lisa (still frowning): I see, but you've got to figure out a way to get it done. It's pivotal to home security functions and to other functions as well . . . those can wait until the next releases . . . I'll agree to that.

Lisa and Doug appear to be at an impasse, and yet they must negotiate a solution to this problem. Can they both "win" here? Playing the role of a mediator, what would you suggest?

7.7 REQUIREMENTS MONITORING

Incremental development is commonplace. This means that use cases evolve, new test cases are developed for each new software increment, and continuous integration of source code occurs throughout a project. *Requirements monitoring* can be extremely useful when incremental development is used. It encompasses five tasks: (1) *distributed debugging* uncovers errors and determines their cause, (2) *run-time verification* determines whether software matches its specification, (3) *run-time validation* assesses whether the evolving software meets user goals, (4) *business activity monitoring* evaluates whether a system satisfies business goals, and (5) *evolution and codesign* provides information to stakeholders as the system evolves.

Incremental development implies the need for incremental validation. Requirements monitoring supports continuous validation by analyzing user goal models against the system in use. For example, a monitoring system might continuously assess user satisfaction and use feedback to guide incremental improvements [Rob10].

7.8 VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. This is true even for agile process models where requirements tend to be written as user stories and/or test cases. The requirements represented

by the model are prioritized by stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

1. Is each requirement consistent with the overall objectives for the system or product?
2. Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
3. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
4. Is each requirement bounded and unambiguous?
5. Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
6. Do any requirements conflict with other requirements?
7. Is each requirement achievable in the technical environment that will house the system or product?
8. Is each requirement testable, once implemented?
9. Does the requirements model properly reflect the information, function, and behavior of the system to be built?
10. Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
11. Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

7.9 SUMMARY

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modeling activities that have been defined for the generic software process. Seven requirements engineering activities—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team and product stakeholders.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings and the development of usage scenarios (user stories).

Elaboration further expands requirements in a model—a collection of scenario-based, activity-based, class-based, and behavioral elements. The model may reference

analysis patterns, characteristics of the problem domain that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. Each requirement needs to be validated against customer needs to ensure that the right system is to be built.

PROBLEMS AND POINTS TO PONDER

7.1. Why is it that many software developers don't pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?

7.2. You have been given the responsibility to elicit requirements from a customer who tells you he is too busy to meet with you. What should you do?

7.3. Discuss some of the problems that occur when requirements must be elicited from three or four different customers.

7.4. Your instructor will divide the class into groups of four or six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security function described in this chapter. Conduct a requirements gathering meeting using the guidelines presented in this chapter.

7.5. Develop a complete use case for one of the following activities:

a. Making a withdrawal at an ATM
b. Using your charge card for a meal at a restaurant
c. Searching for books (on a specific topic) using an online bookstore

7.6. Write a user story for one of the activities listed in Problem 7.5.

7.7. Consider the use case you created in Problem 7.5, and write a nonfunctional requirement for the application.

7.8. Using the template presented in Section 7.5.2, suggest one or more analysis patterns for the following application domains:

a. E-mail software.
b. Internet browsers.
c. Mobile app creation software.

7.9. What does *win-win* mean in the context of negotiation during the requirements engineering activity?

7.10. What do you think happens when requirement validation uncovers an error? Who is involved in correcting the error?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

8 REQUIREMENTS MODELING—A RECOMMENDED APPROACH

The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model is actually a set of models that make up the first technical representation of a system. Software engineers often prefer to include graphical representations of complex model relationships.

KEY CONCEPTS

activity diagram 146
analysis classes 137
attributes 140
behavioral model 149
class-based modeling 127
collaborations 144
CRC modeling 144
events 149
formal use case 135
functional model 146
grammatical parse 137
operations 140
procedural view 146
requirements analysis 127
requirements modeling 129
responsibilities 144
scenario-based modeling 128
sequence diagrams 148
state diagrams 150
swimlane diagram 151
UML models 130
use cases 131
  documentation 135
  exception 134



QUICK LOOK

What is it? Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an analyst) builds these models from requirements elicited from various stakeholders.

Why is it important? Requirements models can be readily evaluated by all stakeholders, resulting in useful feedback at the earliest possible time. Later, as the model is refined, it becomes the basis for software design.

What are the steps? Requirements modeling combines three steps: scenario-based modeling, class modeling, and behavioral modeling.

What is the work product? Usage scenarios, called use cases, describe software functions and usage. In addition, a series of UML diagrams can be used to represent system behavior and other aspects.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency.

For some types of software, a user story (Section 7.3.2) may be the only requirements modeling representation that is required. For others, formal use cases (Section 7.4) and class-based models (Section 8.3) may be developed. Class-based modeling represents the objects that the system will manipulate, the operations (also called *methods* or *services*) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Class-based methods can be used to create a representation of an application that can be understood by nontechnical stakeholders.

In still other situations, complex application requirements may demand an examination of how an application behaves in reaction to either internal or external events. These behaviors need to be modeled (Section 8.5) as well. UML diagrams have become a standard software engineering means of modeling analysis model element relationships and behaviors graphically. As the requirements model is refined and expanding, it evolves into a specification that can be used by software engineers in the creation of the software design.

The important thing to keep in mind when modeling requirements is to only create the models that will be used by the development team. If models developed early in a requirements analysis phase of a project are not referred to during the design and implementation phases, they may not be worth updating. The sections that follow present a series of informal guidelines that will assist in the creation and representation of requirements models.

8.1 REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer,* an *analyst,* or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 7).

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors."
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and how classes collaborate to achieve system requirements.
- *Behavioral models* that depict how the software reacts to internal or external "events."
- *Data models* that depict the information domain for the problem.
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as they move through the system.

These models provide a software designer with information that can be translated to architectural-, interface-, and component-level designs. Finally, the requirements

model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this section, we focus on *scenario-based modeling*—a technique that is very popular throughout the software engineering community. In Sections 8.3 and 8.5 we consider class-based modeling and behavioral modeling. Over the past decade, flow and data modeling have become less commonly used, while scenario and class-based methods, supplemented with behavioral approaches have grown in popularity.[1]

8.1.1 Overall Objectives and Philosophy

Throughout analysis modeling, your primary focus is on *what,* not *how.* What user interaction occurs, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?[2]

In previous chapters, we noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.[3]

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality (software, hardware, data, human elements) and a software design (Chapters 9 through 14) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 8.1.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division between analysis and design modeling tasks is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

8.1.2 Analysis Rules of Thumb

Several rules of thumb [Arl02] are worth considering when creating an analysis model. First, focus on the problem or business domain while keeping the level of abstraction high. Second, recognize that an analysis model should provide insight into

---

1 A presentation of flow-oriented modeling and data modeling is no longer included in this chapter. However, copious information about these older requirements modeling methods can be found on the Web. If you have interest, use the search phrase "structured analysis."

2 It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what.* However, the primary focus should remain on *what.*

3 Alternatively, the software team may choose to create a prototype (Chapter 4) to better understand requirements for the system.

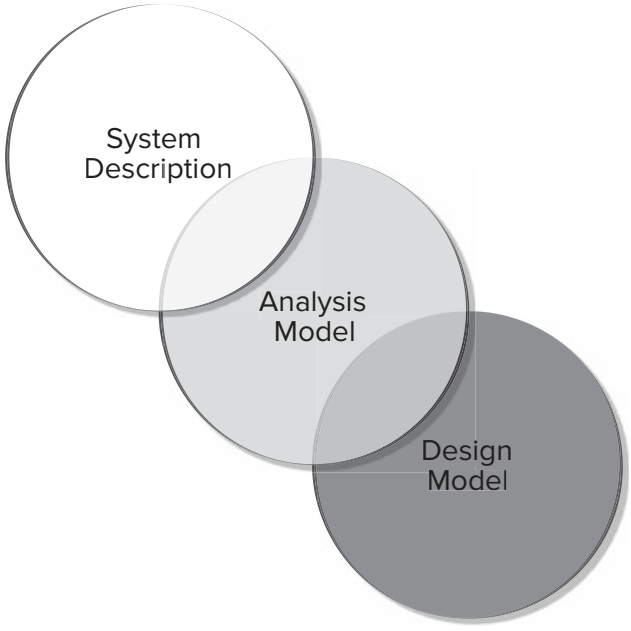


FIGURE 8.1
The analysis model as a bridge between the system description and the design model

the information domain, the function, and the behavior of the software. Third, delay a consideration of software architecture and nonfunctional details until later in the modeling activity. Also, it's important to be aware of the ways in which elements of the software are interconnnected with other elements (we call this *system coupling*).

The analysis model must be structured in a way that provides value to all stakeholders and should be kept as simple as possible without sacrificing clarity.

8.1.3 Requirements Modeling Principles

Over the past four decades, several requirements modeling methods have been developed. Investigators have identified requirements analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. A set of operational principles relates analysis methods:

Principle 1. ***The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize the data that are maintained permanently.

Principle 2. ***The functions that the software performs must be defined.*** Software functions provide direct benefit to end users and those that provide internal support for those features that are user visible. Some functions transform data that

flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements.

Principle 3. ***The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns,* and it is a key strategy in requirements modeling.

Principle 5. ***The analysis task should move from essential information toward implementation detail.*** Analysis modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively, a keyboard command might be typed, a game pad joystick (or mouse) might be pointed in a specific direction, a motion-sensitive device might be waved in the air, or a device that reads the player's body or eye movements directly can be used.

By applying these principles, a software engineer approaches a problem systematically. But how are these principles applied in practice? This question will be answered in the remainder of this chapter.

8.2 SCENARIO-BASED MODELING

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Using UML[4] to model requirements begins with the creation of scenarios in the form of use case diagrams, activity diagrams, and sequence diagrams.

4 UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.

8.2.1 Actors and User Profiles

A UML *actor* models an entity that interacts with a system object. Actors may represent roles played by human stakeholders or external hardware as they interact with system objects by exchanging information. A single physical entity may be portrayed by several actors if the physical entity takes on several roles that are relevant to realizing different system functions.

A UML *profile* provides a way of extending an existing model to other domains or platforms. This might allow you to revise the model of a Web-based system and model the system for various mobile platforms. Profiles might also be used to model the system from the viewpoints of different users. For example, system administrators may have a different view of the functionality of an automated teller machine than end users.

8.2.2 Creating Use Cases

In Chapter 7, we discussed user stories as a way of summarizing the stakeholders' perspective on how they will interact with the proposed system. However, they are written in plain English or the language used by the stakeholders. Developers need more precise means of describing this interaction before beginning to create the software. Alistair Cockburn characterizes a *use case* as a "contract for behavior" [Coc01b]. As we discussed in Chapter 7, the "contract" defines the way in which an actor[5] uses a computer-based system to accomplish some goal. In other words, a use case captures the interactions that occur between producers and consumers of information within the system itself. In this section, we examine how preliminary use cases are developed as part of the analysis modeling activity.[6]

In Chapter 7, we noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a modeling tool.

What to Write About? The first two requirements engineering tasks—inception and elicitation—provide you with the information you'll need to begin writing use cases. Requirements gathering meetings and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 8.4) developed as part of requirements modeling.

---

5 An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor "calls on the system to deliver one of its services" [Coc01b].

6 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis and design is discussed in detail in Chapter 12.

SAFEHOME



Developing Another Preliminary Use Case

The scene: A meeting room, during the second requirements gathering meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: It's time that we begin talking about the *SafeHome* surveillance function. Let's develop a user scenario for access to the surveillance function.

Jamie: Who plays the role of the actor on this?

Facilitator: I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

Meredith: You want to do it the same way we did it last time, right?

Facilitator: Right . . . same way.

Meredith: Well, obviously the reason for surveillance is to allow the homeowner to check out the house while he or she is away, to record and play back video that is captured . . . that sort of thing.

Ed: Will we use compression to store the video?

Facilitator: Good question, Ed, but let's postpone implementation issues for now. Meredith?

Meredith: Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we need to have the AR/VR tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

Facilitator (smiling): Took the words right out of my mouth.

Meredith: Um . . . I want to gain access to the surveillance function either via a mobile device or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a mobile device or PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

Jamie: Those are called thumbnail views.

Meredith: Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

Facilitator: Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a device window.
- Control pan and zoom for a specific camera.

- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format like the one proposed in Chapter 7.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views* (**ACS-DCV**). The stakeholder who takes on the role of the **homeowner** actor might write the following user story:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any mobile device with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera," the original viewing window disappears, and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is free flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98].

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98]:

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."

7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is yes. Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again, the answer to the question is yes. A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."[7] This error condition becomes a secondary scenario.

Is it possible that the actor will encounter some other behavior at this point? Again, the answer to the question is yes. As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with several options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

Each of the situations described in the preceding paragraphs is characterized as a use case exception. An *exception* describes a situation (either a failure condition or

7 In this case, another actor, the **system administrator,** would have to configure the floor plan, install (e.g., assign an equipment ID) all cameras and initialize them, and test each camera to be certain that it is accessible via the system and through the floor plan.

an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that the validation function is invoked, and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based or mobile interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed by asking and answering these questions should be "rationalized" [Coc01b] using the following criteria: An exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

8.2.3 Documenting Use Cases

The informal use cases presented in Section 8.2.2 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case. The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that "gets the use case started" [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 8.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

Most developers like to create graphical representation as they create use cases out of user stories. A diagrammatic representation can facilitate better understanding of the problem by all stakeholders, particularly when the scenario is complex. As we noted earlier in this book, UML provides use case diagramming capability. Figure 8.2 depicts a use case diagram for the *SafeHome* product. The use case diagram helps to show the relations among the use cases in the usage scenario. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

Each modeling notation has limitations, and the UML use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the

SafeHome



Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of cameras placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" button from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Infrequent.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

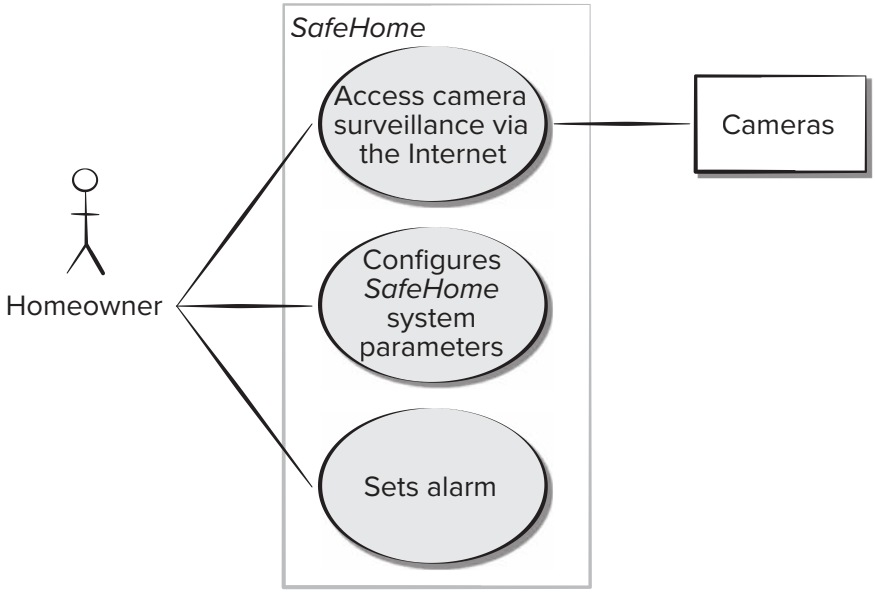
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

FIGURE 8.2

Preliminary use case diagram for the *SafeHome* system



description is unclear, the use case can be misleading or ambiguous. A use case focuses on function and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

8.3 CLASS-BASED MODELING

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you "look around" the problem space of a software application, the classes (and objects) may be more difficult to comprehend.

8.3.1 Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model (Section 8.2) and performing a "grammatical parse" [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

But what should we look for once all the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

This categorization is but one of many that have been proposed in the literature.[8] For example, Budd [Bud96] suggests a taxonomy of classes that includes *producers* (sources) and *consumers* (sinks) of data, *data managers*, *view* or *observer classes*, and *helper classes*.

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[9] for the *SafeHome* security function.

> The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC or a control panel.
>
> During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.
>
> When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.*
>
> The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

8 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 10.3.

9 A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor's point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

Extracting the nouns, we can propose several potential classes:

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that we call each entry in the list a "potential" object. We must consider each further before a final decision is made.

Coad and Yourdon [Coa91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

1. **Retained information.** The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design but is probably better represented as an attribute of another class during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential class, and these attributes apply to all instances of the class.
5. **Common operations.** A set of operations can be defined for the potential class, and these operations apply to all instances of the class.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy most of these characteristics. The decision for inclusion of potential classes in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated. However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must

be made. You should apply the selection characteristics to the list of potential *SafeHome* classes:

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that: (1) the preceding list is not all-inclusive, so additional classes would have to be added to complete the model; (2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., number and type are attributes of **Sensor,** and master password and telephone number may become attributes of **System**); and (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

8.3.2 Defining Attributes and Operations

Attributes describe a class that has been selected for inclusion in the analysis model. It is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class. In addition, the following question should be answered for each class: *What data items (composite and/or elementary) fully define this class in the context of the problem at hand?*

To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation and deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

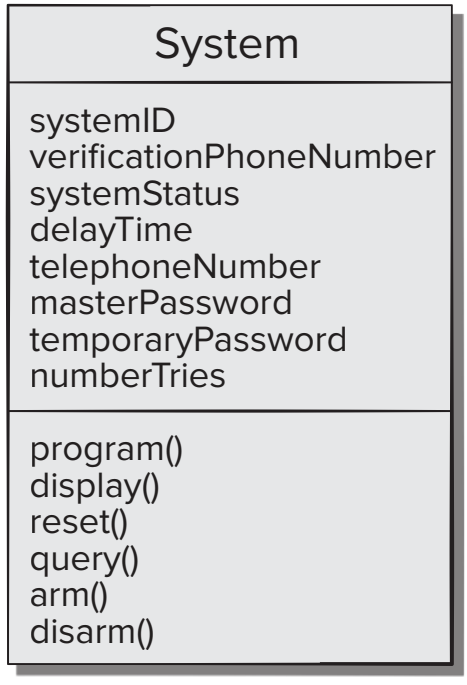
identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

FIGURE 8.3

Class diagram for the System class



Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (Figure 8.3).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 8.3. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations (Section 8.3.3). Therefore, an operation must have "knowledge" of the nature of the class attributes and associations.

8.3.3 UML Class Models

As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied, and verbs are isolated. Some of these verbs will be legitimate operations and can be easily

connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is *assigned* a number and type" or "a master password is *programmed* for *arming and disarming* the system." These phrases indicate several things:

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

SAFEHOME



Class Models

The scene: Ed's cubicle, as analysis modeling begins.

The players: Jamie, Vinod, and Ed, all members of the *SafeHome* software engineering team.

The conversation:

[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

Ed: So, when the homeowner wants to pick a camera, he or she must pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 8.4.)

Jamie: So, **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

Ed: Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 8.3.3.]

Vinod: So, the actual floor plan is made up of walls and contains cameras and sensors that are placed within those walls. How does the floor plan know where to put those objects?

Ed: It doesn't, but the other classes do. See the attributes under, say, **WallSegment,** which is used to build a wall. The wall segment has start and stop coordinates and the *draw()* operation does the rest.

Jamie: And the same goes for windows and doors. Looks like camera has a few extra attributes.

Ed: Yeah, I need them to provide pan and zoom info.

Vinod: I have a question. Why does the camera have an ID, but the others don't? I notice you have an attribute called **nextWall.** How will **WallSegment** know what the next wall will be?

Ed: Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

Jamie: Give me a break . . . I'll bet you've already figured it out.

Ed (smiling sheepishly): True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

Jamie: Looks pretty good to me, but I have a few more questions.

(Jamie asks questions which result in minor modifications.)

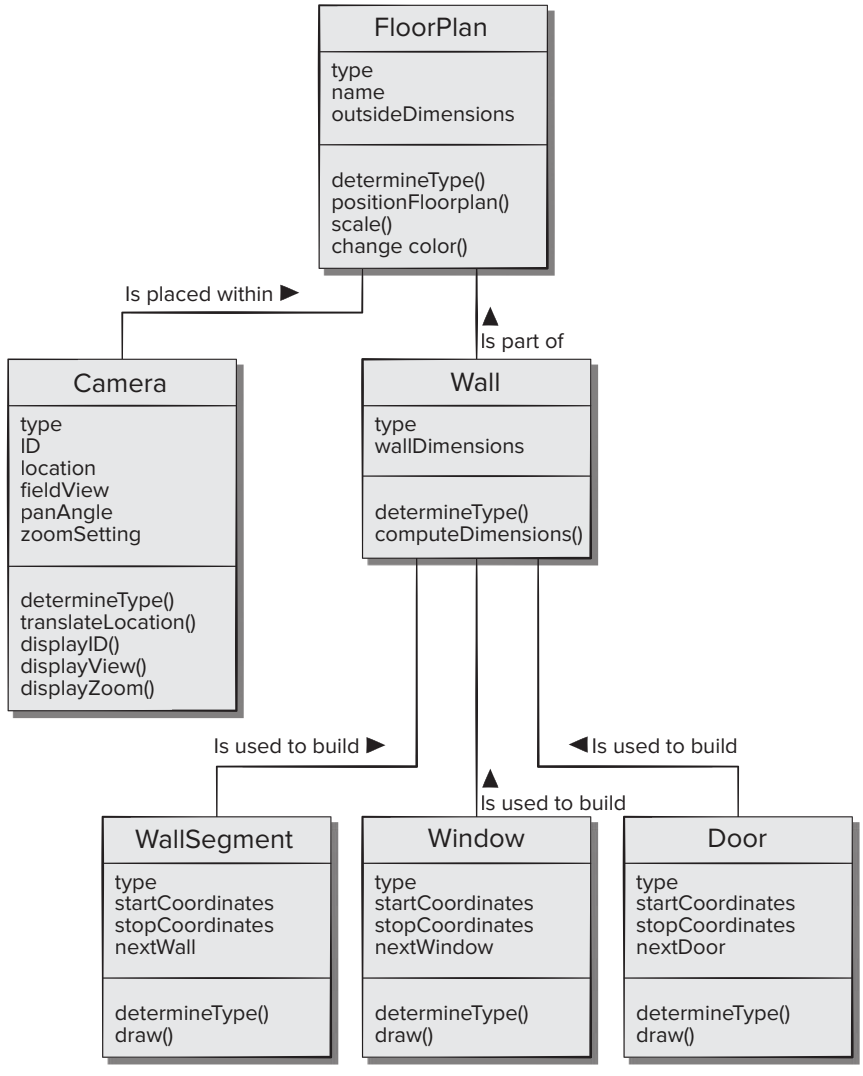
Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

Ed: I'm not quite sure how to do them.

Vinod: It's not hard and they really pay off. I'll show you.

FIGURE 8.4

Class diagram for FloorPlan (see sidebar discussion)



Upon further investigation, it is likely that the operation *program()* will be divided into several more specific suboperations required to configure the system. For example, *program()* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program()* as a single operation.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, we explore this matter in a bit more detail.

In many instances, two analysis classes are related to one another in some fashion. In UML, these relationships are called *associations*. Referring to Figure 8.4, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall.** The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment, Window,** and **Door.**

8.3.4 Class-Responsibility-Collaborator Modeling

Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. A CRC model can be viewed as a collection of index cards. Each index card contains a list of responsibilities on the left side and the corresponding collaborations that enable the responsibilities to be fulfilled on the right side (Figure 8.5). *Responsibilities* are the attributes and operations that are relevant for the class. *Collaborators* are those classes that provide a class with the information needed or action required to complete a responsibility. A simple CRC index card for the **FloorPlan** class is illustrated in Figure 8.5.

The list of responsibilities shown on the CRC card is preliminary and is subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that requires their collaboration.

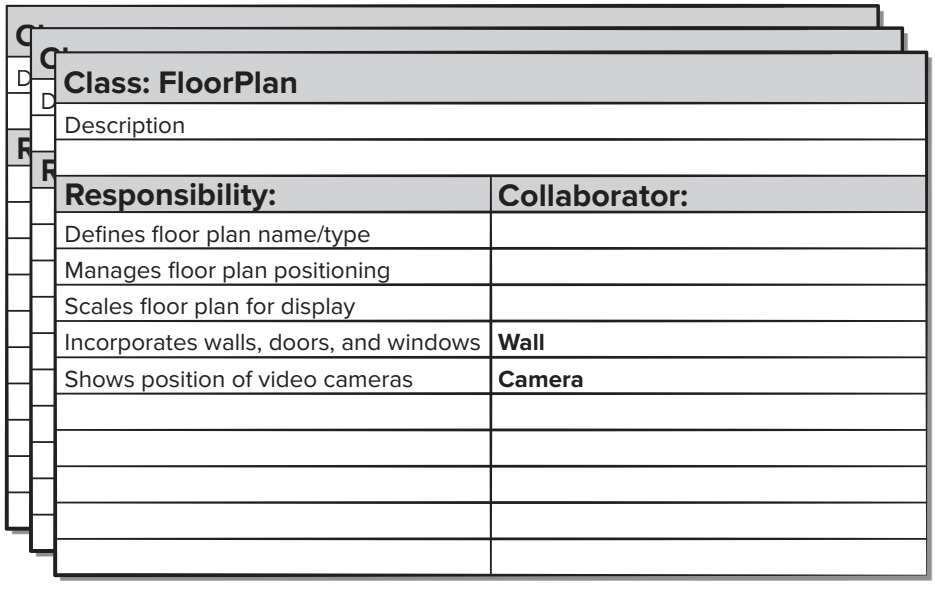
Classes. Basic guidelines for identifying classes and objects were presented earlier in Section 8.3.1.

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) were presented in Section 8.3.2.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a

FIGURE 8.5

A CRC model index card



responsibility itself, or (2) a class can collaborate with other classes. Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class.

As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a status attribute to "not ready." Sensor information can be acquired from each **Sensor** object. The responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor.**

Once a complete CRC model has been developed, the representatives from the stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. No reviewer should have two cards that collaborate.
2. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
3. When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use case requirement.
4. If an error is found, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or revising lists of responsibilities or collaborations on existing cards.

SAFEHOME



CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Vinod and Ed, members of the *SafeHome* software engineering team.

The conversation:

(Vinod has decided to show Ed how to develop CRC cards by showing him an example.)

Vinod: While you've been working on surveillance and Jamie has been tied up with security, I've been working on the home management function.

Ed: What's the status of that? Marketing kept changing its mind.

Vinod: Here's the first-cut use case for the whole function . . . we've refined it a bit, but it should give you an overall view . . .

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a mobile device or an Internet connection to control electronic devices that have wireless interface controllers. The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, and to set my heating and air-conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is home, another is away, a third is overnight travel, and a fourth is extended travel. All these situations will have settings that will be applied to all devices. In the overnight travel and extended travel states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air-conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

Ed: Do the hardware guys have all the wireless interfacing figured out?

Vinod (smiling): They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management, and we can use one as an example. Let's use the **HomeManagementInterface** class.

Ed: Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

Vinod: I thought you didn't understand CRC.

Ed: So, looking at the **HomeManagementInterface** card, when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 8.4.)

Vinod: Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

Ed: Good way to find omissions or errors.

Vinod: Yep.

8.4 FUNCTIONAL MODELING

The *functional model* addresses two application processing elements, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the app to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user. For example, a financial mobile app might implement a variety of financial functions (e.g., computation of mortgage payment). These functions may be implemented using operations within analysis classes, but from the point of view of the end user, the function (more correctly, the data provided by the function) is the visible outcome.

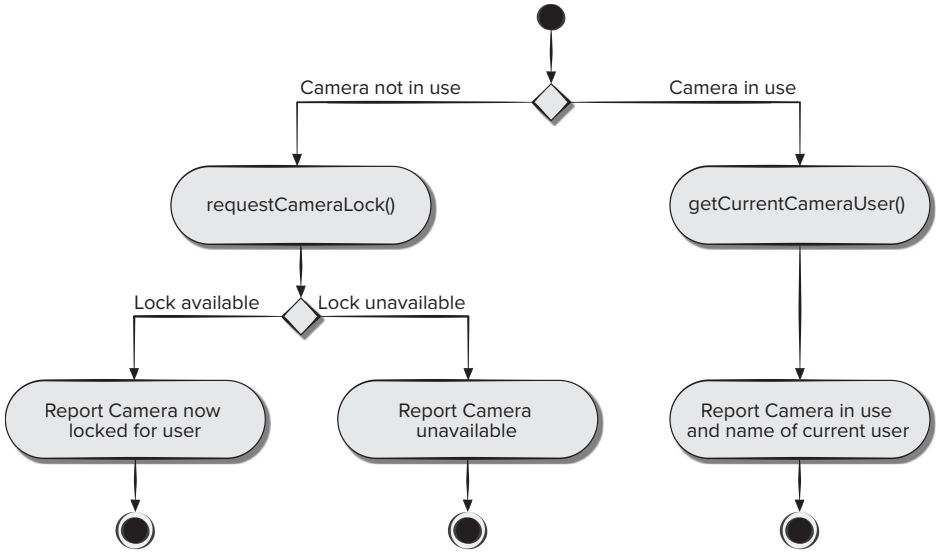
At a lower level of procedural abstraction, the requirements model describes the processing to be performed by analysis class operations. These operations manipulate class attributes and are involved as classes collaborate with one another to accomplish some required behavior.

8.4.1 A Procedural View

Regardless of the level of procedural abstraction, the UML activity diagram can be used to represent processing details. At the analysis level, activity diagrams should be used only where the functionality is relatively complex. Much of the complexity of mobile apps occurs not in the functionality provided, but rather with the nature of the information that can be accessed and the ways in which this can be manipulated.

FIGURE 8.6

Activity diagram for the *takeControlOfCamera()* operation



The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. An activity diagram is like a flowchart. The activity diagram (Figure 8.6) uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring.

An example of a relatively complex functionality for **SafeHomeAssured.com** is addressed by a use case entitled *Get recommendations for sensor layout for my space.* The user has already developed a layout for the space to be monitored, and in this use case, selects that layout and requests recommended locations for sensors within the layout. **SafeHomeAssured.com** responds with a graphical representation of the layout with additional information on the recommended locations for sensors. The interaction is quite simple and the content is somewhat more complex, but the underlying functionality is very sophisticated. The system must undertake a relatively complex analysis of the floor layout to determine the optimal set of sensors. It must examine room dimensions and the location of doors and windows and coordinate these with sensor capabilities and specifications. No small task! A set of activity diagrams can be used to describe processing for this use case.

The second example is the use case *Control cameras.* In this use case, the interaction is relatively simple, but there is the potential for complex functionality, given that this "simple" operation requires complex communication with devices located remotely and accessible across the Internet. A further possible complication relates to negotiation of control when multiple authorized people attempt to monitor and/or control a single sensor at the same time.

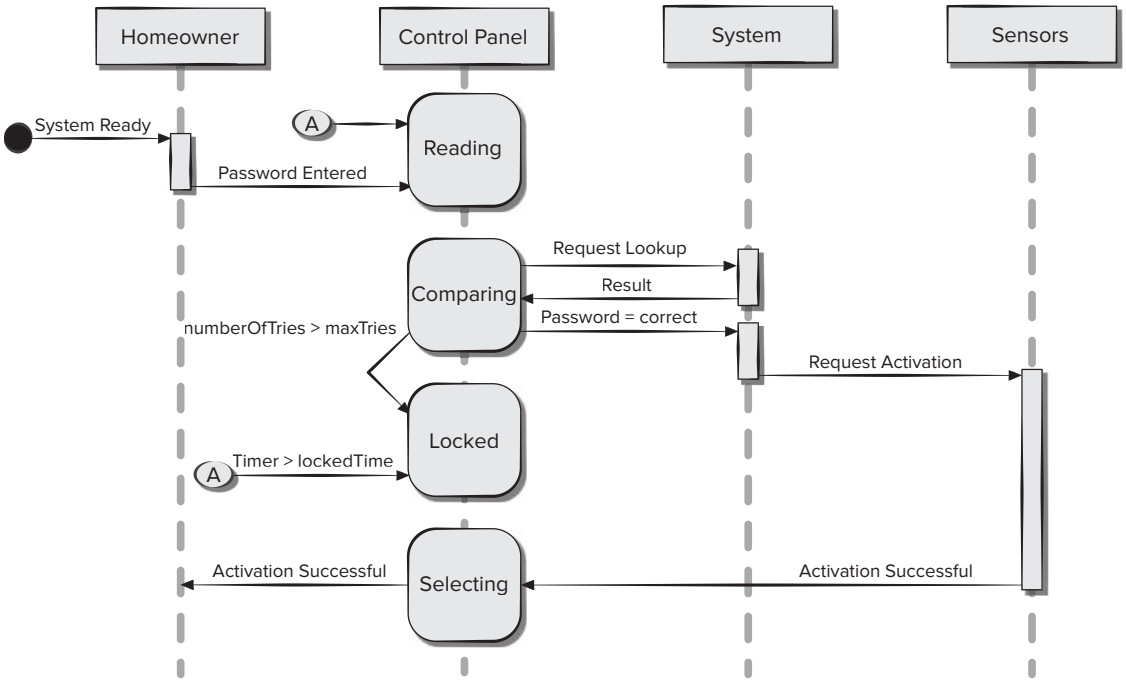
Figure 8.6 depicts an activity diagram for the *takeControlOfCamera()* operation that is part of the **Camera** analysis class used within the *Control cameras* use case. It should be noted that two additional operations are invoked with the procedural flow: *requestCameraLock(),* which tries to lock the camera for this user, and *getCurrentCameraUser(),* which retrieves the name of the user who is currently controlling the camera. The construction details indicating how these operations are invoked and the interface details for each operation are not considered until software design commences.

8.4.2 UML Sequence Diagrams

The UML *sequence diagram* can be used for behavioral modeling. Sequence diagrams can also be used to show how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. The sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 8.7 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

FIGURE 8.7 **Sequence diagram (partial) for the *SafeHome* security function**



The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System,** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System,** which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

8.5 BEHAVIORAL MODELING

A *behavioral model* indicates how software will respond to internal or external events or stimuli. This information is useful in the creation of an effective design for the system to be built. UML activity diagrams can be used to model how system elements respond to internal events. UML state diagrams can be used to model how system elements respond to external events.

To create the model, you should perform the following steps: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a state diagram for the system, and (5) review the behavioral model to verify accuracy and consistency. Each of these steps is discussed in the sections that follow.

8.5.1 Identifying Events with the Use Case

In Section 8.3.3, you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. An event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, reconsider the use case for a portion of the *SafeHome* security function.

> The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,**[10] transmits an event to the object **ControlPanel.**

10 In this example, we assume that each user (homeowner) that interacts with *SafeHome* has an identifying password and is therefore a legitimate object.

The event might be called *password entered*. The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

8.5.2 UML State Diagrams

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current values assigned to an object's attributes. The *active state* of an object indicates the status of the object as it undergoes a continuing transformation or processing. An *event* (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

State Diagrams for Analysis Classes. One component of a behavioral model is a UML state diagram[11] that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 8.8 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

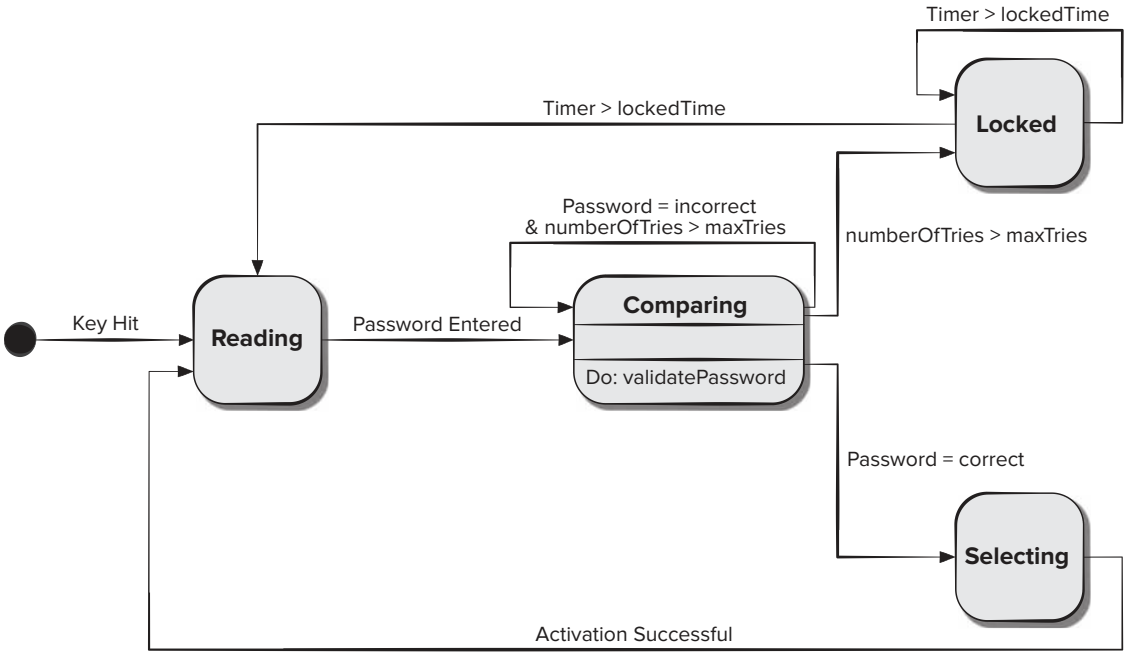
Each arrow shown in Figure 8.8 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 8.8 can be determined by examining the use case:

```
if (password input = 4 digits) then compare to stored password
```

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

11 If you are unfamiliar with UML, a brief introduction to this important modeling notation is presented in Appendix 1.

FIGURE 8.8 State diagram for the ControlPanel class



An *action* occurs concurrently with the state transition or because of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 8.8) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

8.5.3 UML Activity Diagrams

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Many software engineers like to describe activity diagrams as a way of representing how a system reacts to internal events.

An activity diagram for the **ACS-DCV** use case is shown in Figure 8.9. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case. For example, a user may only attempt to enter **userID** and **password** a limited number of times. A decision diamond represents this below: "Prompt for reentry."

The UML swimlane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (Section 8.3.1) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

FIGURE 8.9

Activity diagram for Access camera surveillance via the Internet—display camera views function

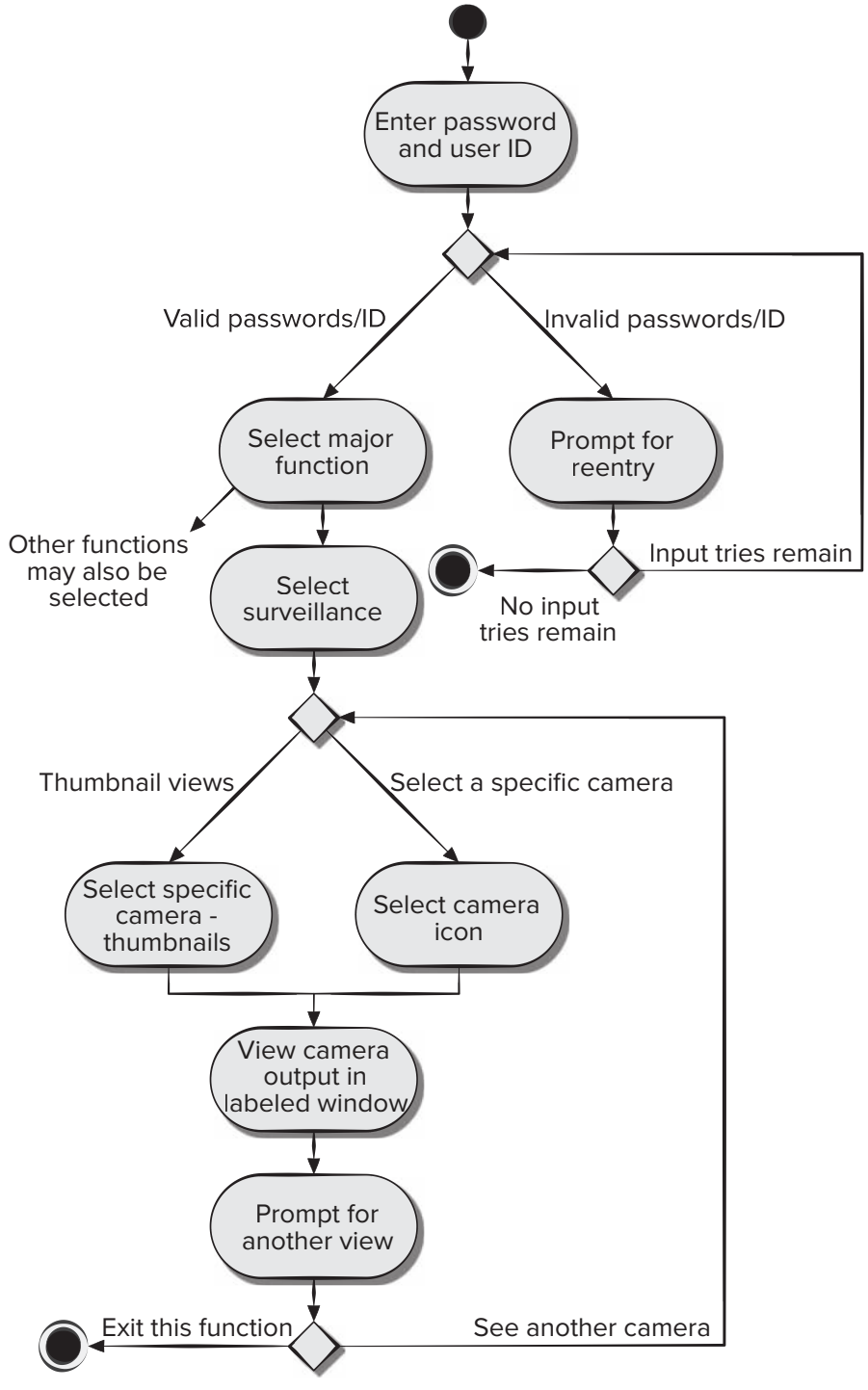
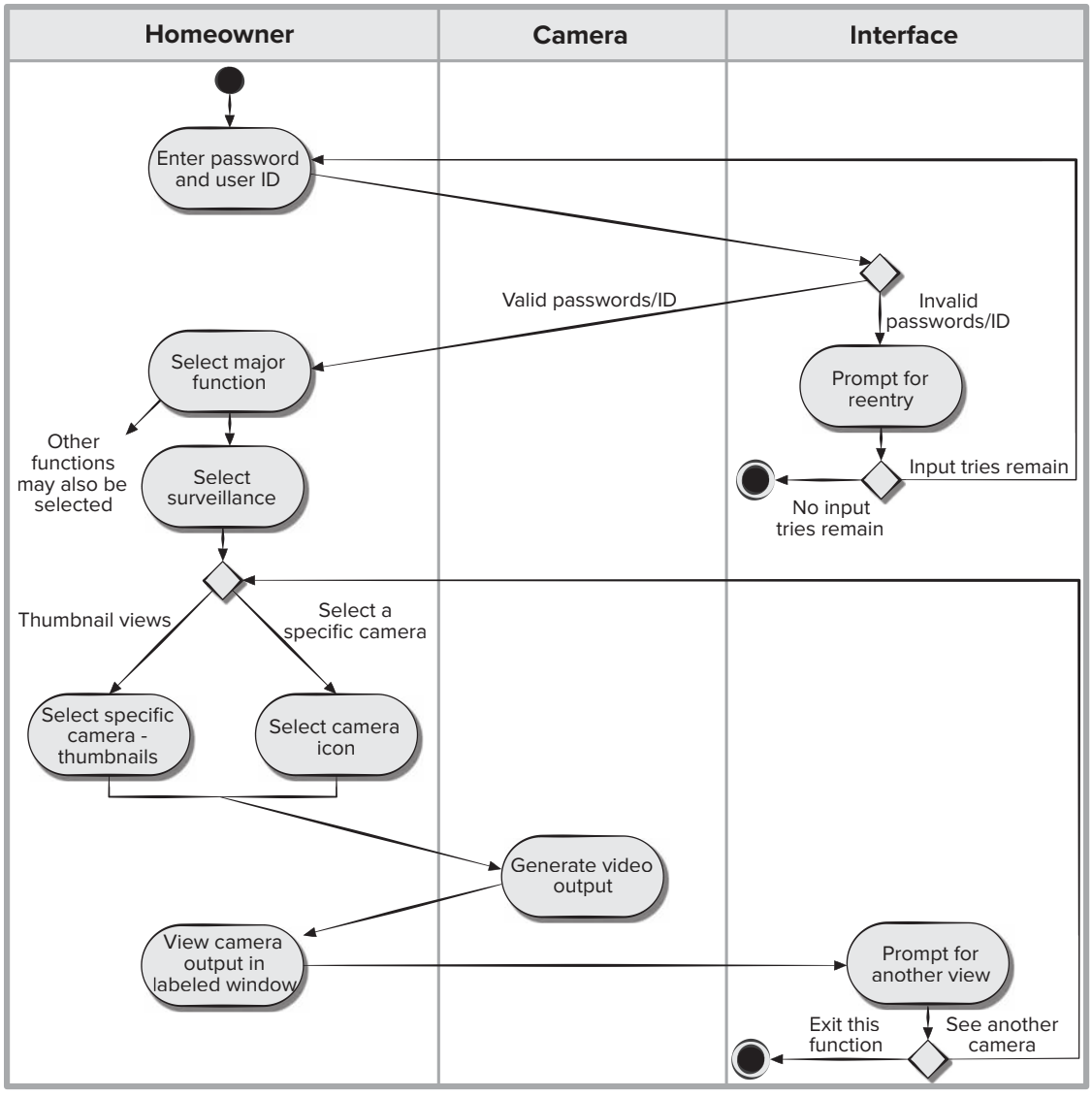


FIGURE 8.10 **Swimlane diagram for Access camera surveillance via the Internet—display camera views function**



Three analysis classes—**Homeowner, Camera,** and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 8.9. Referring to Figure 8.10, the activity diagram is rearranged so that activities associated with an analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the

homeowner. The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the Interface swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. Taken together they can be used to represent the way various actors invoke specific functions (or other procedural steps) to meet the requirements of the system.

8.6 SUMMARY

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level description that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the key steps for a specific function or interaction. The degree of use case formality and detail varies, but they can provide necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a graphical representation that depicts the processing flow within a specific scenario. Temporal relations in a use case can be modeled using sequence diagrams.

Class-based modeling uses information derived from use cases and other written application descriptions to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined using the parse results.

A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations, and dependencies among classes.

Behavioral modeling during requirements analysis depicts dynamic behavior of the software. The behavioral model uses input from scenario-based or class-based elements to represent the states of analysis classes. To accomplish this, states are identified, the events that cause a class (or the system) to make a transition from one state to another are defined, and the actions that occur as transition is accomplished are also identified. UML state diagrams, activity diagrams, swim lane diagrams, and sequence diagrams can be used for behavioral modeling.

PROBLEMS AND POINTS TO PONDER

8.1. Is it possible to begin coding immediately after a requirements model has been created? Explain your answer, and then argue the counterpoint.

8.2. An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.

8.3. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

Draw a UML use case diagram PHTRS system. You'll have to make a number of assumptions about the manner in which a user interacts with this system.

8.4. Write two or three use cases that describe the roles of various actors in the PHTRS described in Problem 8.3.

8.5. Develop an activity diagram for one aspect of PHTRS.

8.6. Develop a swimlane diagram for one or more aspects of PHTRS.

8.7. Develop a class model for the PHTRS system presented in Problem 8.3.

8.8. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 8.3.

8.9. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

8.10. How does a sequence diagram differ from a state diagram? How are they similar?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

9 DESIGN CONCEPTS

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. *Design principles* establish an overriding philosophy that guides the design work you must perform. *Design concepts* must be understood before the mechanics of design practice are applied, and *design practice* leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

KEY CONCEPTS

abstraction 163
architecture 163
cohesion 167
data design 174
design modeling principles 173
design process 159
functional independence 167
good design 160
information hiding 166
modularity 165
patterns 164
quality attributes 160
quality guidelines 160
refactoring 168
separation of concerns 165
software design 157
stepwise refinement 167
technical debt 157

QUICK LOOK

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where requirements and technical considerations come together in the formulation of a product or system. Design creates a representation or model of the software and provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks while continuing communication with the stakeholders.

Why is it important? During the design phase, you model the system or product that needs to be built. The design model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

What are the steps? Design makes use of several different representations of the software. First, the architecture of the system or product must be modeled. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are represented. Finally, the software components that are used to construct the system are designed.

What is the work product? A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design.

How do I ensure that I've done it right? The design model is assessed by the software team (including relevant stakeholders) in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

Design is pivotal to successful software engineering. Some developers are tempted to begin programming once the use cases have been created, without regard to how the software components needed to implement the use cases relate to one another. It is possible to do analysis, design, and implementation iteratively by creating several software increments. It is a bad idea to ignore the design considerations needed to create an appropriate architecture for the evolving software product. *Technical debt* is a concept in software development that refers to costs associated with rework caused by choosing a "quick and dirty" solution right now instead of using a better approach that would take more time. It is impossible to avoid creating technical debt when building a software product incrementally. However, a good development team must try to pay down this technical debt by refactoring (Section 9.3.9) the software on a regular basis. Just like taking out a loan, you can wait until the loan is due and pay a lot of interest or you can pay the loan off a little at a time and pay less interest overall.

One strategy to keep technical debt in check without delaying coding is to make use of the design practices of diversification and convergence. *Diversification* is the practice of identifying possible design alternatives suggested by the elements of the requirements model. *Convergence* is the process of evaluating and rejecting design alternatives that do not meet the constraints imposed by the nonfunctional requirements defined for the software solution. Diversification and convergence combine (1) intuition and judgment based on experience in building similar entities, (2) a set of principles and/or heuristics that guide the way in which the model evolves, (3) a set of criteria that enables quality to be judged, and (4) a process of iteration that ultimately leads to a final design representation. Once a viable design alternative is identified this way, the developers are in a good position to create a software increment that is not likely to be a throwaway prototype.

Software design changes continually as new methods, better analysis, and broader understanding evolve.[1] Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied.

In this chapter, we explore the fundamental concepts and principles that are applicable to all software design, the elements of the design model, and the impact of patterns on the design process. In Chapters 10 through 14 we'll present a variety of software design methods as they are applied to architectural, interface, and component-level design as well as pattern-based, mobile, and user experience design approaches.

9.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

1 Those readers with further interest in the philosophy of software design might have interest in Philippe Kruchen's intriguing discussion of "postmodern" design [Kru05].

Each of the elements of the requirements model (Chapter 8) provide information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 9.1. The requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

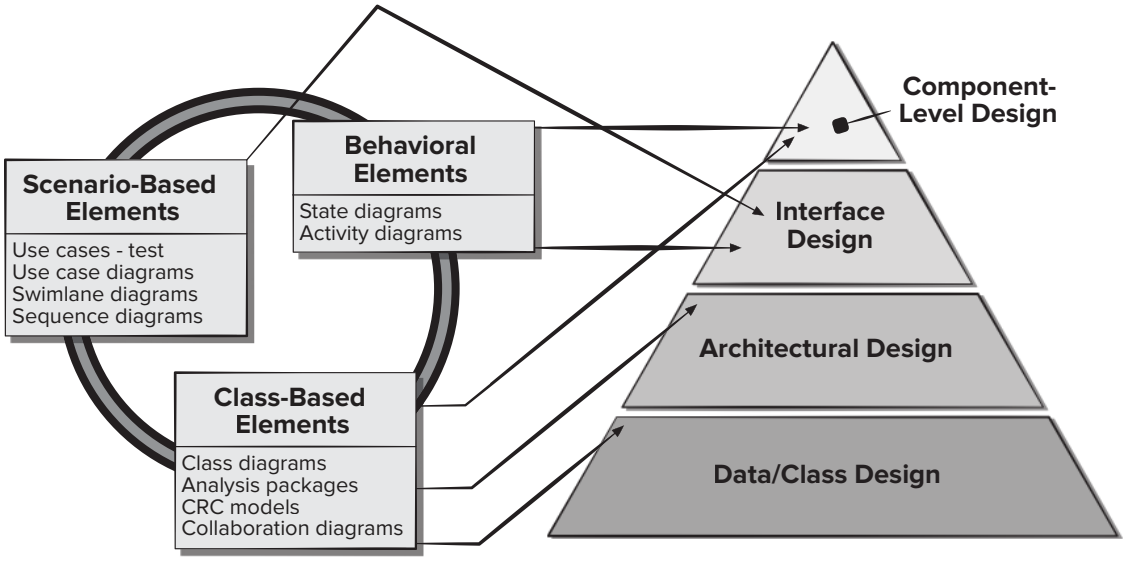
The data/class design transforms class models (Chapter 8) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC model and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural style, and patterns (Chapter 14) that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha15]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

FIGURE 9.1 **Translating the requirements model into the design model**



During design you make decisions that will ultimately affect the success of software construction and, just as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. It provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholders' requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process. Late in the project is when time is short, and many budgeted dollars have already been spent.

SAFEHOME



Design Versus Coding

The scene: Jamie's cubicle, as the team prepares to translate requirements into design.

The players: Jamie, Vinod, and Ed, all members of the *SafeHome* software engineering team.

The conversation:

Jamie: You know, Doug [the team manager] is obsessed with design. I gotta be honest, what I really love doing is coding. Give me C++ or Java, and I'm happy.

Ed: Nah . . . you like to design.

Jamie: You're not listening—coding is where it's at.

Vinod: I think what Ed means is that you don't really like coding; you like to design and express it in code. Code is the language you use to represent the design.

Jamie: And what's wrong with that?

Vinod: Level of abstraction.

Jamie: Huh?

Ed: A programming language is good for representing details like data structures and algorithms, but it's not so good for representing architecture or component-to-component collaboration . . . stuff like that.

Vinod: And a screwed-up architecture can ruin even the best code.

Jamie (thinking for a minute): So, you're saying that I can't represent architecture in code . . . that's not true.

Vinod: You can certainly imply architecture in code, but in most programming languages, it's difficult to get a quick, big-picture read on architecture by examining the code.

Ed: And that's what we want before we begin coding.

Jamie: Okay, maybe design and coding are different, but I still like coding better.

9.2 THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent

refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connections may not be obvious at these lower levels of abstraction.

9.2.1 Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 16. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design should implement all explicit requirements contained in the requirements model, and it must accommodate all the implicit requirements desired by stakeholders.
- The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is a goal of the design process. But how is each of these goals achieved?

Quality Guidelines. To evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 9.3, we discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that (a) has been created using recognizable architectural styles or patterns, (b) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (c) can be implemented in an evolutionary fashion,[2] thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

2 For smaller systems, design can sometimes be developed linearly.

Chance alone will not achieve these design guidelines. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

INFO



Assessing Design Quality—The Technical Review

Design is important because it allows a software team to assess the quality[3] of the software before it is implemented—at a time when errors, omissions, or inconsistencies are easy and inexpensive to correct. But how do we assess quality during design? The software can't be tested, because there is no executable software to test. What to do?

During design, quality is assessed by conducting a series of technical reviews (TRs). TRs are discussed in detail in Chapter 16,[4] but it's worth providing a summary of the technique at this point. A technical review is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed. Each person plays a role. The *review leader* plans the meeting, sets an agenda, and runs the meeting; the *recorder* takes notes so that nothing is missed; and the *producer* is the person whose work product (e.g., the design of a software component) is being reviewed. Prior to the meeting, each person on the review team is given a copy of the design work product and is asked to read it, looking for errors, omissions, or ambiguity. When the meeting commences, the intent is to note all problems with the work product so that they can be corrected before implementation begins. The TR typically lasts between 60 to 90 minutes. After the TR concludes, the review team determines whether further actions are required from the producer before the design work product can be approved as part of the final design model.

9.2.2 The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs [Den73] and methods for refining software structures in a top-down "structured" manner ([Wir71], [Dah72], [Mil72]). Newer design approaches (e.g., [Jac92], [Gam95]) proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture [Kru06] and the design patterns that can be used to implement software architectures and lower levels of design abstractions (e.g., [Hol06], [Sha05]). There is a growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04], which focus on techniques for achieving more effective modularity and architectural structure in the designs that are created.

In the past 10 years, Search-Based Software Engineering (SBSE) techniques have been applied to all phases of the software engineering life cycle, including design [Har12]. SBSE attempts to solve software engineering problems using automated search techniques augmented by operations research and machine learning algorithms to provide design recommendations to software developers. Many modern software systems must accommodate a high degree of variability, both in their deployment environments and

3 The quality factors discussed in Chapter 23 can assist the review team as it assesses quality.

4 You might consider looking ahead to Chapter 16 at this time. Technical reviews are a critical part of the design process and are an important mechanism for achieving design quality.

the number of usage scenarios they expected to satisfy. Design of *variability-intensive systems*[5] requires developers to anticipate future changes in the features to be modified in future versions of the product being designed today [Gal16]. A detailed discussion of the design of variability-intensive systems is beyond the scope of this book.

Several design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapter 8, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, each of these methods has common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

TASK SET



Generic Task Set for Design

Please note: These tasks are often performed iteratively and in parallel. They are rarely completed sequentially and in isolation from one another unless you are following the waterfall process model.

1. Examine the information model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   - Translate an analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select design patterns for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create a behavioral model of the interface.
   - Define interface objects and control mechanisms.
   - Review the interface design, and revise as required.
7. Conduct component-level design. Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.
   - Review each component, and correct all errors uncovered.
8. Develop a deployment model.

5 Variability-intensive systems refers to systems that may be required to be self-modifying based on changes in the run-time environment or families of software products resulting from product line engineering practices for building specialized product variants out of existing software products.

9.3 Design Concepts

Several fundamental software design concepts have evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you define criteria that can be used to partition software into individual components, separate out data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design. These concepts help developers design the software actually needed, rather than simply focusing on creating any old working program.

9.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment (e.g., a user story). At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology to state a solution (e.g., use case). Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented (e.g., pseudocode).

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *use* for a camera in the *SafeHome* system. *Use* implies a long sequence of procedural steps (e.g., activate the *SafeHome* system on a mobile device, log on to the *SafeHome* system, select a camera to preview, locate the camera controls on mobile app user interface, etc.).[6]

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **camera.** Like any data object, the data abstraction for **camera** would encompass a set of attributes that describe the camera (e.g., camera ID, location, field view, pan angle, zoom). It follows that the procedural abstraction *use* would make use of information contained in the attributes of the data abstraction **camera.**

9.3.2 Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha15]. In its simplest form, architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are

6 It should be noted, however, that one set of operations can be replaced with another, if the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *use* would change dramatically if the camera were automatic and attached to a sensor that automatically triggered an alert on your mobile device.

used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garlan [Sha15] describe a set of properties that should be specified as part of an architectural design. *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics (e.g., nonfunctional system requirements)." *Families of related systems* "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."[7]

Given the specification of these properties, the architectural design can be represented using one or more of several different models [Gar95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.

Several different *architectural description languages* (ADLs) have been developed to represent these models [Sha15]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the ways in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The ways in which software architecture is characterized and its role in design are discussed in Chapter 10.

9.3.3 Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00]. Stated in another way, a design pattern describes a design structure that solves a well-defined design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

---

7 These families of related software products sharing common features are called *software product lines*.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different, pattern. Design patterns are discussed in detail in Chapter 14.

9.3.4 Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve.

It follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications for software modularity.

Separation of concerns is manifested in other related design concepts: modularity, functional independence, and refinement. Each will be discussed in the subsections that follow.

9.3.5 Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules,* that are integrated to satisfy problem requirements.

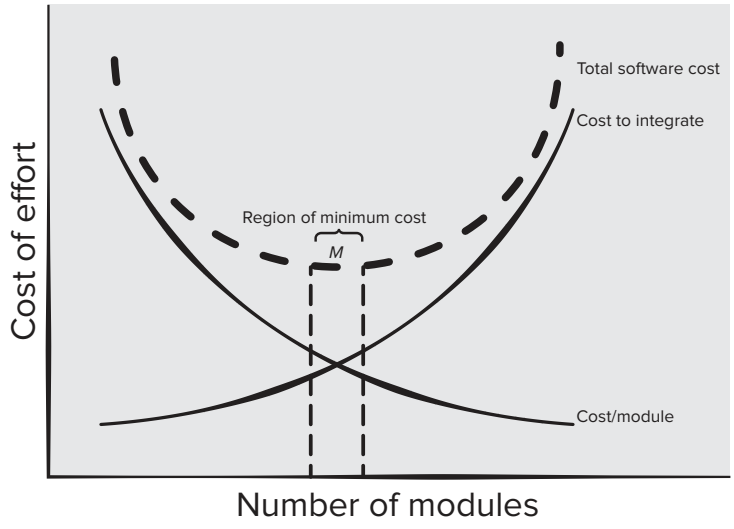
It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 9.2, the effort (cost) to develop an individual software module tends to decrease as the total number of modules increases.

Given the same set of requirements, the more modules used in your program means smaller individual sizes. However, as the number of modules grows, the effort (cost) associated with integrating modules with each other grows. These characteristics lead to a total cost or effort curve, shown in Figure 9.2. There is a number, *M,* of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.

The curves shown in Figure 9.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of *M*. Using too few modules or too many modules should be avoided.

FIGURE 9.2
Modularity and software cost



But how do you know the vicinity of *M*? How modular should you make software? The answers to these questions require an understanding of other design concepts considered in Sections 9.3.6 through 9.3.9.

You modularize a design (and the resulting program) so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

9.3.6 Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of *information hiding* [Par72] suggests that modules should be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

9.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] each allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality. Evaluation of your CRC card model (Chapter 8) can help you spot problems with functional independence. User stories that contain many instances of words such as *and* or *except* are not likely to encourage you to design modules that are "single-minded" system functions.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in Section 9.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnections among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less likely to propagate errors found in one module to other system modules.

9.3.8 Stepwise Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. Successively refining levels of procedural detail is a good way to develop an application. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is a process of *elaboration.* You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the

statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

9.3.9 Refactoring

An important design activity suggested for many agile methods (Chapter 3), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a large component that exhibits low cohesion (i.e., it performs three functions that have only a limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each of which exhibits high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. Refactoring tools [Soa10] are sometimes used to analyze code changes automatically and to "generate a test suite suitable for detecting behavioral changes."

SAFEHOME



Design Concepts

The scene: Vinod's cubicle, as design modeling begins.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

The conversation:

(All four team members have just returned from a morning seminar entitled "Applying Basic Design Concepts," offered by a local computer science professor.)

Vinod: Did you get anything out of the seminar?

Ed: Knew most of the stuff, but it's not a bad idea to hear it again, I suppose.

Jamie: When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

Vinod: Because . . . bottom line . . . it's a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

Shakira: I wasn't an SE grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don't see why this stuff is so important.

Jamie: I've seen your work, Shak, and you know what, you do a lot of this stuff naturally . . . that's why your designs and code work.

Shakira (smiling): Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . that sort of thing.

Ed: Modularity, functional independence, hiding, patterns . . . see.

Jamie: I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

Vinod: Same thing can be applied to design, you know.

Jamie: The only concepts I hadn't heard of before were "design classes" and "refactoring."

Shakira: Refactoring is used in Extreme Programming, I think she said.

Ed: Yep. It's not a whole lot different than refinement, only you do it after the design or code is completed. Kind of like an optimization pass through the software, if you ask me.

Jamie: Let's get back to the *SafeHome* design. I think we should put these concepts on our review checklist as we develop the design model for *SafeHome*.

Vinod: I agree. But as important, let's all commit to think about them as we develop the design.

9.3.10 Design Classes

The analysis model defines a set of analysis classes (Chapter 8). Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented and to create a software infrastructure that supports the business solution.

As the software architecture forms, the level of abstraction is reduced as each analysis class (Chapter 8) is transformed into a design representation. That is, analysis classes represent data objects and associated services that are applied to them. Design classes present significantly more technical detail as a guide for implementation.

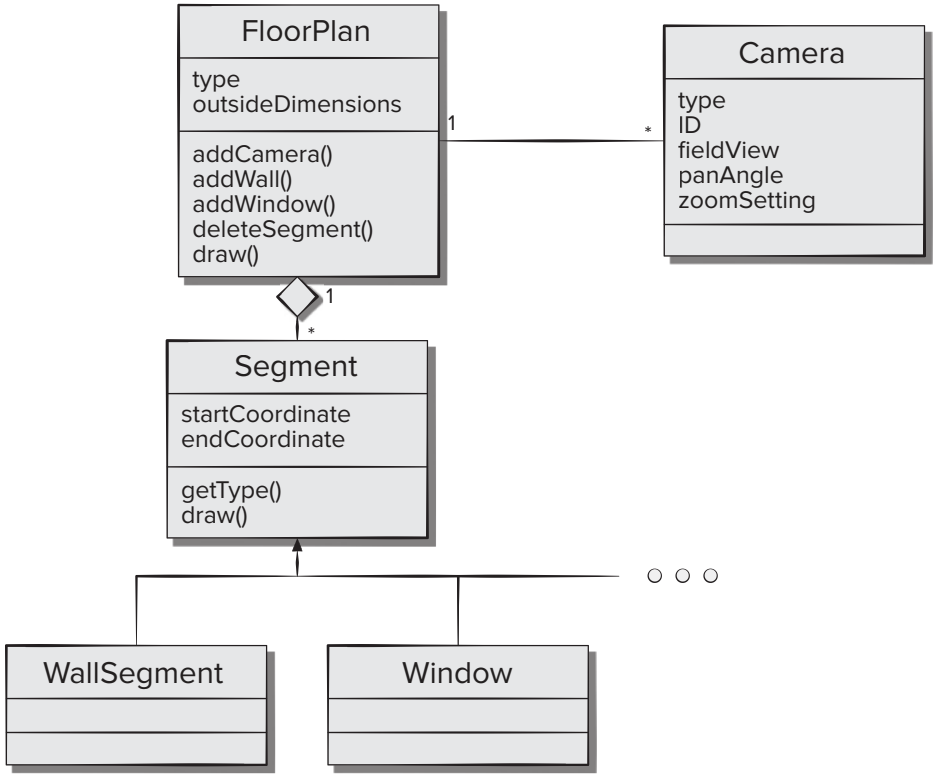
Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **FloorPlan** (Figure 9.3) defined for the *SafeHome* room layout software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a floor plan. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **Segment** (Figure 9.3) for use by the room layout

FIGURE 9.3

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



software might have attributes `startCoordinate` and `endCoordinate` to indicate the start and end points of the segment to be drawn. The method *setCoordinates()* provides the only means for establishing start and end points for the segment.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **FloorPlan** (Figure 9.3) might contain a set of methods for editing the house floor plan. As long as each method focuses solely on attributes associated with the floor plan, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.[8]

8 A less formal way of stating the Law of Demeter is "Each unit should only talk to its friends; don't talk to strangers."

SAFEHOME



Refining an Analysis Class into a Design Class

The scene: Ed's cubicle, as design modeling begins.

The players: Vinod and Ed, members of the *SafeHome* software engineering team.

The conversation:

[Ed is working on the **FloorPlan** class (see sidebar discussion in Section 8.3.3 and Figure 8.4) and has refined it for the design model.]

Ed: So you remember the **FloorPlan** class, right? It's used as part of the surveillance and home management functions.

Vinod (nodding): Yeah, I seem to recall that we used it as part of our CRC discussions for home management.

Ed: We did. Anyway, I'm refining it for design. I want to show how we'll actually implement the **FloorPlan** class. My idea is to implement it as a set of linked lists [a specific data structure]. So . . . I had to refine the analysis class **FloorPlan** (Figure 8.4) and actually, sort of simplify it.

Vinod: The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan,** and obviously, there can be many cameras in the floor plan.

Vinod: Phew, let's see a picture of this new **FloorPlan** design class.

(Ed shows Vinod the drawing shown in Figure 9.3.)

Vinod: Okay, I see what you're trying to do. This allows you to modify the floor plan easily because new items can be added to or deleted from the list—the aggregation—without any problems.

Ed (nodding): Yeah, I think it'll work.

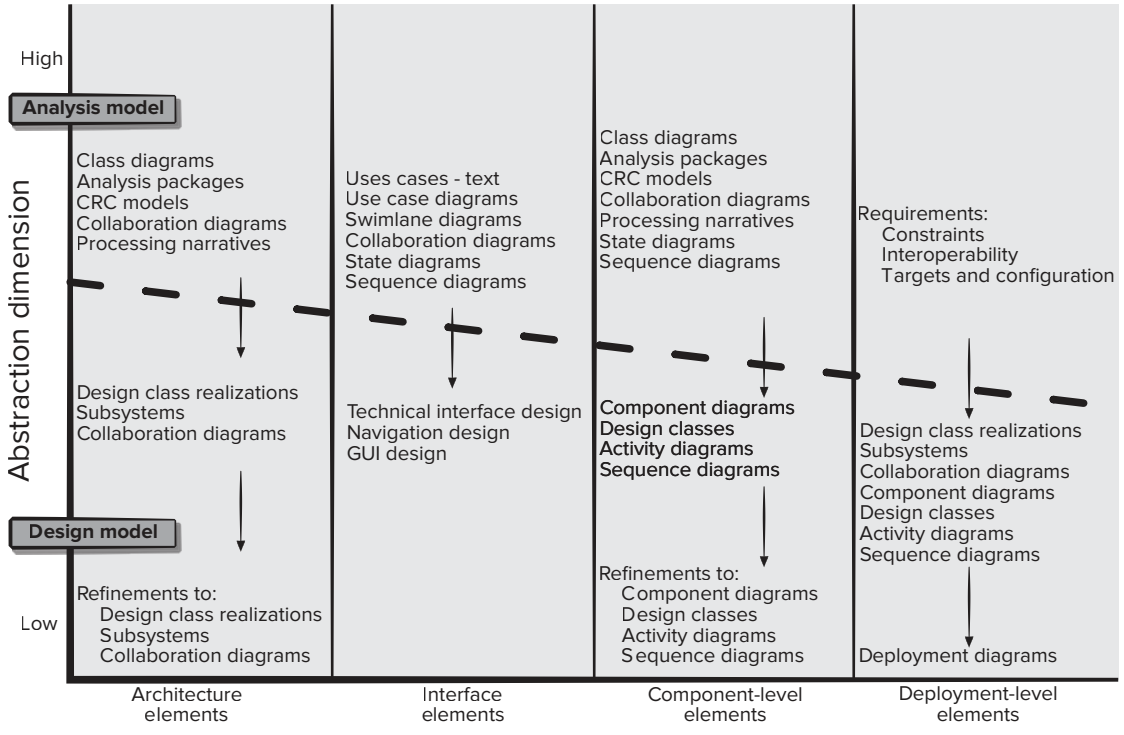
Vinod: So do I.

9.4 THE DESIGN MODEL

The software design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

The design model can be viewed in two different dimensions, as illustrated in Figure 9.4. The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to the figure, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

FIGURE 9.4 Dimensions of the design model



The elements of the design model use many of the same UML diagrams[9] that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases, preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

You can apply design patterns (Chapter 14) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

9 Appendix 1 provides a tutorial on basic UML concepts and notation.

9.4.1 Design Modeling Principles

There is no shortage of methods for deriving the various elements of a software design model. Some methods are data driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern driven, using information about the problem domain (the requirements model) to develop architectural styles and processing patterns. Still others are object oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied, regardless of the method that is used:

Principle 1. ***Design should be traceable to the requirements model.*** The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

Principle 2. ***Always consider the architecture of the system to be built.*** Software architecture (Chapter 10) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. ***Design of data is as important as design of processing functions.*** Data design is an essential element of architectural design. The ways in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4. ***Interfaces (both internal and external) must be designed with care.*** The ways in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. ***User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use.*** The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. ***Component-level design should be functionally independent.*** Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function.

Principle 7. ***Components should be loosely coupled to one another and to the external environment.*** Coupling is achieved in many ways—via a component interface, by messaging, and through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle 8. ***Design representations (models) should be easily understandable.*** The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. ***The design should be developed iteratively.*** With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

Principle 10. ***Creation of a design model does not preclude an agile approach.*** Some proponents of agile software development (Chapter 3) insist that the code is the only design documentation that is needed. Yet the purpose of a design model is to help others who must maintain and evolve the system. It is extremely difficult to understand either the higher-level purpose of a code fragment or its interactions with other modules in a modern multithreaded run-time environment.

Agile design documentation should be kept in sync with the design and development, so that at the end of the project the design is documented at a level that allows the code to be understood and maintained. The design model provides benefit because it is created at a level of abstraction that is stripped of unnecessary technical detail and is closely coupled to the application concepts and requirements. Complementary design information can incorporate a design rationale, including the descriptions of rejected architectural design alternatives.

9.4.2 Data Design Elements

Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer or user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 10.

9.4.3 Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model [Sha15] is derived from three sources: (1) information about the application domain for the software to be built, (2) specific requirements model elements such as use cases or analysis classes, their relationships, and collaborations for the problem at hand, and (3) the availability of architectural styles (Chapter 10) and patterns (Chapter 14).

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 10.

9.4.4 Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. The detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces; to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called *UX* or *user experience design*) is a major software engineering action and is considered in detail in Chapter 12. UX design focuses on ensuring the usability of the UI design. A usable design incorporates carefully chosen aesthetic elements (e.g., layout, color, graphics, information layout), ergonomic elements (e.g., interaction mechanisms, information placement, metaphors, UI navigation), and technical elements (e.g., UX patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture designed to provide the end user with a satisfying user experience.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 7) and verified once the interface design commences.[10] The design of external interfaces should incorporate error checking and appropriate security features.

10 Interface characteristics can change with time. Therefore, a designer should ensure that the specification for the interface is accurate and complete.

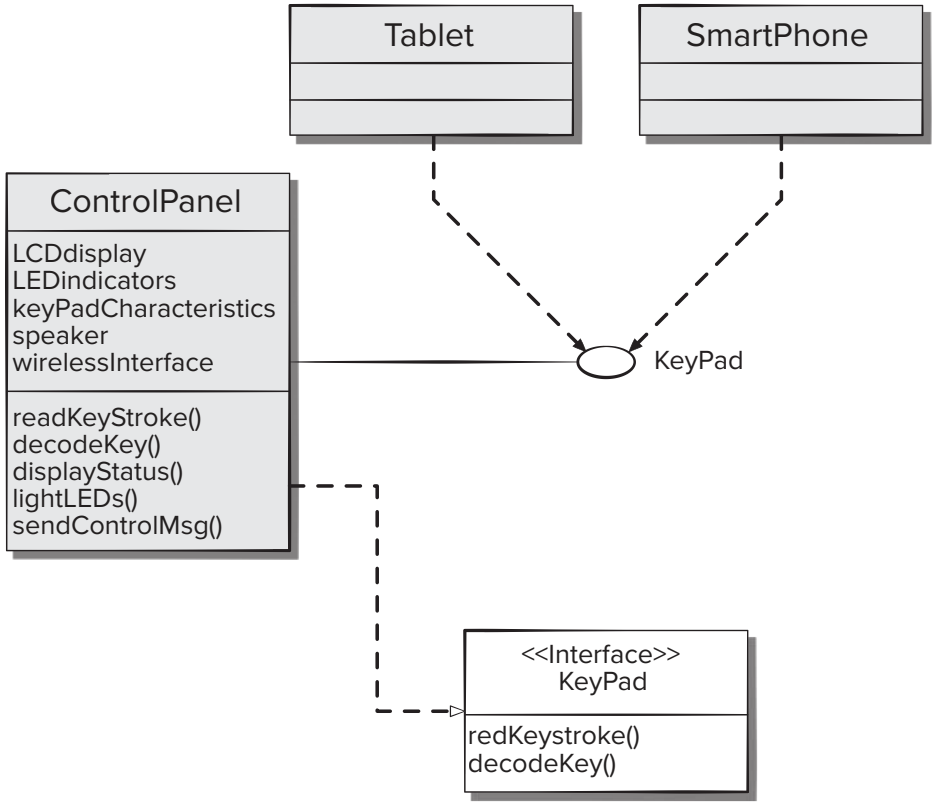


FIGURE 9.5
Interface representation for ControlPanel

The design of internal interfaces is closely aligned with component-level design (Chapter 11). Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested.

In some cases, an interface is modeled in much the same way as a class. An interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

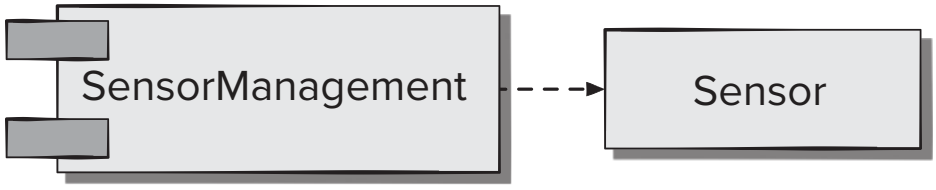
For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain features of the security function. In an advanced version of the system, control panel functions may be implemented via a mobile platform (e.g., smartphone or tablet) and are represented in Figure 9.5.

9.4.5 Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches,

FIGURE 9.6

A UML component diagram



faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form, as shown in Figure 9.6. In this figure, a component named **SensorManagement** (part of the *SafeHome* security function) is represented. A dashed arrow connects the component to a class named **Sensor** that is assigned to it. The **SensorManagement** component performs all functions associated with *SafeHome* sensors including monitoring and configuring them. Further discussion of component design is presented in Chapter 11.

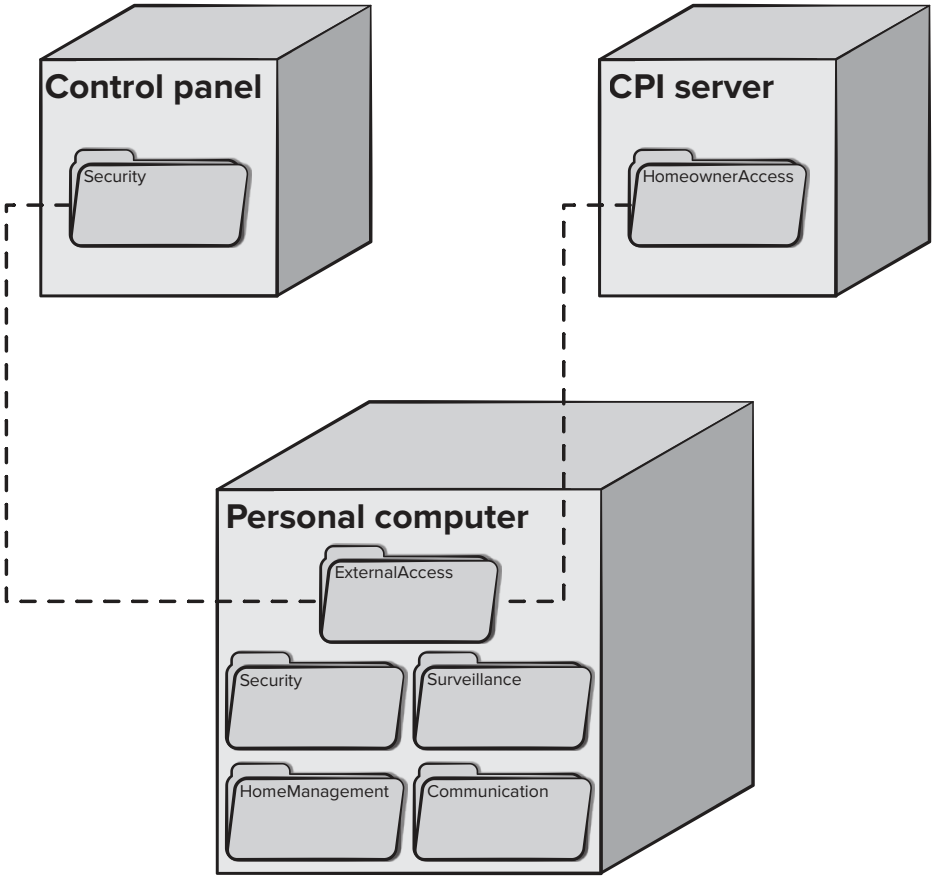
The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Algorithmic structure details for a component can be represented using either pseudocode (a programming languagelike representation described in Chapter 11) or some other diagrammatic form (e.g., flowchart). Data structure details are usually modeled using pseudocode or the programming language to be used for implementation.

9.4.6 Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a mobile device—in this case a PC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system).

During design, a UML deployment diagram is developed and then refined, as shown in Figure 9.7. In the figure, three computing environments are shown (in the full design there would be more details included: sensors, cameras, and the functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

FIGURE 9.7
A UML deployment diagram



The diagram shown in Figure 9.7 is in *descriptor form.* This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac, a Windows-based PC, a Linux box, or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

9.5 SUMMARY

Software design commences as the first iteration of requirements engineering concludes. The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to create a model of software that will implement all customer requirements

correctly and bring delight to those who use it. Software designers must sift through many design alternatives and converge on a solution that best suits the needs of project stakeholders.

The design process moves from a "big picture" view of software to a narrower view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined, communication mechanisms among subsystems are established, components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed at the same time.

Design concepts have evolved over the first 60 years of software engineering work. They describe attributes of computer software that should be present regardless of the software engineering process that is chosen, the design methods that are applied, or the programming languages that are used. In essence, design concepts emphasize the need for abstraction as a mechanism for creating reusable software components; the importance of architecture as a way to better understand the overall structure of a system; the benefits of pattern-based engineering as a technique for designing software with proven capabilities; the value of separation of concerns and effective modularity as a way to make software more understandable, more testable, and more maintainable; the consequences of information hiding as a mechanism for reducing the propagation of side effects when errors do occur; the impact of functional independence as a criterion for building effective modules; the use of refinement as a design mechanism; the application of refactoring for optimizing the design that is derived; the importance of object-oriented classes and the characteristics that are related to them; the need to use abstraction to reduce coupling between components; and the importance of design for testing.

The design model encompasses four different elements. As each of these elements is developed, a more complete view of the design evolves. The architectural element uses information derived from the application domain, the requirements model, and available catalogs for patterns and styles to derive a complete structural representation of the software, its subsystems, and components. Interface design elements model external and internal interfaces and the user interface. Component-level elements define each of the modules (components) that populate the architecture. Finally, deployment-level design elements allocate the architecture, its components, and the interfaces to the physical configuration that will house the software.

PROBLEMS AND POINTS TO PONDER

9.1. Do you design software when you "write" a program? What makes software design different from coding?

9.2. If a software design is not a program (and it isn't), then what is it?

9.3. How do we assess the quality of a software design?

9.4. Describe software architecture in your own words.

9.5. Describe separation of concerns in your own words. Is there a case when a "divide and conquer" strategy may not be appropriate? How might such a case affect the argument for modularity?

9.6. Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.

9.7. How are the concepts of coupling and software portability related? Provide examples to support your discussion.

9.8. Apply a "stepwise refinement approach" to develop three different levels of procedural abstractions for one or more of the following programs: (1) Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check. (2) Iteratively solve for the roots of a transcendental equation. (3) Develop a simple task-scheduling algorithm for an operating system.

9.9. Does *refactoring* mean that you modify the entire design iteratively? If not, what does it mean?

9.10. Briefly describe each of the four elements of the design model.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

10

ARCHITECTURAL DESIGN—A RECOMMENDED APPROACH

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. As we noted in Chapter 9, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The decisions made while considering the data, functional, and behavioral domains serve as guides for the creation of the software architectural design.

KEY CONCEPTS

agility and architecture 185
archetypes 196
architectural considerations 193
architectural decisions 195
architectural description language 184
architectural descriptions 184
architectural design 196
architectural patterns 187
architectural styles 186
architecture 182
architecture conformance checking 204
architecture trade-off analysis method (ATAM) 201
layered architectures 189
refining the architecture 198
taxonomy of architectural styles 187



QUICK LOOK

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated, using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman [Mic09] suggest that software architecture identifies a system's "structural elements and their interfaces," along with the "behavior" of individual components and subsystems. They write that the job of architectural design is to create "coherent, well-planned representations" of the system and software.

Methods to create such representations of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

10.1 SOFTWARE ARCHITECTURE

In their landmark book on the subject, Shaw and Garlan [Sha15] argue that since the earliest days of computer programming, "software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage." Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

10.1.1 What Is Architecture?

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings; the list is almost endless. And finally, it is art.

Architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

Just like the plans for a house are merely a representation of the building, the software architecture representation is not an operational product. Rather, it is a representation that enables you to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of

a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

We believe that a software design can be thought of as an instance of some software architecture. However, the elements and structures that are defined as parts of particular software architectures are the root of every design. It is our recommendation that design should begin with a consideration of the software architecture.

10.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas12] identify three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- The architecture constitutes a relatively small model of how the system components are structured and work together.

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 10.3 through 10.6) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

Making good decisions while defining the software architecture is critical to the success of a software product. The software architecture sets the structure of the system and determines its quality [Das15].

10.1.3 Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. The implication is that different stakeholders will see a given software architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Smolander, Rossi, and Purao [Smo08] have identified multiple metaphors, representing different views of the same architecture that stakeholders use to understand the term *software architecture*. The *blueprint metaphor* seems to be most familiar to the stakeholders who write programs to implement a system. Developers regard architecture descriptions as a means of transferring explicit information from architects to designers to software engineers charged with producing the system components.

The *language metaphor* views architecture as a facilitator of communication across stakeholder groups. This view is preferred by stakeholders with a high customer focus (e.g., managers or marketing experts). The architectural description needs to be concise and easy to understand because it forms the basis for negotiation, particularly in determining system boundaries.

The *decision metaphor* represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance that have resource consequences for the system being designed. Stakeholders such as project managers view architectural decisions as the basis for allocating project resources and tasks. These decisions may affect the sequence of tasks and shape the structure of the software team.

The *literature metaphor* is used to document architectural solutions constructed in the past. This view supports the construction of artifacts and the transfer of knowledge between designers and software maintenance staff. It also supports stakeholders whose concern is reuse of components and designs.

An *architectural description* (AD) represents a system using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns." The IEEE Computer Society standard IEEE-Std-42010:2011(E), *Systems and software engineering—Architectural description* [IEE11], describes the use of architecture viewpoints, architecture frameworks, and architecture description languages as a means of codifying the conventions and common practices for architectural description.

10.1.4 Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole), the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish a historical record that can be useful when design modifications must be made. For agile developers, a lightweight *architectural decision record* (ADR) might simply contain a title, a context (assumptions and constraints), the decision (resolution), status (proposed accepted rejected), and consequences (implications) [Nyg11].

Grady Booch [Boo11a] writes that when setting out to build an innovative product, software engineers often feel compelled to plunge right in, build stuff, fix what doesn't work, improve what does work, and then repeat the process. After doing this a few times, they begin to recognize that the architecture should be defined first and decisions associated with architectural choices must be stated explicitly. It may not be possible to predict the right choices before building a new product. However, if innovators find that architectural decisions are worth repeating after testing their prototypes in the field, then a *dominant design*[1] for this type of product may begin to emerge. Without documenting what worked and what did not, it is hard for software engineers to decide when to innovate and when to use previously created architecture.

1 *Dominant design* describes an innovative software architecture or process that becomes an industry standard after a period of successful adaptation and use in the marketplace.

INFO



Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue: Describe the architectural design issues that are to be addressed.

Resolution: State the approach you've chosen to address the design issue.

Category: Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

Assumptions: Indicate any assumptions that helped shape the decision.

Constraints: Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

Alternatives: Briefly describe the architectural design alternatives that were considered and why they were rejected.

Argument: State why you chose the resolution over other alternatives.

Implications: Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

Related decisions: What other documented decisions are related to this decision?

Related concerns: What other requirements are related to this decision?

Work products: Indicate where this decision will be reflected in the architecture description.

Notes: Reference any team notes or other documentation that was used to make the decision.

10.2 AGILITY AND ARCHITECTURE

The view of some agile developers is that architectural design is equated with "big design upfront." In their view, this leads to unnecessary documentation and the implementation of unnecessary features. However, most agile developers would agree [Fal10] that it is important to focus on software architecture when a system is complex (i.e., when a product has a large number of requirements, lots of stakeholders, or a large number of global users). For this reason, it is important to integrate new architectural design practices into agile process models.

To make early architectural decisions and avoid the rework required to correct the quality problems encountered when the wrong architecture is chosen, agile developers

need to anticipate architectural elements[2] and implied structure that emerges from the collection of user stories gathered (Chapter 7). By creating an architectural prototype (e.g., a *walking skeleton*) and developing explicit architectural work products to communicate the right information to the necessary stakeholders, an agile team can satisfy the need for architectural design.

Using a technique called *storyboarding*, the architect contributes architectural user stories to the project and works with the product owner to prioritize the architectural stories with the business user stories as "sprints" (work units) are planned. The architect works with the team during the sprint to ensure that the evolving software continues to show high architectural quality as defined by the nonfunctional product requirements. If quality is high, the team is left alone to continue development on its own. If not, the architect joins the team for the duration of the sprint. After the sprint is completed, the architect reviews the working prototype for quality before the team presents it to the stakeholders in a formal sprint review. Well-run agile projects make use of iterative work product delivery (including architectural documentation) with each sprint. Reviewing the work products and code as it emerges from each sprint is a useful form of architectural review.

Responsibility-driven architecture (RDA) is a process that focuses on when, how, and who should make the architectural decisions on a project team. This approach also emphasizes the role of architect as being a servant-leader rather than an autocratic decision maker and is consistent with the agile philosophy. The architect acts as facilitator and focuses on how the development team works to accommodate stakeholder's nontechnical concerns (e.g., business, security, usability).

Agile teams usually have the freedom to make system changes as new requirements emerge. Architects want to make sure that the important parts of the architecture were carefully considered and that developers have consulted the appropriate stakeholders. Both concerns may be satisfied by making use of a practice called *progressive sign-off* in which the evolving product is documented and approved as each successive prototype is completed [Bla10].

Using a process that is compatible with the agile philosophy provides verifiable sign-off for regulators and auditors, without preventing the empowered agile teams from making the decisions needed. At the end of the project, the team has a complete set of work products and the architecture has been reviewed for quality as it evolved.

10.3 Architectural Styles

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important,

2 An excellent discussion of architectural agility can be found in [Bro10a].

the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [Bas12].

An *architectural style* is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be refactored (Chapter 27), the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components [Bos00].

An *architectural pattern*, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety, (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00], and (3) architectural patterns (Section 10.3.2) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

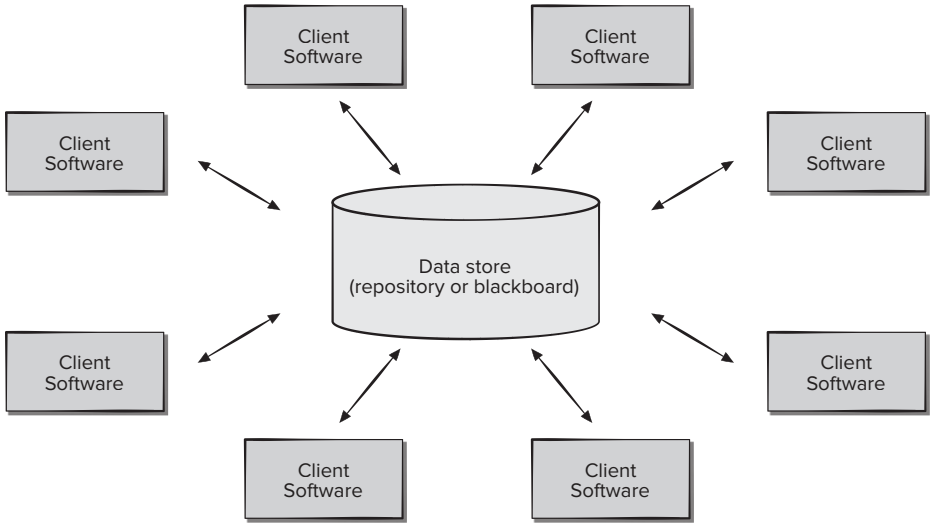
10.3.1 A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles.

Data-Centered Architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 10.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases, the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability* [Bas12]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

FIGURE 10.1
Data-centered architecture



Data-Flow Architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 10.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

FIGURE 10.2 **Data-flow architecture**

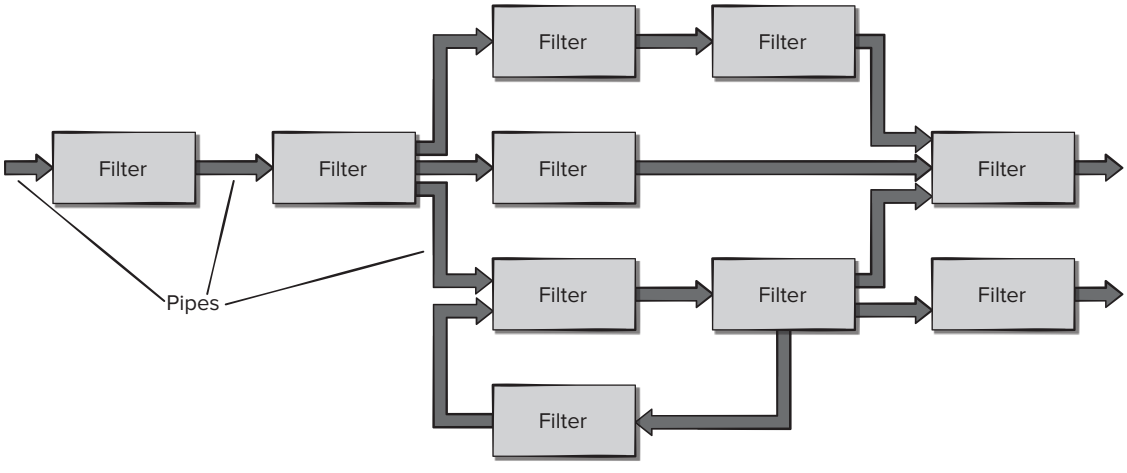
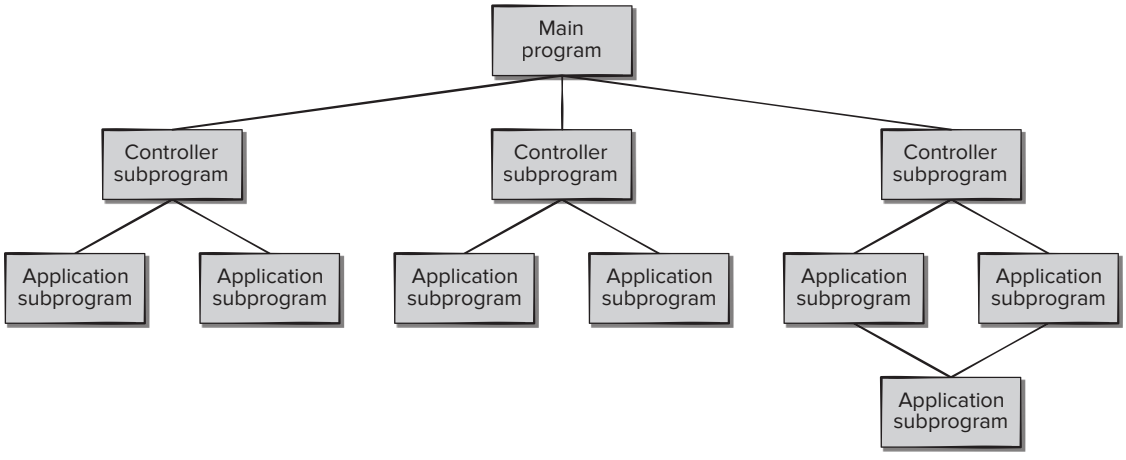


FIGURE 10.3 Main program/subprogram architecture



Call-and-Return Architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. Two substyles [Bas12] that exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes several program components, which in turn may invoke still other components. Figure 10.3 illustrates an architecture of this type.
- *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Object-Oriented Architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing. Figure 10.4 contains a UML communication diagram that shows the message passing for the login portion of a system implemented using an object-oriented architecture. Communications diagrams are described in more details in Appendix 1 of this book.

Layered Architectures. The basic structure of a layered architecture is illustrated in Figure 10.5. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

Model-View-Controller (MVC) architecture [Kra88] is one of a number of suggested mobile infrastructure models often used in Web development. The *model* contains all application-specific content and processing logic. The *view* contains all

FIGURE 10.4

UML communication diagram

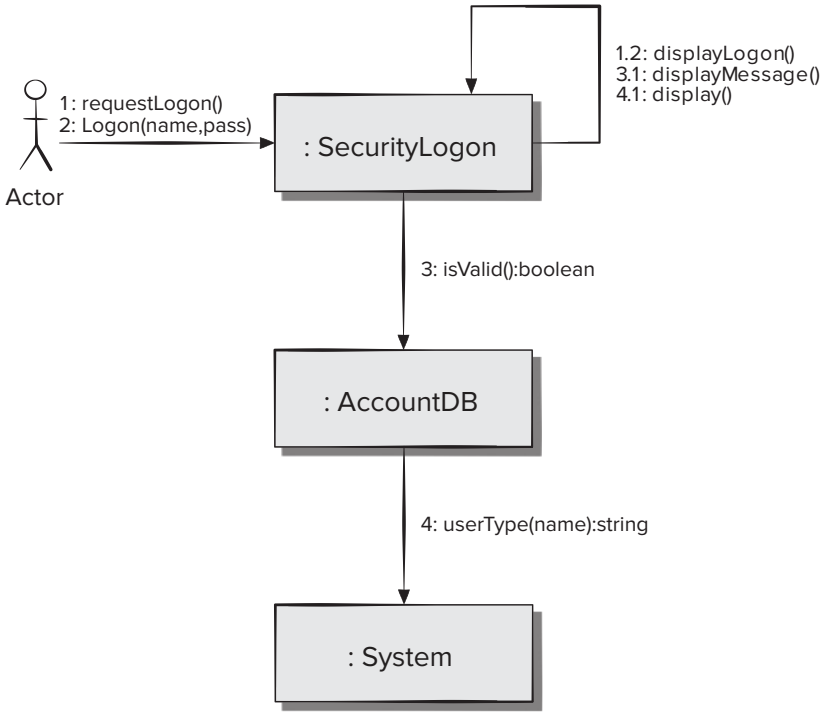


FIGURE 10.5

Layered architecture

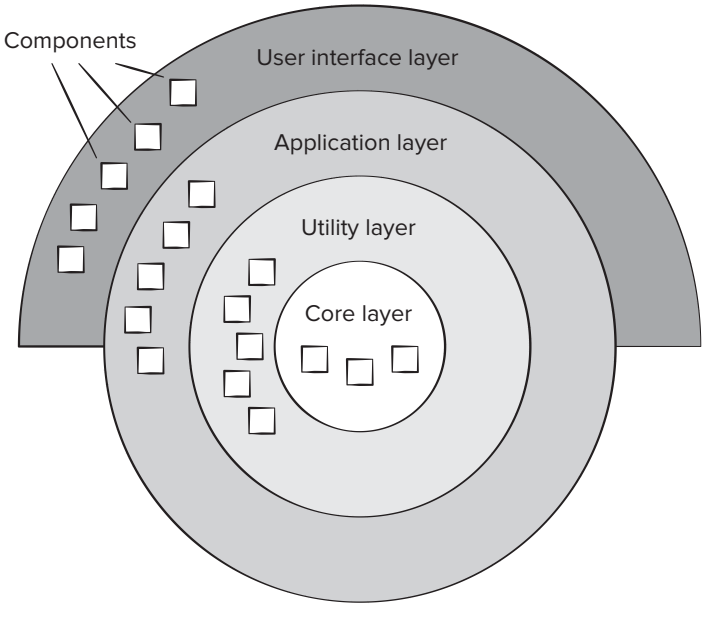
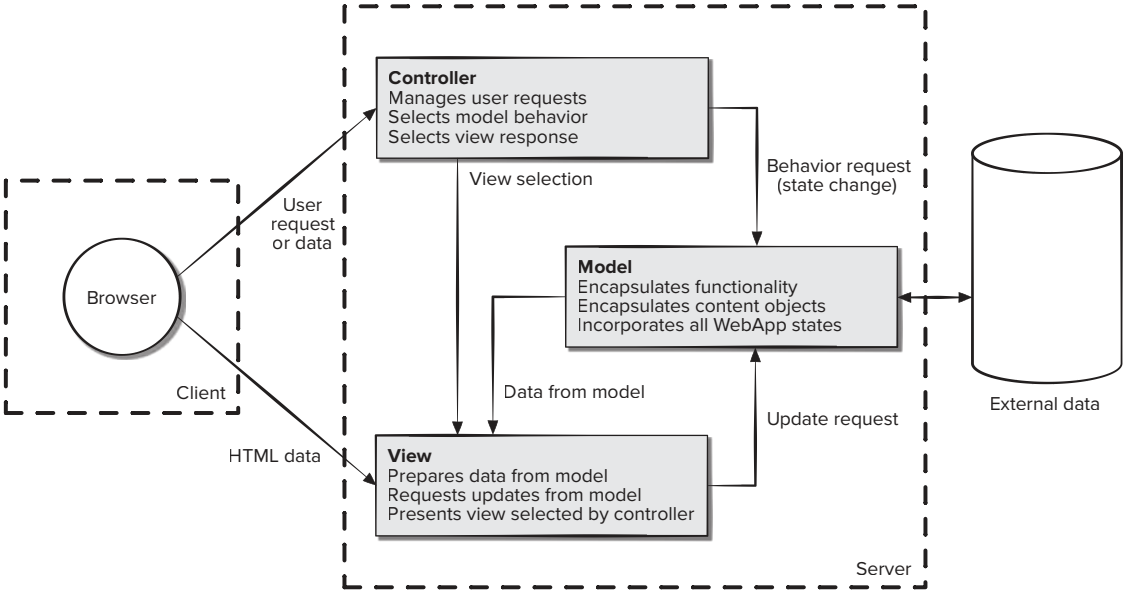


FIGURE 10.6 The MVC architecture



Source: Adapted from Jacyntho, Mark Douglas, Schwabe, Daniel and Rossi, Gustavo, "An Architecture for Structuring Complex Web Applications," 2002, available at http://www-di.inf.puc-rio.br/schwabe/papers/OOHDMJava2%20Report.pdf

interface-specific functions and enables the presentation of content and processing logic required by the end user. The *controller* manages access to the model and the view and coordinates the flow of data between them. A schematic representation of the MVC architecture is shown in Figure 10.6.

Referring to the figure, user requests are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine.

These architectural styles are only a small subset of those available.[3] Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

3 See [Roz11], [Tay09], [Bus07], [Gor06], or [Bas12] for a detailed discussion of architectural styles and patterns.

SAFEHOME



Choosing an Architectural Style

The scene: Jamie's cubicle, as design modeling begins.

The players: Jamie and Ed—members of the *SafeHome* software engineering team.

The conversation:

Ed (frowning): We've been modeling the security function using UML . . . you know, classes, relationships, that sort of stuff. So I guess the object-oriented architecture is the right way to go.

Jamie: But . . .?

Ed: But . . . I have trouble visualizing what an object-oriented architecture is. I get the call-and-return architecture, sort of a conventional process hierarchy, but OO . . . I don't know, it seems sort of amorphous.

Jamie (smiling): Amorphous, huh?

Ed: Yeah . . . what I mean is I can't visualize a real structure, just design classes floating in space.

Jamie: Well, that's not true. There are class hierarchies . . . think of the hierarchy (aggregation) we did for the **FloorPlan** object [Figure 9.3]. An OO architecture is a combination of that structure and the interconnections—you know, collaborations—between the classes. We can show it by fully describing the attributes and operations, the messaging that goes on, and the structure of the classes.

Ed: I'm going to spend an hour mapping out a call-and-return architecture; then I'll go back and consider an OO architecture.

Jamie: Doug'll have no problem with that. He said that we should consider architectural alternatives. By the way, there's absolutely no reason why both of these architectures couldn't be used in combination with one another.

Ed: Good. I'm on it.

Choosing the right architecture style can be tricky. Real-world problems often follow more than one problem frame, and a combination architectural model may result. For example, the model-view-controller (MVC) architecture used in WebApp design[4] might be viewed as combining two problem frames (command behavior and information display). In MVC, the end user's command is sent from the browser window to a command processor (controller) that manages access to the content (model) and instructs the information rendering model (view) to translate it for display by the browser software.

10.3.2 Architectural Patterns

As the requirements model is developed, you'll notice that the software must address several broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to many different customers and allow those customers to find and purchase our goods easily?*

The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way you address the problem to be solved.

4 The MVC architecture is considered in more detail in Chapter 13.

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Previously in this chapter, we noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call and return or object oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns are presented in Chapter 14.

10.3.3 Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas12] provide insight into an architectural style:

> **Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized, or do components operate asynchronously?
>
> **Data.** How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

The answers to these questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

Evolutionary process models (Chapter 2) have become very popular. This implies the software architectures may need to evolve as each product increment is planned and implemented. In Chapter 9, we described this process as refactoring—improving the internal structure of the system without changing its external behavior.

10.4 ARCHITECTURAL CONSIDERATIONS

Buschmann and Henny [Bus10a, Bus10b] suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made.

- **Economy.** The best software is uncluttered and relies on abstraction to reduce unnecessary detail. It avoids complexity due to unnecessary functions and features.

- **Visibility.** As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model later. Important design and domain concepts must be communicated effectively.
- **Spacing.** Separation of concerns (Chapter 9) in a design is sometimes referred to as *spacing*. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- **Symmetry.** Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural symmetry, consider a **customer account** object whose life cycle is modeled directly by a software architecture that requires both *open()* and *close()* methods. Architectural symmetry can be both structural and behavioral.
- **Emergence.** Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of these events that define the system's behavior is an emergent quality. Because it is very difficult to plan for every possible sequence of events, a system architect should create a flexible system that accommodates this emergent behavior.

These considerations do not exist in isolation. They interact with each other and are moderated by each other. For example, spacing can be both reinforced and reduced by economy. Visibility can be balanced by spacing.

The architectural description for a software product is not explicitly visible in the source code used to implement it. As a consequence, code modifications made over time (e.g., software maintenance activities) can cause slow erosion of the software architecture. The challenge for a designer is to find suitable abstractions for the architectural information. These abstractions have the potential to add structuring that improves readability and maintainability of the source code [Bro10b].

SAFEHOME



Evaluating Architectural Decisions

The scene: Jamie's cubicle, as design modeling continues.

The players: Jamie and Ed, members of the *SafeHome* software engineering team.

The conversation:

Ed: I finished my call-return architectural model of the security function.

Jamie: Great! Do you think it meets our needs?

Ed: It doesn't introduce any unneeded features, so it seems to be economic.

Jamie: How about visibility?

Ed: Well, I understand the model, and there's no problem implementing the security requirements needed for this product.

Jamie: I get that you understand the architecture, but you may not be the programmer for this part of the project. I'm a little worried

about spacing. This design may not be as modular as an object-oriented design.

Ed: Maybe, but that may limit our ability to reuse some of our code when we have to create the mobile version of *SafeHome*.

Jamie: What about symmetry?

Ed: Well, that's harder for me to assess. It seems to me the only place for symmetry in the security function is adding and deleting PIN information.

Jamie: That will get more complicated when we add remote security features to the mobile app.

Ed: That's true, I guess.

(They both pause for a moment, pondering the architectural issues.)

Jamie: *SafeHome* is a real-time system, so state transition and sequencing of events will be tough to predict.

Ed: Yeah, but the emergent behavior of this system can be handled with a finite state model.

Jamie: How?

Ed: The model can be implemented based on the call-return architecture. Interrupts can be handled easily in many programming languages.

Jamie: Do you think we need to do the same kind of analysis for the object-oriented architecture we were initially considering?

Ed: I suppose it might be a good idea, because architecture is hard to change once implementation starts.

Jamie: It's also important for us to map the nonfunctional requirements besides security on top of these architectures to be sure they have been considered thoroughly.

Ed: Also, true.

10.5 ARCHITECTURAL DECISIONS

Decisions about system architecture identify key design issues and the rationale behind chosen architectural solutions. System architecture decisions encompass software system organization, selection of structural elements and their interfaces as defined by their intended collaborations, and the composition of these elements into increasingly larger subsystems [Kru09]. In addition, choices of architectural patterns, application technologies, middleware assets, and programming language can also be made. The outcome of the architectural decisions influences the system's nonfunctional characteristics and many of its quality attributes [Zim11] and can be documented with *developer notes*. These notes document key design decisions along with their justification, provide a reference for new project team members, and serve as a repository for lessons learned.

In general, software architectural practice focuses on architectural views that represent and document the needs of various stakeholders. It is possible, however, to define a *decision view* that cuts across several views of information contained in traditional architectural representations. The decision view captures both the architecture design decisions and their rationale.

Service-oriented architecture decision (SOAD)[5] modeling [Zim11] is a knowledge management framework that provides support for capturing architectural decision dependencies in a manner that allows them to guide future development activities.

5 SOAD is analogous to the use of architecture patterns discussed in Chapter 14.

A SOAD *guidance model* contains knowledge about architectural decisions required when applying an architectural style in a particular application genre. It is based on architectural information obtained from completed projects that employed the architectural style in that genre. The guidance model documents places where design problems exist and architectural decisions must be made, along with quality attributes that should be considered in selecting from among potential alternatives. Potential alternative solutions (with their pros and cons) from previous software applications are included to assist the architect in making the best decision possible.

A SOAD *decision model* documents both the architectural decisions required and records the decisions actually made on previous projects with their justifications. The guidance model feeds the architectural decision model in a *tailoring* step that allows the architect to delete irrelevant issues, enhance important issues, or add new issues. A decision model can make use of more than one guidance model and provides feedback to the guidance model after the project is completed. This feedback may be accomplished by *harvesting* lessons learned from project postmortem reviews.

10.6 ARCHITECTURAL DESIGN

As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

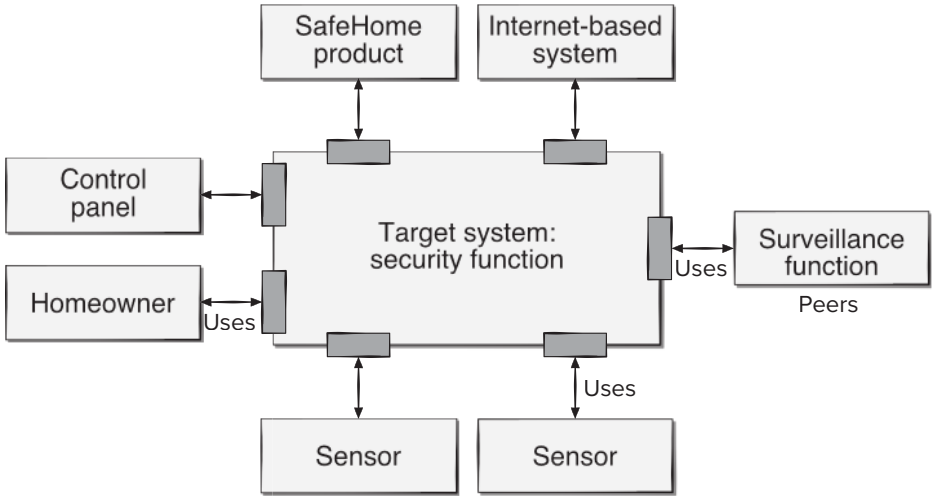
Several questions [Boo11b] must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? What visualizations might there be to help emphasize areas of risk? How can hidden system design patterns be made more obvious to other developers? Can multiple viewpoints show the best way to refactor specific parts of the system? Can design trade-offs be represented in a meaningful way? If a diagrammatic representation of software architecture answers these questions, it will have value to software engineers that use it.

10.6.1 Representing the System in Context

UML does not contain specific diagrams that represent the system in context. Software engineers wishing to stick with UML and represent the system in context would do so with a combination of use case, class, component, activity, sequence, and collaboration diagrams. Some software architects may make use of an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. An architectural context diagram for the *SafeHome* security functions is shown in Figure 10.7.

FIGURE 10.7

Architectural context diagram for the *SafeHome* security function



To illustrate the use of the ACD, consider the home security function of the *SafeHome* product shown in Figure 10.7. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the function. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that produce and consume information that is, respectively, used and produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it (by drawing them below the target system).

As part of the architectural design, the details of each interface shown in Figure 10.7 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

10.6.2 Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

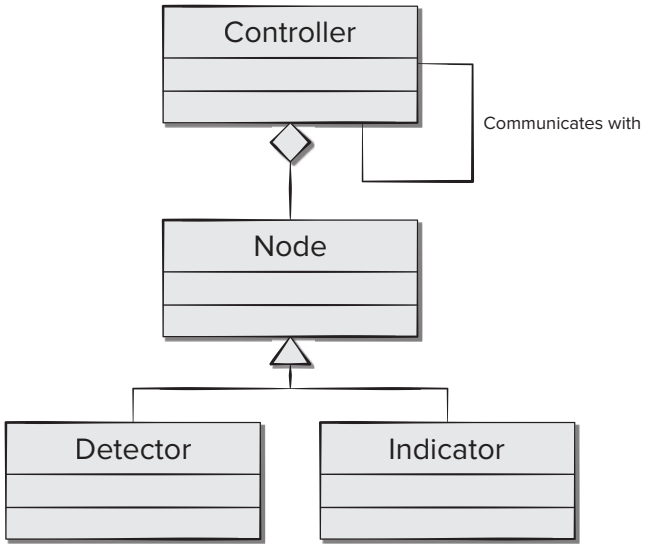
Each of these archetypes is depicted using UML notation, as shown in Figure 10.8. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

10.6.3 Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? To answer this question, you begin with the classes that were described as part of the requirements model.[6] These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

FIGURE 10.8

UML relationships for *SafeHome* security function archetype
Source: Adapted from Bosch, Jan, Design & Use of Software Architectures. Pearson Education, 2000.



6 If a conventional (non-object-oriented) approach is chosen, components may be derived from the subprogram calling hierarchy (see Figure 10.3).

The interfaces depicted in the architecture context diagram (Section 10.6.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

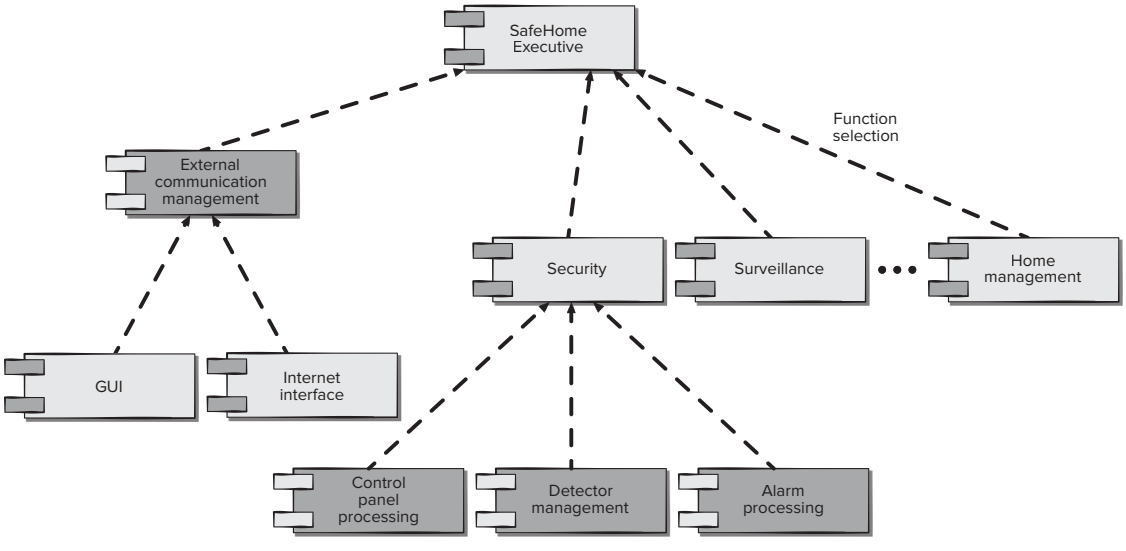
Continuing the *SafeHome* home security function example, you might define the set of top-level components that addresses the following functionality:

- ***External communication management.*** Coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- ***Control panel processing.*** Manages all control panel functionality.
- ***Detector management.*** Coordinates access to all detectors attached to the system.
- ***Alarm processing.*** Verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design (Chapter 11).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 10.9. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm and disarm the security

FIGURE 10.9 **Overall architectural structure for *SafeHome* with top-level components**



function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

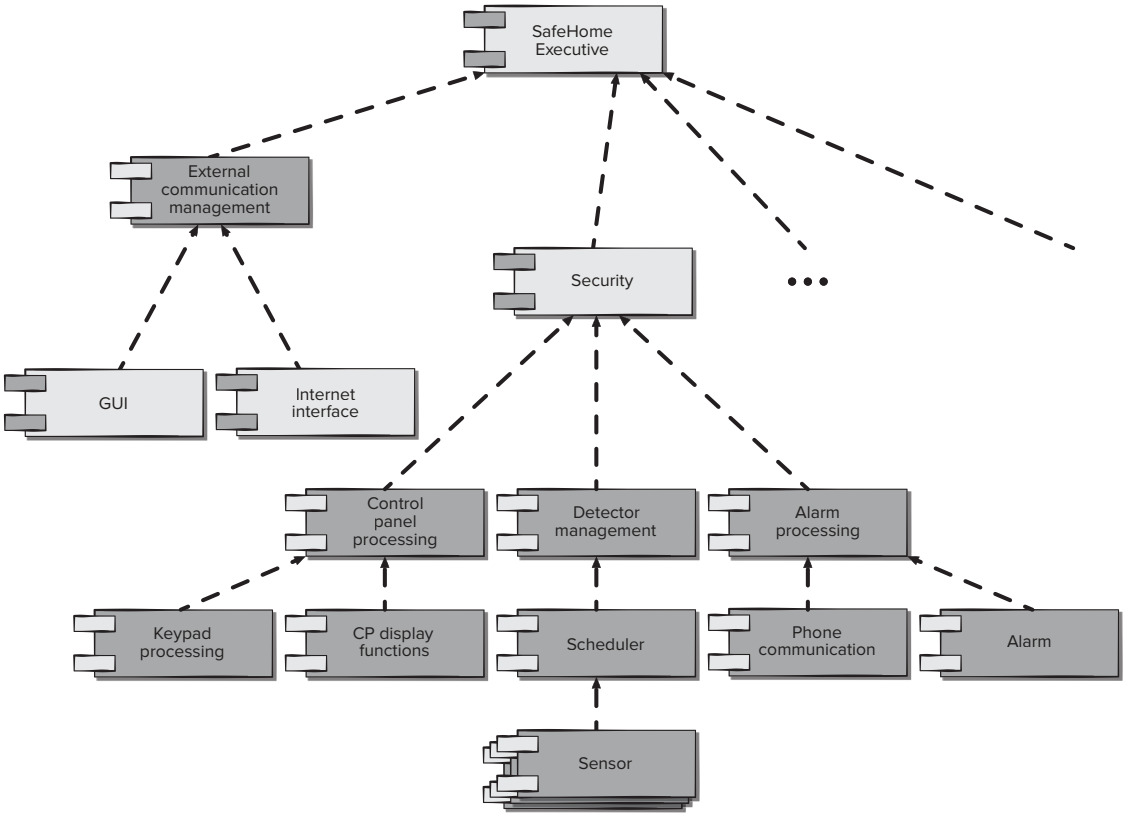
10.6.4 Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure 10.10 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 10.9 are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 10.10.

FIGURE 10.10 **An instantiation of the security function with component elaboration**



10.7 ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state, "To put it bluntly, an architecture is a bet, a wager on the success of a system."

The big question for a software architect and the software engineers who will work to build a system is simple: Will the architectural bet pay off?

To help answer this question, architectural design should result in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved.

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [Kaz98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. **Collect scenarios.** A set of use cases (Chapters 7 and 8) is developed to represent the system from the user's point of view.
2. **Elicit requirements, constraints, and environment description.** This information is required as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. **Describe the architectural styles and patterns that have been chosen to address the scenarios and requirements.** The architectural style(s) should be described using one of the following architectural views:
   - *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.
   - *Process view* for analysis of system performance.
   - *Data flow view* for analysis of the degree to which the architecture meets functional requirements.
4. **Evaluate quality attributes by considering each attribute in isolation.** The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. **Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.** This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. **Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.** The SEI describes this approach in the following manner [Kaz98]:

   Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might

> be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.

SAFEHOME



Architecture Assessment

The scene: Doug Miller's office as architectural design modeling proceeds.

The players: Vinod, Jamie, and Ed, members of the *SafeHome* software engineering team. Also Doug Miller, manager of the software engineering group.

The conversation:

Doug: I know you guys are deriving a couple of different architectures for the *SafeHome* product, and that's a good thing. I guess my question is, how are we going to choose the one that's best?

Ed: I'm working on a call-and-return style, and then either Jamie or I will derive an OO architecture.

Doug: Okay, and how do we choose?

Jamie: I took a CS course in design in my senior year, and I remember that there are a number of ways to do it.

Vinod: There are, but they're a bit academic. Look, I think we can do our assessment and choose the right one using use cases and scenarios.

Doug: Isn't that the same thing?

Vinod: Not when you're talking about architectural assessment. We already have a complete set of use cases. So we apply each to both architectures and see how the system reacts, how components and connectors work in the use case context.

Ed: That's a good idea. Make sure we didn't leave anything out.

Vinod: True, but it also tells us whether the architectural design is convoluted, whether the system has to twist itself into a pretzel to get the job done.

Jamie: Aren't scenarios just another name for use cases?

Vinod: No, in this case a scenario implies something different.

Doug: You're talking about a quality scenario or a change scenario, right?

Vinod: Yes. What we do is go back to the stakeholders and ask them how *SafeHome* is likely to change over the next, say, 3 years. You know, new versions, features, that sort of thing. We build a set of change scenarios. We also develop a set of quality scenarios that defines the attributes we'd like to see in the software architecture.

Jamie: And we apply them to the alternatives.

Vinod: Exactly. The style that handles the use cases and scenarios best is the one we choose.

10.7.1 Architectural Reviews

Architectural reviews are a type of specialized technical review (Chapter 16) that provide a means of assessing the ability of a software architecture to meet the system's quality requirements (e.g., scalability or performance) and to identify any potential

risks. Architectural reviews have the potential to reduce project costs by detecting design problems early.

Unlike requirements reviews that involve representatives of all stakeholders, architecture reviews often involve only software engineering team members supplemented by independent experts. However, software-based systems are built by people with a variety of different needs and points of view. Architects often focus on the long-term impact of the system's nonfunctional requirements as the architecture is created. Senior managers assess the architecture within the context of business goals and objectives. Project managers are often driven by short-term considerations of delivery dates and budget. Software engineers are often focused on their own technology interests and feature delivery. Each of these (and other) constituencies must agree that the software architecture chosen has distinct advantages over any other alternatives. Therefore, a wise software architect should build consensus among members of the software team (and other stakeholders) to achieve the architectural vision for the final software product [Wri11].

The most common architectural review techniques used in industry are: experience-based reasoning, prototype evaluation, scenario review (Chapter 8), and use of checklists. Many architectural reviews occur early in the project life cycle; they should also occur after new components or packages are acquired in component-based design (Chapter 11). One of the most commonly cited problems facing software engineers when conducting architectural reviews is missing or inadequate architectural work products, thereby making review difficult to complete [Bab09].

10.7.2 Pattern-Based Architecture Review

Formal technical reviews (Chapter 16) can be applied to software architecture and provide a means for managing system quality attributes, uncovering errors, and avoiding unnecessary rework. However, in situations in which short build cycles, tight deadlines, volatile requirements, and/or small teams are the norm, a lightweight architectural review process known as *pattern-based architecture review* (PBAR) might be the best option.

PBAR is an evaluation method based on architectural patterns[7] that leverages the patterns' relationships to quality attributes. A PBAR is a face-to-face audit meeting involving all developers and other interested stakeholders. An external reviewer with expertise in architecture, architecture patterns, quality attributes, and the application domain is also in attendance. The system architect is the primary presenter.

A PBAR should be scheduled after the first working prototype or *walking skeleton*[8] is completed. The PBAR encompasses the following iterative steps [Har11]:

1. Identify and discuss the quality attributes most important to the system by walking through the relevant use cases (Chapter 8).
2. Discuss a diagram of the system's architecture in relation to its requirements.

---

7 An *architectural pattern* is a generalized solution to an architectural design problem with a specific set of conditions or constraints. Patterns are discussed in detail in Chapter 14.

8 A *walking skeleton* contains a baseline architecture that supports the functional requirements with the highest priorities in the business case and the most challenging quality attributes.

3. Help the reviewer identify the architecture patterns used and match the system's structure to the patterns' structure.
4. Using existing documentation and past use cases, examine the architecture and quality attributes to determine each pattern's effect on the system's quality attributes.
5. Identify and discuss all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of the issues uncovered during the meeting, and make appropriate revisions to the walking skeleton.

PBARs are well suited to small, agile teams and require a relatively small amount of extra project time and effort. With its short preparation and review time, PBAR can accommodate changing requirements and short build cycles and, at the same time, help improve the team's understanding of the system architecture.

10.7.3 Architecture Conformance Checking

As the software process moves through design and into construction, software engineers must work to ensure that an implemented and evolving system conforms to its planned architecture. Many things (e.g., conflicting requirements, technical difficulties, deadline pressures) cause deviations from a defined architecture. If architecture is not checked for conformance periodically, uncontrolled deviations can cause *architecture erosion* and affect the quality of the system [Pas10].

Static architecture-conformance analysis (SACA) assesses whether an implemented software system is consistent with its architectural model. The formalism (e.g., UML) used to model the system architecture presents the static organization of system components and how the components interact. Often the architectural model is used by a project manager to plan and allocate work tasks, as well as to assess implementation progress.

10.8 SUMMARY

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

Architectural design can coexist with agile methods by applying a hybrid architectural design framework that makes use of existing techniques derived from popular agile methods. Once an architecture is developed, it can be assessed to ensure conformance with business goals, software requirements, and quality attributes.

Several different architectural styles and patterns are available to the software engineer and may be applied within a given architectural genre. Each style describes a system category that encompasses a set of components that perform a function required by a system; a set of connectors that enable communication, coordination, and cooperation among components; constraints that define how components can be integrated

to form the system; and semantic models that enable a designer to understand the overall properties of a system.

In a general sense, architectural design is accomplished using four distinct steps. First, the system must be represented in context. That is, the designer should define the external entities that the software interacts with and the nature of the interaction. Once context has been specified, the designer should identify a set of top-level abstractions, called archetypes, that represent pivotal elements of the system's behavior or function. After abstractions have been defined, the design begins to move closer to the implementation domain. Components are identified and represented within the context of an architecture that supports them. Finally, specific instantiations of the architecture are developed to "prove" the design in a real-world context.

PROBLEMS AND POINTS TO PONDER

10.1. Using the architecture of a house or building as a metaphor, draw comparisons with software architecture. How are the disciplines of classical architecture and the software architecture similar? How do they differ?

10.2. Present two or three examples of applications for each of the architectural styles noted in Section 10.3.1.

10.3. Some of the architectural styles noted in Section 10.3.1 are hierarchical in nature, and others are not. Make a list of each type. How would the architectural styles that are not hierarchical be implemented?

10.4. The terms *architectural style, architectural pattern,* and *framework* (not discussed in this book) are often encountered in discussions of software architecture. Do some research, and describe how each of these terms differs from its counterparts.

10.5. Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 10.3.3.

10.6. Research the ATAM (using [Kaz98]), and present a detailed discussion of the six steps presented in Section 10.7.1.

10.7. If you haven't done so, complete Problem 8.3. Use the design approach described in this chapter to develop a software architecture for the pothole tracking and repair system (PHTRS).

10.8. Use the architectural decision template from Section 10.1.4 to document one of the architectural decisions for PHTRS architecture developed in Problem 10.7.

10.9. Select a mobile application you are familiar with, and assess it using the architecture considerations (economy, visibility, spacing, symmetry, emergence) from Section 10.4.

10.10. List the strengths and weakness of the PHTRS architecture you created for Problem 10.7.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

11 COMPONENT-LEVEL DESIGN

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. Component-level design bridges the gap between architectural design and coding.

KEY CONCEPTS

cohesion 216
component. 207
component-based development 228
content design. 226
coupling 218
dependency inversion principle 214
design guidelines 215
interface segregation principle 214
Liskov substitution principle. 214
object-oriented view 207
open-closed principle 212
process-related view. 211
traditional components. 227
traditional view 209
WebApp component 226

QUICK LOOK

What is it? A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it? A software engineer performs component-level design.

Why is it important? You need to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic.

What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct.

Component-level design will reduce the number of errors introduced during coding. As you translate the design model into source code, you should follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

11.1 What Is a Component?

A *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG03a] defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

As we discussed in Chapter 10, components populate the software architecture and play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

The true meaning of the term *component* will differ depending on the point of view of the software engineer who uses it. In the sections that follow, we examine three important views of what a component is and how it is used as design modeling proceeds.

11.1.1 An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes.[1] Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the analysis model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component). To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived.

The attributes and operations defined during analysis are noted at the top of Figure 11.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation[2] shown in the middle right of the figure. Note that **PrintJob** has two interfaces, *computeJob,*

1 In some cases, a component may contain a single class.

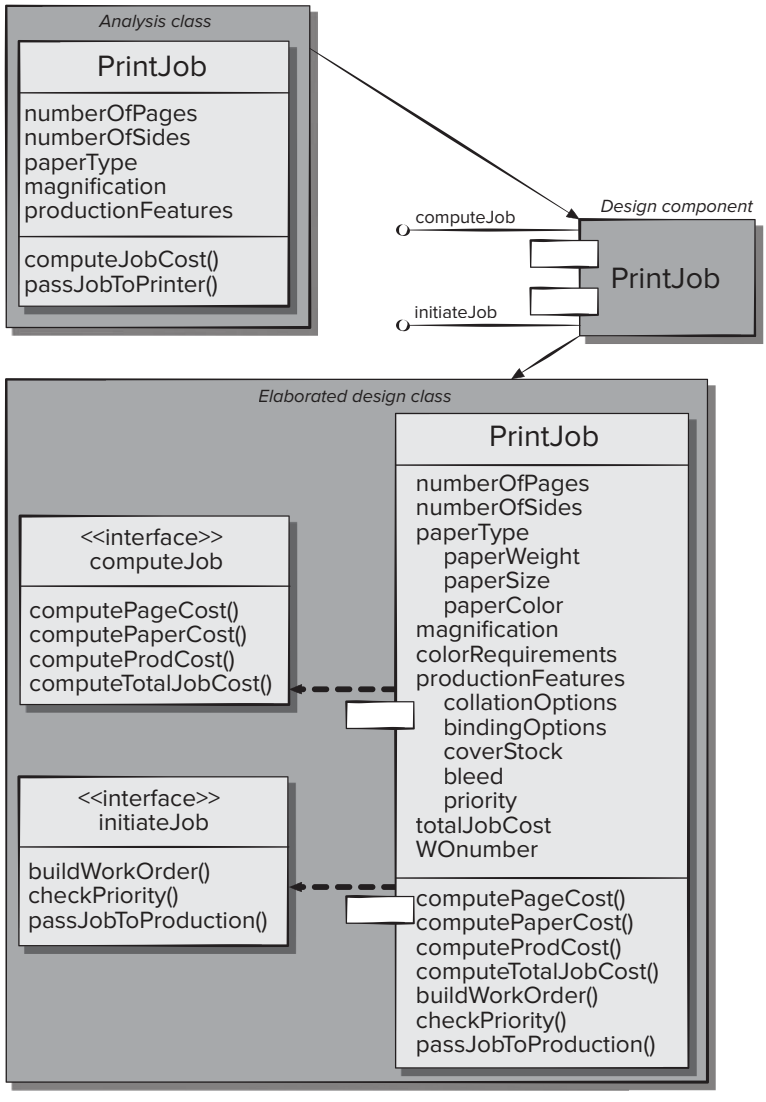
2 Readers who are unfamiliar with UML notation should refer to Appendix 1.

which provides job costing capability, and *initiateJob,* which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob.** Referring to the lower right portion of Figure 11.1, the elaborated design class **PrintJob** contains more

FIGURE 11.1

Elaboration of a design component



detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

This elaboration activity is applied to every component defined as part of the architectural design. Once it is completed, further elaboration is applied to each attribute, operation, and interface. The data structures appropriate for each attribute must be specified. In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed. This procedural design activity is discussed later in this chapter. Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

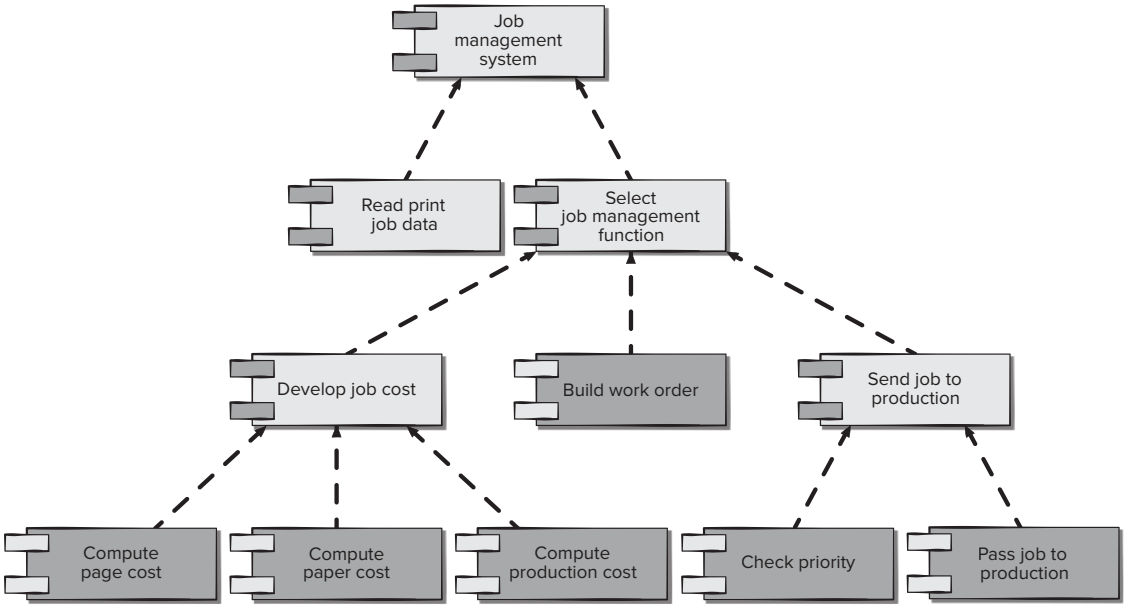
11.1.2 The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a *module,* resides within the software architecture and serves one of three important roles: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. In this case, however, the component elaboration element of the analysis model serves as the basis for the derivation. Each component representing the component hierarchy is mapped (Section 10.6) into a module hierarchy. Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence (Chapter 9) are applied as components are elaborated.

To illustrate this process of design elaboration for traditional components, again consider software to be built for the print shop noted earlier. A hierarchical architecture is derived and shown in Figure 11.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 11.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

FIGURE 11.2 Structure chart for a traditional system

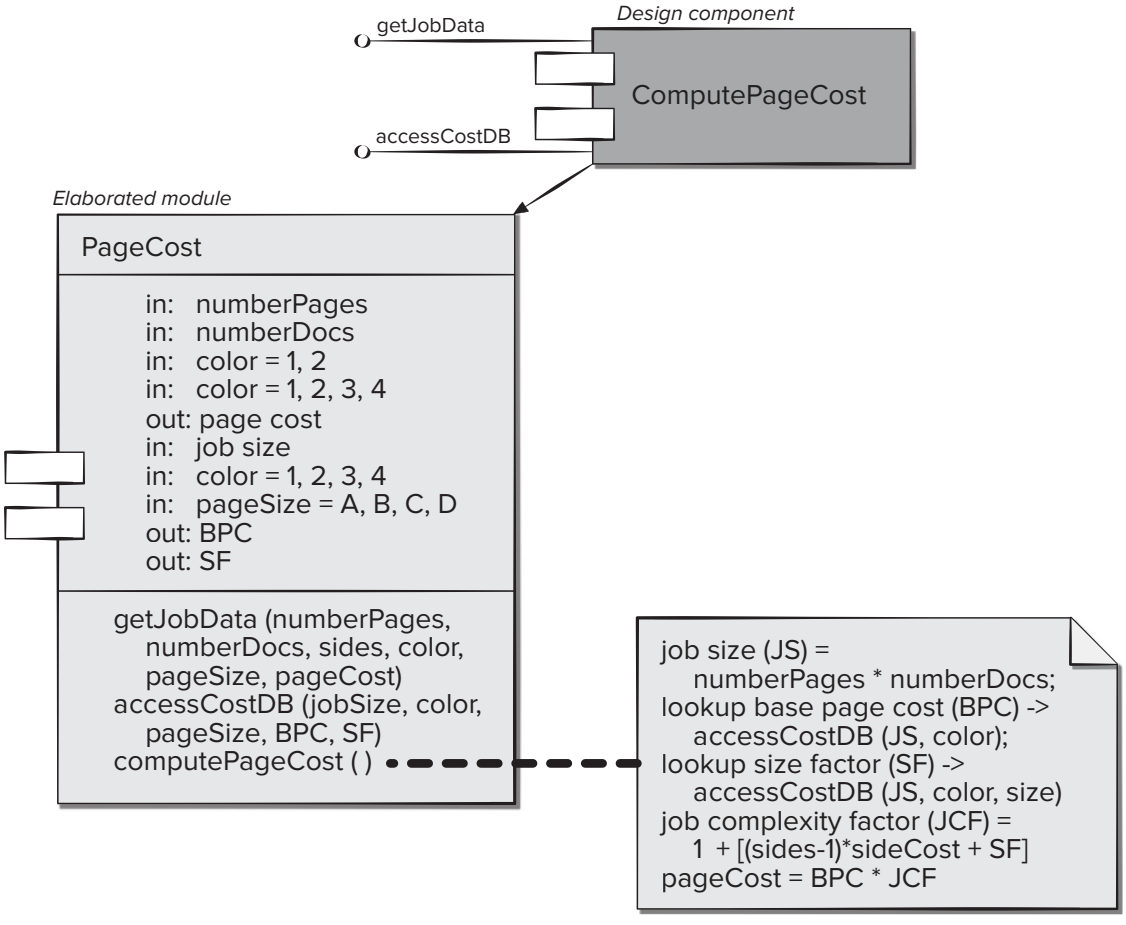


During component-level design, each module in Figure 11.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 9. The behavior of the module is sometimes represented using a state diagram.

To illustrate this process, consider the module *ComputePageCost.* The intent of this module is to compute the printing cost per page based on specifications provided by the customer. Data required to perform this function are: number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements, and size requirements. These data are passed to *ComputePageCost* via the module's interface. *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. Figure 11.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is

FIGURE 11.3 Component-level design for *ComputePageCost*



elaborated to provide algorithm detail and interface detail (Figure 11.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

11.1.3 A Process-Related View

The object-oriented and traditional views of component-level design presented in Sections 11.1.1 and 11.1.2 assume that the component is being designed from scratch. That is, you always create a new component based on specifications derived from the requirements model. There is, of course, another approach.

Over the past four decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. To do this, a catalog of proven design or code-level components needs to be made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you. We will save a discussion on the pros and cons of component-based software engineering (CBSE) for Section 11.4.4.

11.2 DESIGNING CLASS-BASED COMPONENTS

As we have already noted, component-level design draws on information developed as part of the requirements model (Chapter 8) and represented as part of the architectural model (Chapter 10). When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model. The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

11.2.1 Basic Design Principles

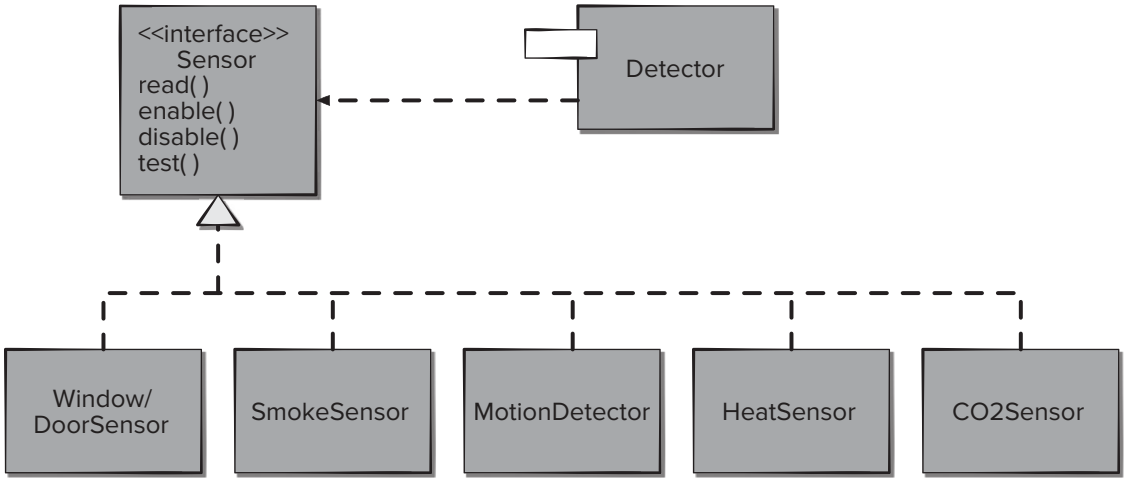
Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur. You can use these principles as a guide as each software component is developed.

The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification"* [Mar00]. This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 11.4. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added, no change is required for the **Detector** class (component). The OCP is preserved.

FIGURE 11.4 Following the OCP



SAFEHOME



The OCP in Action

The scene: Vinod's cubicle.

The players: Vinod and Shakira, members of the *SafeHome* software engineering team.

The conversation:

Vinod: I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.

Shakira (smirking): Not again, jeez!

Vinod: Yeah . . . and you're not going to believe what these guys have come up with.

Shakira: Amaze me.

Vinod (laughing): They call it a doggie angst sensor.

Shakira: Say what?

Vinod: It's for people who leave their pets home in apartments or condos or houses that are close to one another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.

Shakira: You're kidding me, right?

Vinod: Nope. Doug wants to know how much time it's going to take to add it to the security function.

Shakira (thinking a moment): Not much . . . look. [She shows Vinod Figure 11.4.] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an appropriate component . . . uh, class, for it. No change to the **Detector** component at all.

Vinod: So I'll tell Doug it's no big deal.

Shakira: Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.

Vinod: That's not a bad thing, but can you implement now if he wants you to?

Shakira: Yeah, the way we designed the interface lets me do it with no hassle.

Vinod (thinking a moment): Have you ever heard of the open-closed principle?

Shakira (shrugging): Never heard of it.

Vinod (smiling): Not a problem.

The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes"* [Mar00]. This design principle, originally proposed by Barbara Liskov [Lis88], suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a *precondition* that must be true before the component uses a base class and a *postcondition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and post-conditions.

The Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions"* [Mar00]. As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend. Just remember that code is the ultimate concretion. If you dispense with design and hack out code, you're violating DIP.

The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface"* [Mar00]. There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to an individual client category should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

As an example, consider the **FloorPlan** class that is used for the *SafeHome* security and surveillance functions (Chapter 10). For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()* to place, show, group, and remove sensors from the floor plan. The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID().* Hence, the ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them. The interface for security would encompass only the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice().* The interface for surveillance would incorporate the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice(),* along with *showFOV()* and *showDeviceID().*

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design proceeds? Martin [Mar00] suggests additional packaging principles that are applicable to component-level design. These principles follow.

The Reuse/Release Equivalency Principle (REP). *"The granule of reuse is the granule of release"* [Mar00]. When classes or components are designed for reuse,

an implicit contract is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve. Designing components for reuse requires more than good technical design. It also requires effective configuration control mechanisms (Chapter 22).

The Common Closure Principle (CCP). *"Classes that change together belong together"* [Mar00]. Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together"* [Mar00]. When one or more classes with a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operated without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

11.2.2 Component-Level Design Guidelines

In addition to the principles discussed in Section 11.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler [Amb02b] suggests the following guidelines:

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class name **FloorPlan** is meaningful to everyone reading it regardless of technical background. On the other hand, infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning. If a linked list is to be managed as part of the **FloorPlan** implementation, the operation *manageList()* is appropriate, even if a nontechnical person might misinterpret it.[3]

You can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, `<<infrastructure>>` might be used to identify an

3 It is unlikely that someone from marketing or the customer organization (a nontechnical type) would examine detailed design information.

infrastructure component, <<database>> could be used to identify a database that services one or more design classes or the entire system, and <<table>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler [Amb02c] recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex, (2) for consistency, interfaces should flow from the left-hand side of the component box, (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available. These recommendations are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, components' interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency. Following the philosophy of the OCP, this will help to make the system more maintainable.

11.2.3 Cohesion

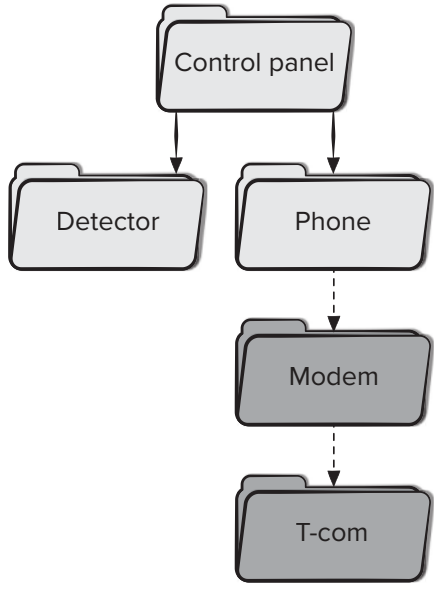
In Chapter 9, we described cohesion as the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to each other and to the class or component itself. Lethbridge and Laganiére [Let04] define several different types of cohesion (listed in order of the level of the cohesion):[4]

> **Functional.** Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns a result.
>
> **Layer.** Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider, for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages, as shown in Figure 11.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.
>
> **Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

4 In general, the higher the level of cohesion, the easier the component is to implement, test, and maintain.

FIGURE 11.5

Layer cohesion



Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. You should strive to achieve these levels of cohesion whenever possible. It is important to note, however, that pragmatic design and implementation issues sometimes force you to opt for lower levels of cohesion.

SAFEHOME



Cohesion in Action

The scene: Jamie's cubicle.

The players: Jamie and Ed, members of the *SafeHome* software engineering team who are working on the surveillance function.

The conversation:

Ed: I have a first-cut design of the **camera** component.

Jamie: Wanna do a quick review?

Ed: I guess . . . but really, I'd like your input on something.

(Jamie gestures for him to continue.)

Ed: We originally defined five operations for **camera**. Look . . .

determineType() tells me the type of camera.

translateLocation() allows me to move the camera around the floor plan.

displayID() gets the camera ID and displays it near the camera icon.

displayView() shows me the field of view of the camera graphically.

displayZoom() shows me the magnification of the camera graphically.

Ed: I've designed each separately, and they're pretty simple operations. So I thought it might be a good idea to combine all of the display operations into just one that's called *displayCamera()*—it'll show the ID, the view, and the zoom. Whaddaya think?

Jamie (grimacing): Not sure that's such a good idea.

Ed (frowning): Why? All of these little ops can cause headaches.

Jamie: The problem with combining them is we lose cohesion, you know, the *displayCamera()* op won't be single-minded.

Ed (mildly exasperated): So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.

Jamie: And what if marketing decides to change the way that we represent the view field?

Ed: I just jump into the *displayCamera()* op and make the mod.

Jamie: What about side effects?

Ed: Whaddaya mean?

Jamie: Well, say you make the change but inadvertently create a problem with the ID display.

Ed: I wouldn't be that sloppy.

Jamie: Maybe not, but what if some support person 2 years from now has to make the mod? He might not understand the op as well as you do, and, who knows, he might be sloppy.

Ed: So you're against it?

Jamie: You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.

Ed (thinking a moment): Maybe we'll go with separate display ops.

Jamie: Good decision.

11.2.4 Coupling

In earlier discussions of analysis and design, we noted that communication and collaboration are essential elements of any object-oriented system. There is, however, a darker side to this important (and necessary) characteristic. As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as possible.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére [Let04] define a spectrum of coupling categories. For example, *content coupling* occurs when one component "surreptitiously modifies data that is internal to another component" [Let04]. This violates information hiding—a basic design concept. *Control coupling* occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result. *External coupling* occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

SAFEHOME



Coupling in Action

The scene: Shakira's cubicle.

The players: Vinod and Shakira, members of the *SafeHome* software team who are working on the security function.

The conversation:

Shakira: I had what I thought was a great idea . . . then I thought about it a little, and it seemed like a not-so-great idea. I finally rejected it, but I just thought I'd run it by you.

Vinod: Sure. What's the idea?

Shakira: Well, each of the sensors recognizes an alarm condition of some kind, right?

Vinod (smiling): That's why we call them sensors, Shakira.

Shakira (exasperated): Sarcasm, Vinod, you've got to work on your interpersonal skills.

Vinod: You were saying?

Shakira: Okay, anyway, I figured . . . why not create an operation within each sensor object called *makeCall()* that would collaborate directly with the **OutgoingCall** component, well, with an interface to the **OutgoingCall** component.

Vinod (pensive): You mean rather than having that collaboration occur out of a component like **ControlPanel** or something?

Shakira: Yeah . . . but then, I said to myself, that means that every sensor object will be connected to the **OutgoingCall** component, and that means that it's indirectly coupled to the outside world and . . . well, I just thought it made things complicated.

Vinod: I agree. In this case, it's a better idea to let the sensor interface pass info to the **ControlPanel** and let it initiate the outgoing call. Besides, different sensors might result in different phone numbers. You don't want the sensor to store that information because if it changes . . .

Shakira: It just didn't feel right.

Vinod: Design heuristics for coupling tell us it's not right.

Shakira: Whatever . . .

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, a designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

11.3 CONDUCTING COMPONENT-LEVEL DESIGN

Earlier in this chapter we noted that component-level design is elaborative in nature. You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated as described in Section 11.1.1.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we

have noted earlier, classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

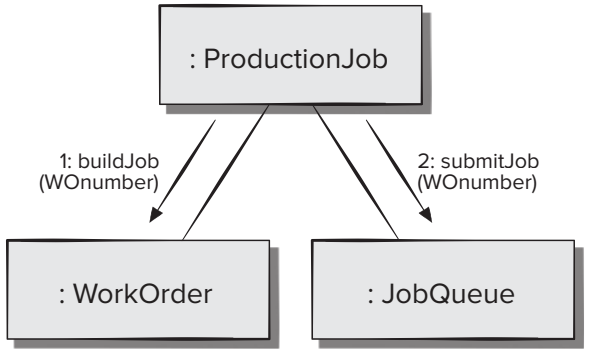
Figure 11.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner [Ben10a]:

```
[guard condition] sequence expression (return value) :=
message name (argument list)
```

where a `[guard condition]` is written in Object Constraint Language (OCL)[5] and specifies any set of conditions that must be met before the message can be sent; `sequence expression` is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; `(return value)` is the name of the information that is returned by the operation invoked by the message; `message name` identifies the operation that is to be invoked; and `(argument list)` is the list of attributes that are passed to the operation.

FIGURE 11.6

Collaboration diagram with messaging



5 OCL is discussed briefly in Appendix 1.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations . . ." [Ben10a]. Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. The elaboration of interfaces is illustrated in Figure 11.1. The operations defined for the design class are categorized into one or more abstract classes. Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

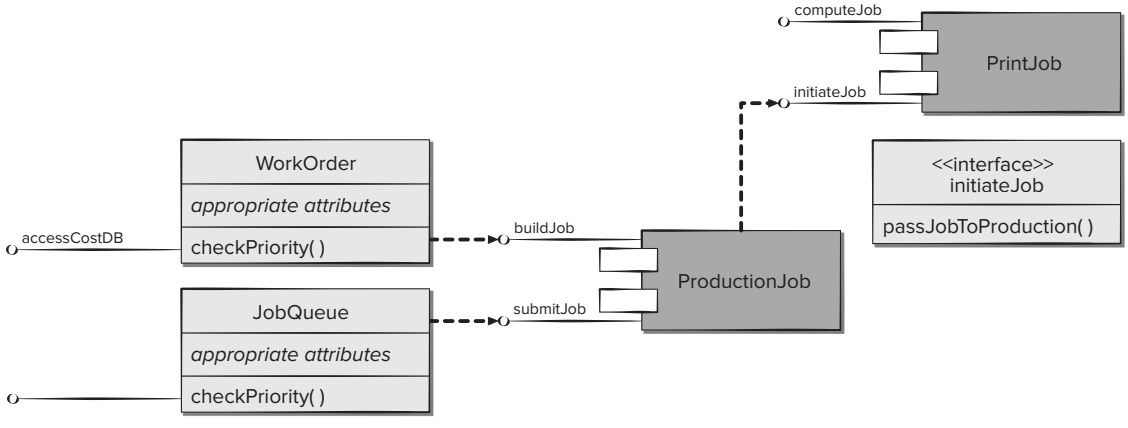
Referring to Figure 11.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. It performs three different subfunctions—building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 11.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly single-minded.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

```
name : type-expression = initial-value {property-string}
```

where `name` is the attribute name, `type expression` is the data type, `initial value` is the value that the attribute takes when an object is created, and `property-string` defines a property or characteristic of the attribute.

FIGURE 11.7 Refactoring interfaces and class definitions for PrintJob



During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 11.1, the attribute list for **PrintJob** lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defined using the UML attribute format noted. For example, `paperType-weight` is defined in the following manner:

```
paperType-weight: string = "A" {contains 1 of 4 values – A, B, C, or D}
```

which defines `paperType-weight` as a string variable initialized to the value A that can take on one of four values from the set {A, B, C, D}.

If an attribute appears repeatedly across several design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through several iterations that apply the stepwise refinement concept (Chapter 9).

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. For example, the operation *computePaperCost()* noted in Figure 11.1 can be expanded in the following manner:

```
computePaperCost (weight, size, color): numeric
```

This indicates that *computePaperCost()* requires the attributes `weight`, `size`, and `color` as input and returns a value that is numeric (actually a dollar value) as output.

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 11.8 depicts a UML activity diagram for *computePaperCost()*. When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code.

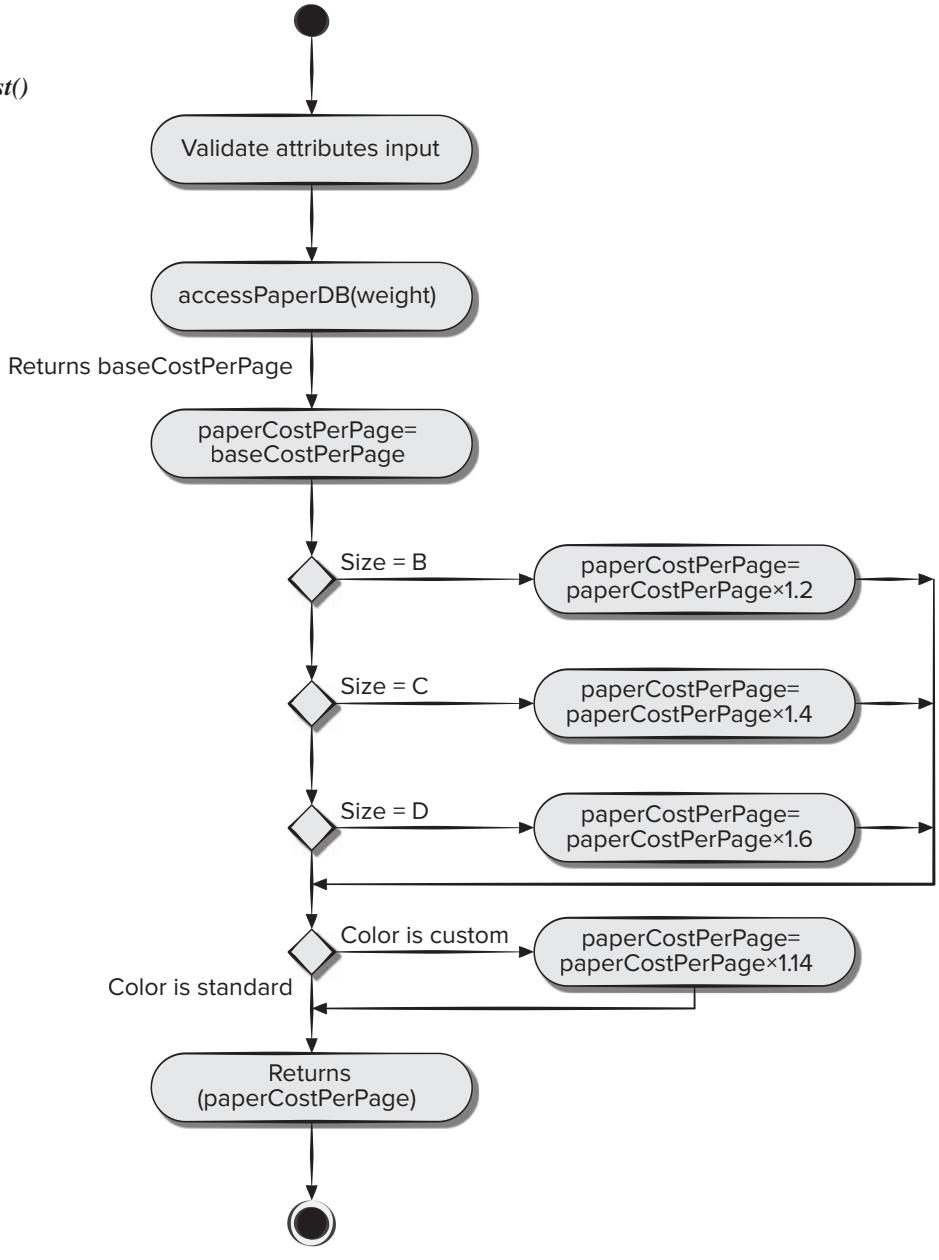
Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, you

FIGURE 11.8

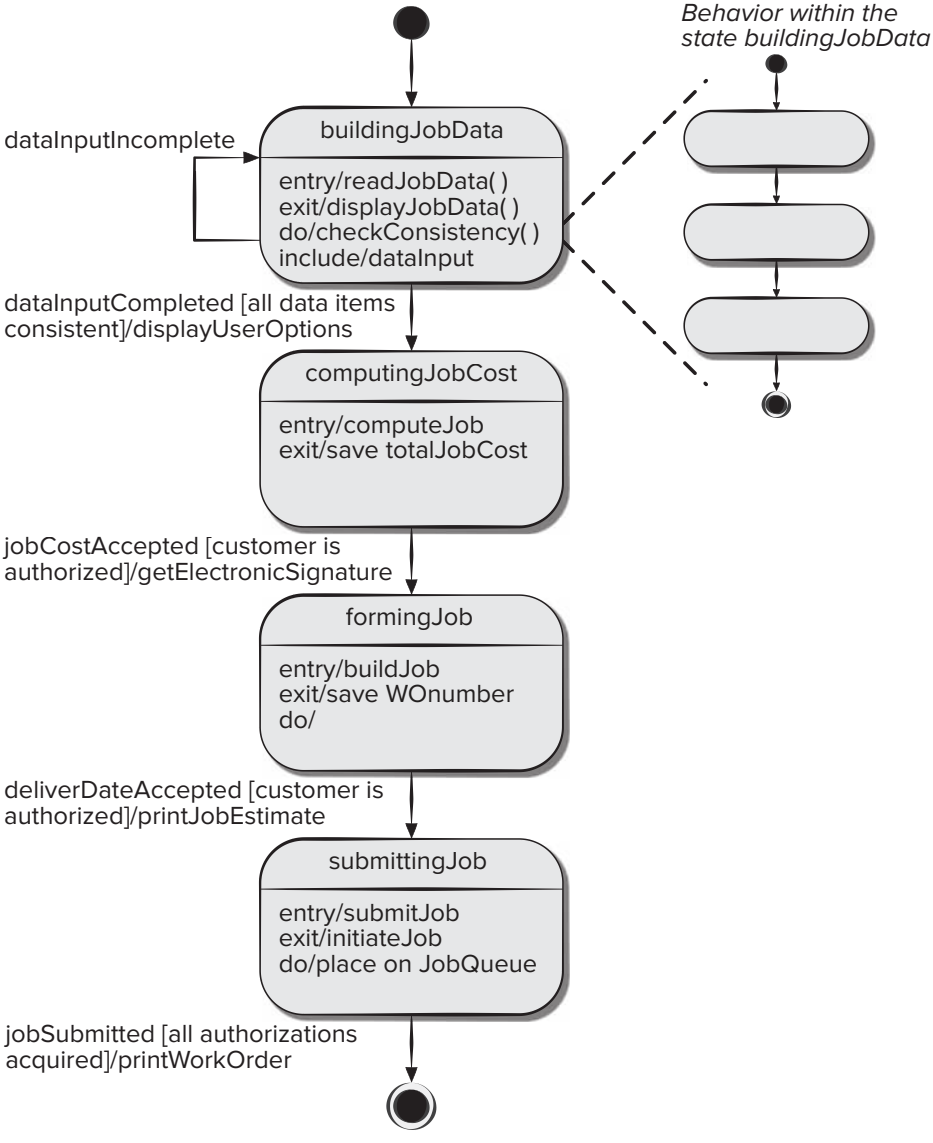
UML activity diagram for *computePaperCost()*



should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) is represented using a UML statechart [Ben10a], as illustrated in Figure 11.9.

FIGURE 11.9

Statechart fragment for PrintJob class



The transition from one state (represented by a rectangle with rounded corners) to another occurs following an event that takes the form of:

```
Event-name (parameter-list) [guard-condition] / action expression
```

where `event-name` identifies the event, `parameter-list` incorporates data that are associated with the event, `guard-condition` is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and `action expression` defines an action that occurs as the transition takes place.

Referring to Figure 11.9, each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that occur while in the state, and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 11.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve), the print job cannot be submitted because there is no way to reach the *submittingJob* state.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams (Chapter 9) are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented (often as subsystems) within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always consider alternatives. Throughout this book, we emphasize that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

In addition, you should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before settling on the final design model. Develop alternatives and consider each carefully, using the design principles and concepts presented in Chapter 9 and in this chapter.

11.4 SPECIALIZED COMPONENT-LEVEL DESIGN

There are many programming languages and many ways to create the components required to implement a software architectural design. The principles described in this chapter provide general advice for designing components. Many software products require the use of specialized program development environments to allow their deployment on targeted end user devices such as cell phones or digital assistants. In this section, we present overviews of some specialized component design techniques.

11.4.1 Component-Level Design for WebApps

The boundary between content and function is often blurred when Web-based systems and applications (WebApps) are considered. Therefore, it is reasonable to ask: What is a WebApp component?

In the context of this chapter, a WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content design at the component level focuses on content objects and the ways they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelationships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation.[6] In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

A good example of a component that might be part of an e-commerce WebApp is the "shopping cart." A shopping cart provides a convenient way for e-commerce customers to store and review their selected items prior to checking out. They can then pay for their selections with a single transaction at the end of their e-commerce session. A carefully designed shopping cart can be reused in several Web store applications by simply editing its content model.

WebApp functionality can be delivered as a series of components developed in parallel with the information architecture to ensure consistency. The shopping cart component described previously contains both content and algorithmic elements. You begin by considering both the requirements model and the initial information architecture. Next, you examine how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A *functional architecture* is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

11.4.2 Component-Level Design for Mobile Apps

Mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. If you are building a mobile app as a thin Web-based client, the only components residing on a mobile device are those required to implement the user interface. Some mobile apps may incorporate the components required to implement the business and/or data layers on the mobile device, subjecting these layers to the limitations of the physical characteristics of the device.

---

6 Content components can also be reused in other WebApps.

Considering the user interface layer first, it is important to recognize that a small display area requires the designer to be more selective in choosing the content (text and graphics) to be displayed. It may be helpful to tailor the content to a specific user group(s) and display only what each group needs. The business and data layers are often implemented by composing Web or cloud service components. If the components providing business and data services reside entirely on the mobile device, connectivity issues are not a significant concern. Intermittent (or missing) Internet connectivity must be considered when designing components that require access to current application data that reside on a networked server.

If a desktop application is being ported to a mobile device, the business-layer components may need to be reviewed to see if they meet nonfunctional requirements (e.g., security, performance, accessibility) required by the new platform. The target mobile device may lack the necessary processor speed, memory, or display real estate. The design of mobile applications is considered in greater detail in Chapter 13.

An example of a component in a mobile application might be the single-window full-screen user interface (UI) typically designed for phones and tablets. With careful design it may be possible to allow the mobile app to sense the display characteristics of the mobile device and adapt its appearance to ensure that text, graphics, and UI controls function correctly on many different screen types. This allows the mobile app to function in similar ways on all platforms, without having to be reprogrammed.

11.4.3 Designing Traditional Components

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra ([Dij65], [Dij76b]) and others (e.g., [Boh66]). A traditional software component implements an element of processing that addresses a function or subfunction in the problem domain or some capability in the infrastructure domain. Often these traditional components are called functions, modules, procedures, or subroutines. Traditional components do not encapsulate data in the same way that object-oriented components do. Most programmers make frequent use of function libraries and data structure templates when developing new software products.

In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures. Complexity metrics (Chapter 23) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which

you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties.

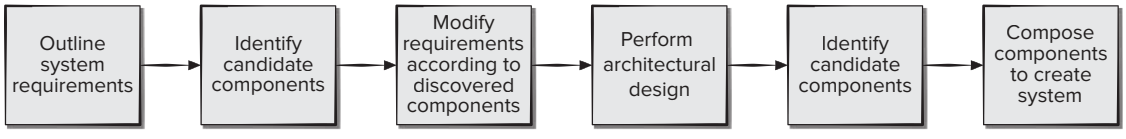
11.4.4 Component-Based Development

In software engineering, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer-based systems must be built in very short time periods and demand a more organized approach to reuse.

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software components (Figure 11.10). Considering this description, many questions arise. Is it possible to construct complex systems by assembling them from a catalog of reusable software components? Can this be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is management willing to incur the added expense associated with creating reusable software components? Can a library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it? Can existing components be found by those who need them? Increasingly, the answer to each of these questions is yes.

Figure 11.10 shows the principle steps in CBSE. You start with the system requirements and refine them to the point that needed components can be identified. The developers would then search the repository to see if any of the components already exist. Each component has its own postconditions and preconditions. Components whose postconditions match a system requirement are identified, and the preconditions of each component are checked. If the preconditions are satisfied, the component is selected for inclusion in the current build. When no components can be selected, the developers must decide whether to modify the requirements or modify a component that most closely matches the original requirements. This is often an iterative process that continues until the architecture design can be implemented, using a combination of existing or newly created components.

FIGURE 11.10 **Component-based software design**



Consider the task of developing autonomous vehicles, either in real life or a video game. The software for these complex systems is typically created by combing several reusable components, where the components provide distinct modular services. Typically they would include many software components: a component that manages obstacle detection, a planning or navigation component, an artificial intelligence component to manage decision making, and a component of some type controlling vehicle movement or braking. Because these types of software modules have the potential to be used in many different vehicles, it would be desirable to be able to house them in a library of components.

Because CBSE makes use of existing components, it can shorten development time and increase quality. Practitioners [Vit03] often attribute the following advantages to CBSE:

- **Reduced lead time.** It is faster to build complete applications from a pool of existing components.
- **Greater return on investment (ROI).** Sometimes savings can be realized by purchasing components rather than redeveloping the same functionality in-house.
- **Leveraged costs of developing components.** Reusing components in multiple applications allows the costs to be spread over multiple projects.
- **Enhanced quality.** Components are reused and tested in many different applications.
- **Maintenance of component-based applications.** With careful engineering, it can be relatively easy to replace obsolete components with new or enhanced components.

Use of components in CBSE is not without risks. Several of these include the following [Kau11]:

- **Component selection risks.** It is difficult to predict component behavior for black-box components, or there may be poor mapping of user requirements to the component architectural design.
- **Component integration risks.** There is a lack of interoperability standards between components; this often requires the creation of "wrapper code" to interface components.
- **Quality risks.** Unknown design assumptions made for the components makes testing more difficult, and this can affect system safety, performance, and reliability.
- **Security risks.** A system can be used in unintended ways, and system vulnerabilities can be caused by integrating components in untested combinations.
- **System evolution risks.** Updated components may be incompatible with user requirements or contain additional undocumented features.

One of the challenges facing widespread component reuse is *architectural mismatch* [Gar09a]—incompatibilities between assumptions made about components and their operating environments.[7] These assumptions often focus on the component

7 This can be a result of several forms of coupling that should be avoided whenever possible.

control model, the nature of the component connections (interfaces), the architectural infrastructure itself, and the nature of the construction process.

Early detection of architectural mismatch can occur if stakeholder assumptions are explicitly documented. In addition, the use of a risk-driven process model emphasizes the definition of early architectural prototypes and points to areas of mismatch. Repairing architectural mismatch is often very difficult without making use of mechanisms like wrappers or adapters.[8] Sometimes it is necessary to completely redesign a component interface or the component itself to remove coupling issues.

11.5 COMPONENT REFACTORING

Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, and software quality assurance (SQA) all contribute to the creation of software components that will be easier to refactor. Most developers would agree that refactoring components to improve quality is a good practice. It is often hard to convince management that it is important to expend resources fixing components that are working correctly instead of adding new functionality to them.

In this book, we focus on the incremental design and delivery of system components. Although there is no quantifiable relationship describing the effects of code changes on architectural quality, most software engineers agree that over time large numbers of changes to a system can lead to the creation of problematic structures in the code base. Failing to address these problems increases the amount of technical debt (Chapter 9) associated with the software system. Reducing this technical debt often involves architectural refactoring, which is generally perceived by developers as both costly and risky. You cannot simply break up large components into smaller components and expect to see an automatic increase in cohesion and a reduction in coupling that will reduce technical debt.

Large software systems may have thousands of components. Making use of data-mining techniques to identify refactoring opportunities can be very beneficial to this work. Automated tools can analyze the source code of system components and make refactoring recommendations to developers, based on generic design rules known to be associated with architectural problems. But it is still up to the developers and their managers to decide which changes to accept and which to ignore [Lin16].

It turns out that many of the error-prone components in a software system are architecturally connected to one another. These flawed architectural connections tend to propagate defects among themselves and accumulate high maintenance costs. If it was possible to automatically identify the technical debt present in the system and the associated maintenance costs, it would be easier to convince developers and managers to spend time refactoring these components. Accomplishing this type of

8 An *adapter* is a software device that allows a client with an incompatible interface to access a component by translating a request for service into a form that can access the original interface.

work requires examining the change histories of the system components [Xia16]. For example, if two or three components are always checked out of the code repository for modification at the same time, it may suggest the components share a common design defect.

11.6 SUMMARY

The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Three different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The traditional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

Several important principles and concepts guide the designer as classes are elaborated. Ideas encompassed in the open-closed principle and the dependency inversion principle, along with concepts such as coupling and cohesion, guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated by specifying messaging details, identifying appropriate interfaces, elaborating attributes, and defining data structures to implement them, describing processing flow within each operation, and representing behavior at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Traditional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of several design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Component-level design for WebApps considers both content and functionality as a Web-based system will deliver it. Content design at the component level focuses on content objects and the ways they may be packaged for presentation to a WebApp end user. Functional design for WebApps focuses on processing functions that manipulate content, perform computations, process database queries, and establish interfaces with other systems. All component-level design principles and guidelines apply.

Component-level design for mobile apps makes use of a multilayered architecture that includes a user interface layer, a business layer, and a data layer. If the mobile app requires the design of components that implement the business and/or data layers on the mobile device, the limitations of the physical characteristics of the device become important constraints on the design.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Component-based software engineering identifies, constructs, catalogs, and disseminates a set of software components for an application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

PROBLEMS AND POINTS TO PONDER

11.1. The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for object-oriented and traditional software. Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

11.2. Why are control components necessary in traditional software and generally not required in object-oriented software?

11.3. Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?

11.4. Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?

11.5. Select three components that you have developed recently, and assess the types of cohesion that each exhibits. If you had to define the primary benefit of high cohesion, what would it be?

11.6. Select three components that you have developed recently, and assess the types of coupling that each exhibits. If you had to define the primary benefit of low coupling, what would it be?

11.7. Develop (1) an elaborated design class, (2) interface descriptions, (3) an activity diagram for one of the operations within the class, and (4) a detailed statechart diagram for one of the *SafeHome* classes that we have discussed in earlier chapters.

11.8. What is a WebApp component?

11.9. Select the code from a small software component and represent it using an activity diagram.

11.10. Why is "chunking" important during the component-level design review process?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

12 User Experience Design

We live in a world of high-technology products, and virtually all of them—consumer electronics, industrial equipment, automobiles, corporate systems, military systems, mobile apps, WebApps, video games, and virtual reality simulations—require human interaction. If a product is to be successful, it must provide a positive user experience (UX). The product needs to exhibit good *usability*—a qualitative measure of the ease and efficiency with which a human can employ the functions and features offered by the high-technology product. A product should incorporate *accessibility* considerations such as assistive technologies when its specified users include people with a range of disabilities within a specified context of use.

KEY CONCEPTS

accessibility 237
accessibility guidelines 259
command labeling 260
customer journey map 244
error handling 260
golden rules 234
help facilities 260
information architecture 235
interface analysis 243
interface consistency 240
interface design 250
internationalization 260
memory load 239
response time 259
storyboard 261
task analysis 247
usability guidelines 257
user experience analysis 243
user interaction design 236
user scenario 261
visual design 237



QUICK LOOK

What is it? User experience (UX) design is the process of enhancing user satisfaction with a product by creating a usable, accessible, and pleasurable interaction between product and its users.

Who does it? A software engineer designs the user experience and user interface assisted by knowledgeable stakeholders.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The user experience has to be right because it molds a user's perception of the software.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. These form the basis for the creation of a screen layout and navigation pathways through the information architecture.

What is the work product? User persona and scenarios are created based on the desired customer journey. Low-fidelity prototypes and digital interface prototypes are developed, evaluated, and modified in an iterative fashion.

How do I ensure that I've done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

For the first three decades of the computing era, usability was not a dominant concern among those who built software. In his classic book on design, Donald Norman [Nor88] argued that it was time for a change in attitude: "To make technology that fits human beings, it is necessary to study human beings. But now we tend to study only the technology. As a result, people are required to conform to technology. It is time to reverse this trend, time to make technology that conforms to people."

As technologists studied human interaction, two dominant issues arose. First, a set of *golden rules* (discussed in Section 12.2) were identified. These applied to all human interaction with technology products. Second, a set of *interaction mechanisms* were defined to enable software designers to build systems that properly implemented the golden rules. These interaction mechanisms, collectively called the *user interface,* have eliminated some of the many egregious problems associated with human interfaces. But even today, we all encounter user interfaces that are difficult to learn, difficult to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

User experience design is a set of incremental process activities that help the development team and the project stakeholders focus on providing a positive experience for users of the software product. UX design is broader than user interface design and usability or accessibility engineering. It must begin early in the project life cycle if it is to be effective. Developers waiting until the end of a project to add user interface functionality are unlikely to provide a pleasurable experience for users.

In this chapter we will focus on user interface design issues in the context of user experience design. Readers wishing a more detailed coverage of UX should examine books by Shneiderman [Shn16], Nielsen [Nei93], and Norman [Nor13].

12.1 User Experience Design Elements

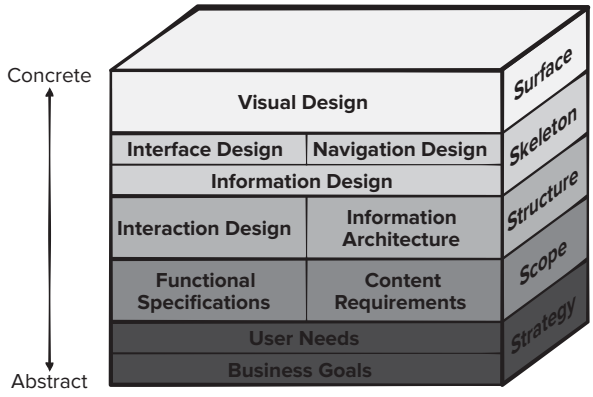
User experience design tries to ensure that no aspect of your software appears in the final release candidate without the explicit decision of the development team and other stakeholders to include it. This means taking into account every reasonable user action and expectation during every step of the development process. To make the task of crafting a positive user experience more manageable, Garret [Gar10] suggests breaking it down into component elements: strategy, scope, structure, skeleton, and surface. The relationships among these component elements and subcomponents are show in Figure 12.1.

The Garrett's organization of UX design can be interpreted as follows for software product development:

- **Strategy.** Identifies user needs and customer business goals that form the basis for all UX design work (Section 12.4)
- **Scope.** Includes both the functional and content (e.g., information, media, services) requirements needed to realize a feature set consistent with the project strategy

FIGURE 12.1

User experience design elements



- **Structure.** Consists of the interaction design [e.g., how the system reacts in response to user actions (Section 12.1.2)] and information architecture [e.g., the organization of the content elements (Section 12.1.1)]
- **Skeleton.** Comprised of three components: information design (e.g., presentation of content in a way to make it understandable to the user), interface design [e.g., arranging interface screen objects to allow the user to work with the system functionality (Section 12.5)], navigation design (e.g., the set of screen elements that allow users to traverse the information architecture)
- **Surface.** Presents visual design or the appearance of the finished project to its users (Section 12.1.4)

Several cross-cutting aspects of UX design are of particular interest to software engineers: information architecture, user interaction design, usability engineering, and visual design.

12.1.1 Information Architecture

As an architectural designer, you must identify information (content) architecture and software architecture. The term *information architecture* is used to connote structures that lead to better organization, labeling, navigation, and searching of content objects. Content architecture focuses on the manner in which content objects (or composite objects such as screens or widgets) are structured for presentation and navigation. Software architecture addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

Architectural design (Chapter 10) is tied to the goals established for a software product, the content to be presented, the users who will visit, and the navigation philosophy that has been established. In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design. Because the software architecture may have a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design. In many cases, a subject matter expert is needed to help project stakeholders organize the content items for efficient assimilation and traversal by product users.

12.1.2 User Interaction Design

Interaction design focuses on the interface between a product and its user. Not too many years ago, the only way for a user to interact with a computer system was typing input on a keyboard and reading output on a display screen of some kind. Today the modes of input and output are quite varied and may include voice input, computer speech generation, touch input, 3D printed output, immersive augmented reality experiences, and sensor tracking of users within their environment. Devices like this are needed to give users ways to control a computer system. Oftentimes these interaction devices interfere with the product providing a natural and pleasurable user experience.

User interaction should be defined by the stakeholders in the user stories (Chapter 7) created to describe how users can accomplish their goals using the software product. This suggests that user interaction design should also include a plan for how information should be presented within such a system and how to enable the user to understand that information. It is important to recall that the purpose of the user interface is to present just enough information to help the users decide what their next action should be to accomplish their goal and how to perform it.

We will describe the user interface design process in more detail in Section 12.5, but initially, there are important questions user interaction designers must ask when devising user interfaces:[1]

- What can users do with a mouse, finger, or stylus to interact with the interface directly?
- What about the appearance (e.g., color, shape, size) gives users clues about how the user interaction functions?
- What information do you provide to let users know what will happen before they perform an action?
- Are there any constraints put in place to help prevent errors?
- Do error messages provide a way for users to correct a problem or explain why an error occurs?
- What feedback do users get once an action is performed?
- Are the interface elements a reasonable size to facilitate interaction?
- What familiar or standard formats should be used to display information and accept input?

12.1.3 Usability Engineering

Usability engineering is part of UX design work that defines the specification, design, and testing of the human-computer interaction portion of a software product. This software engineering action focuses on devising human-computer interfaces that have high usability. Usability engineering provides structured methods for achieving efficiency and elegance in interface design. Terms like *user friendliness* do not provide much guidance here since this is often a very subjective judgment. If developers focus on making a product easy to learn, easy to use, and easy to remember over time, usability can be measured quantitatively and tested for improvements in usability.

1 See https://www.usability.gov/what-and-why/interaction-design.html.

Accessibility is another aspect of usability engineering that should be considered when designing user interactions with the software. *Accessibility* is the degree to which people with special needs (e.g., sight impaired, deaf, elderly, cognitively impaired) are provided with a means to perceive, understand, navigate, and interact with computer products. The goal of accessibility design is to provide hardware or software tools that can remove barriers that may prevent users from successfully completing tasks supported by the software. Usability and accessibility are discussed in greater detail in Section 12.7.

12.1.4 Visual Design

Visual design, also called aesthetic design or graphic design, is an artistic endeavor that complements the technical aspects of the user experience design. Without it, a software product may be functional, but unappealing. With it, a product draws its users into a world that embraces them on a visceral, as well as an intellectual level.

But what is aesthetic? There is an old saying, "Beauty exists in the eye of the beholder." This is particularly appropriate when aesthetic design for many games or mobile apps is considered. To perform effective aesthetic design, you should return to the user hierarchy developed as part of the requirements model (Chapter 8) and ask, "Who are the product's users and what 'look' do they desire?"

Graphic design considers every aspect of the look and feel of a web or mobile app. The graphic design process begins with screen layout and proceeds into a consideration of global color schemes; type fonts, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application. Not every software engineer has artistic talent. If you fall into this category, hire an experienced graphic designer for aesthetic design work.

SAFEHOME



Graphic Design

The scene: Doug Miller's office after the first *SafeHome* room layout interface prototype review.

The players: Doug Miller, SafeHome software engineering project manager, and Vinod Raman, member of the *SafeHome* software engineering team.

The conversation:

Doug: What's your impression of the new room layout design?

Vinod: I like it, but more importantly, our customers like it.

Doug: How much help did you get from the graphic designer we borrowed from marketing?

Vinod: A great deal actually. Marg has a great eye for page layout and suggested an awesome graphic theme for the app screens. Much better than what we came up with on our own.

Doug: That's good. Any issues?

Vinod: We still have to create an alternate screen to take accessibility issues into account for some of our visually impaired users. But we would've had to do that for any app design we had.

Doug: Can we use Marg for that work as well?

Vinod: Yeah, she has a good understanding of usability and accessibility.

Doug: OK, I'll set it up with marketing and borrow her a little longer.

12.2 The Golden Rules

In his book on interface design, Theo Mandel [Man97] coins three *golden rules:*

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

12.2.1 Place the User in Control

During a requirements gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

"What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that."

Your first reaction might be to shake your head and smile, but pause for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel [Man97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *autocorrect* is selected in a text messaging app menu, the software performs autocorrect continually. There is no reason to force the user to remain in autocorrect mode. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using a keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action or any linear sequence of actions.

Streamline interaction as skill levels advance, and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to drag a document into the "trash" is an implementation of direct manipulation.

12.2.2 Reduce the User's Memory Load

A well-designed user interface does not tax a user's memory because the more a user has to remember, the more error-prone the interaction will be. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., ctrl-C to invoke the *copy* function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. A room layout application should allow users to drag furniture from a visual catalog and arrange it on the screen using a touch interface. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest.

SAFEHOME



Violating a User Interface Golden Rule

The scene: Vinod's cubicle, as user interface design begins.

The players: Vinod and Jamie, members of the *SafeHome* software engineering team.

The conversation:

Jamie: I've been thinking about the surveillance function interface.

Vinod (smiling): Thinking is good.

Jamie: I think maybe we can simplify matters some.

Vinod: Meaning?

Jamie: Well, what if we eliminate the floor plan entirely? It's flashy, but it's going to take serious development effort. Instead, we just ask the user to specify the camera he wants to see and then display the video in a video window.

Vinod: How does the homeowner remember how many cameras are set up and where they are?

Jamie (mildly irritated): He's the homeowner; he should know.

Vinod: But what if he doesn't?

Jamie: He should.

Vinod: That's not the point . . . what if he forgets?

Jamie: Uh, we could provide a list of operational cameras and their locations.

Vinod: That's possible, but why should he have to ask for a list?

Jamie: Okay, we provide the list whether he asks or not.

Vinod: Better. At least he doesn't have to remember stuff that we can give him.

Jamie (thinking for a moment): But you like the floor plan, don't you?

Vinod: Uh-huh. Especially since we are creating the room layout application in a related product.

Jamie: Which one will marketing like, do you think?

Vinod: You're kidding, right?

Jamie: No.

Vinod: Duh . . . the one with the flash . . . they love sexy product features . . . they're not interested in which is easier to build.

Jamie (sighing): Okay, maybe I'll create paper prototypes of both.

Vinod: Good idea . . . then we let the customer decide.

12.2.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen

images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a complete product line. A family of applications (i.e., a product line) should implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application encountered. A change (e.g., using alt-S to invoke scaling) will cause confusion.

The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the user experience design process itself.

12.3 User Interface Analysis and Design

Although UX design work is not only about the user interface, design of the user interface is a good place to start understanding the UX process. The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

12.3.1 Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model,* the software engineer creates a *design model,* the end user develops a mental image that is often called the user's *mental model* or the *system perception,* and the implementers of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [Shn16]. In addition, users can be categorized as novices; knowledgeable, intermittent users; or knowledgeable frequent users. Many UX designers like to build user profiles or personas (Section 12.4.2) as way of capturing what is known about each class of users.

The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a mobile app that rates restaurants were asked to describe its operation, the system perception would guide the

response. The accuracy of the description will depend on the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands restaurant rating apps fully but has worked with the specific app only a few times might actually be able to provide a more complete description of its function than the novice who has spent days trying to apply the app effectively.

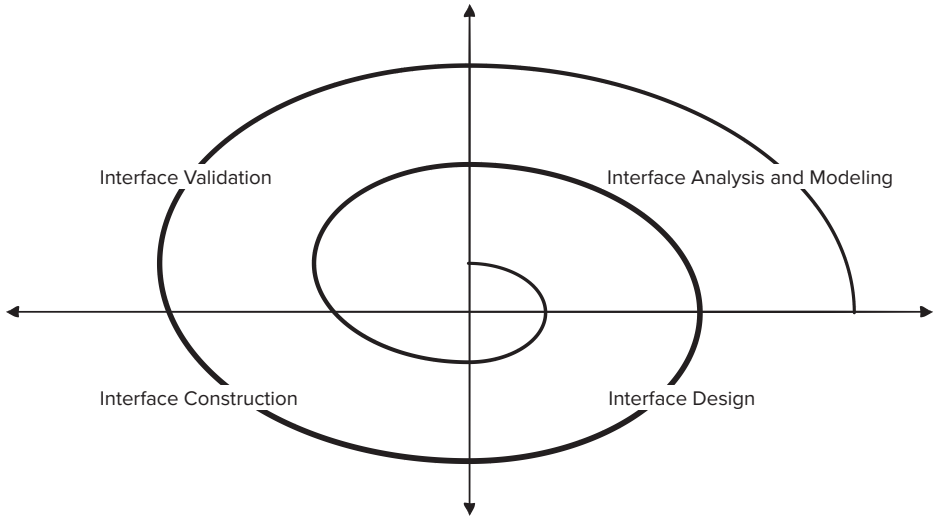
The *implementation model* combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

12.3.2 The Process

The analysis and design process for user interfaces is iterative and can be represented using a process model similar to the one discussed in Chapter 4. Referring to Figure 12.2, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities [Man97]: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 12.2 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

FIGURE 12.2

The user interface design process



Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception (Section 12.4.2) for each class of users.

Once general requirements have been defined, a more detailed *task analysis* is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 12.4.3. Finally, analysis of the user environment focuses on the characteristics of the physical work environment (e.g., location, lighting, position constraints).

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. Interface design is discussed in more detail in Section 12.5.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the user's acceptance of the interface as a useful tool in her work.

As we have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

12.4 USER EXPERIENCE ANALYSIS[2]

A key tenet of all software engineering process models is this: *Understand the problem before you attempt to design a solution.* In the case of user experience design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. In the sections that follow, we examine each of these elements of UX analysis with the intent of establishing a solid foundation for the interface design tasks that follow.

2 It is reasonable to argue that this section should be placed in Chapter 8, since requirements analysis issues are discussed there. It has been positioned here because user experience analysis and design are intimately connected to one another, and the boundary between the two is often fuzzy.

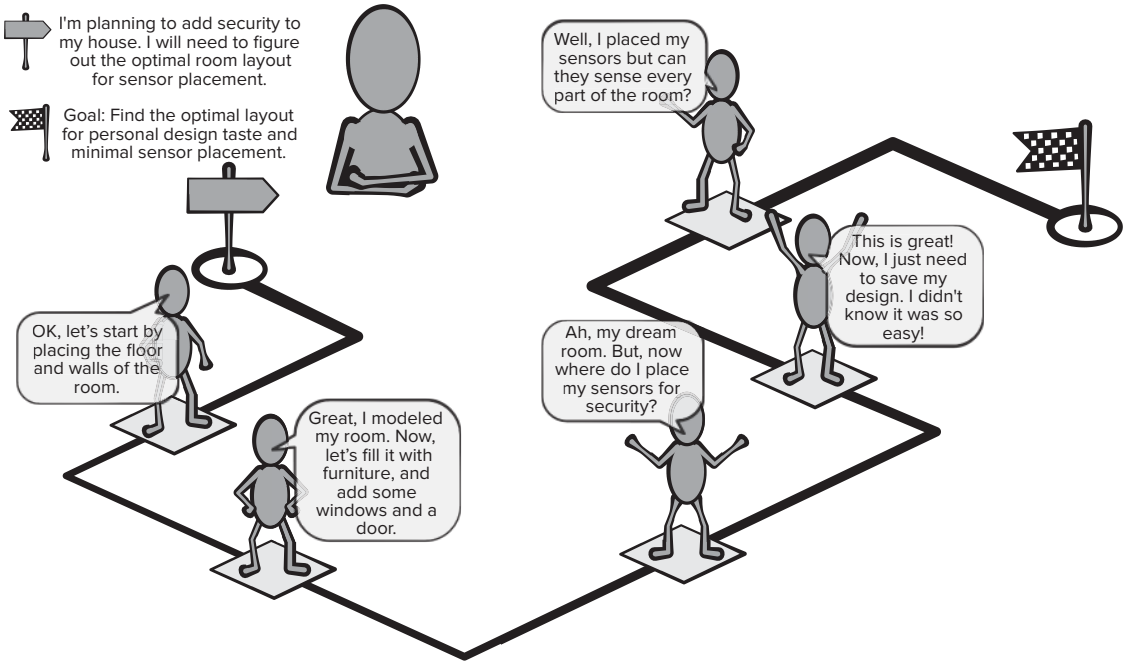
12.4.1 User Research

The phrase *user interface* is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier we noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources (user interviews, sales input, marketing input, support input) can be used to accomplish this.

Many UX developers like to create a *customer journey map* (Figure 12.3) as a means of outlining their goals and plans for the software product. The customer journey map shows how the users will experience the software product as if they were traveling on a physical trip with touchpoints (milestones), obstacles, and ways to monitor their progress. Christensen [Chr13] suggests the following steps to create a customer journey map:

1. **Gather stakeholders.** Locate all affected parties needed to ensure diverse viewpoints are included in the customer journey map.
2. **Conduct research.** Collect all information you can about all the things (thoughts, feelings, actions, motivations, expectations, goals, needs,

FIGURE 12.3 Customer journey map



pain points, barriers, questions) users may experience as they use the software product and define your customer phases. The customer phases will become your touchpoints and are shown as labeled squares in Figure 12.3.

3. **Build the model.** Create a visualization of the touch points (any interaction between the user and product), channels (interaction devices or information streams), and actions taken by the customer (user).
4. **Refine the design.** Recruit a designer to make the deliverable visually appealing and ensure that customer phases are identified clearly.
5. **Identify gaps.** Note any gaps in the customer experience or points of friction or pain (places with information overlap or poor transition between phases).
6. **Implement your findings.** Assign responsible parties to bridge the gaps and resolve pain points found.

12.4.2 User Modeling

In previous chapters, you learned that the user story describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of task analysis, the user story is refined to become a formal use case to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function. When asked, "How do you decide where to put furniture in a room?" an interior designer writes the following informal user story:

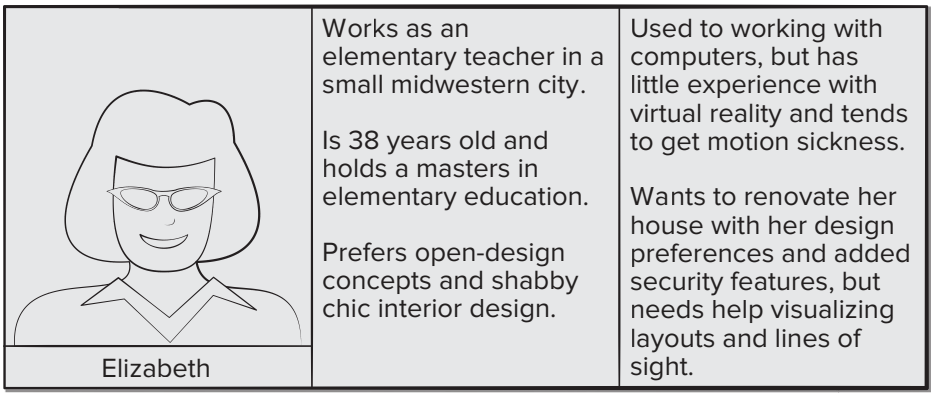
> I begin by sketching the floor plan of the room, the dimensions and the location of windows and doors. I'm very concerned about light as it enters the room, about the view out of the windows (if it's beautiful, I want to draw attention to it), about the running length of an unobstructed wall, about the flow of movement through the room. I then look at the list of furniture my customer and I have chosen. . . . Then, I draw a rendering (a 3-D picture) of the room to give my customer a feel for what it'll look like.

This user story provides a basic description of one important work task for the computer-aided design system from the perspective of one of its users. From it, you can extract tasks, objects, and the overall flow of the interaction. In addition, other features of the system that would please the interior designer might also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through each window. However, if there is more than one type of user, it may be important to define more than one set of user goals for the system described by the user stories.

User experience designers often create fictional user personas to summarize the assumptions made for the different types of users. A *user persona* is a representation of the goals and behavior of a hypothesized group of users. Personas are often

FIGURE 12.4

Persona example



synthesized from data collected during interviews with users. Figure 12.4 shows an example of a user persona. Personas are often used to improve product designers' abilities to see through the eyes of target users [Hil17].

Lene Nielsen [Nie13] describes four tasks that occur during the general process of creating and using personas to guide the user experience design process:

- **Data collection and analysis.** Stakeholders collect as much information about the proposed product users as possible to determine the user groups and begin to emphasize what each group needs.
- **Describe personas.** The developers need to decide how many personas it is reasonable to create, if they will create more than one, and which persona will be their primary focus. The developers create and name each persona including details about their education, lifestyle, values, goals, needs, limitations, desires, attitudes, and behavior patterns.
- **Develop scenarios.** Scenarios are user stories about how personas will use the product being developed. They may focus on the touchpoints and obstacles described in the customer journey. They should show how personas would overcome problems using the system resources if they were the actual users of the system.
- **Acceptance by stakeholders.** Often this is done by validating the scenarios using a review technique or demonstration called *cognitive walkthrough*.[3] Stakeholders assume the role defined by the persona and work through a scenario using a system prototype.

3 A cognitive walkthrough is a method that evaluates whether the order of cues and prompts in a system supports the way people process tasks and anticipates the "next steps" of a system by having users verbalize their decision-making process while using a system representation to complete a user goal.

SAFEHOME



Use Cases for User Interface Design

The scene: Vinod's cubicle, as user interface design continues.

The players: Vinod and Jamie, members of the *SafeHome* software engineering team.

The conversation:

Jamie: I pinned down our marketing contact and had her write a user story for the surveillance interface.

Vinod: From whose point of view?

Jamie: The homeowner, who else is there?

Vinod: There's also the system administrator role. Even if it's the homeowner playing the role, it's a different point of view. The administrator sets the system up, configures stuff, lays out the floor plan, places the cameras . . .

Jamie: All I had her do was play the role of the homeowner when he wants to see video.

Vinod: That's okay. It's one of the major behaviors of the surveillance function interface. But we're going to have to examine the system administrator's behavior as well.

Jamie (irritated): You're right.

(Jamie leaves to find the marketing person. She returns a few hours later.)

Jamie: I was lucky. I found her, and we worked through the administrator's user story together using the homeowner persona. Basically, we're going to define "administration" as one function that's applicable to all other *SafeHome* functions. Here's what we came up with.

(Jamie shows the user story to Vinod.)

User story: I want to be able to set up or edit the system layout at any time. When I set up the system, I select an administration function. It asks me whether I want to do a new setup or whether I want to edit an existing setup. If I select a new setup, the system displays a drawing screen that will enable me to draw the floor plan onto a grid. There will be icons for walls, windows, and doors so that drawing is easy. I just stretch the icons to their appropriate lengths. The system will display the lengths in feet or meters (I can select the measurement system). I can select from a library of sensors and cameras and place them on the floor plan. I get to label each, or the system will do automatic labeling. I can establish settings for sensors and cameras from appropriate menus. If I select edit, I can move sensors or cameras, add new ones or delete existing ones, edit the floor plan, and edit the setting for cameras and sensors. In every case, I expect the system to do consistency checking and to help me avoid mistakes.

Vinod (after reading the scenario): Okay, there are probably some useful design patterns [Chapter 14] or reusable components for GUIs for drawing programs. I'll betcha 50 bucks we can implement some or most of the administrator interface using them.

Jamie: Agreed. I'll check it out.

12.4.3 Task Analysis

The user's goal is to accomplish one or more tasks via the software product. To accomplish this, the software user interface must provide mechanisms that allow the user to achieve her goal. The goal of task or scenario analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?

- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

To answer these questions, you must draw upon techniques that we have discussed earlier in this book, but in this instance, these techniques are applied to the user interface.

In Chapter 8, we discussed stepwise elaboration (also called functional decomposition or stepwise refinement) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach that assists in understanding the human activities the user interface must accommodate.

First, you should define and classify the human tasks that are required to accomplish the goal of the system or app. For example, let's reconsider the computer-aided design system for interior designers discussed earlier. By observing an interior designer at work, you notice that interior design comprises a number of major activities: furniture layout (note the user story discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions, (2) place windows and doors at appropriate locations, (3a) use furniture templates to draw scaled furniture outlines on the floor plan, (3b) use accents templates to draw scaled accents on the floor plan, (4) move furniture outlines and accent outlines to get the best placement, (5) label all furniture and accent outlines, (6) draw dimensions to show location, and (7) draw a perspective-rendering view for the customer. A similar approach could be used for each of the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.[4] The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

12.4.4 Work Environment Analysis

Hackos and Redish [Hac98] discuss work environment analysis this way: "People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people." In some applications the user interface for a computer-based system is placed in a "user-friendly location"

4 However, this may not be the case. The interior designer might want to specify the perspective to be drawn, the scaling, the use of color, and other information. The use case related to drawing perspective renderings would provide the information you need to address this task.

(e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse or touch screen may not be an option, or display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

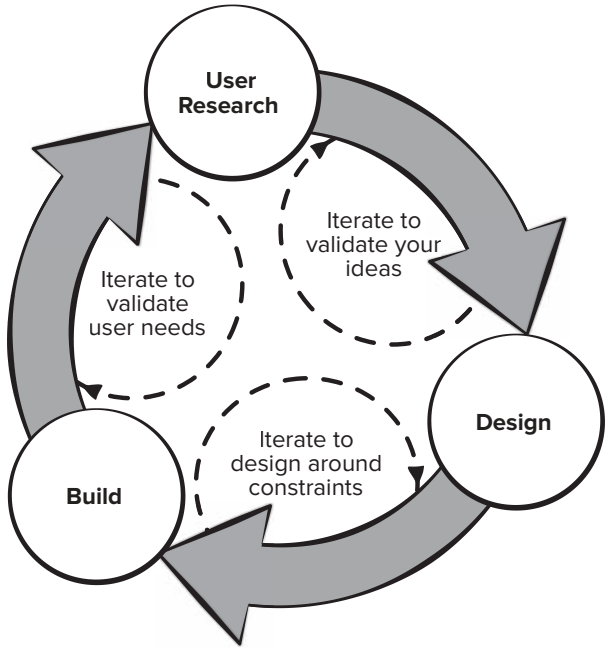
12.5 User Experience Design

As with all iterative processes, there is not a clean division between analysis and design. Figure 12.5 illustrates the practice of cycling between user research and design, as well as cycling between design and construction. The emphasis is on creating incremental prototypes and by testing it with actual users.

Google has defined a 5-day sprint for doing UX design [Kna16], [Goo18], [DXL18]. The steps are outlined below with one day allocated to each:

- **Understand.** Encompasses the user research activities in which the team gathers the information on the problem to be solved (user needs and business goals) for the software product; one way of doing this is to have a series of

FIGURE 12.5
Iterative UX design process



lightning talks (10- to 15-minute presentations) by domain experts on topics such as the business case, competing products, and user profiles. This information is captured on whiteboards (e.g., as customer journey maps, personas, or user task workflow) and remains posted for easy reference throughout the sprint.

- **Sketch.** Individual team members (including all stakeholders) are given the time and space needed to brainstorm solutions to the problems discovered in the understand phase. It is best to do this on paper as quick visual images. Paper drawings and notes are easy to generate, easy to modify, and quite inexpensive. This phase generates lots of ideas because participant ideas are not restricted when creating their solutions.
- **Decide.** Each stakeholder presents his solution sketch, and the team votes to determine the solutions that should be tackled in the prototyping activities that will follow. If there is not a clear consensus following the voting, the development team may decide to consider assumptions that involve constraints posed by the budget, user profiles, available resources (both human and technological), and product business goals.
- **Prototype.** The prototype created during this phase may be a minimally viable product based on the solution selected from the sketch phase, or it may be based on the portions of the customer journey map or storyboard you want to evaluate with potential users in the validate phase. Think of your prototype as an experiment developed to test a hypothesis. That means the team should be developing test cases based on the user stories as the prototype is being built. There is no need to create a fully functional backend for this user interface prototype. It would be best to build a digital prototype using a simple tool (e.g., Keynote[5]). In some cases it may be desirable to create a paper prototype using one of the developers to provide the screen sequences to the users if a prototyping tool is not available.
- **Validate.** Watching users try out your prototype is the best way to discover major issues with its UX design, which in turn lets you start iterating immediately. In a UX design sprint everyone on the development team observes the validation sessions, not just the UX expert or test case designer. This is critical to capturing potential learning opportunities by exposing product decision makers to user feedback in real time. We will talk more about prototype reviews and user testing in Section 12.7.

12.6 User Interface Design

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed in much the same way as described in Chapter 9. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of screen objects and systems actions.

5 See https://keynotopia.com/.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a sensor icon) is dragged and dropped onto a *target object* (e.g., a screen location). The implication of this action is to place a sensor on the room floor plan. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, the code used to simulate the coverage of the room sensors allow for testing of the sensor placement. The code would be associated with each sensor through some type of logical link when the object is placed on the screen, but the code itself is not dragged and dropped via user interaction.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted. If a real-world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

12.6.1 Applying Interface Design Steps

To provide a brief illustration of how design might proceed to create a prototype of a user interface, let's consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary user story (written by the homeowner) for the interface follows:

> **Preliminary user story:** I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my mobile device (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.
>
> To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming *SafeHome.* I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this user story, the following homeowner tasks, objects, and data items are identified:

- *Accesses* the *SafeHome* system
- *Enters* an **ID** and **password** to allow remote access
- *Checks* **system status**
- *Arms* or *disarms SafeHome* system
- *Displays* **floor plan** and **sensor locations**
- *Displays* **zones** on floor plan
- *Changes* **zones** on floor plan
- *Displays* **video camera locations** on floor plan

- *Selects* **video camera** for viewing
- *Views* **video images** (four frames per second)
- *Pans* or *zooms* the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

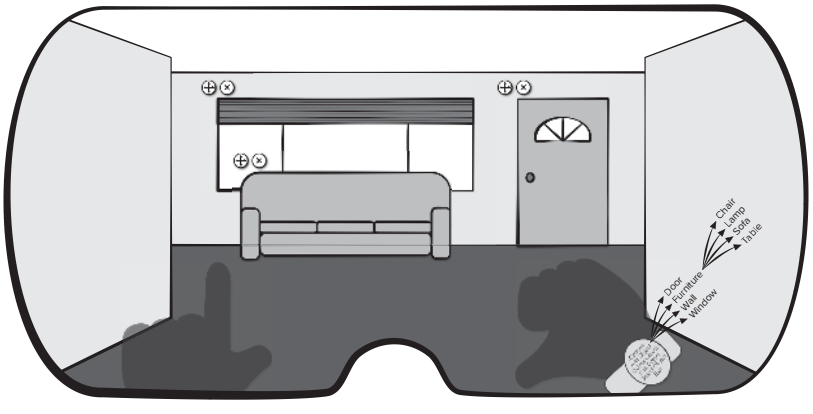
A preliminary sketch of the screen layout for video monitoring is created (Figure 12.6).[6] To invoke the video image, a video camera location icon, *C,* located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper-left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper-left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

12.6.2 User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. A *design pattern* is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

FIGURE 12.6
Preliminary screen layout



6 Note that this differs somewhat from the implementation of these features in earlier chapters. This might be considered a first-draft design based on the new room layout app.

As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

A vast array of interface design patterns has been proposed over the past few decades.[7] A more detailed discussion of user interface design patterns is presented in Chapter 14. In addition, Punchoojit [Pun17] provides a systematic review of many mobile device user interface design patterns (e.g., zooming, lateral access, revealing information in context, control, and conformation).

12.7 Design Evaluation

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

The user interface evaluation cycle takes the form shown in Figure 12.7. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user,[8] who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. We discuss specialized prototype review and testing techniques for graphical user interfaces in Chapter 21.

12.7.1 Prototype Review

The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built?[9] If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model (user stories, storyboard, personas,

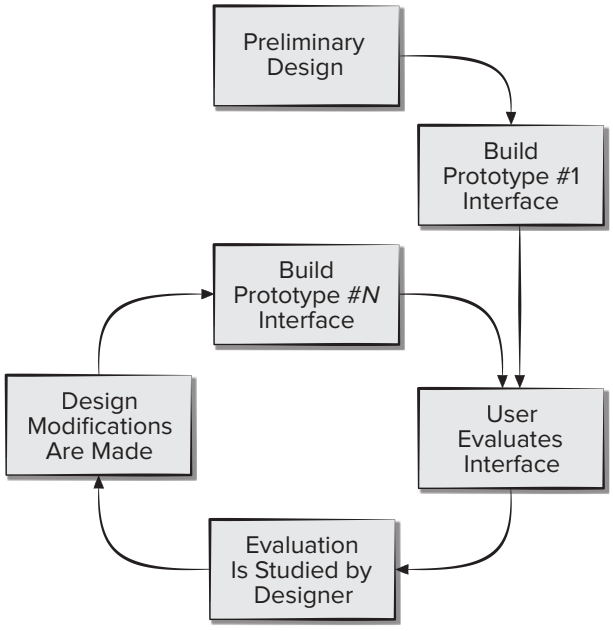
7 Useful suggestions for sites that address user interface patterns can be found at https://www.interaction-design.org/literature/article/10-great-sites-for-ui-design-patterns.

8 It is important to note that experts in ergonomics and interface design may also conduct reviews of the interface. These reviews are called *heuristic evaluations* or *cognitive walk-throughs.*

9 Some software engineers prefer to develop a low-fidelity mockup of the user interface (UI) called a paper prototype to allow stakeholders to test the UI concept before committing any programming resources. The process is described here: http://www.paperprototyping.com/what_examples.html.

FIGURE 12.7

The interface design evaluation cycle



etc.) of the interface has been created, a number of evaluation criteria [Mor81] can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error-handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

SAFEHOME



Interface Design Review

The scene: Doug Miller's office.

The players: Doug Miller, manager of the *SafeHome* software engineering group, and Vinod Raman, a member of the *SafeHome* product software engineering team.

The conversation:

Doug: Vinod, have you and the team had a chance to review the **SafeHomeAssured.com** e-commerce interface prototype?

Vinod: Yeah . . . we all went through it from a technical point of view, and I have a bunch of notes. I e-mailed 'em to Sharon [manager of the WebApp team for the outsourcing vendor for the *SafeHome* e-commerce website] yesterday.

Doug: You and Sharon can get together and discuss the small stuff . . . give me a summary of the important issues.

Vinod: Overall, they've done a good job. Nothing ground breaking, but it's a typical e-commerce interface, decent aesthetics, reasonable layout; they've hit all the important functions . . .

Doug (smiling ruefully): But?

Vinod: Well, there are a few things. . . .

Doug: Such as?

Vinod (showing Doug a sequence of storyboards for the interface prototype): Here's the major functions menu that's displayed on the home page:

Learn about *SafeHome*.

Describe your home.

Get *SafeHome* component recommendations.

Purchase a *SafeHome* system.

Get technical support.

The problem isn't with these functions. They're all okay, but the level of abstraction isn't right.

Doug: They're all major functions, aren't they?

Vinod: They are, but here's the thing . . . you can purchase a system by inputting a list of components . . . no real need to describe the house if you don't want to. I'd suggest only four menu options on the home page:

Learn about *SafeHome*.

Specify the *SafeHome* system you need.

Purchase a *SafeHome* system.

Get technical support.

When you select **Specify the *SafeHome* system you need**, you'll then have the following options:

Select *SafeHome* components.

Get *SafeHome* component recommendations.

If you're a knowledgeable user, you'll select components from a set of categorized pull-down menus for sensors, cameras, control panels, and more. If you need help, you'll ask for a recommendation and that will require that you describe your house. I think it's a bit more logical.

Doug: I agree. Have you talked with Sharon about this?

Vinod: No, I want to discuss this with marketing first; then I'll give her a call.

12.7.2 User Testing

Once the first interactive prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires that allow users to assess the interface prototype can be distributed. If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

We discuss the task of testing virtual environments in more detail in Chapter 21. However, a complete discussion of user interface evaluation methods is beyond the scope of this book. For further information, see [Gao14], [Hus15], [Hac98], and [Sto05].

12.8 USABILITY AND ACCESSIBILITY

Every user interface—whether it is designed for the Web, a mobile device, a traditional software application, a consumer product, or an industrial device—should exhibit the usability characteristics that are described in the sidebar on usability. Dix [Dix99] argues that mobile interfaces should answer three primary questions: *Where am I? What can I do now? Where have I been and where can I go?* Answers to these questions allow a user to understand context and navigate more effectively through the app.

INFO



Usability

In an insightful paper on usability, Larry Constantine [Con95] asks a question that has significant bearing on the subject: "What do users want, anyway?" He answers this way:

> What users really want are good tools. All software systems, from operating systems and languages to data entry and decision support applications, are just tools. End users want from the tools we engineer for them much the same as we expect from the tools we use. They want systems that are easy to learn and that help them do their work. They want software that doesn't slow them down, that doesn't trick or confuse them, that does make it easier to make mistakes or harder to finish the job.

Constantine argues that usability is not derived from aesthetics, state-of-the-art interaction mechanisms, or built-in interface intelligence. Rather, it occurs when the architecture of the interface fits the needs of the people who will be using it.

A formal definition of usability is somewhat illusive. Donahue and his colleagues [Don99] define it in the following manner: "Usability is a measure of how well a computer system . . . facilitates learning; helps learners remember what they've learned; reduces the likelihood of errors; enables them to be efficient, and makes them satisfied with the system."

The only way to determine whether "usability" exists within a system you are building is to conduct usability assessment or testing. Watch users interact with the system and answer the following questions [Con95]:

- Is the system usable without continual help or instruction?
- Do the rules of interaction help a knowledgeable user to work efficiently?
- Do interaction mechanisms become more flexible as users become more knowledgeable?
- Has the system been tuned to the physical and social environment in which it will be used?
- Is the user aware of the state of the system? Does the user know where she is at all times?
- Is the interface structured in a logical and consistent manner?
- Are interaction mechanisms, icons, and procedures consistent across the interface?
- Does the interaction anticipate errors and help the user correct them?
- Is the interface tolerant of errors that are made?
- Is the interaction simple?

If each of these questions is answered yes, it is likely that usability has been achieved.

Among the many measurable benefits derived from a usable system are [Don99]: increased sales and customer satisfaction, competitive advantage, better reviews in the media, better word of mouth, reduced support costs, improved end-user productivity, reduced training costs, reduced documentation costs, and reduced likelihood of litigation from unhappy customers.

12.8.1 Usability Guidelines

The user interface of a software product is its "first impression." Regardless of the value of its content, the sophistication of its processing capabilities and services, and the overall benefit of the application itself, a poorly designed interface will disappoint the potential user and may, in fact, cause the user to go elsewhere. Because of the sheer volume of competing Web and mobile apps in virtually every subject area, the interface must "grab" a potential user immediately.

There are, of course, important differences between conventional and mobile apps. By virtue of the physical constraints imposed by small mobile devices (e.g., smartphones), the mobile interface designer must compress interaction in a focused manner. However, the basic principles discussed in this section continue to apply.

Bruce Tognozzi [Tog01] defines a set of fundamental design principles that lead to better usability:[10]

Anticipation. *An application should be designed so that it anticipates the user's next move.* For example, a user has requested a content object that presents information about a printer driver for a new version of an operating system. The designer of the WebApp should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen directly.

Communication. *The interface should communicate the status of any activity initiated by the user.* Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way).

Consistency. *The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout.* For example, if a mobile app uses a set of four icons (to represent major functions) across the bottom of the display, these icons should appear on every screen and should not be moved to the top of the display. The meaning of the icons should be self-evident within the context of the app.

Controlled Autonomy. *The interface should facilitate user movement throughout the application, but it should do so in a manner that enforces navigation conventions that have been established for the application.* For example, navigation to content requiring controlled access should be controlled by userID and password, and there should be no navigation mechanism that enables a user to circumvent these controls.

Efficiency. *The design of the application and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client-server environment that executes it.* Tognozzi [Tog01] discusses this when he writes: "This simple truth is why it is so important for everyone . . . to appreciate the importance of making user productivity goal one and to understand the vital difference between building an efficient [application] and empowering an efficient user."

---

10 Tognozzi's original principles have been adapted and extended for use in this book. See [Tog01] for further discussion of these principles.

Flexibility. *The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the application in a somewhat random fashion.* In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.

Focus. *The interface (and the content it presents) should stay focused on the user task(s) at hand.* This concept is particularly important for mobile apps that can become very cluttered in the designer's attempts to do too much.

Human Interface Objects. *A vast library of reusable human interface objects has been developed for both Web and mobile apps. Use them.* Any interface object that can be "seen, heard, touched or otherwise perceived" [Tog01] by an end user can be acquired from any one of a number of object libraries.

Latency Reduction. *Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the application should use multitasking in a way that lets the user proceed with work as if the operation has been completed.* In addition to reducing latency, delays must be acknowledged so that the user understands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the application, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providing some entertainment (e.g., an animation or text presentation) while lengthy processing occurs.

Learnability. *An application interface should be designed to minimize learning time and, once learned, to minimize relearning required when the app is revisited.* In general, the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.

Metaphors. *An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user.* Metaphors are an excellent idea because they mirror real-world experience. Just be sure that the metaphor you choose is well known to end users. A metaphor should call on images and concepts from the user's experience, but it does not need to be an exact reproduction of a real-world experience.

Readability. *All information presented through the interface should be readable by young and old.* The interface designer should emphasize readable type styles, user-controllable font sizes, and color background choices that enhance contrast.

Track State. *When appropriate, the state of the user interaction should be tracked and stored so that a user can log off and return later to pick up where he left off.* In general, cookies can be designed to store state information. However, cookies are a controversial technology, and other design solutions may be more palatable for some users.

Visible Navigation. *A well-designed interface provides "the illusion that users are in the same place, with the work brought to them"* [Tog01]. When this approach is used, navigation is not a user concern. Rather, the user retrieves the content object and selects functions that are displayed and executed through the interface.

Nielsen and Wagner [Nie96] suggest a few pragmatic "don'ts" for interface design (based on their redesign of a major WebApp). These provide a nice complement to the principles suggested earlier in this section.

- Don't force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.
- Don't make users scroll unless it is absolutely unavoidable.
- Don't rely on browser functions to assist in navigation.
- Don't allow aesthetics to supersede functionality.
- Don't force the user to search the display to determine how to link to other content or services.

A well-designed interface improves the user's perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound. Additional advice on usability evaluation can be found in [Gao14] and [Hus15}.

12.8.2 Accessibility Guidelines

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. All these can lead to accessibility issues for all users, not just those with special needs. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

Application Accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) that may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines (e.g., [W3C18])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [App13], [Mic13a], and [Zan18]) provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Response Time. System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. *Variability* refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.

Help Facilities. Almost every user of an interactive, computer-based system requires help now and then. Modern software should provide online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

Error Handling. In general, every error message or warning produced by an interactive system should have the following characteristics:

1. Describes the problem in jargon that the user can understand
2. Provides constructive advice for recovering from the error
3. Indicates any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
4. Be accompanied by an audible or visual cue
5. Should never place blame for the error on the user

Menu and Command Labeling. The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and-pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?
- Have appropriate conventions for command usage been established across a family of applications?

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. *Localization* features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM13]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard

[Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

12.9 Conventional Software UX and Mobility

Earlier in this chapter, we noted that all user interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios (use cases) are created and analyzed to define a set of interface objects and actions.

Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are then used to prototype and ultimately implement the interface design model.

When designing for mobility, developers need to pay more attention to differences in screen sizes and user interaction devices. Mobile users of a software product are more likely to expect the product to be easily customized to their preferences and to take advantage of changes in the user's location when they are actively using the app. We will focus on designing for mobile devices in Chapter 13.

12.10 Summary

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power and informational content of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

The development of a user interface begins with a series of analysis tasks. User analysis defines personas for the profiles of various end users and is gathered from a variety of business and technical sources. The user analysis allows the developers to create a customer journey map as a visual representation of the goals for the product. Task analysis defines user tasks and actions using either an elaborative or object-oriented approach, applying use cases, task and object elaboration, workflow analysis, and hierarchical task representations to fully understand the human-computer interaction. Environmental analysis identifies the physical and social structures in which the interface must operate.

After analyzing usage scenarios, interface objects and actions are created and provide a basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. A storyboard may be created to illustrate the navigation through screens developed for the product to

accomplish specific user tasks. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

Like interface design for conventional software, the design mobile app interface describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. A set of interface design principles and an interface design workflow guide the mobile app designer when layout and interface control mechanisms are designed.

The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system. Usability and accessibility issues in the interface may also cause users to find an alternative product that better meets their needs and expectations.

PROBLEMS AND POINTS TO PONDER

12.1. Describe the worst interface that you have ever worked with and critique it relative to the concepts introduced in this chapter. Describe the best interface that you have ever worked with and critique it relative to the concepts introduced in this chapter.

12.2. Consider one of the following interactive applications (or an application assigned by your instructor):

a. A desktop publishing system
b. A computer-aided design system
c. An interior design system (as described in Section 12.4.2)
d. An automated course registration system for a university
e. A library management system
f. An Internet-based polling booth for public elections
g. A home banking system
h. An interactive application assigned by your instructor

Develop a user model, design model, mental model, and an implementation model, for any one of these systems.

12.3. Perform a detailed task analysis for any one of the systems listed in Problem 12.2.

12.4. Create a customer journey map for one of the systems listed in Problem 12.2.

12.5. Continuing Problem 12.2, define interface objects and actions for the application you have chosen. Identify each object type.

12.6. Develop a set of screen layouts and organize them into a storyboard for the system you chose in Problem 12.2.

12.7. Use a prototyping tool like Keynote to create an interactive prototype for the storyboard you created in Problem 12.6.

12.8. Describe your approach to user help facilities for the task analysis design model and task analysis you performed as part of Problems 12.3, 12.4, and 12.5.

12.9. Provide a few examples that illustrate why response time variability can be an issue.

12.10. Develop an approach that would automatically integrate error messages and a user help facility. That is, the system would automatically recognize the error type and provide a help window with suggestions for correcting it. Perform a reasonably complete software design that considers appropriate data structures and algorithms.

12.11. Develop an interface evaluation questionnaire that contains 20 generic questions that would apply to most interfaces. Have 10 classmates complete the questionnaire for an interactive system that you all use. Summarize the results, and report them to your class.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

13 Design for Mobility

Mobile devices—smartphones, tablets, wearable devices, handheld gaming devices, and other specialized products—have become the new face of computing. According to Pew Research Center [Pew18], in the United States 77 percent of people own a smartphone and 50 percent of people own a tablet computer of some kind. Mobile computing has become a dominant force.

Key Concepts

aesthetic design 277
architecture design 278
challenges 265
cloud computing 273
component-level design 282
content architecture 279
content design 277
content objects 277
context-aware apps 274
design
  best practices 285
  mistakes 272
  pyramid 275
  quality 282
graphic design 277
mobile architectures 273
mobile development life cycle 268
model-view-controller 279
navigation design 280
quality checklist 285
technical considerations 266
user interface design 270
WebApp architecture 279

Quick Look

What is it? Mobile design encompasses technical and nontechnical activities that include: establishing the look and feel of the mobile application (including mobile apps, WebApps, virtual reality, and games), creating the aesthetic layout of the user interface, establishing the rhythm of user interaction, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the mobile product.

Who does it? Software engineers, graphic designers, content developers, security specialists, and other stakeholders all participate in the creation of a mobile design model.

Why is it important? Design allows you to create a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end users become involved in large numbers. Design is the place where mobile app quality is established.

What are the steps? Mobile design encompasses six major steps that are driven by information obtained during requirements modeling and are described in this chapter.

What is the work product? A design model that encompasses content, aesthetics, architecture, interface, navigation, and component-level design issues is the primary work product that is produced during mobile design.

How do I ensure that I've done it right? Each element of the design model is reviewed in an effort to uncover errors, inconsistencies, or omissions. In addition, alternative solutions are considered, and the degree to which the current design model will lead to effective implementation on a variety of software platforms and devices is also assessed.

In his authoritative book on Web design, Jakob Nielsen [Nie00] states: "There are essentially two basic approaches to design: the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer." During the first decade of mobile development, the artistic idea was the approach that many developers chose. Design occurred in an ad hoc manner and was usually conducted as HTML was generated. Design evolved out of an artistic vision that evolved as Web page construction occurred.

Even today, many developers use mobile apps as poster children for "limited design." They argue that immediacy and volatility of the mobile market mitigate against formal design; that design evolves as an application is built (coded), and that relatively little time should be spent on creating a detailed design model. This argument has merit, but only for relatively simple apps. When content and function are complex; when the size of the mobile app encompasses hundreds or thousands of content objects, functions, and analysis classes; and when the success of the app will have a direct impact on the success of the business, design cannot and should not be taken lightly. This reality leads us to Nielsen's second approach—"the engineering ideal of solving a problem for a customer."

13.1 THE CHALLENGES

Although mobile devices have many features in common with each other, their users often have very different perceptions of what features they expect to be bundled in each. Some users expect the same features that are provided on their personal computers. Others focus on the freedom that portable devices give them and gladly accept the reduced functionality in the mobile version of a familiar software product. Still others expect unique experiences not possible on traditional computing or entertainment devices. The user's perception of "goodness" might be more important than any of the technical quality dimensions of the mobile product itself.

13.1.1 Development Considerations

Like all computing devices, mobile platforms are differentiated by the software they deliver—a combination of operating system (e.g., Android or iOS) and a small subset of the hundreds of thousands of mobile software products that provide a very wide range of functionality. New tools allow individuals with little formal training to create and sell apps alongside other apps developed by large teams of software developers.

Even though apps can be developed by amateurs, many software engineers think that MobileApps are among the most challenging software systems being built today [Voa12]. Mobile platforms are very complex. Both the Android and iOS operating systems contain over 12 million lines of code. Mobile devices often have mini browsers that will not display the full set of content available on a Web page. Different mobile devices use different operating systems and platform-dependent development environments. Mobile devices tend to have smaller screen sizes and more varied screen sizes than personal computers. This may require greater attention to user interface design issues, including decisions to limit display of some content. MobileApps

must be designed to take intermittent connectivity outages into account, limitations on battery life, and other device constraints[1] [Whi08].

System components in mobile computing environments are likely to change their locations while their apps are running. To maintain connectivity in nomadic networks,[2] coordination mechanisms for discovering devices, exchanging information, maintaining security and communication integrity, and synchronizing actions must be developed. There is always a trade-off between security and other elements of the mobile product design.

In addition, software engineers must identify the proper design trade-offs between the expressive power of the MobileApp and stakeholder security concerns. Developers must seek to discover algorithms (or adapt existing algorithms) that are energy efficient to conserve battery power when possible. Middleware may have to be created to allow different types of mobile devices to communicate with each other in the same mobile networks [Gru00].

Software engineers should craft a user experience that takes advantage of device characteristics and context-aware applications. The nonfunctional requirements (e.g., security, performance, usability) are a bit different from those for either WebApps or desktop software applications. There is always a trade-off between security and other elements of the mobile design. Testing mobile software products (Chapter 21) provides additional challenges because users expect that they will work in a large number of physically different environments. Portability is another challenge for software engineers as there are several popular device platforms. It is expensive to develop and support applications for multiple device platforms [Was10].

13.1.2 Technical Considerations

The low cost of adding Web capabilities to everyday devices such as phones, cameras, and TVs is transforming the way people access information and use network services [Sch11]. Among the many technical considerations that MobileApps should address are the following:

Multiple hardware and software platforms. It is not at all unusual for a mobile product to run on many different platforms (both mobile and stationary) with a range of differing levels of functionality. The reasons for these differences are in part because the hardware and software available are quite different from device to device. This increases both development cost and time. It also can make configuration management (Chapter 22) more difficult.

Many development frameworks and programming languages. Mobile products are currently being written in several distinct programming or scripting languages (e.g., HTML5, JavaScript, Java, Swift, and C#) for a multitude of popular development frameworks (e.g., Android, iOS, Xamarin, Windows, AngularJS). Very few mobile devices allow direct development on a device itself. Instead, mobile developers typically use emulators running on desktop

1 Available at http://www.devx.com/SpecialReports/Article/37693.

2 *Nomadic networks* have changing connections to mobile devices or servers.

development systems. These emulators may or may not accurately reflect the limitations of the device itself. Thin-client applications are often easier to port to multiple devices than applications designed to run exclusively on the mobile device.

Many app stores with different rules and tools. Each mobile platform has its own app store and its own standards for accepting apps (e.g., Apple,[3] Google,[4] Microsoft,[5] and Amazon[6] publish their own standards). Development of a mobile product for multiple platforms must proceed separately, and each version of the app needs its own standards expert.

Very short development cycles. The marketplace for mobile products is very competitive, and software engineers are likely to make use of agile development processes when building MobileApps in an effort to reduce development time [Was10].

User interface limitations and complexities of interaction with sensors and cameras. Mobile devices have smaller screen sizes than personal computers and a richer set of interaction possibilities (touch, gesture, camera, etc.) and usage scenarios based on context awareness. The style and appearance of the user interface is often dictated by the nature of platform-specific development tools [Rot02]. Allowing smart devices to interact with smart spaces offers the potential to create personalized, networked, high-fidelity application platforms such as those seen by merging smartphones and car infotainment systems.[7]

Effective use of context. Users expect MobileApps to deliver personalized user experiences based on the physical location of a device in relation to the available network features. User interface design and context-aware applications are discussed in greater detail in Section 13.4.

Power management. Battery life is often one of the most limiting constraints on many mobile devices. Backlighting, reading and writing to memory, using wireless connections, making use of specialized hardware, and processor speed all impact power usage and need to be considered by software developers [Mei09].

Security and privacy models and policies. Wireless communication is difficult to protect from eavesdropping. Preventing *man-in-the-middle-attacks*[8] in automotive applications can be critical to the safety of the users [Bos11]. Data stored on a mobile device are subject to theft if a device is lost or a malicious app is downloaded. Software policies that increase the level of confidence in the security and privacy of a MobileApp often reduce the usability of the app and the spontaneity of the communication among users [Rot02].

3 https://developer.apple.com/appstore/guidelines.html.
4 http://developer.android.com/distribute/googleplay/publish/preparing.html.
5 http://msdn.microsoft.com/en-us/library/ff941089%28v=vs.92%29.aspx.
6 https://developer.amazon.com/apps-and-games/app-submission/android.
7 When used in an automotive setting, smart devices should be able to restrict access to services that may distract the driver and allow hands-free operation when a vehicle is moving [Bos11].
8 These attacks involve a third party intercepting communications between two trusted sources and impersonating one or both of the parties.

Computational and storage limitations. There is great interest in using mobile devices to control home environmental and security services. When MobileApps are allowed to interact with devices and services in their environment, it is easy to overwhelm the mobile device (storage, processing speed, power consumed) with the sheer volume of information [Spa11]. Developers may need to look for programming shortcuts and means of reducing the demands made on processor and memory resources.

Applications that depend on external services. Building thin mobile clients suggests the need to rely on Web service providers and cloud storage facilities. This increases concerns for both data or service accessibility and security [Rot02].

Testing complexity. Mobile products that run entirely on the device can be tested using traditional software testing methods (Chapters 19 and 20) or using emulators running on personal computers. Thin-client MobileApps are particularly challenging to test. They exhibit many of the same testing challenges found in WebApps, but they have the additional concerns associated with transmission of data through Internet gateways and telephone networks [Was10]. Testing of mobile software products will be discussed in Chapter 21.

13.2 Mobile Development Life Cycle

Burns [Bur16] and her Microsoft colleagues describe a recommendation for an iterative mobile SDLC that contains five major stages:

Inception. Goals, features, and functions of the mobile product are identified to determine the scope and the size of the first increment or feasibility prototype. Developers and stakeholders must be conscious of human, social, cultural, and organizational activities that may reveal hidden aspects of the users' needs and affect the business targets and functionality of the proposed mobile product.

Design. The design includes architectural design, navigation design, interface design, content design. Developers define the app user experience using screen mockups and paper prototypes to help create a proper user interface design that will take different screen sizes and capabilities into account as well as the capabilities of each targeted platform.

Development. Mobile software is coded, functional and nonfunctional. Test cases are created and executed, and usability and accessibility evaluations are conducted as the product evolves.

Stabilization. Most mobile products go through a series of prototypes: feasibility prototype, intended as a proof of concept with perhaps only one complete logic path through the application; alpha prototype, which contains the functionality for minimum viable product; beta prototype, which is largely complete and contains most tested functionality; and lastly the release candidate, which contains all required functionality, for which all scheduled tests have been completed, and which is ready for review by the product owner.

Deployment. Once stabilized, a mobile product is reviewed by a commercial app store and made available for sale and download. For apps intended for internal company use only, a product owner review may be all that is required before deployment.

Mobile development makes use of an agile, spiral engineering process model. The stages are not completed in order like they would be if mobile development was done using the waterfall model. The stages described above are visited repeatedly as developers and stakeholders gain better understanding of the user needs and product business goals.

SAFEHOME



Formulating Mobile Device Requirements

The scene: A meeting room. The first meeting to identify requirements for a mobile version of the *SafeHome* WebApp.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function present in the WebApp.

Vinod (interrupting): My understanding is that people want *SafeHome* functionality to be accessible from mobile devices as well . . . including the home security function?

Marketing person: Yes, that's right . . . we'll have to add that functionality and try to make it context aware to help personalize the user experience.

Facilitator: Context aware in what sense?

Marketing person: People might want to use a smartphone instead of the control panel and avoid logging on to a website when they are in the driveway at home. Or they might not want all family members to have access to the master control dashboard for the system from their phones.

Facilitator: Do you have specific mobile devices in mind?

Marketing person: Well, all smartphones would be nice. We will have a Web version done, so won't the MobileApp run on all of them?

Jamie: Not quite. If we took a mobile phone browser approach, we might be able to reuse a lot of our WebApp functionality. But remember, smartphone screen sizes vary, and they may or may not all have the same touch capabilities. So, at the very least we would have to create a mobile website that takes the features of each device into account.

Ed: Perhaps we should build the mobile version of the website first.

Marketing person: OK, but a mobile website solution wasn't what we had in mind.

Vinod: Each mobile platform seems to have its own unique development environment, too.

Production rep: Can we restrict MobileApp development to only one or two types of smartphones?

Marketing person: I think that might work. Unless I'm mistaken, the smartphone market is dominated by two smartphone platforms right now.

Jamie: There's also security to worry about. We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Also, a phone could get lost or stolen more easily than a laptop.

Doug: Very true.

Marketing: But we still need the same level of security . . . just also be sure to stop an outsider from getting in with a stolen phone.

Ed: That's easier said than done and . . .

Facilitator (interrupting): Let's not worry about those details yet.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: As a starting point, can we identify which elements of WebApp security function are needed in the MobileApp and which will need to be newly created? Then we can decide how many mobile platforms we can support and when we can move forward on this project.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

13.2.1 User Interface Design

Mobile device users expect that minimal learning time will be required to master a MobileApp. To achieve this, MobileApp designers use consistent icon representations and placement across multiple platforms. In addition, designers must be sensitive to the user's expectation of privacy with regard to the display of personal information on the screen of the mobile device. Touch and gesture interfaces, along with sophisticated voice input and facial recognition, are maturing rapidly [Shu12] and have already become part of the user interface designer's toolbox.

Legal and ethical pressure to provide for access by all persons suggests that mobile device interfaces need to account for brand differences, cultural differences, differences in computing experience, elderly users, and users with disabilities (e.g., visual, aural, mobility). The effects of poor usability may mean that users cannot complete their tasks or will not be satisfied with the results. This suggests the importance of user-centered design activities in each of the usability areas (user interface, external accessory interface, and service interface). Accessibility is an important design issue and must be considered when user-centered design is applied.

In trying to meet stakeholder usability expectations, MobileApp developers should attempt to answer these questions to assess the out-of-the-box readiness of the device:

- Is the user interface consistent across applications?
- Is the device interoperable with different network services?
- Is the device acceptable in terms of stakeholder values[9] in the target market area?

Eisenstein [Eis01] claims that the use of abstract, platform-neutral models to describe a user interface greatly facilitates the development of consistent, usable multiplatform user interfaces for mobile devices. Three models in particular are useful. A *platform model* describes the constraints imposed by each platform to be supported. A *presentation model* describes the appearance of the user interface. The *task model* is a structured representation of the tasks a user needs to perform to meet her task goals. In the best case, model-based design (Chapter 9) involves the creation of databases that contain the

9 Brand, ethical preferences, moral preferences, cognitive beliefs.

models and has tool support for generating user interfaces for multiple devices automatically. Utilizing model-based design techniques can also help designers recognize and accommodate the unique contexts and context changes that are present in mobile computing. Without an abstract description of a user interface, the development of mobile user interfaces can be error prone and time consuming.

INFO



Mobile User Interface Design Considerations

Design choices affect performance and should be examined early in the user interface design process. Ivo Weevers [Wee11] posted several mobile user–interface design practices that have proven to be helpful when designing mobile applications:

- **Define user interface brand signatures.** Differentiate the app from its competitors. Make the core signature elements of the brand the most responsive, because users will use them over and over.
- **Focus the portfolio of products.** Target the platforms that are most important to the success of the app and the company. Not all platforms have the same number of users.
- **Identify the core user stories.** Make use of techniques that require stakeholders to prioritize their needs as a way to reduce a lengthy list of requirements and to consider the constrained resources available on mobile devices.
- **Optimize user interface flows and elements.** Users do not like to wait. Identify potential bottlenecks in user work flow and make sure the user is given an indication of progress when delays occur. Make sure that the time to display screen elements is justified in terms of user benefits.
- **Define scaling rules.** Determine the options that will be used when information to be displayed is too large to fit on the screen. Managing functionality, aesthetics, usability, and performance is a continual balancing act.
- **User performance dashboard.** The dashboard is used to communicate the product's current state of completion (e.g., number of use stories implemented), its performance relative to its targets, and perhaps comparisons to its competitors.
- **Champion-dedicated user interface engineering skills.** It is important to understand that the implementation of layout, graphics, and animation has performance implications. Techniques to interleave rendering of display items and program execution can be helpful.

13.2.2 Lessons Learned

de Sá and Carriço [Des08] contend that there are important differences between developing conventional software and developing mobile applications. Software engineers cannot continue to use the same conventional techniques they have used and expect them to be successful. They suggest three approaches for the design of mobile applications:

Usage Scenarios. Described in Chapter 12, usage scenarios must consider context variables (location, user, and device) and transitions between contextual scenarios (e.g., user moves from bedroom to kitchen or switches from stylus to a finger). de Sá and Carriço have identified a set of scenario-variable types that should be considered in developing the user scenarios—locations and settings, movement and posture, devices and usages, workloads and distractions, user preferences.

Ethnographic Observation.[10] This is a widely used method for gathering information about representative users of a software product as it is being designed. It is often difficult to observe users as they change contexts, because the observer must follow users for long periods of time, something that could raise privacy concerns.[11] A complicating factor is that users seem to complete tasks differently in private settings than in social settings. The same users may need to be observed performing tasks in multiple contexts while monitoring transitions, as well as recording user reactions to the changes.

Low-Fidelity Paper Prototypes (e.g., cards or Post-it notes). This is a cost-effective usability assessment approach in user interface design that can be used before any programming takes place. It is important for these prototypes to be similar in size and weight and for their use to be allowed in a variety of contexts. It is also important that the sketches or text displays be true to size and for the final product to be of high quality. Placement and size of user interface widgets (e.g., buttons or scrollbars) must be designed so that they will not disappear when users extend their screens by zooming. The interaction type (e.g., stylus, joy stick, touch screen) needs to be emulated in the low-fidelity prototype (e.g., colored pen or push pin) to check placement and ease of use. Later prototypes may then be created to run on the targeted mobile devices once the layout and placement issues have been resolved.

INFO



MobileApp Design Mistakes

Joh Koester [Koe12] posts several examples of mobile design practices that should be avoided:

- **Kitchen sink.** Avoid adding too many features to the app and too many widgets on the screen. Simple is understandable. Simple is marketable.
- **Inconsistency.** To avoid this, set standards for page navigation, menu use, buttons, tabs, and other user-interface elements. Stick to a uniform look and feel.
- **Overdesigning.** Be ruthless when designing apps. Remove unnecessary elements and wasteful graphics. Do not be tempted to add elements just because you think you should.
- **Lack of speed.** Users do not care about device constraints—they want to view things quickly. Preload what you can. Eliminate what is not needed.
- **Verbiage.** Unnecessarily long, wordy menus and screen displays are indications of a mobile product that has not been tested with users and developers who have not spent enough time understanding the user's task.
- **Nonstandard interaction.** One reason for targeting a platform is to take advantage of the user's experience with the way things are done on that platform. Where standards exist use them. This needs to be balanced with the need to have an application appear and behave the same way on multiple devices when possible.
- **Help-and-FAQ-itis.** Adding online help is not the way to repair a poorly designed user interface. Make sure you have tested your app with your targeted users and repaired the identified defects.

10 *Ethnographic observation* is a means determining the nature of user tasks by watching users in their work environment.

11 Asking users to fill out anonymous questionnaires may have to suffice when direct observation is not possible.

13.3 MOBILE ARCHITECTURES

Services computing[12] and cloud computing[13] enable the rapid development of large-scale distributed applications based on innovative architectural designs [Yau11]. These two computing paradigms have made it easier and more economical to create applications on many different mobile devices (e.g., laptop computers, smartphones, and tablets). These two paradigms allow resource outsourcing and transfer of information technology management to service providers while at the same time mitigating the impact of resource limitations on some mobile devices. A service-oriented architecture provides the architectural style (e.g., REST),[14] standard protocols (e.g., XML[15] and SOAP[16]), and interfaces (e.g., WSDL)[17] needed for mobile development. Cloud computing enables convenient, on-demand network access to a shared pool of configurable computing resources (servers, storage, applications, and services).

Service computing allows mobile developers to avoid the need to integrate service source code into the client running on a mobile device. Instead, the service runs out of the provider's server and is loosely coupled with the applications that make use of it through messaging protocols. A service typically provides an application programming interface (API) to allow it to be treated like an abstract black box.

Cloud computing lets the client (either a user or program) request computing capabilities as needed, across network boundaries anywhere or any time. The cloud architecture has three layers, each of which can be called as a service (Figure 13.1). The *software as service* layer consists of software components and applications hosted by third-party service providers. The *platform as service* layer provides a collaborative development platform to assist with design, implementation, and testing by geographically distributed team members. *Infrastructure as a service* provides virtual computing resources (storage, processing power, network connectivity) on the cloud.

Mobile devices can access cloud services from any location at any time. The risks of identity theft and service hijacking require providers of mobile services and cloud computing to employ rigorous security engineering techniques (Chapter 18) to protect their users.

Taivalsaari [Tai12] points out that making use of cloud storage can allow any mobile device or software features to be updated easily on millions of devices worldwide. In fact, it is possible to virtualize the entire mobile user experience so that all applications are downloaded from the cloud.

12 *Services computing* focuses on architectural design and enables application development through service discovery and composition.

13 *Cloud computing* focuses on the effective delivery of services to users through flexible and scalable resource virtualization and loading balancing.

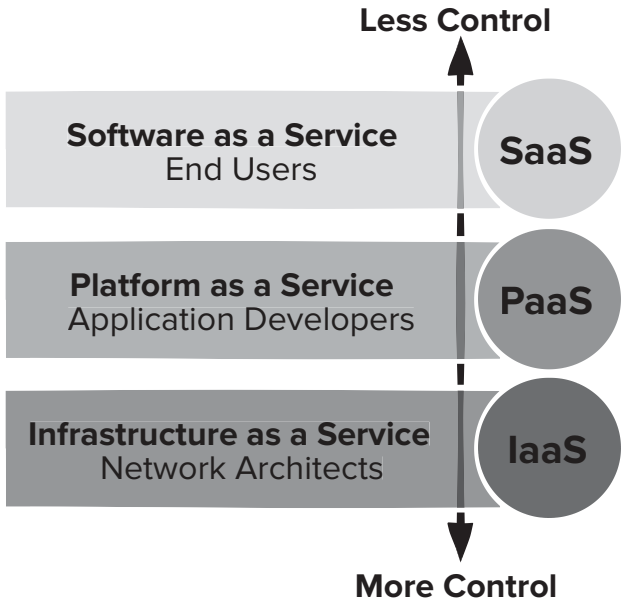
14 *Representation State Transfer* describes a networked Web architectural style where the resource representation (e.g., a Web page) places the client in a new state. The client changes or transfers state with each resource representation.

15 *Extensible Markup Language* (XML) is designed to store and transport data, while HTML is designed to display data.

16 *Simple Object Access Protocol* is a specification for exchanging structured information in the implementation of Web services in computer networks.

17 *Web Services Description Language* is an XML-based language for describing Web services and how to access them.

FIGURE 13.1
Cloud computing layers



13.4 CONTEXT-AWARE APPS

Context allows the creation of new applications based on the location of the mobile device and the functionality to be delivered by the device. Context can also help tailor personal computer applications for mobile devices (e.g., downloading patient information to a device carried by a home health care worker as he arrives at the patient's house).

Using highly adaptive, contextual interfaces is a good way to deal with device limitations (e.g., screen size and memory). To facilitate the development of context-aware user interaction requires the support of corresponding software architectures.

In an early discussion of context-aware applications, Rodden [Rod98] points out that mobile computing merges the real and virtual world by providing functionality that allows a device to be aware of its location, time, and other objects in its surroundings. The device could be in a fixed location like an alarm sensor, embedded in an autonomous device, or be carried around by a human. Because the device can be designed to be used by individuals, groups, or the public, it must detect the presence and identity of the user, as well as the attributes of the context that are relevant or permitted for that user (even if the user is another device).

To achieve context awareness, mobile systems must produce reliable information in the presence of uncertain and rapidly changing data from a variety of heterogeneous sources. Extracting relevant context information by combing data from several sensors proves challenging because of problems with noise, miscalibration, wear and tear, and weather. Event-based communication is preferable to the management of continuous streams of high-abstraction-level data in context-aware applications [Kor03].

In ubiquitous computing environments, multiple users work with a wide range of different devices. The configuration of the devices should be flexible enough to change frequently because of the demands of mobile work practices. It is important for the software infrastructure to support different styles of interaction (e.g., gestures, voice, and pen) and store them in abstractions that can be shared easily.

There are times when one user may desire to work with more than one device simultaneously on the same product (e.g., use a touch-screen device to edit a document image and a personal keyboard to edit document text). It is challenging to integrate mobile devices that are not always connected to the network and have a variety of device constraints [Tan01]. Networked, multiplayer games have had to deal with these problems by storing the game state on each device and sharing change information among other game players' devices in real time.

13.5 WEB DESIGN PYRAMID

What is design in the context of Web engineering? This simple question is more difficult to answer than one might believe. Pressman and Lowe [Pre08] discuss this when they write:

> The creation of an effective design will typically require a diverse set of skills. Sometimes, for small projects, a single developer may need to be multi-skilled. For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists: Web engineers, graphic designers, content developers, programmers, database specialists, information architects, network engineers, security experts, and testers. Drawing on these diverse skills allows the creation of a model that can be assessed for quality and improved *before* content and code are generated, tests are conducted, and end-users become involved in large numbers. If analysis is where *WebApp quality is established,* then design is where the *quality is truly embedded.*

The appropriate mix of design skills will vary depending upon the nature of the WebApp. Figure 13.2 depicts a design pyramid for WebApps. Each level of the pyramid represents a design action that is described in the sections that follow.

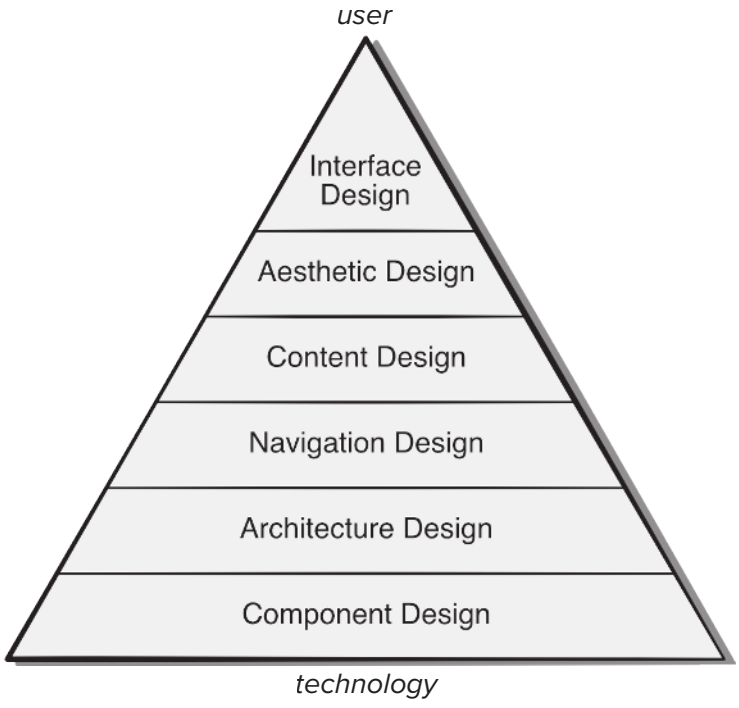
13.5.1 WebApp Interface Design

When a user interacts with a computer-based system, a set of fundamental principles and overriding design guidelines apply. These have been discussed in Chapter 12.[18] Although WebApps present a few special user interface design challenges, the basic principles and guidelines are applicable.

One of the challenges of interface design for WebApps is the indeterminate nature of the user's entry point. That is, the user may enter the WebApp at a "home" location (e.g., the home page) or may be linked into some lower level of the WebApp architecture. In some cases, the WebApp can be designed in a way that reroutes the user to a home location, but if this is undesirable, the WebApp design must provide interface navigation features that accompany all content objects and are available regardless of how the user enters the system.

18 Section 12.1 is dedicated to the user interface design portion of user experience design. If you have not already done so, read it at this time.

FIGURE 13.2
A design pyramid for WebApps



The objectives of a WebApp interface are to: (1) establish a consistent window into the content and functionality provided by the interface, (2) guide the user through a series of interactions with the WebApp, and (3) organize the navigation options and content available to the user. To achieve a consistent interface, you should first use visual design (Section 12.1) to establish a coherent "look." This encompasses many characteristics but must emphasize the layout and form of navigation mechanisms. To guide user interaction, you may draw on an appropriate metaphor[19] that enables the user to gain an intuitive understanding of the interface. To implement navigation options, you can select *navigation menus* positioned consistently on Web pages, *graphic icons* represented in a manner that enable a user to recognize that the icon is a navigation element, and/or *graphic images* that provide a link to a content object or WebApp functionality. It is important to note that one or more of these navigation mechanisms should be provided at every level of the content hierarchy.

Every Web page has a limited amount of "real estate" that can be used to support nonfunctional aesthetics, navigation features, informational content, and user-directed functionality. The development of this real estate is planned during aesthetic design.

19 In this context, a *metaphor* is a representation (drawn from the user's real-world experience) that can be modeled within the context of the interface. A simple example might be a slider switch that is used to control the auditory volume of an .mp4 file.

13.5.2 Aesthetic Design

Aesthetic design, also called visual design or *graphic design,* is an artistic endeavor that complements the technical aspects of WebApp design. We discussed visual design in Section 12.1.4. Page layout is one aspect of aesthetic design that can affect the usefulness (and usability) of a WebApp.

There are no absolute rules when a Web page layout is designed. However, a number of general layout guidelines are worth considering:

Don't be afraid of open space. It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and create visual chaos that is not pleasing to the eye.

Emphasize content. After all, that's the reason the user is there. Nielsen [Nie00] suggests that the typical Web page user should be 80 percent content with the remaining real estate dedicated to navigation and other features.

Organize layout elements from top left to bottom right. The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top left to bottom right.[20] If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.

Group navigation, content, and function geographically within the page. Humans look for patterns in virtually all things. If there are no discernible patterns within a Web page, user frustration is likely to increase (owing to unnecessary searching for needed information).

Don't extend your real estate with the scrolling bar. Although scrolling is often necessary, most studies indicate that users would prefer not to scroll. It is often better to reduce page content or to present necessary content on multiple pages.

Consider resolution and browser window size when designing layout. Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of available space [Nie00]. With the growing use of mobile devices with different screen sizes, this concept becomes increasingly important.

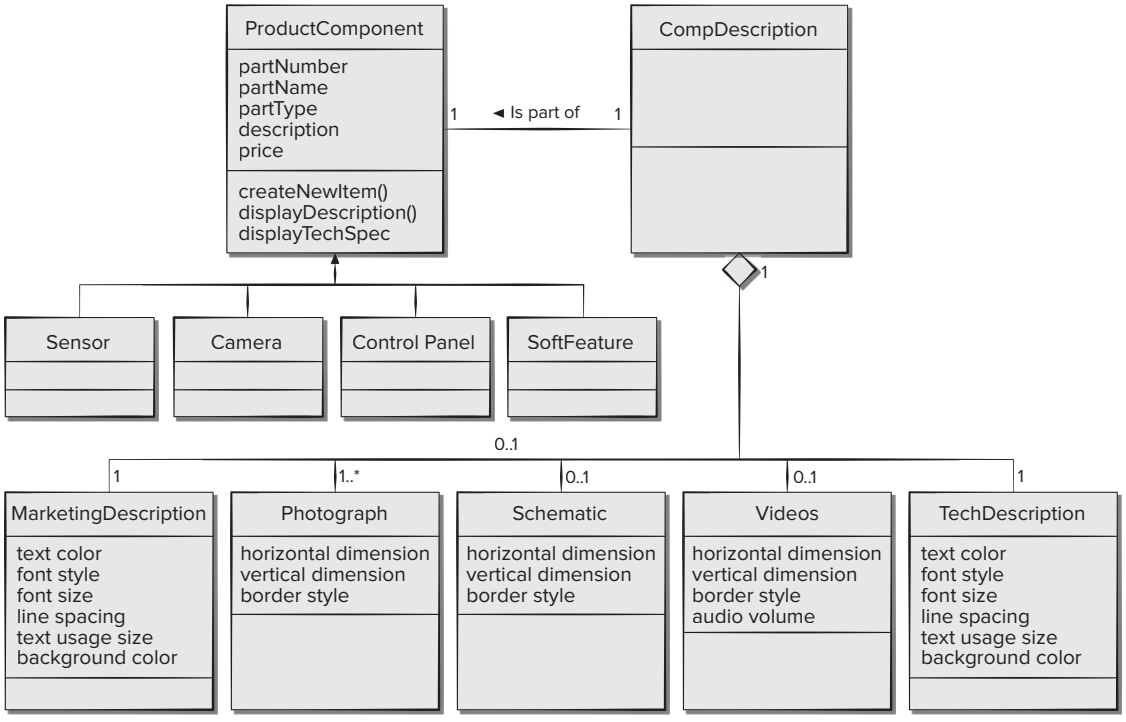
13.5.3 Content Design

We introduced content design in Section 12.1.1. In WebApp design, a content object is more closely aligned with a data object for traditional software. A *content object* has attributes that include content-specific information (normally defined during WebApp requirements modeling) and implementation-specific attributes that are specified as part of design.

As an example, consider an analysis class, **ProductComponent**, developed for the *SafeHome* e-commerce system. The analysis class attribute, `description`, is represented

20 There are exceptions that are cultural and language based, but this rule holds for most users.

FIGURE 13.3 Design representation of content objects



as a design class named **CompDescription** composed of five content objects: **MarketingDescription, Photograph, TechDescription, Schematic,** and **Videos** shown as the bottom row of shaded objects noted in Figure 13.3. Information contained within the content object is noted as attributes. For example, **Photograph** (a jpg image) has the attributes `horizontal dimension`, `vertical dimension`, and `border style`.

UML association and an aggregation[21] may be used to represent relationships between content objects. For example, the UML association shown in Figure 13.3 indicates that one **CompDescription** is used for each instance of the **ProductComponent** class. **CompDescription** is composed on the five content objects shown. However, the multiplicity notation shown indicates that **Schematic** and **Videos** are optional (zero occurrences are possible), one **MarketingDescription** and one **TechDescription** are required, and one or more instances of **Photograph** are used.

13.5.4 Architecture Design

Architecture design is tied to the goals established for a WebApp, the content to be presented, the users who will visit, and the navigation philosophy that has been established. As an architectural designer, you must identify content architecture and

21 Both of these representations are discussed in Appendix 1.

WebApp architecture. *Content architecture*[22] focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation. *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design. Because the WebApp architecture may have a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design.

WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives. Jacyntho and his colleagues [Jac02b] describe the basic characteristics of this infrastructure in the following manner:

> Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages).

The authors suggest a three-layer design architecture that decouples the interface from navigation and from application behavior. They argue that keeping the interface, application, and navigation separate simplifies implementation and enhances reuse.

The *Model-View-Controller* (MVC) architecture [Kra88][23] is a popular WebApp architectural model that decouples the user interface from the WebApp functionality and information content. The *model* (sometimes referred to as the "model object") contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific. The *view* contains all interface-specific functions and enables the presentation of content and processing logic, including all content objects, access to external data and information sources, and all processing functionality required by the end user. The *controller* manages access to the model and the view and coordinates the flow of data between them. In a WebApp, "the view is updated by the controller with data from the model based on user input" [WMT02]. A schematic representation of the MVC architecture is shown in Figure 13.4.

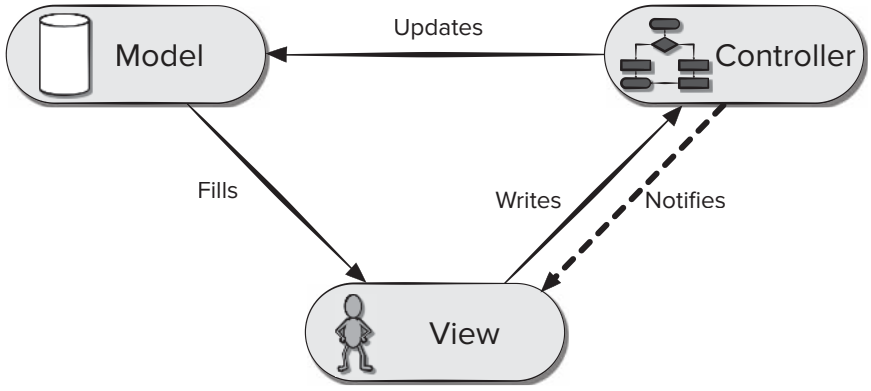
Referring to the figure, user requests or data are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine.

---

22 The term *information architecture* is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.

23 It should be noted that MVC is actually an architectural design pattern developed for the Smalltalk environment (see www.smalltalk.org) and can be used for any interactive application.

FIGURE 13.4
The MVC architecture



In many cases, WebApp architecture is defined within the context of the development environment in which the application is to be implemented. If you have further interest, see [Fow03] for a discussion of development environments and their role in the design of Web application architectures.

13.5.5 Navigation Design

Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, identify the semantics of navigation for different users of the site, and define the mechanics (syntax) of achieving the navigation.

Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases (Chapter 8) developed for each category of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. In addition, the use cases developed for each actor will define a set of classes that encompass one or more content objects or WebApp functions. As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements" [Cac02]. An NSU describes the navigation requirements for each use case. In essence, the NSU shows how an actor moves between content objects or WebApp functions.

An NSU is composed of a set of navigation elements called *ways of navigating* (WoN) [Gna99]. A WoN represents the best navigation pathway to achieve a navigational goal for a specific type of user. Each WoN is organized as a set of *navigational nodes* (NN) that are connected by navigational links. In some cases, a navigational link may be another NSU. Therefore, the overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

To illustrate the development of an NSU, consider the use case **Select SafeHome Components:**

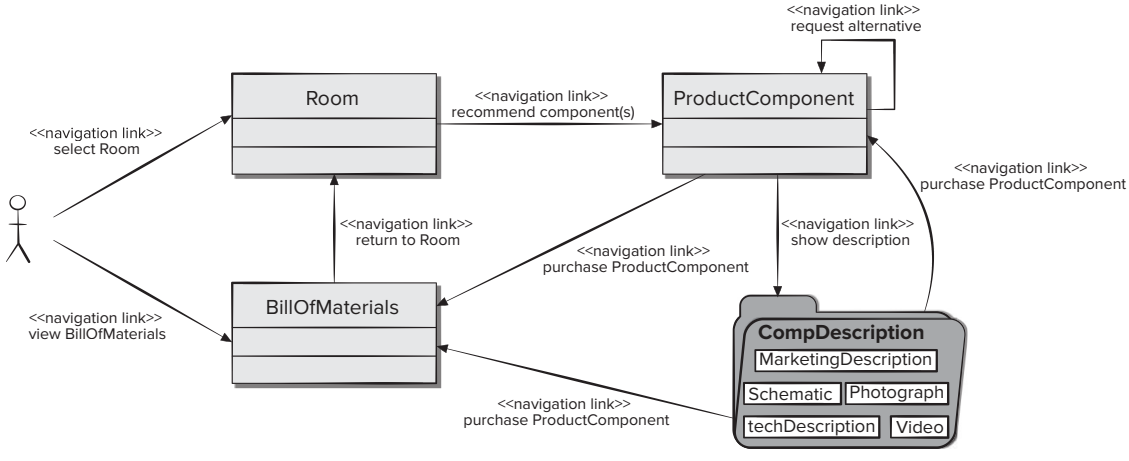
Use Case: *Select SafeHome Components*

The WebApp will recommend product components (e.g., control panels, sensors, cameras) and other features (e.g., PC-based functionality implemented in software) for each room and exterior entrance. If I request alternatives, the WebApp will provide them, if they exist. I will be able to get descriptive and pricing information for each product component. The WebApp will create and display a bill-of-materials as I select various components. I'll be able to give the bill-of-materials a name and save it for future reference (see use case **Save Configuration**).

The underlined items in the use case description represent classes and content objects that will be incorporated into one or more NSUs that will enable a new customer to perform the scenario described in the **Select SafeHome Components** use case.

Figure 13.5 depicts a partial semantic analysis of the navigation implied by the **Select SafeHome Components** use case. Using the terminology introduced earlier, the figure also represents a way of navigating (WoN) for the **SafeHomeAssured.com** WebApp. Important problem domain classes are shown, along with selected content objects (in this case the package of content objects named `CompDescription`, an attribute of the **ProductComponent** class). These items are navigation nodes. Each of the arrows represents a navigation link[24] and is labeled with the user-initiated action that causes the link to occur.

FIGURE 13.5 Creating an NSU



24 These are sometimes referred to as *navigation semantic links* (NSLs) [Cac02].

You can create an NSU for each use case associated with each user role. For example, a **new customer** for **SafeHomeAssured.com** may have three different use cases, all resulting in access to different information and WebApp functions. An NSU is created for each goal.

During the initial stages of navigation design, the WebApp content architecture is assessed to determine one or more WoN for each use case. As noted earlier, a WoN identifies navigation nodes (e.g., content) and then links that enable navigation between them. The WoN are then organized into NSUs.

As design proceeds, your next task is to define the mechanics of navigation. Most websites make use of one or more of the following navigation options for implementing each NSU: individual navigation links, horizontal or vertical navigation bars (lists), tabs, or access to a complete site map. If a site map is defined, it should be accessible from every page. The map itself should be organized so that the structure of WebApp information is readily apparent.

In addition to choosing the mechanics of navigation, you should also establish appropriate navigation conventions and aids. For example, icons and graphical links should look "clickable" by beveling the edges to give the image a three-dimensional look. Audio or visual feedback should be designed to provide the user with an indication that a navigation option has been chosen. For text-based navigation, color should be used to indicate navigation links and to provide an indication of links already traveled. These are but a few of dozens of design conventions that make navigation user-friendly.

13.6 Component-Level Design

Mobile apps deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that are appropriate for the app's business domain, (3) provide sophisticated database query and access, and (4) establish data interfaces with external corporate systems. To achieve these (and many other) capabilities, you must design and construct program components that are identical in form to software components for traditional software.

The design methods discussed in Chapters 11 and 12 apply to mobile components with little, if any, modification. The implementation environment, programming languages, and design patterns, frameworks, and software may vary somewhat, but the overall design approach remains the same. To be cost conscious, you can design mobile components in such a way that they can be used without modification on several different mobile platforms.

13.7 Mobility and Design Quality

Every person has an opinion about what makes a "good" mobile app. Individual viewpoints vary widely. Some users enjoy flashy graphics; others want simple text. Some demand copious information; others desire an abbreviated presentation. Some like sophisticated analytical tools or database access; others like to keep it simple.

In fact, the user's perception of "goodness" (and the resultant acceptance or rejection of a mobile app as a consequence) might be more important than any technical discussion of mobile app quality. Mobile design quality attributes are virtually the same as WebApp quality characteristics.

But how is mobile quality perceived? What attributes must be exhibited to achieve goodness in the eyes of end users and at the same time exhibit the technical characteristics of quality that will enable you to correct, adapt, enhance, and support the mobile product over the long term?

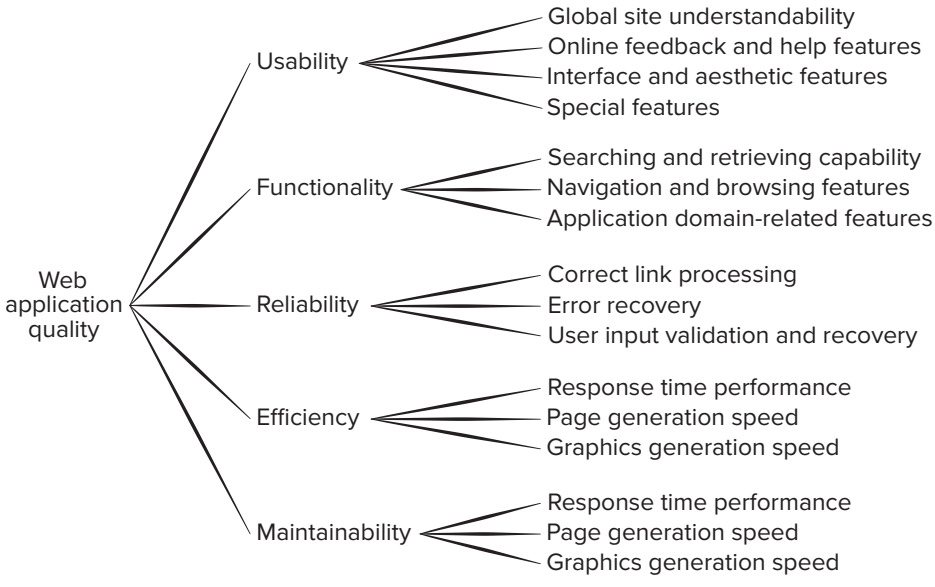
In reality, all the technical characteristics of design quality discussed in Chapter 12 and the generic quality attributes presented in Chapter 19 apply to mobile apps. However, the most relevant of these generic attributes—usability, functionality, reliability, efficiency, and maintainability—provide a useful basis for assessing the quality of mobile systems as well. Andreou [And05] suggests that end-user satisfaction with a mobile app is dictated by the same important quality factors—functionality, reliability, usability, efficiency, and maintainability—but adds portability to the list.

Olsina and his colleagues [Ols99] have prepared a "quality requirement tree" that identifies a set of technical attributes—usability, functionality, reliability, efficiency, and maintainability—that lead to high-quality mobile products.[25] Figure 13.6 summarizes their work. The criteria noted in the figure are of particular interest if you design, build, and maintain mobile products over the long term.

FIGURE 13.6

Quality requirements tree

Source: Olsina, Luis, Lafuente, Guillermo and Rossi, Gustavo, "Specifying Quality Characteristics and Attributes for Web Sites," Proceedings of the 1st International Conference on Software Engineering Workshop on Web Engineering, ACM, Los Angeles, May 1999.



25 These quality attributes are quite similar to those presented in Chapters 9 and 15. The implication is that quality characteristics are universal for all software.

Offutt [Off02] extends the five major quality attributes noted in Figure 13.6 by adding the following attributes:

Security. Mobile products have become heavily integrated with critical corporate and government databases. E-commerce applications extract and then store sensitive customer information. For these and many other reasons, mobile security is paramount in many situations. The key measure of security is the ability of the mobile app and its server environment to rebuff unauthorized access and/or thwart an outright malevolent attack. Security engineering is discussed in Chapter 18. For additional information on WebApp and mobile app security, see [Web13], [Pri10], [Vac06], and [Kiz05].

Availability. Even the best mobile product will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a Web-based mobile resource is available for use. But Offutt [Off02] suggests that "using features available on only one browser or one platform" makes the mobile product unavailable to those with a different browser or platform configuration. The user will invariably go elsewhere.

Scalability. Can the mobile product and its server environment be scaled to handle 100, 1,000, 10,000, or 100,000 users? Will the app and the systems with which it is interfaced handle significant variation in volume, or will responsiveness drop dramatically (or cease altogether)? It is important to design a mobile environment that can accommodate the burden of success (i.e., significantly more end users) and become even more successful.

Time to Market. Although time to market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first mobile product to address a specific market segment often captures a disproportionate number of end users.

Content Quality. Billions of Web pages are available for those in search of information. Even well-targeted Web searches result in an avalanche of content. With so many sources of information to choose from, how does the user assess the quality (e.g., veracity, accuracy, completeness, timeliness) of the content that is presented within a mobile product? This is part of what data science tries to address. The basics of data science are introduced in Appendix 2 of this book.

Tillman [Til00] suggests a useful set of criteria for assessing the quality of content: Can the scope and depth of content be easily determined to ensure that it meets the user's needs? Can the background and authority of the content's authors be easily identified? Is it possible to determine the currency of the content, the last update, and what was updated? Are the content and its location stable (i.e., will they remain at the referenced URL)? Is content credible? Is content unique? That is, does the mobile product provide some unique benefit to those who use it? Is content valuable to the targeted user community? Is content well organized? Indexed? Easily accessible? These questions represent only a small sampling of the issues that should be addressed as the design of a mobile product evolves.

INFO



Mobile Product—Quality Checklist

The following checklist provides a set of questions that will help both software engineers and end users assess overall mobile product quality:

- Can content, function, and/or navigation options be tailored to the user's preferences?
- Can content and/or functionality be customized to the bandwidth at which the user communicates? Does the app account for weak or lost signals in an acceptable manner?
- Can content, function, and/or navigation options be made context aware according to the user's preferences?
- Has adequate consideration been given to the power availability on the target device(s)?
- Have graphics, media (audio, video), and other Web or cloud services been used appropriately?
- Is the overall page design easy to read and navigate? Does the app take screen-size differences into account?
- Does the user interface conform to the display and interaction standards adopted for the targeted mobile device(s)?
- Does the app conform to the reliability, security, and privacy expectations of its users?
- What provisions have been made to ensure an app remains current?
- Has the mobile product been tested in all targeted user environments and for all targeted devices?

13.8 MOBILITY DESIGN BEST PRACTICES

There are several guidelines for developing mobile products[26] and for developing apps for specific platforms like Apple's iOS[27] or Google's Android.[28] Schumacher [Sch09] has collected many best practice ideas and has posted several specially adapted to the design of mobile applications and Web pages. Some important considerations when designing mobile touch-screen applications listed by Schumacher include:

- **Identify your audience.** The application must be written with the expectations and backgrounds of its users in mind. Experienced users want to do things quickly. Less experienced users will appreciate a handholding approach when they are first using the app.
- **Design for context of use.** It is important to consider how the user will interact with the real world while using the mobile product. Watching a movie on an airplane calls for a different user interface than checking the weather before you leave the office.
- **There is a fine line between simplicity and laziness.** Creating an intuitive user interface on a mobile device is much harder than simply removing features found in the user interface for the application running on a larger device. The user interface should provide all the information that enables a user to make her next decision.

26 See http://www.w3.org/TR/mwabp/.
27 See https://developer.apple.com/design/human-interface-guidelines/.
28 See http://developer.android.com/guide/components/index.html.

- **Use the platform as an advantage.** Touch-screen navigation is not intuitive and must be learned by all new users. This learning task will be easier if the user interface designers adhere to standards that have been set for the platform.
- **Make scrollbars and selection highlighting more salient.** Scrollbars are often hard to locate on touch devices because they are too small. Make sure that menu or icon borders are wide enough for color changes to catch the users' attention. When color coding is used, make sure there is sufficient contrast between foreground and background colors to allow them to be distinguishable by any color-blind users.
- **Increase discoverability of advanced functionality.** Hot keys and other shortcuts are sometimes included in mobile products to allow experienced users to complete their tasks more quickly. You can increase the discoverability of features like these by including visual design clues in the user interface.
- **Use clear and consistent labels.** Widget labels should be recognized by all app users, regardless of standards used by specific platforms. Use abbreviations cautiously and avoid them if possible.
- **Clever icons should never be developed at the expense of user understanding.** Icons often only make sense to their designers. Users must be able to learn their meaning quickly. It is hard to guarantee that icons are meaningful across all languages and user groups. A good strategy to enhance recognition is to add a text label beneath a novel icon.
- **Support user expectations for personalization.** Mobile device users expect to be able to personalize everything. At the very least, developers should try to allow users to set their location (or detect it automatically) and select content options that may be available at that location. It is important to indicate to users what features can be personalized and how users can personalize them.
- **Long scrolling forms trump multiple screens on mobile devices.** Experienced mobile device users want all information on a single input screen even if this requires scrolling. Novice users often become experienced quickly and will grow tired of multiple input screens.

Developing native applications for multiple device platforms can be costly and time consuming. Development costs can be reduced by using technologies familiar to Web developers (e.g., JavaScript, CSS, and HTML) to create mobile products that will be accessed using a Web browser on a mobile device.

There are no guarantees that a desktop program or a WebApp can be easily adapted for implementation as a mobile product. However, many of the agile software engineering practices (Chapter 3) used to create desktop computer applications can be used to create stand-alone apps or mobile client software, and many of the practices used to create quality WebApps apply to the creation of Web services used by mobile products.

The most important architectural design decision is often whether to build a thin or fat client. The model-view-controller (MVC) architecture (Section 13.3) is commonly used in mobile products. Because the mobile architecture has a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design. The architectural design must take device resources

into account (storage, processor speed, and network connectivity). The design should include provisions for discoverable services and movable devices.

Usability testing and deployment testing take place during each prototype development cycle. Code reviews that focus on security issues should be included as part of the implementation activities. These code reviews should be based on the appropriate security objectives and threats identified in the system design activities. Security testing is a routine part of system testing (Chapter 21).

13.9 SUMMARY

The quality of a mobile product—defined in terms of functionality, reliability, usability, efficiency, security, maintainability, scalability, and portability—is introduced during design. A good mobile product should be based on the following design goals: simplicity, ubiquity, personalization, flexibility, and localization.

Interface design describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. In addition, the interface for a good mobile product will promote the brand signature and focus on its targeted device platform(s). A set of core user stories is used to trim unnecessary features from the app to manage its resource requirements. Context-aware devices make use of discoverable services to help personalize the user experience.

Content design is critically important and takes the screen and other limitations of mobile devices into account. Aesthetic design, also called graphic design, describes the "look and feel" of the mobile product and includes color schemes, graphic layout, the use of graphics, and related aesthetic decisions. Aesthetic design must also take device limitations into account.

Architecture design identifies the overall hypermedia structure for the mobile product and encompasses both content architecture and mobile architecture. It is critical to determine how much of the mobile functionality will reside on the mobile device and how much will be provided by Web or cloud services.

Navigation design represents the navigational flow between content objects and for all mobile functions. Navigation syntax is defined by the widgets available on the targeted mobile device(s), and the semantics are often determined by the mobile platform. Content chunking must take intermittent service interruptions and user demands for fast performance into account.

Component design develops the detailed processing logic required to implement the components that are used to build a complete MobileApp function. The design techniques described in Chapter 12 may be applicable for the engineering of mobile components.

PROBLEMS AND POINTS TO PONDER

13.1. Explain why deciding to develop a MobileApp for several devices can be a costly design decision. Is there a way to mitigate the risks of supporting the wrong platform?

13.2. In this chapter we listed many quality attributes for mobile products. Select the three that you believe are most important, and make an argument that explains why each should be emphasized in mobile design work.

13.3. You are a MobileApp designer for *Project Planning Corporation*, a company that builds productivity software. You want to implement the equivalent of a digital three-ring binder that allows tablet users to organize and categorize electronic documents of several types under user-defined tabs. For example, a kitchen remodeling project might require a pdf catalog, a jpg or layout drawing, an MS Word proposal, and an Excel spreadsheet stored under a Cabinetry tab. Once defined, the binder and its tab content can be stored either on the tablet or on some cloud storage. The application needs to provide five key functions: binder and tab definition, digital document acquisition from a Web location or the device, binder management functions, page display functions, and a notes function to allow a Post-it note to be added to any page. Develop an interface design for the three-ring application, and implement it as a paper prototype.

13.4. What is the most aesthetically pleasing MobileApp you have ever used and why?

13.5. Create user stories for the three-ring application described in Problem 13.3.

13.6. What might be considered to make the three-ring application a context-aware MobileApp?

13.7. Reconsidering the *ProjectPlanning* three-ring application described in Problem 13.3, select a development platform for the first working prototype. Discuss why you made the choice.

13.8. Do a bit of additional research on the MVC architecture and decide whether it would be an appropriate MobileApp architecture for the three-ring discussed in Problem 13.3.

13.9. Describe three context-aware features that would be desirable to add to a *SafeHome* MobileApp.

13.10. You are a WebApp designer for FutureLearning Corporation, a distance learning company. You intend to implement an Internet-based "learning engine" that will enable you to deliver course content to a student. The learning engine provides the basic infrastructure for delivering learning content on any subject (content designers will prepare appropriate content). Develop a prototype interface design for the learning engine.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

14 PATTERN-BASED DESIGN

Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution for this?* The answer is almost always *yes!* The problem is finding the solution; ensuring that it does, in fact, fit the problem you've encountered; understanding the constraints that may restrict the ways in which the solution is applied; and finally, translating the proposed solution into your design environment.

KEY CONCEPTS

anti-patterns 302
architectural patterns 299
behavioral patterns 292
component-level design patterns 300
creational patterns 292
design mistakes 298
design patterns 290
frameworks 293
kinds of patterns 291
machine learning 294
pattern languages 297
pattern-organizing table 298
structural patterns 292
system of forces 290
user interface design patterns 304

QUICK LOOK

What is it? Pattern-based design creates a new application by finding a set of proven solutions to a clearly delineated set of problems. Each problem and its solution are described by a design pattern that has been cataloged and vetted by other software engineers who have encountered the problem and implemented the solution while designing other applications.

Who does it? A software engineer examines each problem encountered for a new application and then attempts to find a relevant solution by searching one or more pattern repositories.

Why is it important? Have you ever heard the phrase "reinventing the wheel"? It happens all the time in software development, and it's a waste of time and energy. By using existing design patterns, you can acquire a proven solution for a specific problem. As each pattern is applied, solutions are integrated and the application to be built moves closer to a complete design.

What are the steps? The requirements model is examined to isolate the hierarchical set of problems to be solved. The problem space is partitioned so that subsets of problems associated with specific software functions and features can be identified. Problems can also be organized by type: architectural, component-level, algorithmic, user interface, and so forth. Once a subset of problems is defined, one or more pattern repositories are searched to determine if a design pattern, represented at an appropriate level of abstraction, exists.

What is the work product? A design model that depicts the architectural structure, user interface, and component-level detail is developed.

How do I ensure that I've done it right? As each design pattern is translated into some element of the design model, work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

But what if the solution were codified in some manner? What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem? It turns out that software problems have been codified and described using a standardized template, and solutions to them (along with constraints) have been proposed. Called *design patterns,* this codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

The early history of software patterns begins not with a computer scientist but a building architect, Christopher Alexander, who recognized that a recurring set of problems was encountered whenever a building was designed. He characterized these recurring problems and their solutions as *patterns,* describing them in the following manner [Ale77]: "Each pattern describes a problem that occurs repeatedly in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice." Alexander's ideas were first translated into the software world in books by Gamma [Gam95], Buschmann [Bus07], and their many colleagues.[1] Today, dozens of pattern repositories exist, and pattern-based design can be applied in many different application domains.

14.1 DESIGN PATTERNS

A *design pattern* can be characterized as "a three-part rule which expresses a relation between a certain context, a problem, and a solution" [Ale79]. For software design, *context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment. A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how the problem can be interpreted within its context and how the solution can be effectively applied.

It is reasonable to argue that most problems have multiple solutions, but that a solution is effective only if it is appropriate within the context of the existing problem. It is the system of forces that causes a designer to choose a specific solution. The intent is to provide a solution that best satisfies the system of forces, even when these forces are contradictory. Finally, every solution has consequences that may have an impact on other aspects of the software and may themselves become part of the system of forces for other problems to be solved within the larger system.

An effective design pattern: (1) captures a specific solution to a bounded problem, (2) provides a solution that has been proven in practice, (3) identifies an approach to a problem that is not obvious, (4) identifies the relationship(s) between the design and other architectural elements, and (5) is elegant in its approach and its utility.

1 Earlier discussions of software patterns do exist, but these two classic books were the first cohesive treatments of the subject.

A design pattern saves you from "reinventing the wheel," or worse, inventing a "new wheel" that is slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer.

14.1.1 Kinds of Patterns

One of the reasons that software engineers are interested in (and intrigued by) design patterns is that human beings are inherently good at pattern recognition. If we weren't, we'd be frozen in space and time—unable to learn from experience, unwilling to venture forward because of our inability to recognize situations that might lead to high risk, unhinged by a world that seems to have no regularity or logical consistency. Luckily, none of this occurs because we do recognize patterns in virtually every aspect of our lives.

In the real world, the patterns we recognize are learned over a lifetime of experience. We recognize them instantly and inherently understand what they mean and how they might be used. Some of these patterns provide us with insight into recurring phenomenon. For example, you're on your way home from work on the interstate when your navigation system (or car radio) informs you that a serious accident has occurred on the interstate in the opposing direction. You're 4 miles from the accident, but already you begin to see traffic slowing, recognizing a pattern that we'll call **RubberNecking.** People in the travel lanes moving in your direction are slowing at the sight of the accident to get a better view of what happened on the opposite side of the highway. The **RubberNecking** pattern yields remarkably predictable results (a traffic jam), but it does nothing more than describe a phenomenon. In patterns jargon, it might be called a *nongenerative* pattern because it describes a context and a problem, but it does not provide any clear-cut solution.

When software design patterns are considered, we strive to identify and document *generative* patterns. That is, we identify a pattern that describes an important and repeatable aspect of a system and that provides us with a way to build that aspect within a system of forces that are unique to a given context. In an ideal setting, a collection of generative design patterns could be used to "generate" an application or computer-based system whose architecture enables it to adapt to change. Sometimes called *generativity,* "the successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution which emerges indirectly as a result of the smaller solutions" [App00].

Design patterns span a broad spectrum of abstraction and application. *Architectural patterns* describe broad-based design problems that are solved using a structural approach. *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the way they communicate with one another, and their placement within a larger architecture. *Interface design patterns* describe common user interface problems and their solutions with a system of forces that includes the specific characteristics of end users. *WebApp patterns* address a problem set that is encountered

when building WebApps and often incorporates many of the other pattern categories just mentioned. *Mobile patterns* describe commonly encountered problems when developing solutions for mobile platforms. At a lower level of abstraction, *idioms* describe how to implement all or part of a specific algorithm or data structure for a software component within the context of a specific programming language. Don't force a pattern, even if it addresses the problem at hand. If the context and forces are wrong, look for another pattern.

In their seminal book on design patterns, Gamma and his colleagues[2] [Gam95] focus on three types of patterns that are particularly relevant to object-oriented design: creational patterns, structural patterns, and behavioral patterns.

INFO



Creational, Structural, and Behavioral Patterns

A wide variety of design patterns that fit into creational, structural, and behavioral categories have been proposed and can be found on the Web. Here is a sampling of patterns for each type. Comprehensive descriptions of each of these patterns can be obtained via links at **www.wikipedia.org.**

Creational Patterns

- **Abstract factory pattern.** Centralize decision of what factory to instantiate.
- **Factory method pattern.** Centralize creation of an object of a specific type, choosing one of several implementations.
- **Builder pattern.** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structural Patterns

- **Adapter pattern.** "Adapts" one interface for a class into one that a client expects.
- **Aggregate pattern.** A version of the composite pattern with methods for aggregation of children.
- **Composite pattern.** A tree structure of objects where every object has the same interface.
- **Container pattern.** Create objects for the sole purpose of holding other objects and managing them.
- **Proxy pattern.** A class functioning as an interface to another thing.
- **Pipes and filters.** A chain of processes where the output of each process is the input of the next.

Behavioral Patterns

- **Chain of responsibility pattern.** Command objects are handled or passed on to other objects by logic-containing processing objects.
- **Command pattern.** Command objects encapsulate an action and its parameters.
- **Iterator pattern.** Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator pattern.** Provides a unified interface to a set of interfaces in a subsystem.
- **Visitor pattern.** A way to separate an algorithm from an object.
- **Hierarchical visitor pattern.** Provides a way to visit every node in a hierarchical data structure such as a tree.

2 Gamma and his colleagues [Gam95] are often referred to as the "Gang of Four" (GoF) in patterns literature.

Creational patterns focus on the "creation, composition, and representation" of objects and provide mechanisms that make the instantiation of objects easier within a system and enforce "constraints on the type and number of objects that can be created within a system" [Maa07]. *Structural patterns* focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. *Behavioral patterns* address problems associated with the assignment of responsibility between objects and the way communication is affected between objects.

14.1.2 Frameworks

Patterns themselves may not be sufficient to develop a complete design. In some cases, it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework,* for design work. A *framework* is a reusable "mini-architecture" that serves as a foundation from which other design patterns can be applied. That is, you can select a "*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context . . . which specifies their collaboration and use within a given domain" [Amb98].

A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain. The plug points enable you to integrate problem-specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.

Gamma and his colleagues [Gam95] note that patterns are more abstract than frameworks. A framework can be "embodied in code," while a pattern is generally code independent. A framework often encompasses more than a single pattern and is therefore a larger architectural element than a pattern. Finally, a framework resides within a specific application domain, but patterns can be applied in any domain in which the problem to be addressed is encountered.

The designer of a framework will argue that one, reusable, mini-architecture is applicable to all software to be developed within a limited domain of application. To be most effective, frameworks are applied with no changes. Additional design elements may be added, but only via the plug points that allow the designer to flesh out the framework skeleton.

14.1.3 Describing a Pattern

Pattern-based design begins with the recognition of patterns within the application you intend to build, continues with a search to determine whether others have addressed the pattern, and concludes with the application of an appropriate pattern to the problem at hand. The second of these three tasks is often the most difficult. How do you find patterns that fit your needs?

An answer to this question must rely on effective communication of the problem the pattern addresses, the context in which the pattern resides, the system of forces that mold the context, and the solution that is proposed. To communicate this information unambiguously, a standard form or template for pattern descriptions is required. Although several different pattern templates have been proposed, almost all contain a major

subset of the content suggested by Gamma and his colleagues [Gam95]. A simplified pattern template is shown in the sidebar.

INFO



Design Pattern Template

Pattern name. Describes the essence of the pattern in a short but expressive name.

Problem. Describes the problem that the pattern addresses.

Motivation. Provides an example of the problem.

Context. Describes the environment in which the problem resides, including the application domain.

Forces. Lists the system of forces that affect the way the problem must be solved; includes a discussion of limitation and constraints that must be considered.

Solution. Provides a detailed description of the solution proposed for the problem.

Intent. Describes the pattern and what it does.

Collaborations. Describes how other patterns contribute to the solution.

Consequences. Describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern.

Implementation. Identifies special issues that should be considered when implementing the pattern.

Known uses. Provides examples of actual uses of the design pattern in real applications.

Related patterns. Cross-references related design patterns.

The names of design patterns should be chosen with care. One of the key technical problems in pattern-based design is the inability to find existing patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name.

A pattern template provides a standardized means for describing a design pattern. Each of the template entries represents characteristics of the design pattern that can be searched (e.g., via a database) so that the appropriate pattern can be found.

14.1.4 Machine Learning and Pattern Discovery

Software patterns can be described as best practice solutions to known problems. Information about where design patterns have been implemented in a software design is useful information for developers to have when creating or maintaining a software system. Sadly, this information is lost due to poor documentation practices of the original developers. In recent years, there has been significant interest in making use of automatic techniques to identify new patterns present, but not documented, in existing software products [Alh12].

One way to do this is to create an artificial intelligence (AI) system capable of recognizing design patterns after examining many similar software systems. The same software pattern may be implemented in many ways. *Machine learning*[3] techniques

3 Machine learning is an AI technique that uses statistical techniques to allow a system to learn from examples and improve its performance without being explicitly programmed. For those readers interested in pursuing this subject in greater detail, see [Kub17].

can provide a means of teaching a system to recognize the presence of a pattern in the software source code. The machine learning system repeatedly examines a training set containing both good and bad examples of software patterns using specific quantitative criteria of "good" and "bad" examples. This process continues until the system has learned to recognize most of the good patterns in the training set. Often these training sets are derived from large open-source software systems available on the Internet [Zan15].

Once trained, the tool can be used to locate software patterns in new systems outside the training set. To be useful, the software patterns are collected in a repository. Ideally, this repository can be searched for software patterns applicable to the problems developers need to solve [Amp13].

14.2 PATTERN-BASED SOFTWARE DESIGN

The best designers in any field have an uncanny ability to see patterns that characterize a problem and the corresponding patterns that can be combined to create a solution. Throughout the design process, you should look for every opportunity to apply existing design patterns (when they meet the needs of the design) rather than creating new ones.

14.2.1 Pattern-Based Design in Context

Pattern-based design is not used in a vacuum. The concepts and techniques discussed for architectural, component-level, and user interface design (Chapter 10 through 12) are all used in conjunction with a pattern-based approach.

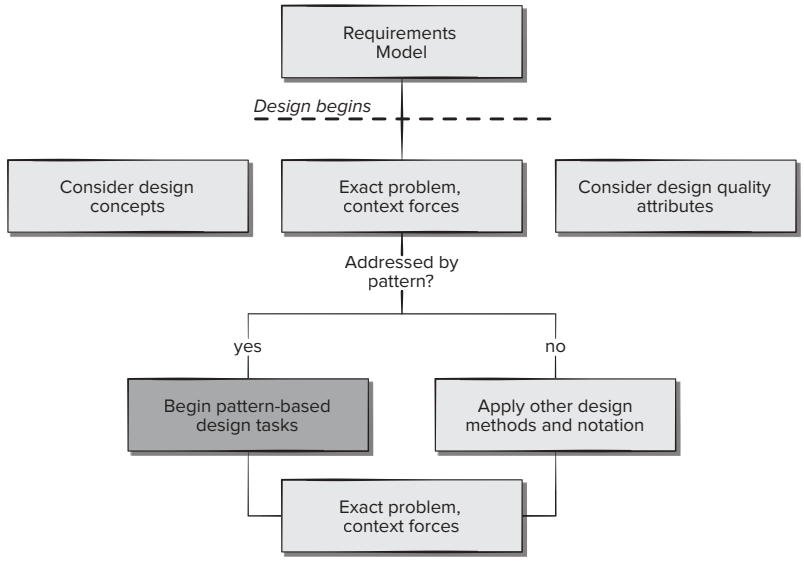
In Chapter 9, we noted that a set of quality guidelines and attributes serve as the basis for all software design decisions. The decisions themselves are influenced by a set of fundamental design concepts (e.g., separation of concerns, stepwise refinement, functional independence) that are achieved using heuristics that have evolved over many decades, and best practices (e.g., techniques, modeling notation) that have been proposed to make design easier to perform and more effective as a basis for construction.

The role of pattern-based design in all of this is illustrated in Figure 14.1. A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system. The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway. It may imply the design in an abstract manner, but the requirements model does little to represent the design explicitly.

As you begin your work as a designer, it's always important to keep quality attributes (Chapter 9) in mind. These attributes establish a way to assess software quality but do little to help you achieve it. Therefore, you should apply proven techniques for translating the abstractions contained in the requirements model into a more concrete form that is the software design. To accomplish this, you'll use the methods and modeling tools available for architectural, component-level, and interface design. But, only when you're faced with a problem, context, and system of forces that have not been solved before. If a solution already exists, use it! And that means applying a pattern-based design approach.

FIGURE 14.1

Pattern-based design in context



14.2.2 Thinking in Patterns

Pattern-based design implies a "new way of thinking" [Sha05] that begins by considering context—the big picture. As context is evaluated, you extract a hierarchy of problems that must be solved. Some of these problems will be global in nature, while others will address specific features and functions of the software. All will be affected by a system of forces that will influence the nature of the solution that is proposed.

Shalloway and Trott [Sha05] suggest the following approach[4] that enables a designer to think in patterns:

1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with "big picture" patterns that establish a context or skeleton for further design work.
4. "Work inward from the context" [Sha05], looking for patterns at lower levels of abstraction that contribute to the design solution.
5. Repeat steps 1 to 4 until the complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.

It's important to note that patterns are not independent entities. Design patterns that are present at a high level of abstraction will invariably influence the ways other patterns are applied at lower levels of abstraction. In addition, patterns often collaborate

4 Based on the work of Christopher Alexander [Ale79].

with one another. The implication is that when you select an architectural pattern, it may very well influence the component-level design patterns you choose. Likewise, when you select a specific interface design pattern, you are sometimes forced to use other patterns that collaborate with it.

To illustrate, consider the **SafeHomeAssured.com** WebApp. If you consider the big picture, the WebApp must address how to provide information about *SafeHome* products and services, how to sell *SafeHome* products and services to customers, and how to establish Internet-based monitoring and control of an installed security system. Each of these fundamental problems can be further refined into a set of subproblems.

For example, *How to sell* via the Internet implies an **e-commerce** pattern that itself implies many patterns at lower levels of abstraction. The **e-commerce** pattern (likely, an architectural pattern) implies mechanisms for setting up a customer account, displaying the products to be sold, selecting products for purchase, and so forth. Hence, if you think in patterns, it is important to determine whether a pattern for setting up an account exists. If **SetUpAccount** is available as a viable pattern for the problem context, it may collaborate with other patterns such as **BuildInputForm, ManageFormsInput,** and **ValidateFormsEntry.** Each of these patterns delineates problems to be solved and solutions that may be applied.

14.2.3 Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:

1. **Examine the requirements model and develop a problem hierarchy.** Describe each problem and subproblem by isolating the problem, the context, and the system of forces that apply. Work from broad problems (high level of abstraction) to smaller subproblems (at lower levels of abstraction).
2. **Determine if a reliable pattern language has been developed for the problem domain.** A *pattern language* encompasses a collection of patterns, each described using a standardized template (Section 14.1.3) and interrelated to show how these patterns collaborate to solve problems across an application domain. The *SafeHome* software team would look for a pattern language developed specifically for home security products. If that level of pattern language specificity could not be found, the team would partition the *SafeHome* software problem into a series of generic problem domains (e.g., digital device monitoring problems, user interface problems, digital video management problems) and search for appropriate pattern languages.
3. **Beginning with a broad problem, determine whether one or more architectural patterns are available for it.** If an architectural pattern is available, be certain to examine all collaborating patterns. If the pattern is appropriate, adapt the design solution proposed and build a design model element that adequately represents it. For example, a broad problem for the **SafeHomeAssured.com** WebApp is addressed with an **e-commerce** pattern (Section 14.2.2). This pattern will suggest a specific architecture for addressing e-commerce requirements.
4. **Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems and search for appropriate patterns to address them.** It may be necessary to search through other pattern repositories as

well as the list of patterns that corresponds to the architectural solution. If an appropriate pattern is found, adapt the design solution proposed and build a design model element that adequately represents it. Be certain to apply step 7.

5. **Repeat steps 2 to 4 until all broad problems have been addressed.** The implication is to begin with the big picture and elaborate to solve problems at increasingly more detailed levels.
6. **If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.** Proceed in a manner similar to steps 3 to 5.
7. **Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.** Be certain to examine context and forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.
8. **Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.**

Although this design approach is fundamentally top-down, Gillis [Gil06] suggests that "it's more organic than that, more inductive than deductive, more bottom-up than top-down." In addition, the pattern-based approach must be used in conjunction with other software design concepts and techniques.

14.2.4 Building a Pattern-Organizing Table

As pattern-based design proceeds, you may encounter trouble organizing and categorizing candidate patterns from multiple pattern languages and repositories. To help organize your evaluation of candidate patterns, Microsoft [Mic13b] suggests the creation of a *pattern-organizing table* that takes the general form shown in Figure 14.2.

A pattern-organizing table can be implemented as a spreadsheet model using the form shown in the figure. An abbreviated list of problem statements, organized by data and content, architecture, component level, and user interface issues, is presented in the left-hand (shaded) column. Four pattern types—database, application, implementation, and infrastructure—are listed across the top row. The names of candidate patterns are noted in the cells of the table.

To provide entries for the organizing table, you'll search through pattern languages and repositories for patterns that address an individual problem statement. When one or more candidate patterns is found, it is entered in the row corresponding to the problem statement and the column that corresponds to the pattern type. The name of the pattern is entered as a hyperlink to the URL of the Web address that contains a complete description of the pattern.

14.2.5 Common Design Mistakes

Several common mistakes may occur when pattern-based design is used. In some cases, not enough time has been spent to understand the underlying problem and its context and forces, and you may select a pattern that looks right but is inappropriate for the solution required. Once the wrong pattern is selected, you refuse to see your error and force-fit the pattern. In other cases, the problem has forces that are not

FIGURE 14.2

A pattern-organizing table
Source: Adapted from Microsoft, "Prescriptive Architecture: Integration and Patterns," MSDN, May 2004.

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

considered by the pattern you've chosen, resulting in a poor or erroneous fit. Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

Can these mistakes be avoided? In most cases, the answer is yes. Every good designer looks for a second opinion and welcomes review of her work. The review techniques discussed in Chapter 16 can help to ensure that the pattern-based design you've developed will result in a high-quality solution for the software problem to be solved.

14.3 ARCHITECTURAL PATTERNS

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed on the design.[5]

Architectural patterns are a bit different. For example, every house (and every architectural style for houses) employs a **Kitchen** pattern. The **Kitchen** pattern and

5 This implies that there will be a central foyer and hallway, that rooms will be placed to the left and right of the foyer, that the house will have two (or more) stories, that the bedrooms of the house will be upstairs, and so on. These "rules" are imposed once the decision is made to use the center-hall colonial style.

patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room. In addition, the pattern might address problems associated with countertops, lighting, wall switches, a central island, flooring, and so on. Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the "solution" suggested by the **Kitchen** pattern.

A software architecture may have several architectural patterns that address issues such as concurrency, persistence, and distribution. Before a representative architectural pattern can be chosen in a specific domain, it must be assessed for its appropriateness for the application and the overall architectural style, as well as the context and system of forces that it specifies.

14.4 COMPONENT-LEVEL DESIGN PATTERNS

Component-level design patterns provide you with proven solutions that address one or more subproblems extracted from the requirements model. In many cases, design patterns of this type focus on some functional element of a system. For example, the **SafeHomeAssured.com** application must address the following design subproblem: *How do I get product specifications and related information for any* SafeHome *device?*

Having stated the subproblem that must be solved, you should now consider context and the system of forces that affect the solution. Examining the appropriate requirements model use case, you learn that the consumer uses the specification for a *SafeHome* device (e.g., a security sensor or camera) for informational purposes. However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.

The solution to the subproblem involves a search. Because searching is a very common problem, it should come as no surprise that there are many search-related patterns. Looking through several patterns repositories, you find the following patterns, along with the problem that each solves:

AdvancedSearch. Users must find a specific item in a large collection of items.

HelpWizard. Users need help on a certain topic related to the website or when they need to find a specific page within the site.

SearchArea. Users must find a page.

SearchTips. Users need to know how to control the search engine.

SearchResults. Users must process a list of search results.

SearchBox. Users must find an item or specific information.

For **SafeHomeAssured.com**, the number of products is not particularly large, and each has a relatively simple categorization, so **AdvancedSearch** and **HelpWizard** are probably not necessary. Similarly, the search is simple enough not to require **SearchTips**. The description of **SearchBox**, however, is given (in part) as:

Search Box

(*Adapted from* www.welie.com/patterns/showPattern.php?patternID=search)

Problem:	The users need to find an item or specific information.
Motivation:	Any situation in which a keyword search is applied across a collection of content objects organized as Web pages.
Context:	Rather than using navigation to acquire information or content, the user wants to do a direct search through content contained on multiple Web pages. Any website that already has primary navigation. User may want to search for an item in a category. User might want to further specify a query.
Forces:	The website already has primary navigation. Users may want to search for an item in a category. Users might want to further specify a query using simple Boolean operators.
Solution:	Offer search functionality consisting of a search label, a keyword field, a filter if applicable, and a "go" button. Pressing the return key has the same function as selecting the go button. Also provide Search Tips and examples in a separate page. A link to that page is placed next to the search functionality. The edit box for the search term is large enough to accommodate three typical user queries (typically around 20 characters). If the number of filters is more than 2, use a combo box for filters selection, otherwise a radio button. The search results are presented on a new page with a clear label containing at least "Search results" or similar. The search function is repeated in the top part of the page with the entered keywords, so that the users know what the keywords were.

The pattern description continues with other entries, as described in Section 14.1.3.

The pattern goes on to describe how the search results are accessed, presented, matched, and so on. Based on this, the **SafeHomeAssured.com** team can design the components required to implement the search or (more likely) acquire existing reusable components.

SAFEHOME



Applying Patterns

The scene: Informal discussion during the design of a software increment that implements sensor control via the Internet for **SafeHomeAssured.com.**

The players: Jamie, responsible for design, and Vinod, **SafeHomeAssured.com** chief system architect.

The conversation:

Vinod: So how is the design of the camera control interface coming along?

Jamie: Not too bad. I've designed most of the capability to connect to the actual sensors without too many problems. I've also started thinking about the interface for the users to move,

pan, and zoom the cameras from a remote device, but I'm not sure I've got it right yet.

Vinod: What have you come up with?

Jamie: Well, the requirements are that the camera control needs to be highly interactive—as the user moves the control, the camera should move as soon as possible. So, I was thinking of having a set of buttons laid out like a normal camera, but when the user clicks them, it controls the camera.

Vinod: Hmmm. Yeah, that would work, but I'm not sure it's right—for each click of a control you need to wait for the whole client-server communication to occur, and so you won't get a good sense of quick feedback.

Jamie: That's what I thought—and why I wasn't very happy with the approach, but I'm not sure how else I might do it.

Vinod: Well, why not just use the **Interactive-DeviceControl** pattern?

Jamie: Uhmmm—what's that? I haven't heard of it.

Vinod: It's basically a pattern for exactly the problem you are describing. The solution it proposes is basically to create a control connection to the server with the device, through which control commands can be sent. That way you don't need to send normal HTTP requests. And the pattern even shows how you can implement this using some simple AJAX techniques. You have some simple client-side JavaScript that communicates directly with the server and sends the commands as soon as the user does anything.

Jamie: Cool! That's just what I needed to solve this thing. Where do I find it?

Vinod: It's available in an online repository. Here's the URL.

Jamie: I'll go check it out.

Vinod: Yep—but remember to check the consequences field for the pattern. I seem to remember that there was something in there about needing to be careful about issues of security. I think it might be because you are creating a separate control channel and so bypassing the normal Web security mechanisms.

Jamie: Good point. I probably wouldn't have thought of that! Thanks.

14.5 Anti-Patterns

Design patterns provide you with proven solutions that address one or more problems extracted from the requirements model. *Anti-patterns* describe commonly used solutions to design problems that usually have negative effects on software quality. In other words, they describe bad solutions to the design problems or at the very least describe the consequence of applying a design pattern in the wrong context. Anti-patterns can provide tools to help developers recognize when these problems exist and may provide detailed plans for reversing the underlying problem causes and implementing better solutions to these problems [Bro98]. Anti-patterns can provide valuable guidance to developers when they are looking for ways to refactor software products to improve their quality. In addition, they can be used by technical reviewers (Chapter 16) to uncover area of concern.

Brown and his colleagues [Bro98] make the following comparisons between pattern and anti-pattern descriptions. Design patterns are usually written from the bottom up. A design pattern description starts with a recurring solution to a problem and then adds forces, symptoms, and context elements of the situation in which the solution is to be applied. Anti-patterns are written from the top down.

An anti-pattern description takes a recurring design problem, or a bad development practice, and then lists its symptoms and negative consequences. It is then possible to include a recommended procedure for reducing the consequences documented in the anti-pattern. **The Blob** anti-pattern is presented in the following example.

The Blob

(*Adapted from* http://antipatterns.com/briefing/sld024.htm)

Problem:

Single class with large number of attributes, operations, or both.

Symptoms and Consequences:

- A disparate collection of unrelated attributes and operations encapsulated in a single class.
- An overall lack of cohesiveness of the attributes and operations is typical of **The Blob**.
- A single controller class with associated simple, data-object classes.
- A single controller class encapsulates the entire functionality like a procedural main program.
- **The Blob** limits the ability to modify the system without affecting the functionality of other objects.
- Modifications to other objects in the system are also likely to impact **The Blob's** class.
- **The Blob** class is typically too complex for reuse and testing.
- **The Blob** class may be expensive to load into memory and uses excessive resources.

Typical Causes:

- Lack of an object-oriented architecture.
- The team may be lacking appropriate abstraction skills.
- Lack of defined software architecture.
- Lack of programming language support for architectural design.
- In iterative projects, developers tend to add little pieces of functionality to existing classes.
- Defining system architecture during requirements analysis often leads to **The Blob**.

Solution:

- The solution involves a form of re-factoring.
- The key is to move behavior away from **The Blob**.
- Reallocate behavior to other data objects in ways that make **The Blob** less complex.

A complete discussion of anti-patterns is beyond the scope of this book. Several anti-patterns with colorful and descriptive names appear in the sidebar that follows. We suspect you will recognize many of the anti-pattern names as development practices you were told to avoid when you learned to program.

INFO



Selected Anti-Patterns

A wide variety of anti-design patterns have been identified to assist developers in making refactoring decisions. Comprehensive descriptions of each of these patterns can be obtained via links at **https://en.wikipedia.org/wiki/Anti-pattern.**

- **Big ball of mud.** A system with no recognizable structure
- **Stovepipe system.** A barely maintainable assemblage of ill-related components
- **Boat anchor.** Retaining a part of a system that no longer has any use
- **Lava flow.** Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- **Spaghetti code.** Program whose structure is barely comprehensible, especially because of misuse of code structures
- **Copy and paste programming.** Copying existing code several times rather than creating generic solutions
- **Silver bullet.** Assume that a favorite technical solution will always solve a larger process or problem
- **Programming by permutation.** Trying to approach a solution by successively modifying the code to see if it works

14.6 USER INTERFACE DESIGN PATTERNS

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of 10 categories of patterns as described by Tidwell [Tid11] and vanWelie [Wel01]. A few representative categories (discussed with a simple example[6]) follow:

Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.

Pattern:	**Top-level navigation**
Brief description:	Used when a site or application implements several major functions. Provides a top-level menu, often coupled with a logo or identifying graphic, that enables direct navigation to any of the system's major functions.
Details:	Major functions (generally limited to between four and seven function names) are listed across the top of the display (vertical column formats are also possible) in a horizontal line of text. Each name provides a link to the appropriate function or information source. Often used with the **BreadCrumbs** pattern discussed later.
Navigation elements:	Each function/content name represents a link to the appropriate function or content.

Page layout. Address the general organization of pages (for websites) or distinct screen displays (for interactive applications).

6 An abbreviated pattern template is used here. Full pattern descriptions (along with dozens of other patterns) can be found at [Tid11] and [Wel01].

Pattern:	**Card stack**
Brief description:	Used when several specific subfunctions or content categories related to a feature or function must be selected in random order. Provides the appearance of a stack of tabbed cards, each selectable with a mouse click and each representing specific subfunctions or content categories.
Details:	Tabbed cards are a well-understood metaphor and are easy for the user to manipulate. Each tabbed card (divider) may have a slightly different format. Some may require input and have buttons or other navigation mechanisms; others may be informational. May be combined with other patterns such as **drop-down list, fill-in-the-blanks**, and others.
Navigation elements:	A mouse click on a tab causes the appropriate card to appear. Navigation features within the card may also be present, but in general, these should initiate a function that is related to card data, not cause an actual link to some other display.

E-commerce. Specific to websites, these patterns implement recurring elements of e-commerce applications.

Pattern:	**Shopping cart**
Brief description:	Provides a list of items selected for purchase.
Details:	Lists item, quantity, product code, availability (in stock, out of stock), price, delivery information, shipping costs, and other relevant purchase information. Also provides ability to edit (e.g., remove, change quantity).
Navigation elements:	Contains ability to proceed with shopping or go to checkout.

Each of the preceding example patterns (and all patterns within each category) would also have a complete component-level design, including design classes, attributes, operations, and interfaces. If you have further interest, see [Cho16], [Gas17], [Kei18], [Tid11], [Hoo12], and [Wel01] for further information. The role of anti-patterns and their effects on UX design can be found in [Sch15] and [Gra18].

14.7 MOBILITY DESIGN PATTERNS

Throughout this chapter you've learned that there are different types of patterns and how they can be categorized. By their nature, mobile applications are all about the interface. In many cases, mobile UI patterns [Mob12] are represented as a collection of "best of breed" screen images for apps in a variety of different categories. Typical examples might include:

Check-in screens. How do I check in from a specific location, make a comment, and share comments with friends and followers on a social network?

Maps. How do I display a map within the context of an app that addresses some other subject? For example, review a restaurant and represent its location within a city.

Popovers. How do I represent a message or information (from the app or another user) that arises in real time or as the consequence of a user action?

Sign-up flows. How do I provide a simple way to sign in or register for information or functionality?

Custom tab navigation. How do I represent a variety of different content objects in a manner that enables the user to select the one she wants?

Invitations. How do I inform the user that he must participate in some action or dialog? Typical examples might include making use of a dialog box, a tool tip, or a self-playing video demo.

Additional information about mobile UI patterns can be found in [Nei14], [Hoo12], [McT16], [Abr17], and [Pun17]. In addition to UI patterns, Meier and his colleagues [Mei12] propose a variety of more general pattern descriptions for mobile apps.

14.8 SUMMARY

Design patterns provide a codified mechanism for describing problems and their solution in a way that allows the software engineering community to capture design knowledge for reuse. A pattern describes a problem, indicates the context enabling the user to understand the environment in which the problem resides, and lists a system of forces that indicate how the problem can be interpreted within its context and how the solution can be applied. In software engineering work, we identify and document generative patterns. These patterns describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that is unique to a given context.

Architectural patterns describe broad-based design problems that are solved using a structural approach. Data patterns describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. Component patterns (also referred to as design patterns) address problems associated with the development of subsystems and components, the ways in which they communicate with one another, and their placement within a larger architecture. Software anti-patterns describe commonly occurring solutions to a problem that lead to the creation of software products with poor quality. Interface design patterns describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. Mobile patterns address the unique nature of the mobile interface and functionality and control elements that are specific to mobile platforms.

A framework provides an infrastructure in which patterns may reside and idioms describe programming language–specific implementation detail for all or part of a specific algorithm or data structure. A standard form or template is used for pattern descriptions. A pattern language encompasses a collection of patterns, each described using a standardized template and interrelated to show how these patterns collaborate to solve problems across an application domain.

Pattern-based design is used in conjunction with architectural, component-level, and user interface design methods. The design approach begins with an examination of the requirements model to isolate problems, define context, and describe the system of forces. Next, pattern languages for the problem domain are searched to determine if patterns exist for the problems that have been isolated. Once appropriate patterns have been found, they are used as a design guide.

PROBLEMS AND POINTS TO PONDER

14.1. Discuss the three "parts" of a design pattern, and provide a concrete example of each from some field other than software.

14.2. What is the difference between a pattern and an anti-pattern?

14.3. How do architectural patterns differ from component patterns?

14.4. What is a framework, and how does it differ from a pattern? What is an idiom, and how does it differ from a pattern?

14.5. Using the design pattern template presented in Section 14.1.3, develop a complete pattern description for a pattern suggested by your instructor.

14.6. Develop a skeletal pattern language for a sport with which you are familiar. You can begin by addressing the context, the system of forces, and the broad problems that a coach and team must solve. You need only specify pattern names and provide a one-sentence description for each pattern.

14.7. When Christopher Alexander says, "good design cannot be achieved simply by adding together performing parts," what do you think he means?

14.8. Using the pattern-based design tasks noted in Section 14.2.3, develop a skeletal design for the "interior design system" described in Section 12.3.2.

14.9. Build a pattern-organizing table for the patterns you used in Problem 14.8.

14.10. Using the design pattern template presented in Section 14.1.3, develop a complete pattern description for the **Kitchen** pattern mentioned in Section 14.3.

Design element: Quick Look icon magnifying glass: © Roger Pressman

PART

Three

QUALITY AND SECURITY

In this part of *Software Engineering: A Practitioner's Approach,* you'll learn about the principles, concepts, and techniques that are applied to manage and control software quality. These questions are addressed in the chapters that follow:

- What are the generic characteristics of high-quality software?
- How do we review quality, and how are effective reviews conducted?
- What is software quality assurance?
- What strategies are applicable for software testing?
- What methods are used to design effective test cases?
- Are there realistic methods that will ensure that software is correct?
- How can we manage and control changes that always occur as software is built?
- What measures and metrics can be used to assess the quality of requirements and design models, source code, and test cases?
- How do we ensure the security concerns for a product are being addressed during the software life cycle?

Once these questions are answered, you'll be better prepared to ensure that high-quality software has been produced.

CHAPTER

15 QUALITY CONCEPTS

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised. Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, *CIO Magazine* trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do" [Lev01]. Sadly, at least one survey of the state of software quality practices done in 2014 suggests that maintenance and software evolution activities make up as much as 90 percent of the total software development costs [Nan14]. Poor software quality caused by a rush to release products without adequate testing continues to plague the software industry.

KEY CONCEPTS

cost of quality 317
good enough 316
liability 320
machine learning. 322
management actions 321
quality 311
quality dilemma. 315
quality dimensions 314
quality factors 312
quantitative quality assessment. 315
risks 319
security 320

QUICK LOOK

What is it? The answer isn't as easy as you might think. You know quality when you see it, and yet, it can be an elusive thing to define. But for computer software, quality is something that we must define, and that's what we'll do in this chapter.

Who does it? Everyone—software engineers, managers, all stakeholders—involved in the software process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time to market.

What are the steps? To achieve high-quality software, four activities must occur: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure.

What is the work product? Software that meets its customer's needs, performs accurately and reliably, and provides value to all who use it.

How do I ensure that I've done it right? Track quality by examining the results of all quality control activities, and measure quality by examining errors before delivery and defects released to the field.

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both*—and that's the problem. In this chapter, we consider software quality as a concept and examine why it's worthy of serious consideration whenever software engineering practices are applied.

15.1 What Is Quality?

In his mystical book *Zen and the Art of Motorcycle Maintenance,* Robert Persig [Per74] commented on the thing we call *quality*:

> Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others; that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about . . . Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Indeed—what is it?

At a somewhat more pragmatic level, David Garvin [Gar84] of the Harvard Business School suggests that "quality is a complex and multifaceted concept" that can be described from five different points of view. The *transcendental view* argues (like Persig) that quality is something you immediately recognize but cannot explicitly define. The *user view* sees quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality. The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality. The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product. Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all these views and more.

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, and the design quality of a product increases if the product is manufactured according to specifications.

In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass [Gla98] argues that a more "intuitive" relationship is in order:

$$\text{user satisfaction} = \text{compliant product} + \text{good quality} + \text{delivery within budget and schedule}$$

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco [DeM98] reinforces this view when he states: "A product's quality is a function of how much it changes the world for the better." This view of quality contends that if a software product provides substantial benefit to its end users, they may be willing to tolerate occasional reliability or performance problems. A modern view of software quality requires attention to customer satisfaction as well as conformance to the product requirements [Max16].

15.2 SOFTWARE QUALITY

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *software* quality? In the most general sense, software quality can be defined as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

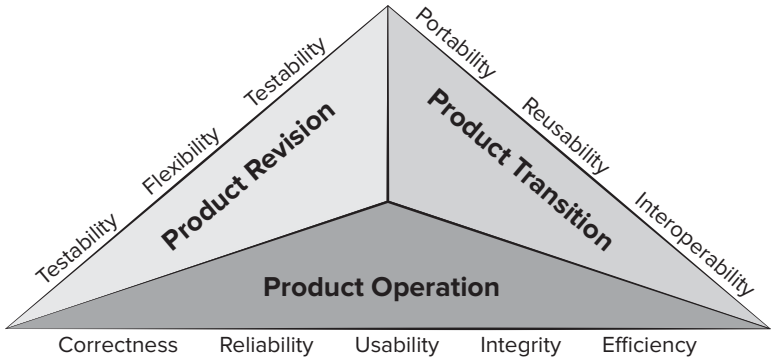
1. An *effective software process* establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high-quality software. Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.
2. A *useful product* delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software.
3. By *adding value for both the producer and user* of a software product, high-quality software provides benefits for the software organization and the end-user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process. The end result is (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial for the business.

15.2.1 Quality Factors

Several software quality models and standards have been proposed in the software engineering literature. David Garvin [Gar84] writes that quality is a multifaceted

FIGURE 15.1

McCall's software quality factors



phenomena and requires the use of multiple perspectives to assess it. McCall and Walters [McC77] proposed a useful way to think about and organize factors affecting software quality. Their software quality factors (shown in Figure 15.1) focus on three software product aspects: its operation characteristics, its ability to undergo change, and its adaptability to new environments. McCall's quality factors provide a basis for engineering software that provides high levels of user satisfaction by focusing on the overall user experience delivered by the software product. This cannot be done unless developers ensure that the requirements specification is correct and that defects are removed early in the software development process [Max16].

The ISO 25010 quality model is the newest standard (created in 2011 and revised in 2017).[1] This standard defines two quality models. The *quality in use* model describes five characteristics that are appropriate when considering using the product in a particular context (e.g., using the product on a specific platform by a human). The *product quality* model describes eight characteristics that focus on both the static and dynamic nature of computer systems.

- **Quality in Use Model**
  - **Effectiveness.** Accuracy and completeness with which users achieve goals
  - **Efficiency.** Resources expended to achieve user goals completely with desired accuracy
  - **Satisfaction.** Usefulness, trust, pleasure, comfort
  - **Freedom from risk.** Mitigation of economic, health, safety, and environmental risks
  - **Context coverage.** Completeness, flexibility
- **Product Quality**
  - **Functional suitability.** Complete, correct, appropriate
  - **Performance efficiency.** Timing, resource utilization, capacity
  - **Compatibility.** Coexistence, interoperability

1 The ISO 25010 can be found at https://www.iso.org/standard/35733.html.

- **Usability.** Appropriateness, learnability, operability, error protection, aesthetics, accessibility
- **Reliability.** Maturity, availability, fault tolerance, recoverability
- **Security.** Confidentiality, integrity, accountability, authenticity
- **Maintainability.** Modularity, reusability, modifiability, testability
- **Portability.** Adaptability, installability, replaceability

The addition of the quality in use model helps to emphasize the importance of customer satisfaction in the assessment of software quality. The product quality model points out the importance of assessing both the functional and nonfunctional requirements for the software product [Max16].

15.2.2 Qualitative Quality Assessment

The quality dimensions and factors presented in Section 15.2.1 focus on the complete software product and can be used as a generic indication of the quality of an application. Your software team can develop a set of quality characteristics and associated questions that would probe the degree to which each factor has been satisfied.[2] For example, ISO 25010 identifies *usability* as an important quality factor. If you were asked to evaluate a user interface and assess its usability, how would you proceed?

Although it's tempting to develop quantitative measures for the quality factors noted in Section 15.2.1, you can also create a simple checklist of attributes that provide a solid indication that the factor is present. You might start with the sub-characteristics suggested for usability in ISO 25010: appropriateness, learnability, operability, error protection, aesthetics, and accessibility. You and your team might decide to create a user questionnaire and a set of structured tasks for users to perform. You might observe the users while they perform these tasks and have them complete the questionnaire when they finish. We will discuss usability testing in more detail in Chapter 21.

To conduct your assessment, you and your team will need to address specific, measurable (or at least, recognizable) attributes of the interface. Your tasks might be focused on answering the following questions:

- How quickly can users determine whether the software product can be used to help them complete their task or not? (appropriateness)
- How long does it take users to learn how to use the system functions needed to complete their task? (learnability)
- Is the user able to recall how to use system functions in subsequent testing sessions without having to relearn them? (learnability)
- How long does it take users to complete tasks using the system? (operability)
- Does the system try to prevent users from making errors? (error protection)
- Does the system allow users to undo operations that may have resulted in errors? (error protection)
- Do answers give favorable responses to questions about the appearance of the user interface? (aesthetics)

2 These characteristics and questions would be addressed as part of a software review (Chapter 16).

- Does the interface conform to the expectations set forth by the golden rules from Chapter 12? (accessibility)
- Does the user interface conform to the accessibility checklist items required for the intended users? (accessibility)

As the interface design is developed, the software team would review the design prototype and ask the questions noted. If the answer to most of these questions is yes, it is likely that the user interface exhibits high quality. A collection of questions similar to these would be developed for each quality factor to be assessed. In the case of usability, it is always important to observe representative users interact with the system. For some other quality factors it may be important to test the software in the wild (or at least in the production environment).

15.2.3 Quantitative Quality Assessment

In the preceding subsections, we have presented a set of qualitative factors for the "measurement" of software quality. The software engineering community strives to develop precise measures for software quality and is sometimes frustrated by the subjective nature of the activity. Cavano and McCall [Cav78] discuss this situation:

> Subjectivity and specialization . . . apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as some way to derive quantitative measurements of software quality for objective analysis . . . Since there is no such thing as absolute knowledge, one should not expect to measure software quality exactly, for every measurement is partially imperfect.

Several software design defects can be detected using software metrics. The process consists of finding code fragments that suggest the presence of things like high coupling or unnecessary levels of complexity. Internal code attributes can be described quantitatively using software metrics. Any time software metric values computed for a code fragment fall outside the range of acceptable values, it signals the existence of a quality problem that should be investigated [Max16].

In Chapter 23, we'll present a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure *quality* but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

15.3 The Software Quality Dilemma

In an interview [Ven03] published on the Web, Bertrand Meyer discusses what we call the *quality dilemma:*

> If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway. Either you missed the market window, or you simply exhausted all your resources. So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

It's fine to state that software engineers should strive to produce high-quality systems. It's even better to apply good practices in your attempt to do so. But the situation discussed by Meyer is real life and represents a dilemma for even the best software engineering organizations. When you're faced with the quality dilemma (and everyone is faced with it at one time or another), try to achieve balance—enough effort to produce acceptable quality without burying the project.

15.3.1 "Good Enough" Software

Stated bluntly, if we are to accept the argument made by Meyer, is it acceptable to produce "good enough" software? The answer to this question must be yes, because software companies do it every day [Rod17]. They create software with known bugs and deliver it to a broad population of end users. They recognize that some of the functions and features delivered in version 1.0 may not be of the highest quality and plan for improvements in version 2.0. They do this knowing that some customers will complain, but they recognize that time to market may trump better quality as long as the delivered product is "good enough."

Exactly what is "good enough"? Good enough software delivers high-quality functions and features that end users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality.

This idea may resonate with many readers. If you're one of them, we can only ask you to consider some of the arguments against "good enough." It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in. As we noted earlier, it can argue that it will improve quality in subsequent versions. By delivering a good enough version 1.0, it has cornered the market.

If you work for a small company, be wary of this philosophy. When you deliver a good enough (buggy) product, you risk permanent damage to your company's reputation. You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.

If you work in certain application domains (e.g., real-time embedded software) or build application software that is integrated with hardware (e.g., automotive software, telecommunications software), delivering software with known bugs can be negligent and open your company to expensive litigation. In some cases, it can even be criminal. No one wants good enough aircraft avionics software! Ebert writes that the software process model should provide clear criteria to guide developers to determine what is really "good enough" for the intended application domain [Ebe14].

So, proceed with caution if you believe that "good enough" is a shortcut that can solve your software quality problems. It can work, but only for a few and only in a limited set of application domains.[3]

3 A worthwhile discussion of the pros and cons of "good enough" software can be found in [Bre02].

15.3.2 The Cost of Quality

The argument goes something like this: *We know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want.* On its face, this argument seems reasonable (see Meyer's comments earlier in this section). There is no question that quality has a cost, but lack of quality also has a cost—not only to end users who must live with buggy software, but also to the software organization that has built and must maintain it. The real question is this: *Which cost should we be worried about?* To answer this question, you must understand both the cost of achieving quality and the cost of low-quality software.

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization should collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities. Don't be afraid to incur significant prevention costs. Rest assured that your investment will provide an excellent return.

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include: (1) the cost of conducting technical reviews (Chapter 16) for software engineering work products, (2) the cost of data collection and metrics evaluation (Chapter 23), and (3) the cost of testing and debugging (Chapters 19 through 21).

Failure costs are those that would disappear if no errors appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when you detect an error in a product prior to shipment. Internal failure costs include: (1) the cost required to perform rework (repair) to correct an error, (2) the cost that occurs when rework inadvertently generates side effects that must be mitigated, and (3) the costs associated with the collection of quality metrics that allow an organization to assess the modes of failure. *External failure costs* are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real. Bad things happen when low-quality software is produced.

In an indictment of software developers who refuse to consider external failure costs, Cem Kaner [Kan95] states:

> Many of the external failure costs, such as goodwill, are difficult to quantify, and many companies therefore ignore them when calculating their cost-benefit tradeoffs. Other external failure costs can be reduced (e.g. by providing cheaper, lower-quality, post-sale support, or by charging customers for support) without increasing customer satisfaction. By ignoring the costs to our customers of bad products, quality engineers encourage quality-related decision-making that victimizes our customers, rather than delighting them.

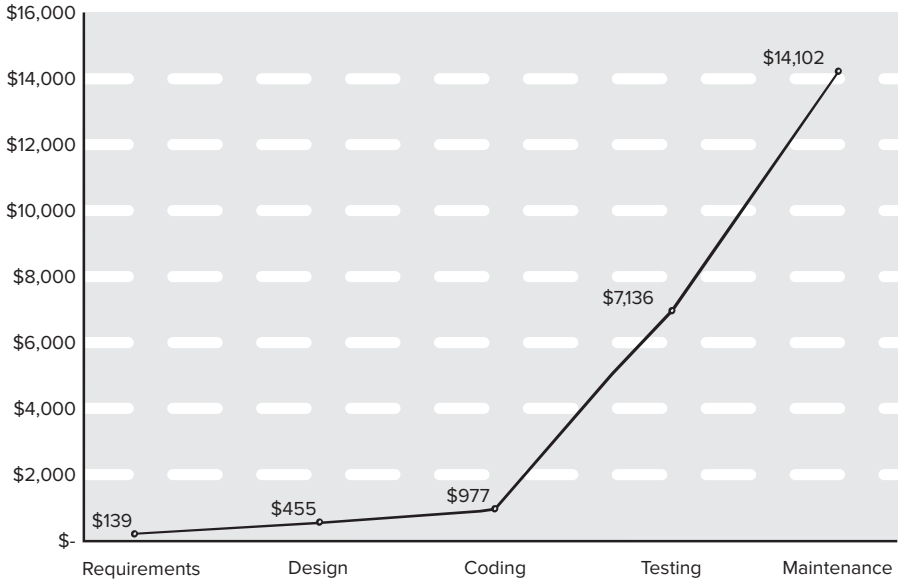


FIGURE 15.2

Relative cost of correcting errors and defects

Source: Boehm, Barry and Basili, Victor R., "Software Defect Reduction Top 10 List," IEEE Computer, vol. 34, no. 1, January 2001.

As expected, the relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 15.2, based on data collected by Boehm and Basili [Boe01b] and illustrated by Cigital Inc. [Cig07], illustrates this phenomenon.

The industry average cost to correct a defect during code generation is approximately $977 per error. The industry average cost to correct the same error if it is discovered during system testing is $7,136 per error. Cigital Inc. [Cig07] considers a large application that has 200 errors introduced during coding.

> According to industry average data, the cost of finding and correcting defects during the coding phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 × $977) is approximately $195,400.
>
> Industry average data shows that the cost of finding and correcting defects during the system testing phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 × $7,136) would have been approximately $356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 × $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been $2,472,100 ($2,115,300 + $356,800).

Even if your software organization has costs that are half of the industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

SAFEHOME



Quality Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and other members of the product software engineering team.

The conversation:

Doug: I was looking at an industry report on the costs of repairing software defects. It's pretty sobering.

Jamie: We're already working on developing test cases for each functional requirement.

Doug: That's good, but I was noticing that it costs eight times as much to repair a defect that is discovered in testing than it does if the defect is caught and repaired during coding.

Vinod: We're using pair programming, so we should be able to catch most of the defects during coding.

Doug: I think you're missing the point. Quality is more than simply removing coding errors. We need to look at the project quality goals and ensure that the evolving software products are meeting them.

Jamie: Do you mean things like usability, security, and reliability?

Doug: Yes, I do. We need to build checks into the software process to monitor our progress toward meeting our quality goals.

Vinod: Can't we finish the first prototype and then check it for quality?

Doug: I'm afraid not. We must establish a culture of quality early in the project.

Vinod: What do you want us to do, Doug?

Doug: I think we will need to find a technique that will allow us to monitor the quality of the *SafeHome* products. Let's think about this and revisit this again tomorrow.

15.3.3 Risks

In Chapter 1 of this book, we wrote, "people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right." The implication is that low-quality software increases risks for both the developer and the end user.[4] In the preceding subsection, we discussed one of these risks (cost). But the downside of poorly designed and implemented applications does not always stop with dollars and time. An extreme example [Gag04] might serve to illustrate.

Throughout the month of November 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, 5 of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.

4 In an article titled "And the 'Most Shocking Software Failure' Award Goes To . . ." Chelsea Frischnecht provides a few brief examples of what can go wrong. The article can be found at https://www.tricentis.com/blog/2017/03/01/software-fail-awards/.

The three Panamanian medical physicists, who tweaked the software to provide additional capability, were charged with second-degree murder. The U.S. company was faced with serious litigation in two countries. Gage and McCormick comment:

> This is not a cautionary tale for medical technicians, even though they can find themselves fighting to stay out of jail if they misunderstand or misuse technology. This also is not a tale of how human beings can be injured or worse by poorly designed or poorly explained software, although there are plenty of examples to make the point. This is a warning for any creator of computer programs: that software quality matters, that applications must be foolproof, and that—whether embedded in the engine of a car, a robotic arm in a factory or a healing device in a hospital—poorly deployed code can kill.

Poor quality leads to risks, some of them very serious.[5]

15.3.4 Negligence and Liability

The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity. The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., health care administration or homeland security).

Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. Litigation ensues.

In most cases, the customer claims that the developer has been negligent (in the manner in which it has applied software practices) and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.

15.3.5 Quality and Security

As the criticality of Web-based and mobile systems grows, application security has become increasingly important. Stated simply, software that does not exhibit high quality is easier to hack, and as a consequence, low-quality software can indirectly increase the security risk with all its attendant costs and problems.

In an interview in *ComputerWorld,* author and security expert Gary McGraw comments [Wil05]:

> Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws.

To build a secure system, you must focus on quality, and that focus must begin during design. The concepts and methods discussed in Part Two of this book lead to

5 In early 2019, an error in the flight control software produced by a major aircraft manufacturer was directly linked to two crashes and the deaths of 346 people.

a software architecture that reduces "flaws." A more detailed discussion of security engineering is presented in Chapter 18.

15.3.6 The Impact of Management Actions

Software quality is often influenced as much by management decisions as it is by technology decisions. Even the best software engineering practices can be subverted by poor business decisions and questionable project management actions.

In Part Four of this book we discuss project management within the context of the software process. As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

Estimation Decisions. A software team is rarely given the luxury of providing an estimate for a project *before* delivery dates are established and an overall budget is specified. Instead, the team conducts a "sanity check" to ensure that delivery dates and milestones are rational. In many cases there is enormous time-to-market pressure that forces a team to accept unrealistic delivery dates. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.

Scheduling Decisions. When a software project schedule is established (Chapter 25), tasks are sequenced based on dependencies. For example, because component **A** depends on processing that occurs within components **B, C,** and **D,** component **A** cannot be scheduled for testing until components **B, C,** and **D** are fully tested. A project schedule would reflect this. But if time is very short, and **A** must be available for further critical testing, you might decide to test **A** without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, **A** may have defects that are hidden, only to be discovered much later. Quality suffers.

Risk-Oriented Decisions. Risk management (Chapter 26) is one of the key attributes of a successful software project. You really do need to know what might go wrong and establish a contingency plan if it does. Too many software teams prefer blind optimism, establishing a development schedule under the assumption that nothing will go wrong. Worse, they don't have a way of handling things that do go wrong. As a consequence, when a risk becomes a reality, chaos reigns, and as the degree of craziness rises, the level of quality invariably falls.

The software quality dilemma can best be summarized by stating Meskimen's law: *There's never time to do it right, but always time to do it over again.* Our advice: Taking the time to do it right is almost never the wrong decision.

15.4 ACHIEVING SOFTWARE QUALITY

Software quality doesn't just appear. It is the result of good project management and solid software engineering practice. Management and practice are applied within the context of four broad activities that help a software team achieve high software quality: software engineering methods, project management techniques, quality control actions, and software quality assurance.

15.4.1 Software Engineering Methods

If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions and factors discussed in Section 15.2.

In Part Two of this book, we presented a wide array of concepts and methods that can lead to a reasonably complete understanding of the problem and a comprehensive design that establishes a solid foundation for the construction activity. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially.

15.4.2 Project Management Techniques

The impact of poor management decisions on software quality has been discussed in Section 15.3.6. The implications are clear: If (1) a project manager uses estimation to verify that delivery dates are achievable, (2) schedule dependencies are understood and the team resists the temptation to use shortcuts, (3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.

In addition, the project plan should include explicit techniques for quality and change management. Techniques that lead to good project management practices are discussed in Part Four of this book.

15.4.3 Machine Learning and Defect Prediction

Defect prediction [Mun17] is an important part of identifying software components that may have quality concerns. Defect prediction models use statistical techniques to examine the relationships among combinations of software metrics and software components containing known software defects. They can be an efficient and effective way for software developers to quickly identify defect-prone classes. This can reduce costs and development times [Mal16].

Machine learning is an application of artificial intelligence (AI) techniques that provide systems with the ability to learn and improve from experience without being explicitly programmed. Stated another way, machine learning focuses on the development of computer programs that can access data and use the data to learn for themselves. Machine learning techniques can be used to automate the process of discovering predictive relationships between software metrics and defective components [Ort17], [Li16], [Mal16].

Machine learning systems process large data sets containing representative combinations of metrics for defective and nondefective software components. These data are used to tune classification algorithms. Once the system has built a prediction model though this type of training, it can be used for quality assessment and defect prediction on data associated with future software products. Building these types of classifiers is a big part of what modern data scientists do. More discussion on the use of data science and software engineering appears in Appendix 2 of this book.

15.4.4 Quality Control

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that

they are complete and consistent. Code may be inspected to uncover and correct errors before testing commences. A series of testing steps is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fails to meet quality goals. Quality control activities are discussed in detail throughout the remainder of Part Three of this book.

15.4.5 Quality Assurance

Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working. Of course, if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues. Software quality assurance is discussed in detail in Chapter 17.

15.5 SUMMARY

Concern for the quality of the software-based systems has grown as software becomes integrated into every aspect of our daily lives. But it is difficult to develop a comprehensive description of software quality. In this chapter, quality has been defined as an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

A wide variety of software quality dimensions and factors has been proposed over the years. All try to define a set of characteristics that, if achieved, will lead to high software quality. McCall's and the ISO 25010 quality factors establish characteristics such as reliability, usability, maintainability, functionality, and portability as indicators that quality exists.

Every software organization is faced with the software quality dilemma. In essence, everyone wants to build high-quality systems, but the time and effort required to produce "perfect" software are simply unavailable in a market-driven world. The question becomes, Should we build software that is "good enough"? Although many companies do just that, there is a significant downside that must be considered.

Regardless of the approach that is chosen, quality does have a cost that can be discussed in terms of prevention, appraisal, and failure. Prevention costs include all software engineering actions that are designed to prevent defects in the first place. Appraisal costs are associated with those actions that assess software work products to determine their quality. Failure costs encompass the internal price of failure and the external effects that poor quality precipitates.

Software quality is achieved through the application of software engineering methods, solid management practices, and comprehensive quality control—all supported by a software quality assurance infrastructure. In the chapters that follow, quality control and assurance are discussed in some detail.

PROBLEMS AND POINTS TO PONDER

15.1. Describe how you would assess the quality of a university before applying to it. What factors would be important? Which would be critical?

15.2. Using the definition of software quality proposed in Section 15.2, do you think it's possible to create a useful product that provides measurable value without using an effective process? Explain your answer.

15.3. Using the subattributes noted for the ISO 25010 quality factor "maintainability" in Section 15.2.1, develop a set of questions that explore whether or not these attributes are present. Follow the example shown in Section 15.2.2.

15.4. Describe the software quality dilemma in your own words.

15.5. What is "good enough" software? Name a specific company and specific products that you believe were developed using the good enough philosophy.

15.6. Considering each of the four aspects of the cost of quality, which do you think is the most expensive and why?

15.7. Do a Web search and find three other examples of "risks" to the public that can be directly traced to poor software quality. Consider beginning your search at http://catless.ncl.ac.uk/risks.

15.8. Are *quality* and *security* the same thing? Explain.

15.9. Explain why it is that many of us continue to live by Meskimen's law. What is it about the software business that causes this?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

16

REVIEWS—A RECOMMENDED APPROACH

Software reviews are a "filter" for the software process work flow. Too few, and the flow is "dirty." Too many, and the flow slows to a trickle. Reviews are applied at various points during software engineering and serve to uncover errors and defects. Software reviews "purify" software engineering work products, including requirements and design models, code, and testing data. Using metrics, you can determine which reviews work and emphasize them and at the same time remove ineffective reviews from the flow to accelerate the process.

KEY CONCEPTS

bugs 326
cost effectiveness 329
defect amplification 327
defects 326
error density 328
errors 326
informal reviews 331
record keeping 333
review reporting 333
technical reviews 326

QUICK LOOK

What is it? You'll make mistakes as you develop software engineering work products. There's no shame in that—as long as you try hard, very hard, to find and correct the mistakes before they are delivered to end users. Technical reviews are the most effective mechanism for finding mistakes early in the software process.

Who does it? Software engineers perform technical reviews, also called peer reviews, with their colleagues. As we discussed in Chapters 3 and 4, sometimes it is wise to include other stakeholders in these reviews.

Why is it important? If you find an error early in the process, it is less expensive to correct. In addition, errors have a way of amplifying as the process proceeds. So a relatively minor error left untreated early in the process can be amplified into a major set of errors later in the project. Finally, reviews save time by reducing the amount of rework that will be required late in the project.

What are the steps? Your approach to reviews will vary depending on the type of review you select. In general, six steps are employed, although not all are used for every type of review: planning, preparation, structuring the meeting, noting errors, making corrections (done outside the review), and verifying that corrections have been performed properly.

What is the work product? The output of a review is a list of issues and/or errors that have been uncovered. In addition, the technical status of the work product is also indicated.

How do I ensure that I've done it right? First, select the type of review that is appropriate for your development culture. Follow the guidelines that lead to successful reviews. If the reviews that you conduct lead to higher-quality software, you've done it right.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software architecture to an audience of customers, management, and technical staff is also a form of review. In this book, however, we focus on *technical* or *peer reviews,* exemplified by *casual reviews, walkthroughs,* and *inspections.* A technical review (TR) is the most effective filter from a quality control standpoint. Conducted by software engineers and other stakeholders for all project team members, the TR is an effective means for uncovering errors and improving software quality.

16.1 Cost Impact of Software Defects

Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end users (or to another framework activity in the software process). In earlier chapters, we used the term *error* to depict a quality problem that is discovered by software engineers (or others) *before* the software is released to the end user (or to another framework activity in the software process).

INFO



Bugs, Errors, and Defects

The goal of software quality control, and in a broader sense, quality management in general, is to remove quality problems in the software. These problems are referred to by various names—*bugs, faults, errors,* or *defects,* to name a few. Are these terms synonymous, or are there subtle differences between them?

In this book we make a clear distinction between an *error* (a quality problem found *before* the software is released to other stakeholders or end users) and a *defect* (a quality problem found only *after* the software has been released to end users or other stakeholders).[1] We make this distinction because errors and defects have very different economic, business, psychological, and human impacts. As software engineers, we want to find and correct as many errors as possible before the customer and/or end user encounter them. We want to avoid defects—because defects (justifiably) make software people look bad.

It is important to note, however, that the temporal distinction made between errors and defects in this book is *not* mainstream thinking. The general consensus within the software engineering community is that defects and errors, faults, and bugs are synonymous. That is, the point in time that the problem was encountered has no bearing on the term used to describe the problem. Part of the argument in favor of this view is that it is sometimes difficult to make a clear distinction between pre- and postrelease (e.g., consider an incremental process used in agile development).

Regardless of how you choose to interpret these terms, recognize that the point in time at which a problem is discovered does matter and that software engineers should try hard—*very* hard—to find problems before their customers and end users encounter them.

1 If software process improvement is considered, a quality problem that is propagated from one process framework activity (e.g., **modeling**) to another (e.g., **construction**) can also be called a "defect" because the problem should have been found before a work product (e.g., a design model) was "released" to the next activity.

The primary objective of a formal technical review (FTR) is to find errors before they are passed on to another software engineering activity or released to the end user. The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, review techniques have been shown to be up to 75 percent effective [Jon86] in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent activities in the software process. We have known this for decades, and yet there are still many developers who do not believe the time spent on reviews is almost always less than the time required to rewrite bad code [Yad17].

16.2 Defect Amplification and Removal

Defect amplification is a concept originally proposed almost four decades ago [IBM81]. It helps to justify the effort expended on software reviews. In essence, defect amplification makes the following argument—an error introduced early in the software engineering work flow (e.g., during requirement modeling) and undetected, can and often will be amplified into multiple errors during design. If those errors are not uncovered (using effective reviews), they themselves may be further amplified into still more errors during coding. A single error introduced early and not uncovered and corrected can amplify into multiple errors later in the process. *Defect propagation* is a term used to describe the impact an undiscovered error has on future development activities or product behavior [Vit17].

As a development team moves deeper into the software process, the cost of finding and fixing an error grows. This simple reality is exacerbated by defect amplification and propagation because a single error may become multiple errors downstream. The cost of finding and fixing a single error can be signficant, but the cost to find and fix multiple errors propagated by a single earlier error is substantially more significant.

To conduct reviews, you must expend time and effort, and your development organization must spend money. However, the reality of defect amplification and propagation leaves little doubt that you can pay now or pay much more later. This is what *technical debt* (Chapter 11) is all about [Xia16] [Vit17].

16.3 Review Metrics and Their Use

Technical reviews are one of many actions that are required as part of good software engineering practice. Each action requires dedicated human effort. Because available project effort is finite, it is important for a software engineering organization to understand the effectiveness of each action by defining a set of metrics (Chapter 23) that can be used to assess their efficacy.

Although many metrics can be defined for technical reviews, a relatively small subset can provide useful insight. The following review metrics can be collected for each review that is conducted:

- **Preparation effort,** E_p. The effort (in person-hours) required to review a work product prior to the actual review meeting
- **Assessment effort,** E_a. The effort (in person-hours) that is expended during the actual review
- **Rework effort,** E_r. The effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- **Review effort,** E_{review}. Represents the sum of effort measures for reviews:

  $$E_{\text{review}} = E_p + E_a + E_r$$

- **Work product size (WPS).** A measure of the size of the work product that has been reviewed (e.g., the number of UML models, the number of document pages, or the number of lines of code)
- **Minor errors found,** $\text{Err}_{\text{minor}}$. The number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- **Major errors found,** $\text{Err}_{\text{major}}$. The number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)
- **Total errors found,** Err_{tot}. Represents the sum of the errors found:

  $$\text{Err}_{\text{tot}} = \text{Err}_{\text{minor}} + \text{Err}_{\text{major}}$$

- **Error density.** Represents the errors found per unit of work product reviewed:

  $$\text{Error density} = \frac{\text{Err}_{\text{tot}}}{\text{WPS}}$$

How might these metrics be used? As an example, consider a requirements model that is reviewed to uncover errors, inconsistencies, and omissions. It would be possible to compute the error density in several different ways. Assume the requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials. The review uncovers 18 minor errors and 4 major errors. Therefore, $\text{Err}_{\text{tot}} = 22$. Error density is 1.2 errors per UML diagram, or 0.68 errors per requirements model page.

If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases), the percentage of errors uncovered for each review can be computed against the total number of errors found for all reviews. In addition, the error density for each work product can be computed.

Once data are collected for many reviews conducted across many projects, average values for error density enable you to estimate the number of errors to be found in a new document before it is reviewed. For example, if the average error density for a requirements model is 0.68 errors per page, and a new requirements model is 40 pages long, a rough estimate suggests that your software team will find around 27 errors during the review of the document. If you find only 9 errors, you've either done an extremely good job in developing the requirements model or your review approach was not thorough enough.

It is difficult to measure the cost effectiveness of any technical review in real time. A software engineering organization can assess the effectiveness of reviews and their cost benefit only after reviews have been completed, review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured (via testing).

Returning to the previous example, the average error density for requirements models was determined to be 0.68 per page. The effort required to correct a minor model error (immediately after the review) has been found to require 4 person-hours. The effort required for a major requirement error has been found to be 18 person-hours. Examining the review data collected, you find that minor errors occur about six times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.

Requirements-related errors uncovered during testing require an average of 45 person-hours to find and correct (no data are available on the relative severity of the error). Using the averages noted, we get:

$$\begin{aligned}\text{Effort saved per error} &= E_{\text{testing}} - E_{\text{reviews}} \\ &= 45 - 6 = 39 \text{ person-hours/error}\end{aligned}$$

Because 22 errors were found during the review of the requirements model, a savings of about 858 person-hours of testing effort would be achieved. And that's just for requirements-related errors. Errors associated with design and code would add to the overall benefit.

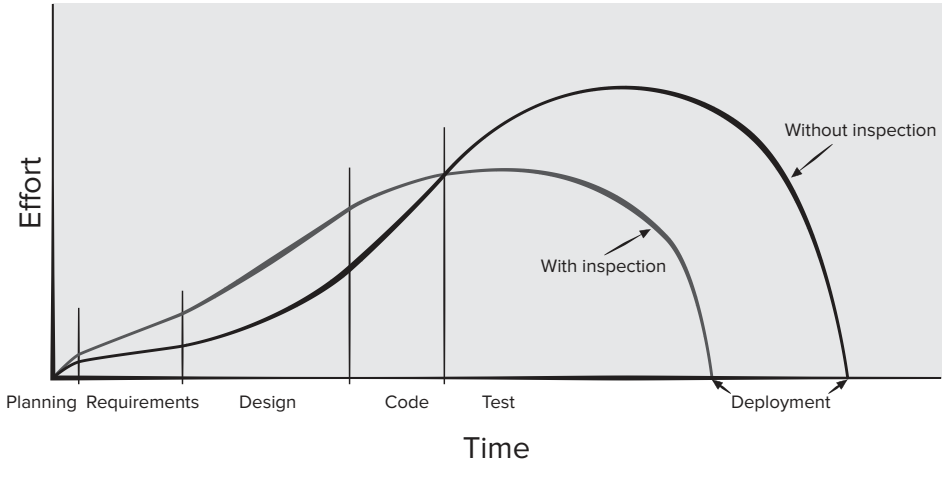
The bottom line—effort saved leads to shorter delivery cycles and improved time to market. The example presented in this section suggests this may be true. More importantly, industry data for software reviews has been collected for more than three decades and is summarized qualitatively using the graphs shown in Figure 16.1.

Referring to the figure, the effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced. It is important to note

FIGURE 16.1

Effort expended with and without reviews

Source: Fagan, Michael E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, vol. SE-12, no. 7, July 1986, 744–751.



that the deployment date for development with reviews is sooner than the deployment date without reviews. Reviews don't take time; they save it!

16.4 Criteria for Types of Reviews

Technical reviews can be classified as either formal or informal or somewhere in between these two extremes. The level of formality is chosen to match the type of product to be built, the project time line, and the people who are doing the work. Figure 16.2 depicts a reference model for technical reviews [Lai02] that identifies four characteristics that contribute to the formality with which a review is conducted.

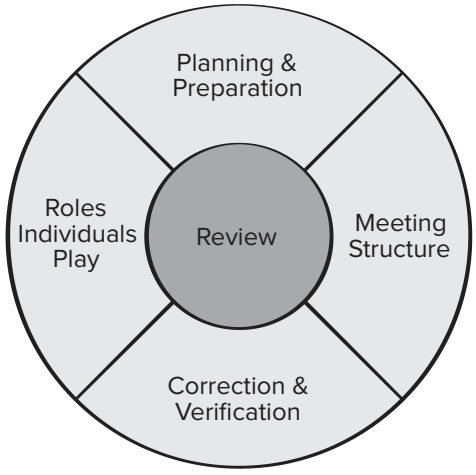
Each of the reference model characteristics helps to define the level of review formality. The formality of a review increases when (1) distinct roles are explicitly defined for the reviewers, (2) there is a sufficient amount of planning and preparation for the review, (3) a distinct structure for the review (including tasks and internal work products) is defined, and (4) follow-up by the reviewers occurs for any corrections that are made.

An element that is not presented in this model is the frequency of the reviews themselves. If you are using an agile prototyping model (Chapter 4) that contains relatively short sprints, your team may opt for less formal reviews because the reviews are happening fairly often. This usually means that defects are caught sooner and more often.

To understand the reference model, let's assume that you've decided to review the interface design for **SafeHomeAssured.com.** You can do this in a variety of different ways that range from relatively casual to extremely rigorous. If you decide that the casual approach is most appropriate, you ask a few colleagues (peers) to examine the interface prototype in an effort to uncover potential problems. All of you decide that there will be no advance preparation, but that you will evaluate the prototype in a reasonably structured way—looking at layout first, aesthetics next, navigation options after that, and so on. As the designer, you decide to take a few notes, but nothing formal.

FIGURE 16.2

Reference model for technical reviews



But what if the interface is pivotal to the success of the entire project? What if human lives depended on an interface that was ergonomically sound? You might decide that a more rigorous approach was necessary. A review team would be formed. Each person on the team would have a specific role to play—leading the team, recording findings, presenting the material, and so on. Each reviewer would be given access to the work product (in this case, the interface prototype) before the review and would spend time looking for errors, inconsistencies, and omissions. A set of specific tasks would be conducted based on an agenda that was developed before the review occurred. The results of the review would be formally recorded, and the team would decide on the status of the work product based on the outcome of the review. Members of the review team might also verify that the corrections made were done properly.

In this book we consider two broad categories of technical reviews: informal reviews and more formal technical reviews. Within each category, a number of different approaches can be chosen. These are presented in the sections that follow.

16.5 INFORMAL REVIEWS

Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product, or the review-oriented aspects of pair programming (Chapter 3).

A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

One way to improve the efficacy of a desk check review is to develop a set of simple review checklists[2] for each major work product produced by the software team. The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product. For example, let's reexamine a desk check of the interface prototype for **SafeHomeAssured.com.** Rather than simply playing with the prototype at the designer's workstation, the designer and a colleague examine the prototype using a checklist for interfaces:

- Is the layout designed using standard conventions? Left to right? Top to bottom?
- Does the presentation need to be scrolled?
- Are color and placement, typeface, and size used effectively?
- Are all navigation options or functions represented at the same level of abstraction?
- Are all navigation choices clearly labeled?

2 Literally hundreds of technical review checklists can be found via a web search. For example, a useful code review checklist can be downloaded from https://courses.cs.washington.edu/courses/cse403/12wi/sections/12wi_code_review_checklist.pdf.

and so on. Any errors or issues noted by the reviewers are recorded by the designer for resolution at a later time. Desk checks may be scheduled in an ad hoc manner, or they may be mandated as part of good software engineering practice. In general, the amount of material to be reviewed is relatively small and the overall time spent on a desk check spans little more than 1 or 2 hours.

In Chapter 3, we described *pair programming* in the following manner: XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance.

Pair programming (Section 3.5.1) can be characterized as a continuous desk check. Rather than scheduling a review at some point in time, pair programming encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality.

Some software engineers argue that the inherent redundancy built into pair programming is wasteful of resources. After all, why assign two people to a job that one person can accomplish? The answer to this question can be found in Section 16.3. If the quality of the work product produced as a consequence of pair programming is significantly better than the work of an individual, the quality-related savings can more than justify the "redundancy" implied by pair programming.

16.6 FORMAL TECHNICAL REVIEWS

A *formal technical review* (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are: (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR is actually a class of reviews that includes *walkthroughs* and *inspections*. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs, see [Rad02], [Wie02], or [Fre90].

16.6.1 The Review Meeting

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than 2 hours of work for each person.
- The duration of the review meeting should be less than 2 hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing the focus, the FTR has a higher likelihood of uncovering errors.

The focus of the FTR is on a work product (e.g., a self-contained portion of a requirements model, a detailed component design, or source code for a component). The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required. The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation. Each reviewer is expected to spend between 1 and 2 hours reviewing the product, making notes, and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of a *recorder,* that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each. Don't point out errors too harshly. One way to be gentle is to ask a question that enables the producer to discover the error.

At the end of the review, all attendees of the FTR must decide whether to: (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

16.6.2 Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting, and a *review issues list* is produced. In addition, a *formal technical review summary report* is completed. A review summary report answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single-page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow-up to the review leader.

16.6.3 Review Guidelines

Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse than no review at all. The following represents a minimum set of guidelines for formal technical reviews:

1. **Review the product, not the producer.** An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.
2. **Set an agenda and maintain it.** One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. **Limit debate and rebuttal.** When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. **Enunciate problem areas, but don't attempt to solve every problem noted.** A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. **Take written notes.** It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. **Limit the number of participants and insist upon advance preparation.** Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance.
7. **Develop a checklist for each product that is likely to be reviewed.** A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even testing work products.
8. **Allocate resources and schedule time for FTRs.** For reviews to be effective, they should be scheduled as a task during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.

9. **Conduct meaningful training for all reviewers.** The training should stress both process-related issues and the human psychological side of reviews.
10. **Review your early reviews.** Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a software organization should experiment to determine what approach works best in a local context.

In an ideal setting, every software-engineering work product would undergo a formal technical review. In the real world of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized. However, full FTR resources should be used on those work products that are likely to be error prone.

SAFEHOME



Quality Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and other members of the product software engineering team.

The conversation:

Doug: I know we didn't spend time developing a quality plan for this project, but we're already into it and we have to consider quality . . . right?

Jamie: Sure. We've already decided that as we develop the requirements model [Chapter 8], Ed has committed to develop a testing procedure for each requirement.

Doug: That's really good, but we're not going to wait until testing to evaluate quality, are we?

Vinod: No! Of course not. We've got reviews scheduled into the project plan for this software increment. We'll begin quality control with the reviews.

Jamie: I'm a bit concerned that we won't have enough time to conduct all the reviews. In fact, I know we won't.

Doug: Hmmm. So what do you propose?

Jamie: I say we select those elements of the requirements and design model that are most critical to *SafeHome* and review them.

Vinod: But what if we miss something in a part of the model we don't review?

Jamie: Maybe . . . but I'm not sure we even have time to review every element of the models.

Vinod: What do you want us to do, Doug?

Doug: Let's steal something from Extreme Programming [Chapter 3]. We'll develop the elements of each model in pairs—two people—and conduct an informal review of each as we go. We'll then target "critical" elements for a more formal team review, but keep those reviews to a minimum. That way, everything gets looked at by more than one set of eyes, but we still maintain our delivery dates.

Jamie: That means we're going to have to revise the schedule.

Doug: So be it. Quality trumps schedule on this project.

16.7 Postmortem Evaluations

Many lessons can be learned if a software team takes the time to evaluate the results of a software project after the software has been delivered to end users. Baaz and his colleagues [Baa10] suggest the use of a *postmortem evaluation* (PME) as a mechanism to determine what went right and what went wrong when software engineering process and practice is applied in a specific project.

Unlike an FTR that focuses on a specific work product, it is more like a Scrum retrospective (Section 3.4.5). A PME examines the entire software project, focusing on both "*excellences* (that is, achievements and positive experiences) and *challenges* (problems and negative experiences)" [Baa10]. Often conducted in a workshop format, a PME is attended by members of the software team and stakeholders. The intent is to identify excellences and challenges and to extract lessons learned from both. The objective is to suggest improvements to both process and practice going forward. Many software engineers regard PME documents as some of the most valuable documents to save in the project archive.

16.8 Agile Reviews

It's not surprising that some software engineers balk at the thought of including reviews of any kind in agile development processes. Yet failing to catch defects early can be costly in terms of time and resources. Ignoring technical debt does not make it go away. Agile developers have the same need to find defects early (and often) that all software developers have. Admittedly, it may be somewhat more difficult to get agile developers to make use of metrics, but many of them can be collected unobtrusively.

If we take a closer look at the Scrum framework (Section 3.4), there are several places where informal and formal reviews take place. During the sprint planning meeting, user stories are reviewed and ordered according to priority, before selecting the user stories to include in the next sprint. The daily Scrum meeting is an informal way to ensure that the team members are all working on the same priorities and to catch any defects that may prevent completing the sprint on time. Agile developers often use pair programming, another informal review technique. The sprint review meeting is often conducted using guidelines similar to those discussed for a formal technical review. The code producers walk through the user stories selected for the sprint and demonstrate to the product owner that all functionality is present. Unlike the FTR, the product owner has the final word on whether to accept the sprint prototype or not.

If we look at the evaluate prototype portion of our recommended process model (Section 4.5), this task is also likely to be conducted as a formal technical review with development risk assessment added to it. We mentioned previously (Section 16.7) that the sprint retrospective meeting is really similar to the project postmortem meeting in that the development team is trying to capture its lessons learned. A major aspect of software quality assurance is being able to repeat your successes and avoid repeating your mistakes.

16.9 SUMMARY

The intent of every technical review is to find errors and uncover issues that would have a negative impact on the software to be deployed. The sooner an error is uncovered and corrected, the less likely that error will propagate to other software engineering work products and amplify itself, resulting in significantly more effort to correct it.

To determine whether quality control activities are working, a set of metrics should be collected. Review metrics focus on the effort required to conduct the review and the types and severity of errors uncovered during the review. Once metrics data are collected, they can be used to assess the efficacy of the reviews you do conduct. Industry data indicate that reviews provide a significant return on investment.

A reference model for review formality identifies the roles people play, planning and preparation, meeting structure, correction approach, and verification as the characteristics that indicate the degree of formality with which a review is conducted. Informal reviews are casual in nature but can still be used effectively to uncover errors. Formal reviews are more structured and have the highest probability of leading to high-quality software.

Informal reviews are characterized by minimal planning and preparation and little record keeping. Desk checks and pair programming fall into the informal review category.

A formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors. Formal technical reviews establish defined roles for each reviewer, encourage planning and advance preparation, require the application of defined review guidelines, and mandate record keeping and status reporting.

PROBLEMS AND POINTS TO PONDER

16.1. Explain the difference between an *error* and a *defect.*

16.2. Why can't we just wait until testing to find and correct all software errors?

16.3. Assume that 10 errors have been introduced in the requirements model and that each error will be amplified by a factor of 2:1 into design and an additional 20 design errors are introduced and then amplified 1.5:1 into code where an additional 30 errors are introduced. Assume further that all unit testing will find 30 percent of all errors, integration will find 30 percent of the remaining errors, and validation tests will find 50 percent of the remaining errors. No reviews are conducted. How many errors will be released to the end users?

16.4. Reconsider the situation described in Problem 16.3, but now assume that requirements, design, and code reviews are conducted and are 60 percent effective in uncovering all errors at that step. How many errors will be released to the field?

16.5. Reconsider the situation described in Problems 16.3 and 16.4. If each of the errors released to the field costs $4800 to find and correct and each error found in review costs $240 to find and correct, how much money is saved by conducting reviews?

16.6. Describe the meaning of Figure 16.1 in your own words.

16.7. Can you think of a few instances in which a desk check might create problems rather than provide benefits?

16.8. A formal technical review is effective only if everyone has prepared in advance. How do you recognize a review participant who has not prepared? What do you do if you're the review leader?

16.9. How is technical debt addressed in agile process models?

16.10. Consider the review guidelines presented in Section 16.6.3. Which do you think is most important and why?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

17 SOFTWARE QUALITY ASSURANCE

The software engineering approach described in this book works toward a single goal: to produce on-time, high-quality software. Yet many readers will be challenged by the question: "What is software quality?"

KEY CONCEPTS

Bayesian inference 351
elements of software quality assurance 341
formal approaches 347
genetic algorithms 352
goals 346
ISO 9001:2015 standard 354
Six Sigma 349
software reliability 350
software safety 352
SQA plan 354
SQA tasks 343
statistical software quality assurance 347



QUICK LOOK

What is it? It's not enough to talk the talk by saying that software quality is important. You have to (1) explicitly define what is meant when you say "software quality," (2) create a set of activities that will help ensure that every software engineering work product exhibits high quality, (3) perform quality control and assurance activities on every software project, (4) use metrics to develop strategies for improving your software process and, as a consequence, the quality of the end product.

Who does it? Everyone involved in the software engineering process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time to market.

What are the steps? Before software quality assurance (SQA) activities can be initiated, it is important to define *software quality* at several different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

What is the work product? A Software Quality Assurance Plan is created to define a software team's SQA strategy. During modeling and coding, the primary SQA work product is the output of technical reviews (Chapter 16). During testing (Chapters 19 through 21), test plans and procedures are produced. Other work products associated with process improvement may also be generated.

How do I ensure that I've done it right? Find errors before they become defects! That is, work to improve your defect removal efficiency (Chapter 23), thereby reducing the amount of rework that your software team must perform.

Philip Crosby [Cro79], in his landmark book on quality, provides a wry answer to this question:

> The problem of quality management is not what people don't know about it. The problem is what they think they do know . . .
>
> In this regard, quality has much in common with sex. Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

Indeed, quality is a challenging concept—one that we addressed in some detail in Chapter 15.[1]

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! *Software quality assurance* (often called *quality management*) is an umbrella activity (Chapter 2) that is applied throughout the software process.

Software quality assurance (SQA) encompasses (Figure 17.1): (1) an SQA process, (2) specific quality assurance and quality control tasks (including technical reviews and a multitiered testing strategy), (3) effective software engineering practice (methods and tools), (4) control of all software work products and the changes made to them (Chapter 22), (5) a procedure to ensure compliance with software development standards (when applicable), and (6) measurement and reporting mechanisms.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does "the right things at the right time in the right way."

FIGURE 17.1

Software quality assurance



1 If you have not read Chapter 15, you should do so now.

17.1 Background Issues

Quality control and assurance are essential activities for any business that produces products to be used by others. Prior to the twentieth century, quality control was the sole responsibility of the craftsperson who built a product. As time passed and mass production techniques became commonplace, quality control became an activity performed by people other than the ones who built the product.

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement [Dem86] as key elements of quality management.

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE17]. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" [Sch01] that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors noted in Chapter 15? Have software engineering practices been conducted according to preestablished standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

17.2 Elements of Software Quality Assurance

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner [Hor03]:

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers (Chapter 16). Their

intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing (Chapters 19 through 21) is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices (Chapter 22) have been instituted.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement (Chapter 28) and is a key proponent and sponsor of educational programs.

Vendor management. Three categories of software are acquired from external software vendors—*shrink-wrapped packages* (e.g., Microsoft Office), a *tailored shell* [Hor03] that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and *contracted software* that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible) and incorporating quality mandates as part of any contract with an external vendor.

Security management. With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for mobile apps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security (Chapter 18).

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. Although the analysis and mitigation of risk (Chapter 26) is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

In addition to each of these concerns and activities, SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern.

17.3 SQA Processes and Product Characteristics

As we begin a discussion of software quality assurance, it's important to note that SQA procedures and approaches that work in one software environment may not work as well in another. Even within a company that adopts a consistent approach[2] to software engineering, different software products may exhibit different levels of quality [Par11].

The solution to this dilemma is to understand the specific quality requirements for a software product and then select the process and specific SQA actions and tasks that will be used to meet those requirements. The Software Engineering Institute's CMMI and ISO 9000 standards are the most commonly used process frameworks. Each proposes "a syntax and semantics" [Par11] that will lead to the implementation of software engineering practices that improve product quality. Rather than instantiating either framework in its entirety, a software organization can "harmonize" the two models by selecting elements of both frameworks and matching them to the quality requirements of an individual product.

17.4 SQA Tasks, Goals, and Metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Modern software quality assurance is often data driven, as shown in Figure 17.2. The product stakeholders define goals and quality measures, problem areas are identified, indicators are measured, and a determination is made as to whether or not process changes are needed. Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

17.4.1 SQA Tasks

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies evaluations to be performed, audits and

2 For example, CMMI-defined process and practices (Chapter 28).

FIGURE 17.2

Software quality assurance



reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance, and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these activities, the SQA group coordinates the control and management of change (Chapter 22) and helps to collect and analyze software metrics.

SAFEHOME



Software Quality Assurance

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and other members of the product software engineering team.

The conversation:

Doug: How are things going with the informal reviews?

Jamie: We're conducting informal reviews of the critical project elements in pairs as we code but before testing. It's going faster than I thought.

Doug: That's good, but I want to have Bridget Thorton's SQA group conduct audits of our work products to ensure that we're following our processes and meeting our quality goals.

Venod: Aren't they already doing the bulk of the testing?

Doug: Yes, they are. But QA is more than testing. We need to be sure that our documents are evolving along with our code and that we're making sure we don't introduce errors as we integrate new components.

Jamie: I really don't want to be evaluated based on their findings.

Doug: No worries. The audits are focused on conformance of our work products to the requirements and the process activities we've defined. We'll only be using audit results to try to improve our processes as well as our software products.

Vinod: I must believe it's going to take more of our time.

Doug: In the long run it will save us time when we find defects earlier. It also costs less to fix defects if they're caught early.

Jamie: That sounds like a good thing then.

Doug: It's also important to identify the activities where defects were introduced and add review tasks to catch them in the future.

Vinod: That'll help us determine if we're sampling carefully enough with our review activities.

Doug: I think SQA activities will make us a better team in the long run.

17.4.2 Goals, Attributes, and Metrics

The SQA activities described in the preceding section are performed to achieve a set of pragmatic goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality. Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Table 17.1 (adapted from [Hya96]) identifies the attributes that are indicators for the existence of quality for each of the goals discussed. Metrics that can be used to indicate the relative strength of an attribute are also shown.

TABLE 17.1

Software quality goals, attributes, and metrics

Source: Adapted from Hyatt, L., and Rosenberg, L., "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," NASA SATC, 1996.

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors
	Understandability	Percent internal comments
		Variable naming conventions
		Percent reused components
	Reusability	Percent reused component
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

17.5 Formal Approaches to SQA

In the preceding sections, we have argued that software quality is everyone's job and that it can be achieved through competent software engineering practice as well as through the application of technical reviews, a multitiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards and process frameworks. In addition, quality can be defined in terms of a broad array of quality attributes and measured (indirectly) using a variety of indices and metrics.

Over the past three decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements is available. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

Attempts to prove programs correct are not new. Dijkstra [Dij76a] and Linger, Mills, and Witt [Lin79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts. Although formal methods are interesting to some software engineering researchers, commercial developers rarely make use of formal methods in 2018.

17.6 Statistical Software Quality Assurance

Statistical quality assurance reflects a growing trend throughout the industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

This relatively simple concept represents an important step toward the creation of an adaptive software process in which changes are made to improve those elements of the process that introduce error.

17.6.1 A Generic Example

To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on errors and defects for a period of 1 year. Some of the errors are uncovered as software is being developed. Other defects are encountered after the software has been released to its end users.

Although hundreds of different problems are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or erroneous specifications (IES)
- Misinterpretation of customer communication (MCC)
- Intentional deviation from specifications (IDS)
- Violation of programming standards (VPS)
- Error in data representation (EDR)
- Inconsistent component interface (ICI)
- Error in design logic (EDL)
- Incomplete or erroneous testing (IET)
- Inaccurate or incomplete documentation (IID)
- Error in programming language translation of design (PLT)
- Ambiguous or inconsistent human/computer interface (HCI)
- Miscellaneous (MIS)

To apply statistical SQA, a table is built (see Table 17.2). The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, you might implement requirements gathering techniques (Chapter 7) to improve the quality of customer communication and specifications. To improve EDR, you might acquire tools for data modeling and perform more stringent data design reviews.

TABLE 17.2

Data collection for statistical SQA

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement (e.g., [Rya11], [Art97]). In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

17.6.2 Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a business-management strategy designed to improve the quality of process outputs by minimizing variation and causes of defects in processes. It is a subset of the Total Quality Management (TQM) methodology with a heavy focus on statistical applications used to reduce costs and improve quality" [Voe18]. The term *Six Sigma* is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high-quality standard. The Six Sigma methodology defines three core steps:

- *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication.
- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

- *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
- *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject. If you have further interest, see [Voe18], [Pyz14], and [Sne18].

17.7 SOFTWARE RELIABILITY

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

Software reliability, unlike many other quality factors, can be measured directly and estimated using historical and developmental data. *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [Mus87]. To illustrate, program *X* is estimated to have a reliability of 0.999 over eight elapsed processing hours. In other words, if program *X* were to be executed 1000 times and require a total of 8 hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 999 times.

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term *failure*? In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds, while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

17.7.1 Measures of Reliability and Availability

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear (see Chapter 1) is not a factor.

There has been an ongoing debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet to be established, it is worthwhile to consider a few simple concepts that apply to both system elements.

If we consider a computer-based system, a simple measure of reliability is *mean time between failure* (MTBF):[3]

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

where the acronyms MTTF and MTTR are *mean time to failure* and *mean time to repair*,[4] respectively.

3 It is important to note that MTBF and related measures are based on CPU time, not wall clock time.

4 Although debugging (and related corrections) may be required following a failure, in many cases the software will work properly after a restart with no other change.

Many researchers argue that MTBF is a far more useful measure than other quality-related software metrics discussed in Chapter 23. Stated simply, an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 3000 processor hours without failure. Many defects in this program may remain undetected for tens of thousands of hours before they are discovered. The MTBF of such obscure errors might be 30,000 or even 60,000 processor hours. Other defects, not yet discovered, might have a failure rate of 4000 or 5000 hours. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

However, MTBF can be problematic for two reasons: (1) it projects a time span between failures but does not provide us with a projected failure rate, and (2) MTBF can be misinterpreted to mean average life span even though this is *not* what it implies.

An alternative measure of reliability is *failures in time* (FIT)—a statistical measure of how many failures a component will have over 1 billion hours of operation. Therefore, 1 FIT is equivalent to one failure in every billion hours of operation.

In addition to a reliability measure, you should also develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software. Of course, some aspects of availability have nothing to do with failure. For example, scheduling downtime (for support functions) causes the software to be unavailable. For a comprehensive discussion of software reliability measures, see [Laz11].

17.7.2 Use of AI to Model Reliability

Some software engineers view data science as the application of artificial intelligence techniques to solve software engineering problems. One of the things artificial intelligence methods attempt to do is provide reasonable solutions to problems where the needed data may be incomplete. *Software reliability* is defined as the probability of failure-free software operation for a specified time period in a specified environment. This means that we can never know the exact moment when a software product will fail because we will never have the complete data needed to calculate the probability.

Software engineers have used statistical techniques based on Bayes' theorem[5] in quantitative decision-making situations for several years. *Bayesian inference* is a method of statistical inference in which Bayes' theorem is used to update the probability

5 Bayes' theorem for conditional probabilities is P(A|B) = (P(B|A) * P(A)) / P(B). For more details, see http://www.statisticshowto.com/bayes-theorem-problems/.

for a hypothesis (such as system reliability) as more evidence or information becomes available. Bayesian inference can be used to estimate probabilistic quantities using historic data even when some information is missing. Using Bayesian techniques has allowed for real-time solutions to probability estimation problems that are beyond human reasoning [Tos17].

Proactive failure prediction using machine learning was discussed briefly in Section 15.4.3. It would be nice to be able to predict system failures in subsequent sprints before you deliver the prototype being developed in the current sprint. Making use of predictive data analytics such as a regression model involving MTBF has been used to estimate where and what types of defects might occur in future prototypes [Bat18].

A *genetic algorithm* is a heuristic search method used in artificial intelligence and computing. It is used for finding near-optimal solutions to search problems based on the theory of natural selection and evolutionary biology. Genetic algorithms can be used to grow reliability models by discovering relationships in historic system data. These models have been used to identify software components that may fail in the future. Sometimes these models have been created based on metrics estimated from UML models before any codes has been written [Pad17]. This type of work is very important to developers interested in refactoring software products or reusing software components in other products.

17.7.3 Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be: (1) causes uncontrolled acceleration that cannot be stopped, (2) does not respond to depression of brake pedal (by turning off), (3) does not engage when switch is activated, and (4) slowly loses or gains speed. Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.[6] To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If and only if a set of external environmental conditions is met, the improper position of the mechanical device will cause a disastrous failure. Analysis techniques [Eri15] such as fault tree analysis, real-time logic, and Petri net models can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

6 This approach is similar to the risk analysis methods described in Chapter 26. The primary difference is the emphasis on technology issues rather than project-related topics.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum but are evaluated in the context of an entire computer-based system and its environment.

A comprehensive discussion of software safety is beyond the scope of this book. If you have further interest in software safety and related system issues, see [Fir13], [Har12a], and [Lev12].

17.8 THE ISO 9000 QUALITY STANDARDS[7]

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management [ANS87]. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing, and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard.

The requirements delineated by ISO 9001:2015 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. For a software organization to become registered to ISO 9001:2015, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. If you desire further information on ISO 9001:2015, see [ISO14].

7 This section, written by Michael Stovsky, has been adapted from *Fundamentals of ISO 9000*, a workbook developed for *Essential Software Engineering,* a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

INFO



The ISO 9001:2015 Standard

The following outline defines the basic elements of the ISO 9001:2015 standard. Comprehensive information on the standard can be obtained from the International Organization for Standardization (**www.iso.ch**) and other Internet sources (e.g., **www.praxiom.com**).

Establish the elements of a quality management system.
- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system.
- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.
- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction.
- Define a quality plan that addresses objectives, responsibilities, and authority.
- Define communication mechanisms among stakeholders.

Establish review mechanisms for the quality management system.
- Identify review methods and feedback mechanisms.
- Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

Establish control mechanisms.
- For planning.
- For customer requirements.
- For technical activities (e.g., analysis, design, testing).
- For project monitoring and management.

Define methods for remediation.
- Assess quality data and metrics.
- Define approach for continuous process and quality improvement.

17.9 THE SQA PLAN

The *SQA plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

A standard for SQA plans has been published by the IEEE [IEE17]. The standard recommends a structure that identifies: (1) the purpose and scope of the plan, (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA, (3) all applicable standards and practices that are applied during the software process, (4) SQA actions and tasks (including reviews and audits) and their placement throughout the software process, (5) the tools and methods that support SQA actions and tasks, (6) software configuration management (Chapter 22) procedures, (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and (8) organizational roles and responsibilities relative to product quality.

17.10 SUMMARY

Software quality assurance is a software engineering umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, oversight of quality control activities such as technical reviews and software testing, procedures for change management, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

In summary, you should note the words of Dunn and Ullman [Dun82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

PROBLEMS AND POINTS TO PONDER

17.1. Some people say that "variation control is the heart of quality control." Because every program that is created is different from every other program, what are the variations that we look for and how do we control them?

17.2. Is it possible to assess the quality of software if the customer keeps changing what it is supposed to do?

17.3. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss the differences.

17.4. Can a program be correct and still not be reliable? Explain.

17.5. Can a program be correct and still not exhibit good quality? Explain.

17.6. Why is there often tension between a software engineering group and an independent software quality assurance group? Is this healthy?

17.7. You have been given the responsibility for improving the quality of software across your organization. What is the first thing that you should do? What's next?

17.8. Besides counting errors and defects, are there other countable characteristics of software that imply quality? What are they and can they be measured directly?

17.9. The MTBF concept for reliability is open to criticism. Explain why.

17.10. Consider two safety-critical systems that are controlled by computer. List at least three hazards for each that can be directly linked to software failures.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

18 Software Security Engineering

Contributed by: Nancy Mead *Carnegie Mellon University Software Engineering Institute*

Stop and take a look around. Where do you see software being deployed? Sure, it's in your laptop, tablet, and cell phone. What about your appliances—refrigerator, dishwasher, and so forth? How about your car? Financial transactions—ATM, online banking, financial software, tax software? Your supplier of electricity? Definitely using software. Do you have any wearable devices on? A fit-bit? Maybe you have medical devices, like a pacemaker. The bottom line is: Software is all around us, on us, and sometimes in us. Every software product has the potential to be hacked, sometimes with dire consequences. This is the reason that we, as software engineers, need to be concerned about software security.

Key Concepts

attack patterns 363
attack surface 366
maturity models 370
measurement 368
misuse and abuse cases.....363
requirements engineering 360
secure coding 367
secure development life-cycle activities 359
security life-cycle models.....357
security process improvement.....370
security risk analysis.....364
software security engineering, importance of 357
threat modeling, prioritization, and mitigation.....365

Quick Look

What is it? Software security engineering encompasses a set of techniques that can improve the security of software *while* it is under development.

Who does it? Although software engineers do not need to become security experts, they need to collaborate with security experts. The security experts may be members of the software team, on a separate specialized team, or they may be outside consultants.

Why is it important? The media continually reports instances of hacking—whether by gangsters, corporate competitors, hostile nations, or any other bad actor. The consequences for critical infrastructure, financial institutions, health care, and all aspects of modern life are significant.

What are the steps? There are a number of steps that can be taken to ensure that software is secure. We will discuss some of them here and provide pointers to resources for further exploration.

What is the work product? As you will see, there are many work products that are developed in the process of secure software engineering. The ultimate work product, or course, is the software that you have developed using secure software engineering practices.

How do I ensure that I've done it right? Everything that we will discuss as a method to improve software security, whether at the organizational or project level, can and should be reviewed by the interested parties. In addition, secure development processes can be improved, if they are found to be lacking.

18.1 Why Software Security Engineering Is Important

Software security is about more than just securing operational software with firewalls, strong passwords, and encryption. It's also about developing software in such a way that it is more secure from the outset. Techniques are now available that will help us develop software that is significantly more secure than it would be otherwise.

In this chapter, we're going to look as some of the models and techniques that can help us achieve better software security. We'll start by looking at security process models. Then we'll examine specific process activities, including such activities as requirements engineering, misuse or abuse cases, security risk analysis, threat modeling, attack surface, secure coding, and measurement. We'll also consider security process improvement models. Finally, we'll summarize and provide you with a list of references so that you can dive into any of these topics in more depth.

Software security engineering research is a very active area. In this book we're only providing an overview of methods and tools to support actual practice. There are many books (for example, [Mea16], [Shu13], and [Hel18]) and other resources devoted exclusively to software security engineering, and we will point you to some of them.

18.2 Security Life-Cycle Models

The Microsoft Security Development Lifecycle (SDL) [Mea16] [Mic18] is an industry-leading software security process. A Microsoft-wide initiative and a mandatory policy since 2004, the SDL enabled Microsoft to embed security and privacy in its software and culture. The SDL introduces security and privacy early and throughout all phases of the development process and is without question the most widely known and used security development life-cycle model.

Microsoft defined a collection of principles it calls *Secure by Design, Secure by Default, Secure in Deployment, and Communications* (SD3+C) to help determine where security efforts are needed. These are as follows [Mic10]:

Secure by Design

Secure architecture, design, and structure. Developers consider security issues part of the basic architectural design of software development. They review detailed designs for possible security issues, and they design and develop mitigations for all threats.

Threat modeling and mitigation. Threat models are created, and threat mitigations are present in all design and functional specifications.

Elimination of vulnerabilities. No known security vulnerabilities that would present a significant risk to the anticipated use of the software remain in the code after review. This review includes the use of analysis and testing tools to eliminate classes of vulnerabilities.

Improvements in security. Less secure legacy protocols and code are deprecated, and, where possible, users are provided with secure alternatives that are consistent with industry standards.

Secure by Default

Least privilege. All components run with the fewest possible permissions.

Defense in depth. Components do not rely on a single threat mitigation solution that leaves users exposed if it fails.

Conservative default settings. The development team is aware of the attack surface for the product and minimizes it in the default configuration.

Avoidance of risky default changes. Applications do not make any default changes to the operating system or security settings that reduce security for the host computer. In some cases, such as for security products, it is acceptable for a software program to strengthen (increase) security settings for the host computer. The most common violations of this principle are games that either open firewall ports without informing the user or instruct users to open firewall ports without informing users of possible risks.

Less commonly used services off by default. If fewer than 80 percent of a program's users use a feature, that feature should not be activated by default. Measuring 80 percent usage in a product is often difficult because programs are designed for many different personas. It can be useful to consider whether a feature addresses a core/primary use scenario for all personas. If it does, the feature is sometimes referred to as a P1 feature.

Secure in Deployment

Deployment guides. Prescriptive deployment guides outline how to deploy each feature of a program securely, including providing users with information that enables them to assess the security risk of activating non-default options (and thereby increasing the attack surface).

Analysis and management tools. Security analysis and management tools enable administrators to determine and configure the optimal security level for a software release.

Patch deployment tools. Deployment tools aid in patch deployment.

Communications

Security response. Development teams respond promptly to reports of security vulnerabilities and communicate information about security updates.

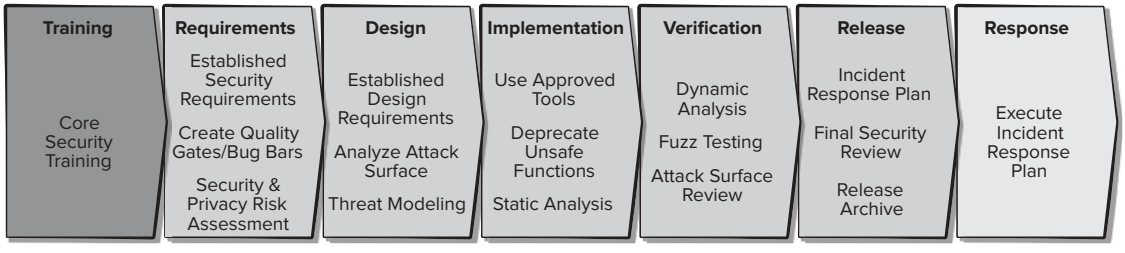
Community engagement. Development teams proactively engage with users to answer questions about security vulnerabilities, security updates, or changes in the security landscape.

The secure software development process model looks like the one shown in Figure 18.1.

The Microsoft SDL documentation describes what architects, designers, developers, and testers are required to do for each of the 16 recommended practices. The data that Microsoft collected after implementing the SDL shows a significant reduction in

FIGURE 18.1 **Secure Software Development Process Model at Microsoft**

Adapted from Shunn, A., et al. Strengths in Security Solutions, Software Engineering Institute, Carnegie Mellon University, 2013. Available at http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=77878.



vulnerabilities, which led to a need for fewer patches, and thus a significant cost savings. We recommend that you browse the SDL website to learn more about these practices. Since the SDL was developed, there have been numerous papers, books, training, and so on, to go with the SDL model.[1]

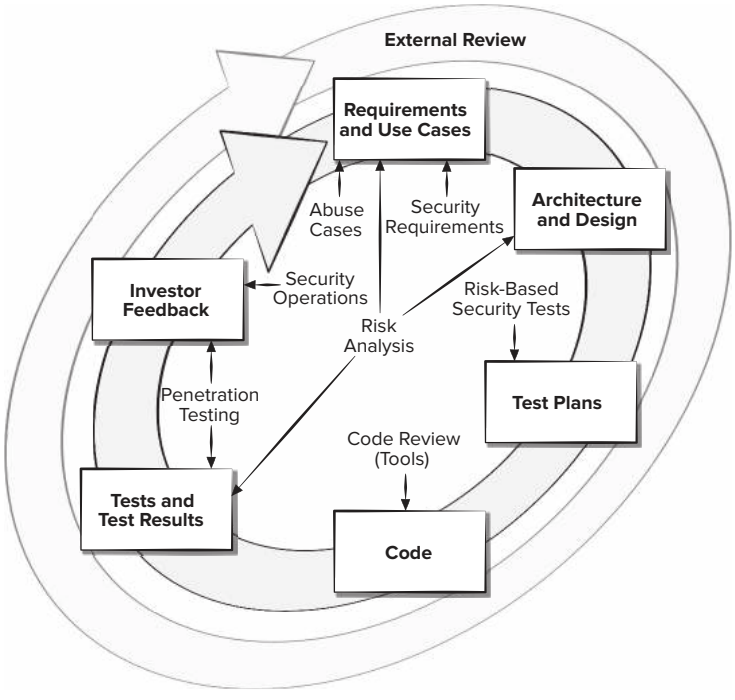
18.3 Secure Development Life-Cycle Activities

A different approach that is independent of a life-cycle model is the touchpoints for software security [McG06], which argues that the *activities* (touchpoints) are what matter, not the model. The activities can be incorporated into any life-cycle model, and thus are considered to be process agnostic. The touchpoints later formed the basis for BSIMM, a maturity model that will be discussed later in this chapter. Some organizations consider the touchpoints to be the minimum set of activities that should be performed in secure software development. A pictorial version of the touchpoints is shown in Figure 18.2. In this diagram, the recommended security activities appear above the corresponding software development activity, or life-cycle phase:

With the SDL and touchpoints in mind, we'll look at some of the important secure software development activities that are associated with them.

FIGURE 18.2

Software security touchpoints



1 More recently, Microsoft has shown how the SDL activities can be integrated with an agile development approach: https://www.microsoft.com/en-us/SDL/Discover/sdlagile.aspx.

18.4 SECURITY REQUIREMENTS ENGINEERING

Although security requirements are an important part of secure software development, in practice they are often neglected. When they exist, they are often an add-on, copied from a generic list of security features. The requirements engineering that is needed to get a better set of security requirements seldom takes place [All08].

Requirements engineering practice typically addresses desired user features. Therefore, attention is given to the functionality of the system from the user's perspective, but little attention is given to what the system should *not* do [Bis02]. Users expect systems to be secure, and these assumptions need to make their way into security requirements for software systems before they are developed, not after the fact. Often the users' assumptions about security are overlooked because system features are the primary focus.

In addition to security life-cycle models, there are many process models that are specific to security requirements. These include: core security requirements artifacts [Mof04], Software Cost Reduction (SCR) [Hei02], SQUARE (Security QUAlity Requirements Engineering) [Mea05], and Security Requirements Engineering Process (SREP) [Mel06]. For the remainder of this section, we'll consider SQUARE as a representative example of security life-cycle models.

18.4.1 SQUARE

SQUARE is a representative security requirements engineering process model, but it's important to keep in mind that if you already have a development process model, such as the one presented in Chapter 4, you can just pick up some of the SQUARE steps to enhance your existing model. There's no need to develop a whole new process to address security in your software development activities. We suggest that you add security definitions to your glossary; perform risk analysis, including identification of potential attacks via misuse cases or threat modeling; develop mitigation strategies; and perform categorization and prioritization of candidate security requirements.

The SQUARE process model provides for eliciting, categorizing, and prioritizing security requirements for software-intensive systems. Its focus is to build security concepts into the early stages of the development life cycle. It can also be used for fielded systems and those undergoing improvements and modifications. The process is shown in Table 18.1, followed by brief descriptions of each step.

18.4.2 The SQUARE Process

The SQUARE process is best applied by the project's requirements engineers and security experts with supportive executive management and stakeholders. Let's take a look at the steps.[2]

> **Step 1. Agree on definitions.** So that there is not semantic confusion, this step is needed as a prerequisite to security requirements engineering. On a

2 To dig deeper, see the resources available at the SEPA 9/e website.

TABLE 18.1 The SQUARE process

Number	Step	Input	Techniques	Participants	Output
1	Agree on definitions.	Candidate definitions from IEEE and other standards	Structured interviews, focus group	Stakeholders, requirements team	Agreed-to definitions
2	Identify assets and security goals.	Definitions, candidate assets and goals, business drivers, policies and procedures, examples	Facilitated work session, surveys, interviews	Stakeholders, requirements engineer	Assets and security goals
3	Develop artifacts to support security requirements definition.	Potential artifacts (e.g., scenarios, misuse cases, templates, forms)	Work session	Requirements engineer	Needed artifacts: scenarios, misuse cases, models, templates, forms
4	Perform (security) risk assessment.	Misuse cases, scenarios, security goals	Risk assessment method, analysis of anticipated risk against organizational risk tolerance, including threat analysis	Requirements engineer, risk expert, stakeholders	Risk assessment results
5	Select elicitation techniques.	Goals, definitions, candidate techniques, expertise of stakeholders, organizational style, culture, level of security needed, cost-benefit analysis, etc.	Work session	Requirements engineer	Selected elicitation techniques
6	Elicit security requirements.	Artifacts, risk assessment results, selected techniques	Accelerated Requirements Method, Joint Application Development, interviews, surveys, model-based analysis, checklists, lists of reusable requirements types, document reviews	Stakeholders facilitated by requirements engineer	Initial cut at security requirements
7	Categorize requirements as to level (system, software, etc.) and whether they are requirements or other kinds of constraints.	Initial requirements, architecture	Work session using a standard set of categories	Requirements engineer, other specialists as needed	Categorized requirements
8	Prioritize requirements.	Categorized requirements and risk assessment results	Prioritization methods such as Analytical Hierarchy Process (AHP), Triage, Win-Win, etc.	Stakeholders facilitated by requirements engineer	Prioritized requirements
9	Inspect requirements.	Prioritized requirements, candidate formal inspection technique	Inspection method such as Fagan and peer reviews	Inspection team	Initial selected requirements, documentation of decision-making process and rationale

given project, team members tend to have definitions in mind, based on their prior experience, but those definitions are often different from one another [Woo05]. Sources such as the Institute for Electrical and Electronics Engineers (IEEE) and the Software Engineering Body of Knowledge (SWEBOK) provide a range of definitions to select from or tailor [SWE14].

Step 2. Identify assets and security goals. This step occurs at the project's organizational level and is needed to support software development. Different stakeholders usually have concerns about different assets, and thus have different goals. For example, a stakeholder in human resources may be concerned about maintaining the confidentiality of personnel records, whereas a stakeholder in a research area may be concerned with ensuring that research project information is not accessed, modified, or stolen.

Step 3. Develop artifacts. This step is necessary to support all subsequent security requirements engineering activities. Often, organizations do not have key documents needed to support requirements definition, or they may not be up to date. This means that a lot of time may be spent backtracking to try to obtain documents, or the team will have to bring them up to date before going further.

Step 4. Perform risk assessment. This step requires an expert in risk assessment methods, the support of the stakeholders, and the support of a security requirements engineer. There are a number of risk assessment methods, but regardless of the one that you choose, the outcomes of risk assessment can help in identifying the high-priority security exposures.

Step 5. Select elicitation technique. This step becomes important when there are diverse stakeholders. A more formal elicitation technique, such as the Accelerated Requirements Method [Hub99], Joint Application Design [Woo89], or structured interviews, can be effective in overcoming communication issues when there are stakeholders with different cultural backgrounds. In other cases, elicitation may simply consist of sitting down with a primary stakeholder to try to understand that stakeholder's security requirements needs.

Step 6. Elicit security requirements. This step encompasses the actual elicitation process using the selected technique. Most elicitation techniques provide detailed guidance on how to perform elicitation. This builds on the artifacts that were developed in earlier steps.

Step 7. Categorize requirements. This step allows the security requirements engineer to distinguish among essential requirements, goals (desired requirements), and architectural constraints that may be present. This categorization also helps in the prioritization activity that follows.

Step 8. Prioritize requirements. This step depends on the prior step and may also involve performing a cost-benefit analysis to determine which security requirements have a high payoff relative to their cost. Of course prioritization may also depend on other consequences of security breaches, such as loss of life, loss of reputation, and loss of consumer confidence.

Step 9. Requirements inspection. This review activity can be accomplished at varying levels of formality, discussed in Chapter 16. Once inspection is complete, the project team should have an initial set of prioritized security requirements that can be revisited as needed later in the project.

18.5 Misuse and Abuse Cases and Attack Patterns

Misuse (or *abuse*) cases can help you view your software in the same way that attackers do. By thinking about negative events, you can better understand how to develop secure software. A misuse case can be thought of as a use case that the attacker initiates.

One of the goals of misuse cases [Sin00] is to decide up front how software should react to potential attacks. You can also use misuse and normal use cases together to conduct threat and hazard analysis [Ale03].

We suggest creating misuse cases through brainstorming. Teaming security experts with subject matter experts (SMEs) covers a lot of ground quickly. During brainstorming, software security experts ask many questions of developers to help identify the places where the system is likely to have weaknesses. This involves a careful look at all user interfaces and considers events that developers assume can't happen, but that attackers can actually cause to happen.

Here are some questions that need to be considered: How can the system distinguish between valid and invalid input data? Can it tell whether a request is coming from a legitimate application or a rogue application? Can an insider cause a system to malfunction? Trying to answer such questions helps developers to analyze their assumptions and allows them to fix problems up front.

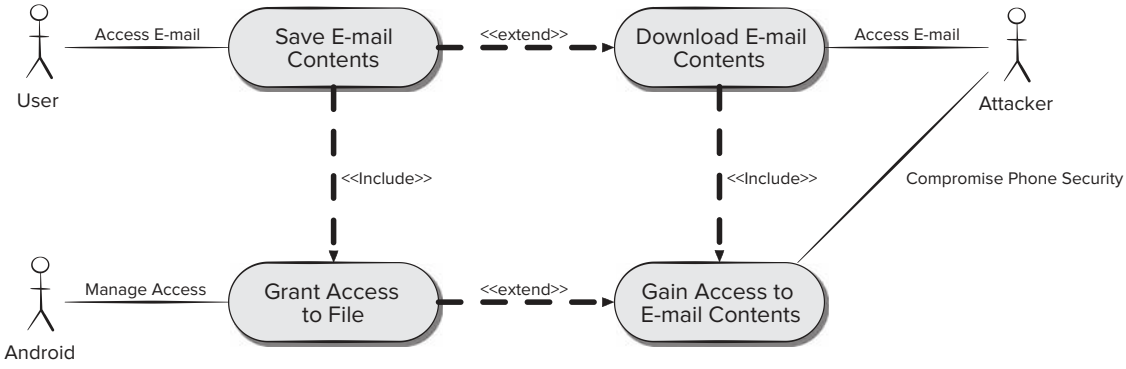
Misuse cases can be in table or diagram form. Figure 18.3 provides an example misuse case that shows how DroidCleaner malware can successfully attack a cell phone using an open-source e-mail application called K-9. This is extracted from a much larger report that you may wish to study [Ali14].

In this misuse case, the user keeps e-mail on the phone's external storage area. The attacker gains access to the phone's storage by compromising the operating system. A common way for the attacker to gain access to the phone is by tricking the user into installing a Trojan, to which the user unwittingly grants access to the drive during the install process. The attacker is then able to use the Trojan to download files, including the e-mail contents file.

Attack patterns can provide some help by providing a blueprint for creating an attack. For example, buffer overflow is one type of security exploitation. Attackers trying to capitalize on a buffer overflow make use of similar steps [OWA16]. Attack patterns can document these steps (e.g., timing, resources, techniques) as well as practices software developers can use to prevent or mitigate their success [Hog04]. When you're trying to develop misuse and abuse cases, attack patterns can help.

Misuse cases need to be prioritized as they are generated. In addition, they need to strike the right balance between cost and benefit. The project budget may not allow

FIGURE 18.3 **Misuse case (exploited by DroidCleaner): Data in an e-mail stored on the smartphone is stolen**



a software team to implement all defined mitigation strategies at once. In such cases, the strategies can be prioritized and implemented incrementally. The team can also exclude certain cases as being extremely unlikely.

Templates for misuse and abuse cases appear in a number of references. They can be text or diagrams and may be supported by tools. Good sources for templates are in materials by Sindre and Opdahl [Sin01] and Alexander [Ale02].

18.6 SECURITY RISK ANALYSIS

A wide variety of security risk assessment methods have been proposed. Typical examples include SEI CERT's Security Engineering Risk Analysis (SERA) method[3] and the NIST Risk Management Framework (RMF).[4]

RMF has emerged as an approach that is widely used, providing guidelines for the users. The RMF steps for security are:

- **Categorize** the information system and the information processed, stored, and transmitted by that system based on an impact analysis.
- **Select** an initial set of baseline security controls for the information system based on the security categorization; using an organizational assessment of risk and local conditions, tailor and supplement the security control baseline as needed.
- **Implement** the security controls, and describe how the controls are employed within the information system and its operational environment.
- **Assess** the security controls using appropriate assessment procedures to determine the extent to which the controls are implemented correctly, operating as intended, and producing the desired outcome with respect to meeting the security requirements for the system.

3 See https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=485410.
4 See https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final.

- **Authorize** the information system operation based on a determination of the risk to organizational operations and assets, individuals, or other organizations (including national defense), from the operation of the information system that this risk is acceptable.
- **Monitor** the security controls in the information system on an ongoing basis including assessing control effectiveness, documenting changes to the system or its environment of operation, conducting security impact analyses of the associated changes, and reporting the security state of the system to designated organizational officials.

It's important to note that NIST also provides a set of security controls to select from, thus simplifying the risk assessment work. Recently, the RMF has been modified to include privacy concerns.

18.7 Threat Modeling, Prioritization, and Mitigation

A threat modeling method (TMM) is an approach for creating an abstraction of a software system, aimed at identifying attackers' abilities and goals, and using that abstraction to generate and catalog possible threats that the system must mitigate [Shu16].

STRIDE (an acronym for six threat categories) is representative of a number of threat modeling methods [Mea18] and is the most well-established TMM, representing the state of the practice. At its core, STRIDE requires breaking down a system into its various elements, assessing each of these elements for their vulnerability to threats, and then mitigating these threats [Her06]. In practice, a typical STRIDE implementation includes modeling a system with data flow diagrams (DFDs),[5] mapping the DFD elements to the six threat categories, determining the specific threats via checklists or threat trees, and documenting the threats and steps for their prevention [Sca15]. STRIDE can be implemented manually; however, a free Microsoft Secure Development Lifecycle (SDL) Threat Modeling Tool [Mic17] can also be used. Table 18.2 identifies the security property associated with each of the six threat categories.

TABLE 18.2

Threat categories and security properties

Threat	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Nonrepudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

5 A brief tutorial on data flow diagrams can be downloaded from https://ratandon.mysite.syr.edu/cis453/notes/DFD_over_Flowcharts.pdf.

TABLE 18.3
Threat categories of DFD system elements

Element	Spoofing	Tampering	Repudiation	Information Disclosure	Denial of Service	Elevation of Privilege
Data flows		X		X	X	
Data stores		X		X	X	
Processes	X	X	X	X	X	X
External entity	X		X			

DFDs are designed to show how a system works by using standard symbols to graphically represent the interaction between data stores (e.g., databases, files, registries), processes (e.g., DLLs, Web services), data flows (e.g., function calls, remote procedure calls), and external entities (e.g., people, other systems) [Sho14]. Once complete, each of these system elements in turn can be associated with one or more relevant threat categories, as depicted in Table 18.3.

In the next stage, the typical STRIDE user works through a checklist (that may be in the form of a threat tree) of specific threats that are associated with each match between a DFD element and threat category. Such checklists are accessible through STRIDE reference books or tools.

Once the threats have been identified, mitigation strategies can be developed and prioritized. Typically, prioritization is based on cost and value considerations. Considering the cost of implementing the mitigation strategy is important, but it's equally important to also consider the cost of *not* implementing it, which is reflected in value. Remember that risks that are realized result in costs that are not only expressed in terms of dollars, but could also reflect loss of reputation, loss of trust, and even loss of life.

18.8 ATTACK SURFACE

An *attack surface* can be defined[6] in the following manner:

> The attack surface describes all of the different points where an attacker could get into a system, and where they could get data out.
>
> The attack surface of an application is:
>
> 1. the sum of all paths for data/commands into and out of the application, and
> 2. the code that protects these paths (including resource connection and authentication, authorization, activity logging, data validation and encoding), and
> 3. all valuable data used in the application, including secrets and keys, intellectual property, critical business data, personal data and PII, and
> 4. the code that protects these data (including encryption and checksums, access auditing, and data integrity and operational security controls). [OWA18]

6 See https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.md.

The OWASP Foundation [OWA18] states that attack surface analysis is:

> . . . targeted to be used by developers to understand and manage application security risks as they design and change an application, as well as by application security specialists doing a security risk assessment. The focus here is on protecting an application from external attack—it does not take into account attacks on the users or operators of the system (e.g., malware injection, social engineering attacks), and there is less focus on insider threats, although the principles remain the same. The internal attack surface is likely to be different to the external attack surface and some users may have a lot of access.
>
> Attack Surface Analysis is about mapping out what parts of a system need to be reviewed and tested for security vulnerabilities. The point of attack surface analysis is to understand the risk areas in an application, to make developers and security specialists aware of what parts of the application are open to attack, to find ways of minimizing this, and to notice when and how the Attack Surface changes and what this means from a risk perspective.

18.9 SECURE CODING

Secure coding is just what the name implies—coding in such a way that vulnerabilities are not introduced as a result of coding errors. It's not surprising that most software vulnerabilities occur because of sloppy and erroneous coding practices, many of which can be easily avoided.

For example, a condition known as *buffer overflow* results from one of the most well-known and common coding errors. OWASP[7] describes it as follows:

> A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.

Buffer overflow is just one example of coding errors that can result in vulnerabilities. Fortunately, a number of coding standards now exist to provide guidance on secure coding. The SEI/CERT website[8] provides a list of the top-10 secure coding practices:

1. **Validate input.** Validate input from all untrusted data sources.
2. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
3. **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies.
4. **Keep it simple.** Keep the design as simple and as small as possible.
5. **Default deny.** Base access decisions on permission rather than exclusion.

7 See https://www.owasp.org/index.php/Buffer_overflow_attack.

8 See https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices.

6. **Adhere to the principle of least privilege.** Every process should execute with the least set of privileges necessary to complete the job.
7. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components.
8. **Practice defense in depth.** Manage risk with multiple defensive strategies.
9. **Use effective quality assurance techniques.**
10. **Adopt a secure coding standard.**

SEI CERT and others also provide secure coding standards.[9] In addition to using a secure coding standard, you should inspect for coding errors that lead to vulnerabilities. This is just a natural add-on to your normal code inspection and review process (Chapter 16). *Static analysis tools*[10] are used to automatically analyze code and are another mechanism for detecting vulnerabilities due to coding errors.

18.10 MEASUREMENT

Developing adequate measures of software security is a difficult problem, and one for which there are differing viewpoints. On the one hand, you can look at the development processes followed and assess whether the resultant software is likely to be secure. On the other hand, you can look at the incidence of vulnerabilities and successful break-ins and measure those as a way of assessing software security. However, neither of these measurement approaches will allow you to say with 100 percent certainty that our software is secure. When you add supporting software such as operating systems and external interoperable systems, the measurement of software security becomes even more difficult. Nevertheless some progress has been made.

Measures of software quality can go a long way toward measuring software security. Specifically, vulnerabilities invariably point to software defects. Although not all software defects are security problems, vulnerabilities in software generally result from a defect of some kind, whether it is in the requirements, architecture, or code. Therefore, measures such as defect and vulnerability counts [Woo14] are useful. Microsoft uses measures such as attack surface analysis and tries to keep the attack surface (places where software can be compromised) to a minimum.

Just as use of maturity models such as CMMI (Chapter 28) suggests that higher-quality software will result, mature security development processes, such as those emphasized by BSIMM,[11] will result in more secure software. In some cases, organizations are encouraged to identify the unique set of security metrics that are relevant to them. BSIMM makes reference to this, as does SAMM.[12]

9 See https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards.

10 A list of commercially available tools can be found at https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

11 See https://www.bsimm.com/.

12 See https://www.owasp.org/index.php/OWASP_SAMM_Project.

It's important to note that none of the measurement characteristics implied by various maturity models is perfect. If you follow good secure software development processes, does that guarantee that the software is secure? No! If you find a lot of vulnerabilities, does that mean that most of them have been found or that there are still more to be found because this is a particularly poor piece of software? There are no simple answers. However, to assess vulnerabilities and associated software security, we have to collect data so that patterns can be analyzed over time. If we don't collect data about software security, we will never be able to measure its improvement.

Tables 18.4 and 18.5 provide examples of how to assess software security during each life-cycle phase. The full tables and discussion can be found in [Mea17] and [Alb10].

TABLE 18.4

Examples of life-cycle-phase measures

Life-Cycle Phase	Example Software Security Measures
Requirements engineering	Percentage of relevant software security principles reflected in requirements-specific actions (assuming security principles essential for a given development project have been selected)
	Percentage of security requirements that have been subject to analysis (risk, feasibility, cost-benefit, performance trade-offs) prior to being included in the specification
	Percentage of security requirements covered by attack patterns, misuse and abuse cases, and other specified means of threat modeling and analysis
Architecture and design	Percentage of architectural and design components subject to attack surface analysis and measurement
	Percentage of architectural and design components subject to architectural risk analysis
	Percentage of high-value security controls covered by a security design pattern

TABLE 18.5

Example measures based on the seven principles of evidence

Principle	Description
Risk	Number of active and latent threats, categorized
	Incidents reported by category of threat
	Likelihood of occurrence for each threat category
	Financial and/or human safety estimate of impact for each threat category
Trusted dependencies	Number of levels of subcontracting in the supply chain (in other words, have the subcontractors, in turn, executed subcontracts, and what is the depth of this activity?)
	Number of suppliers by level
	Hierarchical and peer dependencies between suppliers by level
	Number of (vetted) trusted suppliers in the supply chain by level

18.11 Security Process Improvement and Maturity Models

A number of process improvement and maturity models are available for software development in general, such as the Capability Maturity Model Integration (CMMI).[13] For cyber security maturity, the CMMI Institute offers a newer product, the Cyber Capability Maturity Management platform.[14] OWASP offers the Software Assurance Maturity Model (SAMM).[15] SAMM is an open framework to help organizations formulate and implement a strategy for software security that is tailored to the specific risks facing the organization.

A comprehensive discussion of these models is beyond the scope of this book. To provide a simple overview, consider the overall objective of SAMM:

- Evaluate an organization's existing software security practices.
- Build a balanced software security assurance program in well-defined iterations.
- Demonstrate concrete improvements to a security assurance program.
- Define and measure security-related activities throughout an organization.

Perhaps the most well-known maturity model that is specifically for software security is the Building Security in Maturity Model (BSIMM). BSIMM has periodic releases, typically every year or two. The BSIMM model and its recent summarized assessment results can be downloaded from the BSIMM website.[16] According to the BSIMM developers, BSIMM is meant to be used by those who create and execute a software security initiative.

All the maturity models mentioned here (and others) have benefits, and the essential elements of the models are freely available. However, sometimes the assessment is done by external entities. It's possible to do an internal self-assessment and define the associated improvement program, but it requires dedicated resources and effort. Alternatively, some of these organizations offer assessment programs, thus providing an external view of the strengths and areas for improvement within a software organization.

18.12 Summary

All software engineers should have an understanding of what it takes to develop secure software. The steps that are needed to improve the security of your software products are relevant in each activity in the software process, regardless of which process model is used.

Although there are still many open questions, and technologies that need additional research, there are many resources available today to assist with this challenge. For every activity that normally takes place in the software process, try to incorporate

13 See https://cmmiinstitute.com/.
14 See https://cmmiinstitute.com/products/cybermaturity.
15 See https://www.owasp.org/index.php/OWASP_SAMM_Project.
16 See https://www.bsimm.com/.

security aspects. Models such as Microsoft SDL and the SQUARE model can be assessed to determine which steps you can incorporate into your development process.

Add security to a risk analysis activity, especially using detailed guidance available from NIST. Given the number of secure coding standards already available, it is certainly possible for anyone to learn how to code securely. Inspect your code for remaining vulnerabilities. Learn how to identify security holes and develop and prioritize mitigation strategies. Perform static analysis testing on your code. Visit the OWASP and BSIMM websites, among others, to learn about maturity in software security engineering.

As software becomes ever more ubiquitous, the number of vulnerabilities and successful hacks grow as well. It will take all our efforts to stem the tide, but many of the tools already exist to tackle this problem. The consequences of failing to address software security are high, and the benefits of developing secure software are huge.

PROBLEMS AND POINTS TO PONDER

18.1. What is the most important thing that a software team can do to improve software security?

18.2. If you were recommending one activity for your organization to improve software security, what would it be? If you were recommending multiple activities, what are they, and what would be the priorities, considering that it's not likely that all of them will be implemented at once?

18.3. How could you incorporate software security into your existing process model or into a new process model?

18.4. Sit down with a colleague and identify security risks on a software project that is in development. Come up with mitigation strategies and prioritize them.

18.5. Are you collecting measurement data that could be used or repurposed to help measure software security? If not, is there data that could easily be collected for this purpose?

18.6. Use the Internet to find out the details needed to create a phishing attack pattern.

18.7. Explain some of the problems that might be encountered if you wait until after the system is completed to address security risks.

18.8. Use the Internet to determine the average cost to the consumer of a single incidence of identity theft.

18.9. Consider a MobileApp that you make use of on your personal phone. List three to five security risks that developers should consider when developing apps like this one.

18.10. Determine the security requirements of a bill-paying wallet-type MobileApp.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

19 SOFTWARE TESTING—COMPONENT LEVEL

Software component testing incorporates a strategy that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Within the testing strategy, software component testing implements a collection of component testing tactics that address test planning, test-case design, test execution, and resultant data collection and evaluation. Both component testing strategy and tactics are considered in this chapter.

KEY CONCEPTS

basis path testing 384
black-box testing 388
boundary value analysis 389
class testing 390
control structure testing 386
cyclomatic complexity 385
debugging 373
equivalence partitioning 389
independent test group 375
integration testing 375
interface testing 388
object-oriented testing 390
scaffolding 379
system testing 376
testing methods 373
testing strategies 373
unit testing 376
validation testing 376
verification 373
white-box testing 383

QUICK LOOK

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed. A software component testing strategy considers testing of individual components and integrating them into a working system.

Who does it? A software component testing strategy is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering action. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected.

What are the steps? Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component or on a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested, they must be integrated until the complete system is constructed.

What is the work product? A *test specification* documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the types of test cases that will be conducted.

How do I ensure that I've done it right? An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

To be effective, a component testing strategy should be flexible enough to promote a customized testing approach but rigid enough to encourage reasonable planning and management tracking as the project progresses. Component testing remains a responsibility of individual software engineers. Who does the testing, how engineers communicate their results with one another, and when testing is done is determined by the software integration approach and design philosophy adopted by the development team.

These "approaches and philosophies" are what we call *strategy and tactics*—topics to be discussed in this chapter. In Chapter 20 we discuss the integration testing techniques that often end up defining the team development strategy.

19.1 A Strategic Approach to Software Testing

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason, a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

A number of software testing strategies have been proposed in the literature [Jan16] [Dak14] [Gut15]. All provide you with a template for testing, and all have the following generic characteristics:

- To perform effective testing, you should conduct technical reviews (Chapter 16). By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing incorporates a set of tactics that accommodate the low-level tests necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

19.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of tasks that ensure that software correctly implements a specific function. *Validation* refers to a different set

of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

> Verification: "Are we building the product right?"
> Validation: "Are we building the right product?"

The definition of V&V encompasses many software quality assurance activities (Chapter 19).[1]

Verification and validation include a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering, and testing cannot be applied as a fix at the end of the process. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

19.1.2 Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error-free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigates against thorough testing.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyzes, models, and then creates a computer program and its documentation. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) destructive. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than to uncover errors. Unfortunately, errors will be nevertheless present. And, if the software engineer doesn't find them, the customer will!

---

1 It should be noted that there is a strong divergence of opinion about what types of testing constitute "validation." Some people believe that *all* testing is verification and that validation is conducted when requirements are reviewed and approved, and later, by the user when the system is operational. Other people view unit and integration testing (Chapters 19 and 20) as verification and higher-order testing (Chapter 21) as validation.

There are often a number of misconceptions that you might infer from the preceding discussion: (1) that the developer of software should do no testing at all, (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly, and (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts *integration testing*—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

The role of an *independent test group* (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

However, you don't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

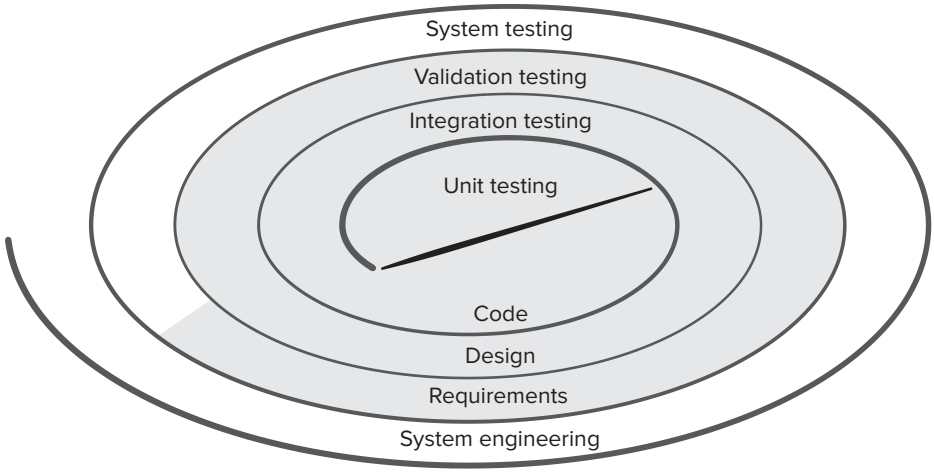
The ITG is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering team.

19.1.3 The Big Picture

The software process may be viewed as the spiral illustrated in Figure 19.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance,

FIGURE 19.1

Testing strategy

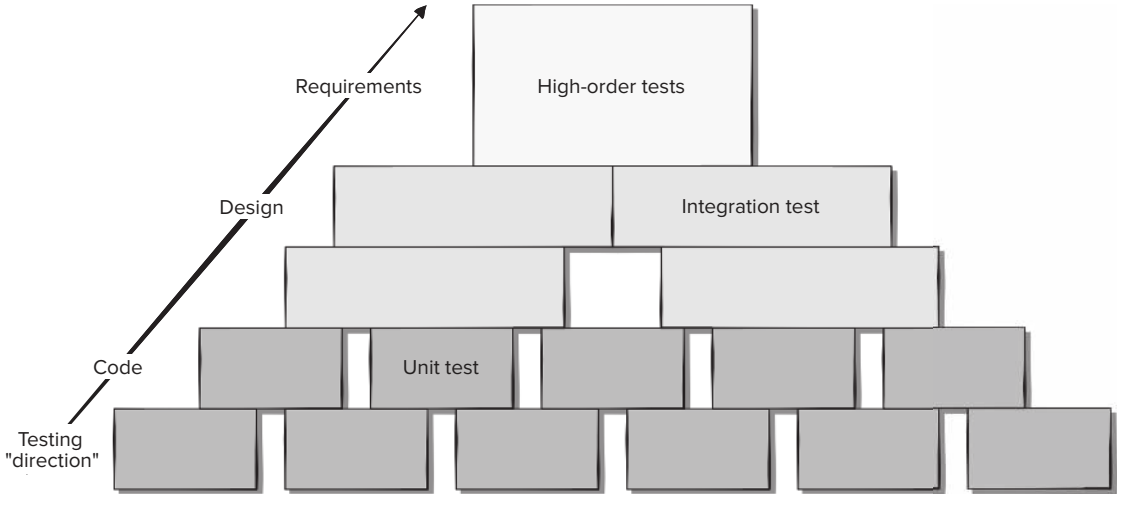


constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral (Figure 19.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter *validation testing,* where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 19.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing.* Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Test-case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.

FIGURE 19.2 **Software testing steps**



After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering (discussed in Chapter 21). Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function and performance is achieved.

SAFEHOME



Preparing for Testing

The scene: Doug Miller's office, as component-level design continues and construction of certain components begins.

The players: Doug Miller, software engineering manager; Vinod, Jamie, Ed, and Shakira, members of the *SafeHome* software engineering team.

The conversation:

Doug: It seems to me that we haven't spent enough time talking about testing.

Vinod: True, but we've all been just a little busy. And besides, we have been thinking about it . . . in fact, more than thinking.

Doug (smiling): I know . . . we're all overloaded, but we've still got to think down the line.

Shakira: I like the idea of designing unit tests before I begin coding any of my components, so that's what I've been trying to do. I have a pretty big file of tests to run once code for my components is complete.

Doug: That's an Extreme Programming [an agile software development process, see Chapter 3] concept, no?

Ed: It is. Even though we're not using Extreme Programming per se, we decided that it'd be a good idea to design unit tests before we build the component—the design gives us all of the information we need.

Jamie: I've been doing the same thing.

Vinod: And I've taken on the role of the integrator, so every time one of the guys passes a component to me, I'll integrate it and run a series of regression tests [see Section 20.3 for a discussion on regression testing] on the partially integrated program. I've been working to design a set of appropriate tests for each function in the system.

Doug (to Vinod): How often will you run the tests?

Vinod: Every day . . . until the system is integrated . . . well, I mean until the software increment we plan to deliver is integrated.

Doug: You guys are way ahead of me!

Vinod (laughing): Anticipation is everything in the software biz, Boss.

19.1.4 Criteria for "Done"

A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The *statistical quality assurance* approach (Section 17.6) suggests statistical use techniques [Rya11] that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. By collecting metrics during software testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"

19.2 PLANNING AND RECORDKEEPING

Many strategies can be used to test software. At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in the hope of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints all stakeholders. At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.

A testing strategy that is chosen by many software teams (and the one we recommend) falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units (sometimes on a daily basis), and culminating with tests that exercise the constructed system as it evolves. The remainder of this chapter will focus on component-level testing and test-case design.

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

The best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [Gil95] argues that a software testing strategy will succeed only when software testers: (1) specify product requirements in a quantifiable manner long before testing commences, (2) state testing objectives explicitly, (3) understand the users of the software and develop a profile for each user category, (4) develop a testing plan that emphasizes "rapid cycle testing,"[2] (5) build "robust" software that is designed to test

2 Gilb [Gil95] recommends that a software team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

itself (the concept of antibugging is discussed briefly in Section 9.3), (6) use effective technical reviews as a filter prior to testing, (7) conduct technical reviews to assess the test strategy and test cases themselves, and (8) develop a continuous improvement approach (Chapter 28) for the testing process.

These principles are reflected in agile software testing as well. In agile development, the test plan needs to be established before the first sprint meeting and reviewed by stakeholders. This plan merely lays out the rough time line, standards, and tools to be used. The test cases and directions for their use are developed and reviewed by the stakeholders as the code needed to implement each user story is created. Testing results are shared with all team members as soon as practical to allow changes in both existing and future code development. For this reason, many teams choose to keep their test recordkeeping in online documents.

Test recordkeeping does not need to be burdensome. The test cases can be recorded in a Google Docs spreadsheet that briefly describes the test case, contains a pointer to the requirement being tested, contains expected output from the test case data or the criteria for success, allows testers to indicate whether the test was passed or failed and the dates the test case was run, and should have room for comments about why a test may have failed to aid in debugging. This type of online form can be viewed as needed for analysis, and it is easy to summarize at team meetings. Test-case design issues are discussed in Section 19.3.

19.2.1 Role of Scaffolding

Component testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors. Each test case should be coupled with a set of expected results.

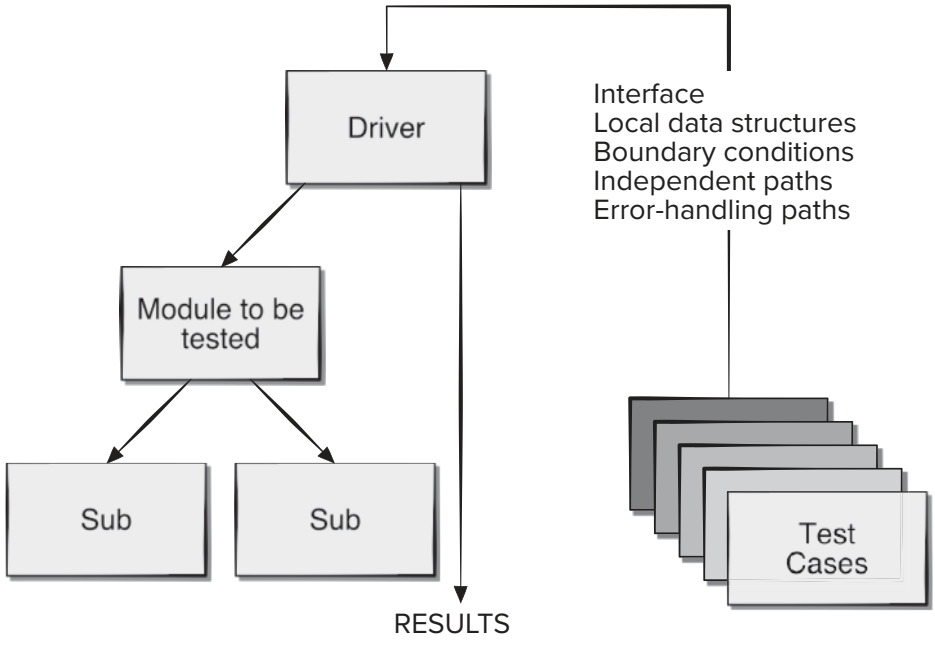
Because a component is not a stand-alone program, some type of *scaffolding* is required to create a testing framework. As part of this framework, driver and/or stub software must often be developed for each unit test. The unit-test environment is illustrated in Figure 19.3. In most applications a *driver* is nothing more than a "main program" that accepts test-case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (invoked by) the component to be tested. A *stub* or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent testing "overhead." That is, both are software that must be coded (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit-tested with "simple" scaffolding software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

19.2.2 Cost-Effective Testing

Exhaustive testing requires every possible combination of input values and test-case orderings be processed by the component being tested (e.g., consider the *move generator* in a computer chess game). In some cases, this would require the creation of

FIGURE 19.3
Unit-test environment



a near-infinite number of data sets. The return on exhaustive testing is often not worth the effort, since testing alone cannot be used to prove a component is correctly implemented. There are some situations in which you will not have the resources to do comprehensive unit testing. In these cases, testers should select modules crucial to the success of the project and those that are suspected to be error-prone because they have complexity metrics as the focus for your unit testing. Some techniques for minimizing the number of test cases required to do a good job testing are discussed in Sections 19.4 through 19.6.

INFO



Exhaustive Testing

Consider a 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules.

Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

19.3 TEST-CASE DESIGN

It is a good idea to design unit test cases before you develop code for a component. This ensures that you'll develop code that will pass the tests or at least the tests you thought of already.

Unit tests are illustrated schematically in Figure 19.4. The module interface is tested to ensure that information properly flows into and out of the program unit under test (Section 19.5.1). Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once (Section 19.4.2). Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing (Section 19.5.3). And finally, all error-handling paths are tested.

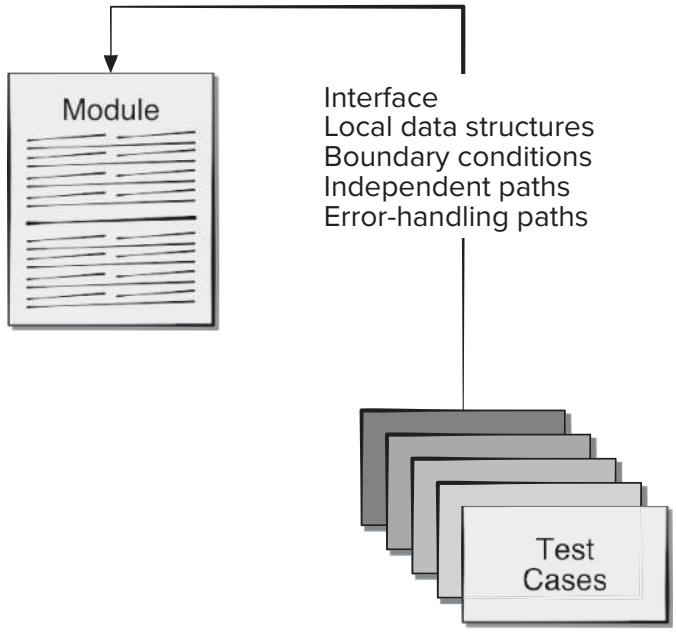
Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit-testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, or when the maximum or minimum allowable value is encountered. Test cases that exercise

FIGURE 19.4

Unit test



data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon [You75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test the error handling. Be sure that you design tests to execute every error-handling path. If you don't, the path may fail when it is invoked, exacerbating an already dicey situation.

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

SAFEHOME



Designing Unique Tests

The scene: Vinod's cubical.

The players: Vinod and Ed, members of the *SafeHome* software engineering team.

The conversation:

Vinod: So these are the test cases you intend to run for the *passwordValidation* operation.

Ed: Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.

Vinod: So let's see . . . you note that the correct password will be 8080, right?

Ed: Uh-huh.

Vinod: And you specify passwords 1234 and 6789 to test for error in recognizing invalid passwords?

Ed: Right, and I also test passwords that are close to the correct password, see . . . 8081 and 8180.

Vinod: Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

Ed: Well, they're different values.

Vinod: That's true, but if 1234 doesn't uncover an error . . . in other words . . . the *passwordValidation* operation notes that it's an invalid password, it's not likely that 6789 will show us anything new.

Ed: I see what you mean.

Vinod: I'm not trying to be picky here . . . it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.

Ed: Not a problem . . . I'll give this a bit more thought.

19.3.1 Requirements and Use Cases

In requirements engineering (Chapter 7) we suggested starting the requirements gathering process by working with the customers to generate user stories that developers can refine into formal use cases and analysis models. These use cases and models can be used to guide the systematic creation of test cases that do a good job of testing the functional requirements of each software component and provide good test coverage overall [Gut15].

The analysis artifacts do not provide much insight into the creation of test cases for many nonfunctional requirements (e.g., usability or reliability). This is where the customer's acceptance statements included in the user stories can form the basis for writing test cases for the nonfunctional requirements associated with components. Test-case developers make use of additional information based on their professional experience to quantify acceptance criteria to make it testable. Testing nonfunctional requirements may require the use of integration testing methods (Chapter 20) or other specialized testing techniques (Chapter 21).

The primary purpose of testing is to help developers discover defects that were previously unknown. Executing test cases that demonstrate the component is running correctly is often not good enough. As we mentioned earlier (Section 19.3), it is important to write test cases that exercise the error-handling capabilities of a component. But if we are to uncover new defects, it is also important to write test cases to test that a component does not do things it is not supposed to do (e.g., accessing privileged data sources without proper permissions). These may be stated formally as *anti-requirements*[3] and may require specialized security testing techniques (Section 21.7) [Ale17]. These so-called *negative test cases* should be included to make sure the component behaves according to the customer's expectations.

19.3.2 Traceability

To ensure that the testing process is auditable, each test case needs to be traceable back to specific functional or nonfunctional requirements or anti-requirements. Often nonfunctional requirements need to be traceable to specific business or architectural requirements. Many agile developers resist the concept of traceability as an unnecessary burden on developers. But many test process failures can be traced to missing traceability paths, inconsistent test data, or incomplete test coverage [Rem14]. Regression testing (discussed in Section 20.3) requires retesting selected components that may be affected by changes made to other software components that it collaborates with. Although this is more often considered an issue in integration testing (Chapter 20), making sure that test cases are traceable to requirements is an important first step and needs to be done during component testing.

19.4 WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing or structural testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

3 Anti-requirements are sometimes described during the creation of abuse cases that describe a user story from the perspective of a malicious user and are part of threat analysis (discussed in Chapter 18).

19.4.1 Basis Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe [McC76]. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

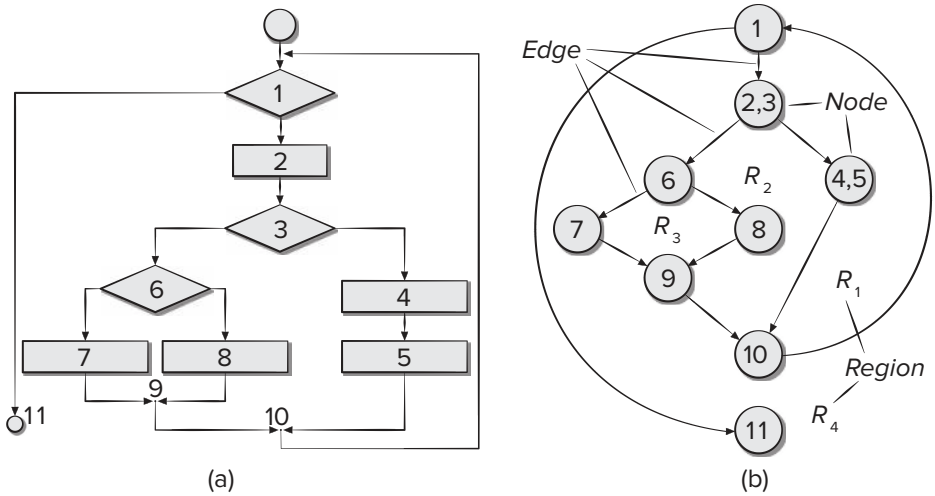
Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph), must be introduced.[4] A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allows you to trace program paths more readily.

To illustrate the use of a flow graph, consider the procedural design representation in Figure 19.5a. Here, a flowchart is used to depict program control structure. Figure 19.5b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 19.5b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow

FIGURE 19.5

(a) Flowchart and (b) flow graph



4 In actuality, the basis path method can be conducted without the use of flow graphs. However, they serve as a useful notation for understanding control flow and illustrating the approach.

graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 19.5b is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 19.5b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do you know how many paths to look for? The computation of cyclomatic complexity provides the answer. *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

   $V(G) = E - N + 2$

   where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

   where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 19.5b, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. $V(G) = 11$ edges $- 9$ nodes $+ 2 = 4$.
3. $V(G) = 3$ predicate nodes $+ 1 = 4$.

Therefore, the cyclomatic complexity of the flow graph in Figure 19.5b is 4.

More important, the value for $V(G)$ provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements. So in this case we would need to define at most four test cases to exercise each independent logic path.

SAFEHOME



Using Cyclomatic Complexity

The scene: Shakira's cubicle.

The players: Vinod and Shakira—members of the *SafeHome* software engineering team who are working on test planning for the security function.

The conversation:

Shakira: Look . . . I know that we should unit-test all the components for the security function, but there are a lot of 'em and if you consider the number of operations that have to be exercised, I don't know . . . maybe we should forget white-box testing, integrate everything, and start running black-box tests.

Vinod: You figure we don't have enough time to do component tests, exercise the operations, and then integrate?

Shakira: The deadline for the first increment is getting closer than I'd like . . . yeah, I'm concerned.

Vinod: Why don't you at least run white-box tests on the operations that are likely to be the most error-prone?

Shakira (exasperated): And exactly how do I know which are the most error-prone?

Vinod: *V* of *G*.

Shakira: Huh?

Vinod: Cyclomatic complexity—*V* of *G*. Just compute *V*(*G*) for each of the operations within each of the components and see which have the highest values for *V*(*G*). They're the ones that are most likely to be error-prone.

Shakira: And how do I compute *V* of *G*?

Vinod: It's really easy. Here's a book that describes how to do it.

Shakira (leafing through the pages): Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V*(*G*) will be the candidates for white-box tests.

Vinod: Just remember that there are no guarantees. A component with a low *V*(*G*) can still be error-prone.

Shakira: Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

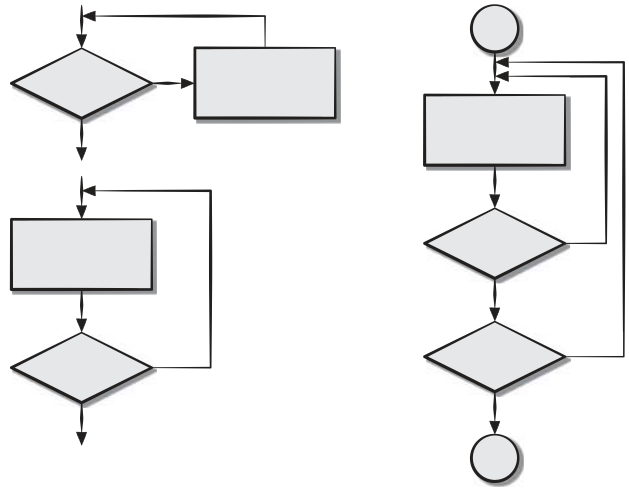
19.4.2 Control Structure Testing

The basis path testing technique described in Section 19.4.1 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. *Data flow testing* [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.

FIGURE 19.6

Classes of loops



Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Two different classes of loops [Bei90] can be defined: simple loops and nested loops. (Figure 19.6).

Simple Loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n - 1$, n, $n + 1$ passes through the loop.

Nested Loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

19.5 BLACK-BOX TESTING

Black-box testing, also called *behavioral testing* or *functional testing,* focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria [Mye79]: test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

19.5.1 Interface Testing

Interface testing is used to check that the program component accepts information passed to it in the proper order and data types and returns information in proper order and data format [Jan16]. Interface testing is often considered part of integration testing. Because most components are not stand-alone programs, it is important to make sure that when the component is integrated into the evolving program it will not break the build. This is where the use stubs and drivers (Section 19.2.1) become important to component testers.

Stubs and drivers sometimes incorporate test cases to be passed to the component or accessed by the component. In other cases, debugging code may need to be inserted inside the component to check that data passed was received correctly (Section 19.3). In still other cases, the testing framework should contain code to check that data returned from the component is received correctly. Some agile developers prefer to do interface testing using a copy of the production version of the evolving program with some of this debugging code added.

19.5.2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

19.5.3 Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b.*
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering

analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

19.6 OBJECT-ORIENTED TESTING

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation X is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation X, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation X is used varies in subtle ways, it is necessary to test operation X in the context of each of the subclasses. This means that testing operation X in a stand-alone fashion (the conventional unit-testing approach) is usually ineffective in the object-oriented context.

19.6.1 Class Testing

Class testing for object-oriented (OO) software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

To provide brief illustrations of these methods, consider a banking application in which an **Account** class has the following operations: *open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(),* and *close()* [Kir94]. Each of these operations may be applied for **Account,** but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of **Account** includes the following operations:

```
open•setup•deposit•withdraw•close
```

This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]^n• withdraw•close

A variety of different operation sequences can be generated randomly. For example:

Test case r_1*:*

open•setup•deposit•deposit•balance•summarize•withdraw•close

Test case r_2*:*

open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close

These and other random order tests are conducted to exercise different class instance life histories. Use of test equivalence partitioning (Section 19.5.2) can reduce the number of test cases required.

SAFEHOME



Class Testing

The scene: Shakira's cubicle.

The players: Jamie and Shakira, members of the *SafeHome* software engineering team who are working on test-case design for the security function.

The conversation:

Shakira: I've developed some tests for the **Detector** class [Figure 11.4]—you know, the one that allows access to all of the **Sensor** objects for the security function. You familiar with it?

Jamie (laughing): Sure, it's the one that allowed you to add the "doggie angst" sensor.

Shakira: The one and only. Anyway, it has an interface with four ops: *read(), enable(), disable(),* and *test().* Before a sensor can be read, it must be enabled. Once it's enabled, it can be read and tested. It can be disabled at any time, except if an alarm condition is being processed. So I defined a simple test sequence that will exercise its behavioral life history. (She shows Jamie the following sequence.)

#1: enable•test•read•disable

Jamie: That'll work, but you've got to do more testing than that!

Shakira: I know, I know. Here are some other sequences I've come up with. (She shows Jamie the following sequences.)

#2: enable•test•[read]^n•test•disable
#3: [read]^n
#4: enable•disable•[test | read]

Jamie: So let me see if I understand the intent of these. #1 goes through a normal life history, sort of a conventional usage. #2 repeats the read operation *n* times, and that's a likely scenario. #3 tries to read the sensor before it's been enabled . . . that should produce an error message of some kind, right? #4 enables and disables the sensor and then tries to read it. Isn't that the same as test #2?

Shakira: Actually no. In #4, the sensor has been enabled. What #4 really tests is whether the disable op works as it should. A *read()* or *test()* after *disable()* should generate the error message. If it doesn't, then we have an error in the disable op.

Jamie: Cool. Just remember that the four tests have to be applied for every sensor type since all the ops may be subtly different depending on the type of sensor.

Shakira: Not to worry. That's the plan.

19.6.2 Behavioral Testing

The use of the state diagram as a model that represents the dynamic behavior of a class is discussed in Chapter 8. The state diagram for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure 19.7 [Kir94] illustrates a state diagram for the **Account** class discussed earlier. Referring to the figure, initial transitions move through the *empty acct* and *setup acct* states. The majority of all behavior for instances of the class occurs while in the *working acct* state. A final withdrawal and account closure cause the account class to make transitions to the *nonworking acct* and *dead acct* states, respectively.

The tests to be designed should achieve coverage of every state. That is, the operation sequences should cause the **Account** class to transition through all allowable states:

Test case s_1: `open•setupAccnt•deposit (initial)•withdraw (final)•close`

Adding additional test sequences to the minimum sequence,

Test case s_2: `open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close`

Test case s_3: `open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close`

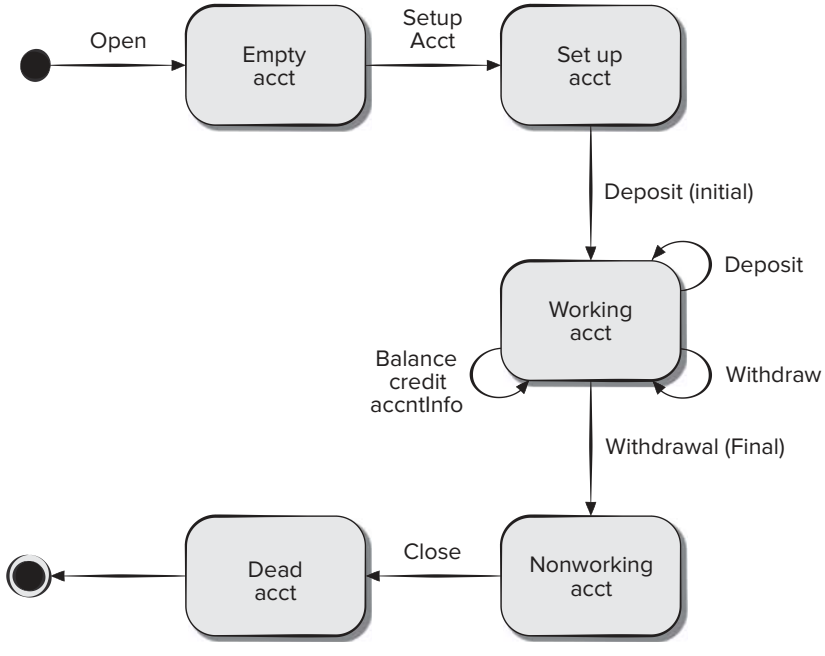


FIGURE 19.7

State diagram for the Account class
Source: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 1994, 79.

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple state diagrams are used to track the behavioral flow of the system.

The state model can be traversed in a "breadth-first" [McG94] manner. In this context, breadth-first implies that a test case exercises a single transition and that when a new transition is to be tested, only previously tested transitions are used.

Consider a **CreditCard** object that is part of the banking system. The initial state of **CreditCard** is *undefined* (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a *defined* state; that is, the attributes `card number` and `expiration date`, along with bank-specific identifiers, are defined. The credit card is *submitted* when it is sent for authorization, and it is *approved* when authorization is received. The transition of **CreditCard** from one state to another can be tested by deriving test cases that cause the transition to occur. A breadth-first approach to this type of testing would not exercise *submitted* before it exercised *undefined* and *defined.* If it did, it would make use of transitions that had not been previously tested and would therefore violate the breadth-first criterion.

19.7 SUMMARY

Software testing accounts for the largest percentage of technical effort in the software process. Regardless of the type of software you build, a strategy for systematic test planning, execution, and control begins by considering small elements of the software and moves outward toward the program as a whole.

The objective of software testing is to uncover errors. For conventional software, this objective is achieved through a series of test steps. Unit and integration tests (discussed in Chapter 20) concentrate on functional verification of a component and incorporation of components into the software architecture. The strategy for testing object-oriented software begins with tests that exercise the operations within a class and then moves to thread-based testing for integration (discussed in Section 20.4.1). Threads are sets of classes that respond to an input or event.

Test cases should be traceable to software requirements. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened. The primary objective for test-case design is to derive a set of tests that have the highest likelihood for uncovering errors in software. To accomplish this objective, two different categories of test-case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability.

Unlike testing (a systematic, planned activity), debugging can be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Testing can sometimes help find the root cause of the error. But often, the most valuable resource is the counsel of other members of the software engineering staff.

PROBLEMS AND POINTS TO PONDER

19.1. Using your own words, describe the difference between verification and validation. Do both make use of test-case design methods and testing strategies?

19.2. List some problems that might be associated with the creation of an independent test group. Are an ITG and an SQA group made up of the same people?

19.3. Why is a highly coupled module difficult to unit test?

19.4. Is unit testing possible or even desirable in all circumstances? Provide examples to justify your answer.

19.5. Can you think of any additional testing objectives that are not discussed in Section 19.1.1?

19.6. Select a software component that you have designed and implemented recently. Design a set of test cases that will ensure that all statements have been executed using basis path testing.

19.7. Myers [Mye79] uses the following program as a self-assessment for your ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

19.8. Design and implement the program (with error handling where appropriate) specified in Problem 19.7. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.

19.9. Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.

19.10. In your own words, describe why the class is the smallest reasonable unit for testing within an OO system.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

20

SOFTWARE TESTING—INTEGRATION LEVEL

A single developer may be able to test software components without involving other team members. This is not true for integration testing where a component must interact properly with components developed by other team members. Integration testing exposes many weaknesses of software development groups who have not gelled as a team. Integration testing presents an interesting dilemma for software engineers, who by their nature are constructive people. In fact, all testing requires that developers discard preconceived notions of the "correctness" of software just developed and instead work hard to design test cases to "break" the software. This means that team members need to be able to accept suggestions from other team members that their code is not behaving properly when it is tested as part of the latest software increment.

KEY CONCEPTS

artificial intelligence 403
black-box testing 397
bottom-up integration 399
continuous integration 400
cluster testing 404
fault-based testing 405
integration testing 398
multiple-class partition testing 405
patterns 409
regression testing 402
scenario-based testing 407
smoke testing 400
thread-based testing 404
top-down integration 398
validation testing 407
white-box testing 397



QUICK LOOK

What is it? Integration testing assembles components in a manner that allows the testing of increasingly larger software functions with the intent of finding errors as the software is assembled.

Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved in addition to other stakeholders.

Why is it important? Test cases must be designed using disciplined techniques to ensure that the components have been integrated properly into the complete software product.

What are the steps? Internal program logic is exercised using "white-box" test-case design techniques and software requirements are exercised using "black-box" test-case design techniques.

What is the work product? A set of test cases designed to exercise internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion, and review the test cases you do create for thoroughness.

Beizer [Bei90] describes a "software myth" that all testers face. He writes: "There's a myth that if we were really good at programming, there would be no bugs to catch . . . There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it."

Should testing instill guilt? Is testing really destructive? The answer to these questions is, No!

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and *systems testing* (a series of system integration and validation tests) is conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

In this chapter, we discuss techniques for software integration testing strategies applicable to most software applications. Specialized software testing strategies are discussed in Chapter 21.

20.1 SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that have the highest likelihood of uncovering a whole class of errors.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach

takes an external view of testing and is called *black-box testing*. The second requires an internal view of testing and is termed *white-box testing*.[1] Both are useful in integration testing [Jan16].

20.1.1 Black-Box Testing

Black-box testing alludes to integration testing that is conducted by exercising the component interfaces with other components and with other systems. It examines some fundamental aspect of a system with little regard for the internal logical structure of the software. Instead, the focus is on ensuring the component executes correctly in the larger software build when the input data and software context specified by its preconditions is correct and behaves in the ways specified by its postconditions. It is of course important to make sure that the component behaves correctly when its preconditions are not satisfied (e.g., it can handle bad inputs without crashing).

Black-box testing is based on the requirements specified in user stories (Chapter 7). Test-case authors do not need to wait for the component implementation code to be written once the component interface is defined. Several cooperating components may need to be written to implement the functionality defined by a single user story. Validation testing (Section 20.5) often defines black-box test cases in terms of the end-user visible input actions and observable output behaviors, without any knowledge of how the components themselves were implemented.

20.1.2 White-Box Testing

White-box testing, sometimes called *glass-box testing* or *structural testing,* is an integration testing philosophy that uses implementation knowledge of the control structures described as part of component-level design to derive test cases. White-box testing of software is predicated on close examination of procedural implementation details and data structure implementation details. White-box tests can be designed only after component-level design (or source code) exists. The logical details of the program must be available. Logical paths through the software and collaborations between components are the focus of white-box integration testing.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. Testers should select a reasonable number of important logical paths to exercise once component integration occurs. Important data structures should also be tested for validity after component integration.

1 The terms *functional testing* and *structural testing* are sometimes used in place of black-box and white-box testing, respectively.

20.2 INTEGRATION TESTING

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit-tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; and global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt nonincremental integration, that is, to construct the program using a "big bang" approach. In the big bang approach, all components are combined in advance and the entire program is tested as a whole. Chaos usually results! Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Taking the big bang approach to integration is a lazy strategy that is doomed to failure.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. Integrate incrementally and testing as you go is a more cost-effective strategy. We discuss several common incremental integration testing strategies in the remainder of this chapter.

20.2.1 Top-Down Integration

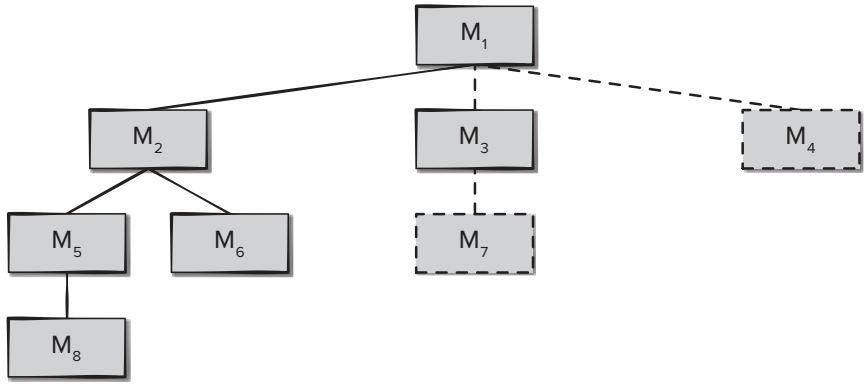
Top-down integration testing is an incremental approach to construction of the software architecture. Modules (also referred to as components in this book) are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Referring to Figure 20.1, *depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics (e.g., components needed to implement one use case). For example, selecting the left-hand path, components M_1, M_2, M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2, M_3, and M_4 would be integrated first. The next control level, M_5, M_6, and so on, follows. The integration process is performed in a series of five steps:

1. The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.

FIGURE 20.1

Top-down integration



2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

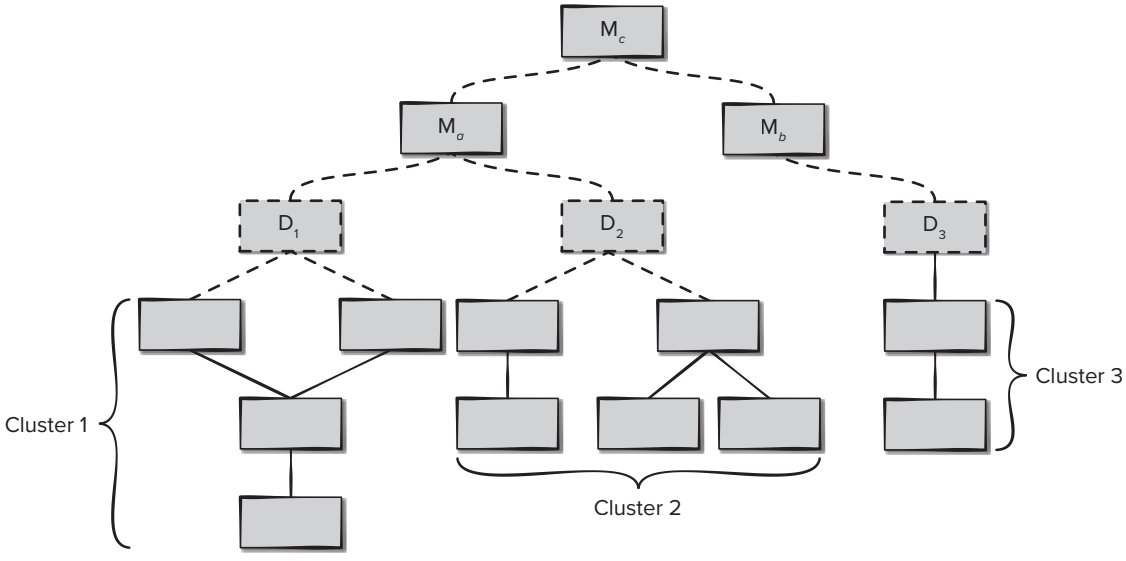
The top-down integration strategy verifies major control or decision points early in the test process. In a "well-factored" program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

20.2.2 Bottom-Up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Bottom-up integration eliminates the need for complex stubs. Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined, moving upward in the program structure.

FIGURE 20.2 Bottom-up integration



Integration follows the pattern illustrated in Figure 20.2. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed, and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

20.2.3 Continuous Integration

Continuous integration is the practice of merging components into the evolving software increment once or more each day. This is a common practice for teams following agile development practices such as XP (Section 3.5.1) or DevOps (Section 3.5.2). Integration testing must take place quickly and efficiently if a team is attempting to always have a working program in place as part of continuous delivery. It is sometimes hard to maintain systems with the use of continuous integration tools [Ste18]. Maintenance and continuous integration issues are discussed in more detail in Section 22.4.

Smoke testing is an integration testing approach that can be used when product software is developed by an agile team using short increment build times. Smoke testing might be characterized as a rolling or continuous integration strategy. The software is rebuilt (with new components added) and smoke tested every day. It is designed as a pacing mechanism for time-critical projects, allowing the software team

to assess the project on a frequent basis. In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing gives both managers and practitioners a realistic assessment of integration testing progress. McConnell [McC96] describes the smoke test in the following manner:

> The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software projects:

- **Integration risk is minimized.** Because smoke tests are conducted daily, incompatibilities and other showstopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- **The quality of the end product is improved.** Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- **Error diagnosis and correction are simplified.** Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- **Progress is easier to assess.** With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

In some ways smoke testing resembles regression testing (discussed in Section 20.3), which helps to ensure that the newly added components do not interfere with the behaviors of existing components that were previously tested. To do this, it is a good idea to rerun a subset of the test cases that executed with the existing software component before the new components were added. The effort required to rerun test cases is not trivial, and automated testing can be used to reduce the time and effort re-created to rerun these test cases [Net18]. A complete discussion of automated testing is

beyond the scope of this chapter, but links to representative tools can be found on the Web pages that supplement this book.[2]

20.2.4 Integration Test Work Products

An overall plan for integration of the software and a description of specific tests is documented in a *test specification.* This work product incorporates a test plan and a test procedure and becomes part of the software configuration. Testing is divided into phases and incremental builds that address specific functional and behavioral characteristics of the software. For example, integration testing for the *SafeHome* security system might be divided into the following test phases: user interaction, sensor processing, communications functions, and alarm processing.

Each integration test phase delineates a broad functional category within the software and generally can be related to a specific domain within the software architecture. Therefore, software increments are created to correspond to each phase.

A schedule for integration, the development of scaffolding software (Section 19.2.1), and related topics are also discussed as part of the test plan. Start and end dates for each phase are established, and "availability windows" for unit-tested modules are defined. When developing a project schedule, you'll have to consider the manner in which integration occurs so that components will be available when needed. A brief description of scaffolding software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results are also included. In the agile world, this level of test-case description occurs when code to implement the user story is being developed so the code can be tested as soon as it is ready for integration.

A history of actual test results, problems, or peculiarities is recorded in a *test report* that can be appended to the *test specification.* It is often best to implement the test report as a shared Web document to allow all stakeholders access to the latest test results and the current state of the software increment. Information contained in this online document can be vital to developers during software maintenance (Section 4.9).

20.3 Artificial Intelligence and Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new input/output (I/O) may occur, and new control logic is invoked. Side effects associated with these changes may cause problems with functions that previously worked flawlessly. In the context of an integration

2 See the SEPA 9e website.

test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression tests should be executed every time a major change is made to the software (including the integration of new components). Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent playback and comparison. The *regression test suite* (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions
- Additional tests that focus on software functions that are likely to be affected by the change
- Tests that focus on the software components that have been changed

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Yoo and Harman [Yoo13] write about potential uses of artificial intelligence (AI) in identifying test cases for use in regression test suites. A software tool could examine the dependencies among the components in the software increment after the new components have been added and generate test cases automatically to use for regression testing. Another possibility would be using machine learning techniques to select sets of test cases that will optimize the discovery of component collaboration errors. This work is promising, but still requires significant human interaction to review the test cases and the recommended order for executing them.

SAFEHOME



Regression Testing

The scene: Doug Miller's office, as integration testing is under way.

The players: Doug Miller, software engineering manager; Vinod, Jamie, Ed, and Shakira, members of the *SafeHome* software engineering team.

The conversation:

Doug: It seems to me that we are not spending enough time retesting software components after new components are integrated.

Vinod: I guess that is true, but isn't it good enough that we are testing the new components' interactions with the components they are supposed to collaborate with?

Doug: Not always. Sometimes components make unintended change to data used by other components. I know we are busy, but it is important to discover these problems early.

Shakira: We do have a test-case repository we have been drawing from. Perhaps we can randomly select several test cases to run using our automated testing framework.

Doug: That's a start. But maybe we should be more strategic in how we select our test cases.

Ed: I suppose we could use our test-case/requirement traceability table and check our CRC card model.

Vinod: I have been using continuous integration, meaning I integrate each component as soon as one of the developers passes it to me. I try to run a series of regression tests on the partially integrated program.

Jamie: I've been trying to design a set of appropriate tests for each function in the system. Maybe I should tag some of the more important ones for Vinod to use for regression testing.

Doug (to Vinod): How often will you run the regression test cases?

Vinod: Every day I integrate a new component I will use the regression test cases . . . until we decide the software increment is done.

Doug: Let's try using Jamie's regression test cases as they are created and see how things go.

20.4 Integration Testing in the OO Context

Object-oriented software does not have an obvious hierarchical control structure, so traditional top-down and bottom-up integration strategies (Section 20.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [Ber93].

There are two different strategies for integration testing of OO systems: thread-based testing and use-based testing [Bin99]. The first, *thread-based testing*, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. An important strategy for integration testing of OO software is thread-based testing. Threads are sets of classes that respond to an input or event.

The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed. Use-based tests focus on classes that do not collaborate heavily with other classes.

The use of scaffolding software also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

20.4.1 Fault-Based Test-Case Design[3]

The object of *fault-based testing* within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, preliminary planning required to perform fault-based testing begins with the analysis model. The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Of course, the effectiveness of these techniques depends on how testers perceive a plausible fault. If real faults in an OO system are perceived to be implausible, then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, and incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

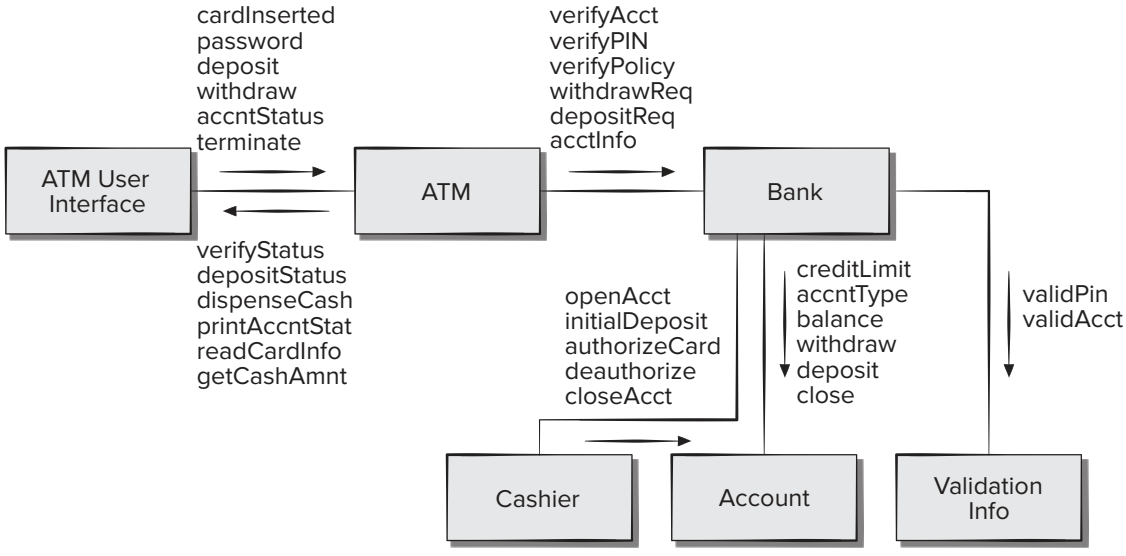
The approach for multiple-class partition testing is similar to the approach used for partition testing of individual classes. A single class is partitioned as discussed in Section 19.6.1. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative approach partitions tests based on the interfaces to a particular class. Referring to Figure 20.3, the **Bank** class receives messages from the **ATM** and **Cashier** classes. The methods within **Bank** can therefore be tested by partitioning them into those that serve **ATM** and those that serve **Cashier.**

Kirani and Tsai [Kir94] suggest the following sequence of steps to generate multiple-class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.

3 Sections 20.4.1 and 20.4.2 have been adapted from an article by Brian Marick originally posted on the Internet newsgroup comp.testing. This adaptation is included with the permission of the author. For further information on these topics, see [Mar94]. It should be noted that the techniques discussed in Sections 20.4.1 and 20.4.2 are also applicable for conventional software.

FIGURE 20.3 Class collaboration diagram for banking application



Source: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 4, 1994, 72.

3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate [Kir94], consider a sequence of operations for the **Bank** class relative to an **ATM** class (Figure 20.3):

```
verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]^n
```

A random test case for the **Bank** class might be

Test case r_3 = `verifyAcct•verifyPIN•depositReq`

To consider the collaborators involved in this test, the messages associated with each of the operations noted in test case r_3 are considered. **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct()* and *verifyPIN()*. **Bank** must collaborate with **Account** to execute *depositReq()*. Hence, a new test case that exercises these collaborations is

Test case r_4 = `verifyAcct [Bank:validAcctValidationInfo]•verifyPIN`
`[Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]`

20.4.2 Scenario-Based Test-Case Design

Fault-based testing misses two main types of errors: (1) incorrect specifications and (2) interactions among subsystems. When errors associated with an incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong

thing or omit important functionality. But in either circumstance, quality (conformance to requirements) suffers. Errors associated with subsystem interaction occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

Scenario-based testing will uncover errors that occur when any actor interacts with the software. Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests. This is very similar to thread testing.

Scenario testing uncovers interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

Test-case design becomes more complicated as integration of the object-oriented system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test-case generation" [Kir94], we expand the banking example introduced in Section 19.6 to include the classes and collaborations noted in Figure 20.3. The direction of the arrows in the figure indicates the direction of messages, and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

20.5 VALIDATION TESTING

Like all testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point, a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a *software requirements specification* has been developed, it describes each user story, all user-visible attributes, and the customer's acceptance criteria for each. The customer's acceptance criteria form the basis for a validation-testing approach.

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability,

compatibility, error recovery, maintainability). If a deviation from specification is uncovered, a *deficiency list* is created. A method for resolving deficiencies (acceptable to stakeholders) must be established. Specialized testing methods for these nonfunctional requirements are discussed in Chapter 21.

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit, is discussed in more detail in Chapter 22.

SAFEHOME



Preparing for Validation

The scene: Doug Miller's office, as component-level design continues and construction of certain components continues.

The players: Doug Miller, software engineering manager; Vinod, Jamie, Ed, and Shakira, members of the *SafeHome* software engineering team.

The conversation:

Doug: The first increment will be ready for validation in what . . . about three weeks?

Vinod: That's about right. Integration is going well. We're smoke testing daily, finding some bugs, but nothing we can't handle. So far, so good.

Doug: Talk to me about validation.

Shakira: Well, we'll use all of the use cases as the basis for our test design. I haven't started yet, but I'll be developing tests for all of the use cases that I've been responsible for.

Ed: Same here.

Jamie: Me too, but we've got to get our act together for acceptance testing and also for alpha and beta testing, no?

Doug: Yes. In fact I've been thinking; we could bring in an outside contractor to help us with validation. I have the money in the budget . . . and it'd give us a new point of view.

Vinod: I think we've got it under control.

Doug: I'm sure you do, but an ITG gives us an independent look at the software.

Jamie: We're tight on time here, Doug. I for one don't have the time to babysit anybody you bring in to do the job.

Doug: I know, I know. But if an ITG works from requirements and use cases, not too much babysitting will be required.

Vinod: I still think we've got it under control.

Doug: I hear you, Vinod, but I am going to overrule on this one. Let's plan to meet with the ITG rep later this week. Get 'em started and see what they come up with.

Vinod: Okay, maybe it'll lighten the load a bit.

At the validation or system level, the details of class connections disappear. The validation of OO software focuses on user-visible actions and user-recognizable outputs from the system. To assist in the derivation of validation tests, the tester should draw upon use cases (Chapters 7 and 8) that are part of the requirements model. The use case provides a scenario that has a high likelihood of uncovered errors in user-interaction requirements. Conventional black-box testing methods (Chapter 19) can be used to drive validation tests. In addition, you may choose to derive test cases from the object-behavior model created as part of object-oriented analysis (OOA).

20.6 TESTING PATTERNS

The use of patterns as a mechanism for describing solutions to specific design problems was discussed in Chapter 15. But patterns can also be used to propose solutions to other software engineering situations—in this case, software testing. *Testing patterns* describe common testing problems and solutions that can assist you in dealing with them.

Much of software testing, even during the past decade, has been an ad hoc activity. If testing patterns can help a software team to communicate about testing more effectively, to understand the motivating forces that lead to a specific approach to testing, and to approach the design of tests as an evolutionary activity in which each iteration results in a more complete suite of test cases, then patterns have accomplished much.

Testing patterns are described in much the same way as design patterns (Chapter 15). Dozens of testing patterns have been proposed in the literature (e.g., [Mar02]). The following three testing patterns (presented in abstract form only) provide representative examples:

Pattern name: **PairTesting**

Abstract: A process-oriented pattern, **pair testing** describes a technique that is analogous to pair programming [Chapter 3] in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Pattern name: **SeparateTestInterface**

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The **SeparateTestInterface** pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component" [Lan01].

Pattern name: **ScenarioTesting**

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement [Kan01].

A comprehensive discussion of testing patterns is beyond the scope of this book. If you have further interest, see [Bin99], [Mar02], [Tho04], [Mac10], and [Gon17] for additional information on this important topic.

20.7 SUMMARY

Integration testing builds the software architecture while at the same time conducting tests to uncover errors associated with interfacing between software components. The objective is to take unit-tested components and build a program structure that has been dictated by design.

Experienced software developers often say, "Testing never ends; it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test-case design, you can

achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

Hetzel [Het84] describes white-box testing as "testing in the small." His implication is that the white-box tests that have been considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens your focus and might be called "testing in the large."

Black-box integration testing is based on the requirements specified in the user stories or some other analysis modeling representation. Test-case authors do not need to wait for the component implementation code to be written, as long as they understand the required functionality of the components undergoing testing. Validation testing is often accomplished with black-box test cases that produce end-user visible input actions and observable output behaviors.

White-box testing requires a close examination of procedural implementation details and data structure implementation details for the components undergoing testing. White-box tests can be designed only after component-level design (or source code) exists. Logical paths through the software and collaborations between components are the focus of white-box integration testing.

Integration testing of OO software can be accomplished using a thread-based or use-based strategy. Thread-based testing integrates the set of classes that collaborate to respond to one input or event. Use-based testing constructs the system in layers, beginning with those classes that do not make use of server classes. Integration test-case design methods can also make use of random and partition tests. In addition, scenario-based testing and tests derived from behavioral models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations.

OO system validation testing is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software. However, scenario-based testing dominates the validation of OO systems, making the use case a primary driver for validation testing.

Regression testing is the process of reexecuting a selected test case following any change made to a software system. Regression tests should be executed whenever new components or changes are added to a software increment. Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.

Testing patterns describe common testing problems and solutions that can assist you in dealing with them. If testing patterns can help a software team to communicate about testing more effectively, to understand the motivating forces that lead to a specific approach to testing, and to approach the design of test cases as an evolutionary activity in which each iteration results in a more complete suite of test cases, then patterns have accomplished much.

PROBLEMS AND POINTS TO PONDER

20.1. How can project scheduling affect integration testing?

20.2. Who should perform validation testing—the software developer or the software user? Justify your answer.

20.3. Will exhaustive testing (even if it is possible for very small programs) guarantee that a program is 100 percent correct?

20.4. Why should "testing" begin with requirements analysis and design?

20.5. Should nonfunctional requirements (e.g., security or performance) be tested as part of integration testing?

20.6. Why do we have to retest subclasses that are instantiated from an existing class, if the existing class has already been thoroughly tested?

20.7. What is the difference between thread-based and use-based strategies for integration testing?

20.8. Develop a complete test strategy for the *SafeHome* system discussed earlier in this book. Document it in a *test specification.*

20.9. Pick one of the *SafeHome* system user stories to use as the basis of scenarios-based testing, and construct a set of integration test cases needed to do integration testing for that user story.

20.10. For the test cases you wrote in Problem 20.9, identify a subset of test cases you will use for regression testing software components that are added to the program.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

21 SOFTWARE TESTING—SPECIALIZED TESTING FOR MOBILITY

The same sense of urgency that drives MobileApp projects also pervades all mobility projects. Stakeholders are worried that they will miss a market window and press to introduce the MobileApp to its intended market. Technical activities that often occur late in the process, such as performance and security testing, are sometimes given short shrift. Usability testing that should occur during the design phase may end up being deferred until just before delivery. These can be catastrophic mistakes. To avoid this situation, you and other team members must ensure that each work product exhibits high quality, or users will move to a competing product [Soa11].

KEY CONCEPTS

accessibility testing 432
alpha test 430
beta test 430
compatibility testing 414
content testing 420
documentation testing 434
internationalization 423
load testing 425
model-based testing 429
navigation testing 421
performance testing 424
real-time testing 426
recovery testing 417
security testing 423
stress testing 425
testing AI systems 428
testing guidelines 413
test strategies for MobileApps 413
test strategies for WebApps 418
usability testing 430



QUICK LOOK

What is it? Mobility testing is a collection of related activities with a single goal: to uncover errors in MobileApp content, function, usability, navigability, performance, capacity, and security.

Who does it? Software engineers and other project stakeholders (managers, customers, end users) all participate in mobility testing.

Why is it important? If end users encounter errors or difficulties within the MobileApp, they will go elsewhere for the personalized content and function they need.

What are the steps? The mobility testing process begins by focusing on user-visible aspects of the MobileApp and proceeds to tests that exercise technology and infrastructure.

What is the work product? A MobileApp test plan is often produced. A suite of test cases is developed for each testing step, and an archive of test results is maintained for future use.

How do I ensure that I've done it right? Although you can never be sure that you've performed every test that is needed, you can be certain that testing has uncovered errors (and that those errors have been corrected). In addition, if you've established a test plan, you can check to ensure that all planned tests have been conducted.

MobileApp requirements and design models cannot be tested solely with executable test cases. You and your team should conduct technical reviews (Chapter 16) that examine usability (Chapter 12) as well as MobileApp performance and security.

There are several important questions to ask when creating a mobility testing strategy [Sch09]:

- Do you have to build a fully functional prototype before you test with users?
- Should you test with the user's device or provide a device for testing?
- What devices and user groups should you include in testing?
- What are the trade-offs associated with lab testing versus remote testing?

We address each of these questions throughout this chapter.

21.1 Mobile Testing Guidelines

MobileApps that run entirely on a mobile device can be tested using traditional software testing methods (Chapters 19 and 20). Alternatively, they can be tested using emulators running on personal computers. Things become more complicated when thin-client MobileApps[1] are to be tested. They exhibit many of the same testing challenges found in WebApps (Section 20.2), but thin-client MobileApps have the additional concerns associated with transmission of data through Internet gateways and telephone networks [Was10].

In general, users expect MobileApps to be context aware and deliver personalized user experiences based on the physical location of a device in relation to available network features. Testing MobileApps in dynamic ad hoc networks for every possible device and network configuration is difficult, if not impossible.

MobileApps are expected to deliver much of the complex functionality and reliability found in desktop applications, but they are resident on mobile platforms with relatively limited resources. The following guidelines provide a basis for mobile application testing [Kea07]:

- Understand the network and device landscape before testing to identify bottlenecks (Section 21.6).
- Conduct tests in uncontrolled real-world test conditions (field-based testing, Section 21.8).
- Select the right automation test tool (Section 21.11).
- Use the Weighted Device Platform Matrix method to identify the most critical hardware/platform combination to test (Section 21.8).
- Check the end-to-end functional flow in all possible platforms at least once (Section 21.10).

1 Thin-client apps typically have software for the user interface running on the mobile device (or Web browser software) and use a network interface to an Internet-based application or cloud-based data storage.

- Conduct performance testing, GUI testing, and compatibility testing using actual devices (Sections 21.8 and 21.11).
- Measure performance only in realistic conditions of wireless traffic and user load (Section 21.8).

21.2 The Testing Strategies

The strategy for testing mobile applications adopts the basic principles for all software testing. However, the unique nature of MobileApps demands the consideration of a number of specialized issues:

- **User-experience testing.** Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices (Section 21.3).
- **Device compatibility testing.** Testers verify that the MobileApp works correctly on all required hardware and software combinations (Section 21.9).
- **Performance testing.** Testers check nonfunctional requirements unique to mobile devices (e.g., download times, processor speed, storage capacity, power availability) (Section 21.8).
- **Connectivity testing.** Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access (Section 21.6).
- **Security testing.** Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users (Section 21.7).
- **Testing in the wild.** The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe (Section 21.9).
- **Certification testing.** Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it.

Technology alone is not sufficient to guarantee commercial success of a MobileApp. Users abandon MobileApps quickly if they do not work well or fail to meet expectations. It is important to recall that testing has two important goals: (1) to create test cases that uncover defects early in the development cycle and (2) to verify the presence of important quality attributes. The quality attributes for MobileApps are based on those set forth in ISO 2050:2011 [ISO17] and encompass functionality, reliability, usability, efficiency, maintainability, and portability (Chapter 17).

Developing a MobileApp testing strategy requires an understanding of both software testing and the challenges that make mobile devices and their network infrastructure unique [Kho12a]. In addition to a thorough knowledge of conventional software testing approaches (Chapters 19 and 20), a MobileApp tester should have a good understanding of telecommunications principles and an awareness of the differences and capabilities of mobile operating systems platforms. This basic knowledge must be complemented with a thorough understanding of the different types of mobile testing (e.g., MobileApp testing, mobile handset testing, mobile website testing), the use

of simulators, test automation tools, and remote data access services (RDA). Each of these topics is discussed later in this chapter.

21.3 User Experience Testing Issues

In a crowded marketplace in which products provide the same functionality, users will choose the MobileApp that is easiest to use. The user interface and its interaction mechanisms are visible to the MobileApp users. It is important to test the quality of the user experience provided by the MobileApp to ensure that it meets the expectations of its users.

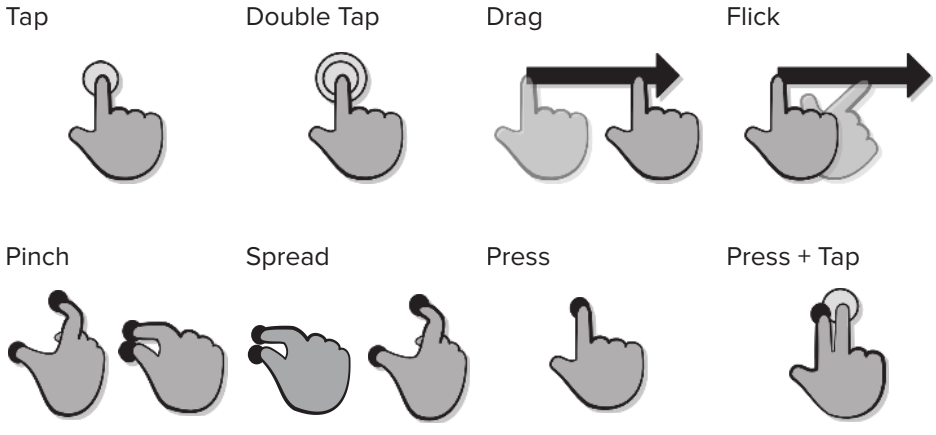
What characteristics of MobileApp usability become the focus of testing, and what specific objectives are addressed? Many of the procedures for assessing the usability of software user interfaces discussed in Chapters 12 and 13 can be used to assess MobileApps. Similarly, many of the strategies used to assess the quality of WebApps (Section 21.5) may be used to test the user interface portion of the MobileApp. There is more to building a good MobileApp user interface than simply shrinking the size of a user interface from an existing desktop application.

21.3.1 Gesture Testing

Touch screens are ubiquitous on mobile devices, and, as a consequence, developers have added multitouch gestures (e.g., swiping, zooming, scrolling, selection) as a means of augmenting the user interaction possibilities without losing screen real estate. Figure 21.1 shows several commonly found gestures on MobileApps. Unfortunately, gesture-intensive interfaces present a number of review and testing challenges.

Paper prototypes, sometimes developed as part of the design, cannot be used to adequately review the adequacy or efficacy of gestures. When testing is initiated, it's difficult to use automated tools to test touch or gesture interface actions. The location

FIGURE 21.1
Mobile app gestures



of screen objects is affected by screen size and resolution, as well as previous user actions, making accurate gesture testing difficult. And even as testing is conducted, gestures are hard to log accurately for replay.

Instead, testers need to create test framework programs that make calls to functions that simulate gesture events. All of this is expensive and time consuming.

Accessibility testing for visually impaired users is challenging because gesture interfaces typically do not provide either tactile or auditory feedback. Usability and accessibility testing for gestures become very important for ubiquitous devices like smartphones. It may be important to test the operation of the device when gesture operations are not available.

Ideally, user stories or use cases are written in sufficient detail to allow their use as the basis for test scripts. It is important to recruit representative users and include all targeted devices to take screen differences into account when testing gestures with a MobileApp. Finally, testers should ensure that the gestures conform to the standards and contexts set for the mobile device or platform.

21.3.2 Virtual Keyboard Input

Because a virtual keyboard may obscure part of the display screen when activated, it is important to test the MobileApp to ensure that important screen information is not hidden from the user while typing. If the screen information must be hidden, it is important to test the ability of the MobileApp to allow page flipping by the user without losing typed information [Sch09].

Virtual keyboards are typically smaller than personal computer keyboards, and therefore, it is difficult to type with 10 fingers. Because the keys themselves are smaller and harder to hit and provide no tactile feedback, the MobileApp must be tested to ensure that it allows easy error correction and can manage mistyped words without crashing.

Predictive technologies (i.e., autocompletion of partially typed words) are often used with virtual keyboards to help expedite user input. It is important to test the correctness of the word completions for the natural language chosen by the user, if the MobileApp is designed for a global market. It is also important to test the usability of any mechanism that allows the user to override a suggested completion.

Virtual keyboard testing is often conducted in the usability laboratory, but some should be conducted in the wild. If virtual keyboard tests uncover significant problems, the only alternative may be to ensure that the MobileApp can accept input from devices other than a virtual keyboard (e.g., a physical keyboard or voice input).

21.3.3 Voice Input and Recognition

Voice input has become an increasingly common method for providing input and commands in hands-busy, eyes-busy situations. Voice input may take several forms with different levels of programming complexity required to process each. Voicemail input occurs when a message is simply recorded for playback later. Discrete word recognition can be used to allow users to verbally select items from a menu with a small number of choices. Continuous speech recognition translates dictated speech into meaningful text strings. Each type of voice input has its own testing challenges.

According to Shneiderman [Shn09], all forms of voice input and processing are hindered by interference from noisy environments. Using voice commands to control a device impresses a greater cognitive load on the user, as compared to pointing to a screen object or pressing a key. The user must think of the correct word or words to get the MobileApp to perform the desired action. However, the breadth and accuracy of speech recognition systems are evolving rapidly, and it is likely the voice recognition will become the dominant form of communication in many MobileApps.

Testing the quality and reliability of voice input and recognition should take environmental conditions and individual voice variation into account. Errors will be made by users of the MobileApp and by the portions of the system processing the input. The MobileApp should be tested to ensure that bad input does not crash the MobileApp or the device. Large numbers of users and environments should be involved to be sure the MobileApp is working with an acceptable error rate. It is important to log errors to help developers improve the ability of the MobileApp to process speech input.

21.3.4 Alerts and Extraordinary Conditions

When a MobileApp runs in a real-time environment, there are factors that may impact its behavior. For example, a Wi-Fi signal may be lost, or an incoming text message, phone call, or calendar alert may be received while the user is working with the MobileApp.

These factors can disrupt the MobileApp user's work flow, yet most users opt to allow alerts and other interruptions as they work. A MobileApp test environment must be able to simulate these alerts and conditions. In addition, you should test the MobileApp's ability to handle alerts and conditions in a production environment on actual devices (Section 21.9).

Part of MobileApp testing should focus on the usability issues relating to alerts and pop-up messages. Testing should examine the clarity and context of alerts, the appropriateness of their location on the device display screen, and when foreign languages are involved, verification that the translation from one language to another is correct.

Many alerts and conditions may be triggered differently on various mobile devices or by network or context changes. Although many of the exception-handling processes can be simulated with a software test harness, you should not rely solely on testing in the development environment. This again emphasizes the importance of testing the MobileApp in the wild on actual devices.

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean time to repair (MTTR) is evaluated to determine whether it is within acceptable limits.

21.4 Web Application Testing

Many Web testing practices are also appropriate for testing thin-client MobileApps and interactive simulations. The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for errors.

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by support staff who use regression tests derived from the tests developed when the WebApp was first engineered. Methods for WebApp testing are considered in Section 21.5.

21.5 Web Testing Strategies

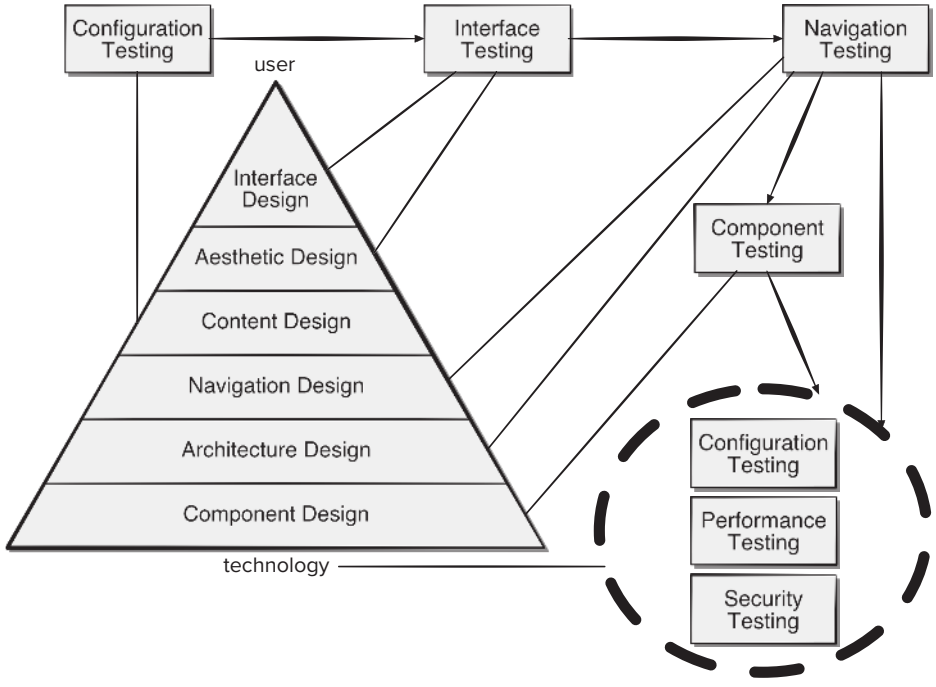
Testing is the process of exercising software with the intent of finding (and ultimately correcting) errors. This fundamental philosophy, first presented in Chapter 20, does not change for WebApps. In fact, because Web-based systems and applications reside on a network and interoperate with many different operating systems, browsers (or other personal communication devices), hardware platforms, communications protocols, and "backroom" applications, the search for errors represents a significant challenge.

Figure 21.2 juxtaposes the mobility testing process with the design pyramid for WebApps (Chapter 13). Note that as the testing flow proceeds from left to right and top to bottom, user-visible elements of the WebApp design (top elements of the pyramid) are tested first, followed by infrastructure design elements.

Because many apps evolve continuously, the testing process is an ongoing activity, conducted by app support staff who use regression tests derived from the tests developed when the app was first engineered.

FIGURE 21.2

The testing process



SAFEHOME



WebApp Testing

The scene: Doug Miller's office.

The players: Doug Miller, manager of the *SafeHome* software engineering group, and Vinod Raman, a member of the product software engineering team.

The conversation:

Doug: What do you think of the **SafeHome Assured.com** e-commerce WebApp V0.0?

Vinod: The outsourcing vendor has done a good job. Sharon [development manager for the vendor] tells me they're testing as we speak.

Doug: I'd like you and the rest of the team to do a little informal testing on the e-commerce site.

Vinod (grimacing): I thought we were going to hire a third-party testing company to validate the WebApp. We're still killing ourselves trying to get the product software out the door.

Doug: We're going to hire a testing vendor for performance and security testing, and our outsourcing vendor is already testing. Just thought another point of view would be helpful, and besides, we'd like to keep costs in line, so . . .

Vinod (sighs): What are you looking for?

Doug: I want to be sure that the interface and all navigation are solid.

Vinod: I suppose we can start with the use cases for each of the major interface functions:

Learn about *SafeHome*.
Specify the *SafeHome* system you need.
Purchase a *SafeHome* system.
Get technical support.

Doug: Good. But take the navigation paths all the way to their conclusion.

Vinod (looking through a notebook of use cases): Yeah, when you select **Specify the SafeHome system you need**, that'll take you to:

Select SafeHome components.
Get *SafeHome* component recommendations.

We can exercise the semantics of each path.

Doug: While you're there, check out the content that appears at each navigation node.

Vinod: Of course . . . and the functional elements as well. Who's testing usability?

Doug: Oh . . . the testing vendor will coordinate usability testing. We've hired a market research firm to line up 20 typical users for the usability study, but if you guys uncover any usability issues . . .

Vinod: I know, pass them along.

Doug: Thanks, Vinod.

21.5.1 Content Testing

Errors in WebApp content can be as trivial as minor typographical mistakes or as significant as incorrect information, improper organization, or violation of intellectual property laws. *Content testing* attempts to uncover these and many other problems before the user encounters them.

Content testing has three important objectives: (1) to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media; (2) to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs; and (3) to find errors in the organization or structure of content that is presented to the end user.

Content testing combines both reviews and the generation of executable test cases. Although technical reviews are not a part of testing, content review should be performed to ensure that content has quality and to uncover semantic errors. Executable testing is used to uncover content errors that can be traced to dynamically derived content that is driven by data acquired from one or more databases.

To accomplish the first objective, automated spelling and grammar checkers may be used. However, many syntactic errors evade detection by such tools and must be discovered by a human reviewer (tester). In fact, a large website might enlist the services of a professional copy editor to uncover typographical errors, grammatical mistakes, errors in content consistency, errors in graphical representations, and cross-referencing errors.

Semantic testing focuses on the information presented within each content object. The reviewer (tester) must answer the following questions:

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?

- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Obtaining answers to each of these questions for a large WebApp (containing hundreds of content objects) can be a daunting task. However, failure to uncover semantic errors will shake the user's faith in the WebApp and can lead to failure of the Web-based application.

21.5.2 Interface Testing

Interface testing exercises interaction mechanisms and validates aesthetic aspects of the user interface. The overall strategy for interface testing is to (1) uncover errors related to specific interface mechanisms (e.g., errors in the proper execution of a menu link or the way data are entered in a form) and (2) uncover errors in the way the interface implements the semantics of navigation, WebApp functionality, or content display. With the exception of WebApp-oriented specifics, the interface strategy noted here is applicable to all types of client-server software. To accomplish this strategy, a number of tactical steps are initiated:

- Interface features are tested to ensure that design rules, aesthetics, and related visual content are available for the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing. For example, tests are designed to exercise all forms, client-side scripting, dynamic HTML, scripts, streaming content, and application-specific interface mechanisms (e.g., a shopping cart for an e-commerce application).
- Each interface mechanism is tested within the context of a use case or network semantic unit (NSU) (Chapter 13) for a specific user category.
- The complete interface is tested against selected use cases and NSUs to uncover errors in the semantics of the interface. It is at this stage that a series of usability tests are conducted.
- The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible.

21.5.3 Navigation Testing

A user travels through a WebApp in much the same way as a visitor walks through a store or museum. There are many pathways to take, stops to make, things to learn and look at, activities to initiate, and decisions to make. This navigation process is predictable in the sense that every visitor has a set of objectives when he arrives. At the same time, the navigation process can be unpredictable because the visitor, influenced by something she sees or learns, may choose a path or initiate an action that is not typical for the original objective. The job of navigation testing is (1) to ensure that the mechanisms that allow the WebApp user to travel through the

WebApp are all functional and (2) to validate that each NSU can be achieved by the appropriate user category.

The first phase of navigation testing actually begins during interface testing. Navigation mechanisms (links and anchors of all types, redirects,[2] bookmarks, frames and frame sets, site maps, and the accuracy of internal search facilities) are tested to ensure that each performs its intended function. Some of the tests noted can be performed by automated tools (e.g., link checking), while others are designed and executed manually. The intent throughout is to ensure that errors in navigation mechanics are found before the WebApp goes online.

Each NSU (Chapter 13) is defined by a set of navigation paths (called "the user journey") that connect navigation nodes (e.g., Web pages, content objects, or functionality). Taken as a whole, each NSU allows a user to achieve specific requirements defined by one or more use cases for a user category. Navigation testing exercises each NSU to ensure that these requirements can be achieved. If NSUs have not been created as part of WebApp analysis or design, you can apply use cases for the design of navigation test cases. You should answer the following questions as each NSU or use case is tested:

- Is the NSU achieved in its entirety without error?
- Is every navigation node (defined for an NSU) reachable within the context of the navigation paths defined for the NSU?
- If the NSU can be achieved using more than one navigation path, has every relevant path been tested?
- If guidance is provided by the user interface to assist in navigation, are directions correct and understandable as navigation proceeds?
- Is there a mechanism (other than the browser "back" arrow) for returning to the preceding navigation node and to the beginning of the navigation path?
- Do mechanisms for navigation within a large navigation node (i.e., a long Web page) work properly?
- If a function is to be executed at a node and the user chooses not to provide input, can the remainder of the NSU be completed?
- If a function is executed at a node and an error in function processing occurs, can the NSU be completed?
- Is there a way to discontinue the navigation before all nodes have been reached, but then return to where the navigation was discontinued and proceed from there?
- Is every node reachable from the site map? Are node names meaningful to end users?
- If a node within an NSU is reached from some external source, is it possible to process to the next node on the navigation path? Is it possible to return to the previous node on the navigation path?
- Does the user understand his location within the content architecture as the NSU is executed?

2 When a server request is forwarded to a nonexistent URL.

Navigation testing, like interface and usability testing, should be conducted by as many different constituencies as possible. You have responsibility for early stages of navigation testing, but later stages should be conducted by other project stakeholders, an independent testing team, and ultimately, by nontechnical users. The intent is to exercise WebApp navigation thoroughly.

21.6 INTERNATIONALIZATION

Internationalization is the process of creating a software product so that it can be used in several countries and with various languages without requiring any engineering changes. *Localization* is the process of adapting a software application for use in targeted global regions by adding locale-specific requirements and translating text elements to appropriate languages. Localization effort may involve taking each country's currency, culture, taxes, and standards (both technical and legal) into account in addition to differences in languages [Sla12]. Launching a MobileApp in many parts of the world without testing it there would be very foolish.

Because it can be very costly to build an in-house testing facility in each country for which localization is planned, outsourcing testing to local vendors in each country is often more cost effective [Reu12]. However, using an outsourcing approach risks a degradation of communication between the MobileApp development team and those who are performing localization tests.

Crowdsourcing has become popular in many online communities.[3] Reuveni [Reu12] suggests that crowdsourcing could be used to engage localization testers dispersed around the globe outside of the development environment. To accomplish this, it is important to find a community that prides itself on its reputation and has a track record of successes. An easy-to-use real-time platform allows community members to communicate with the project decision makers. To protect intellectual property, only trustworthy community members who are willing to sign nondisclosure agreements are allowed to participate.

21.7 SECURITY TESTING

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. Given enough time and resources, thorough security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will

3 *Crowdsourcing* is a distributed problem-solving model where community members work on solutions to problems posted to the group.

be obtained. Security assurance and security engineering are discussed in more detail in Chapter 18.

Mobile security is a complex subject that must be fully understood before effective security testing can be accomplished.[4] MobileApps and the client-side and server-side environments in which they are housed represent an attractive target for external hackers, disgruntled employees, dishonest competitors, and anyone else who wishes to steal sensitive information, maliciously modify content, degrade performance, disable functionality, or embarrass a person, organization, or business.

Security tests are designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment. Each of these domains can be attacked, and it is the job of the security tester to uncover weaknesses that can be exploited by those with the intent to do so.

On the client side, vulnerabilities can often be traced to preexisting bugs in browsers, e-mail programs, or communication software. On the server side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client side or used to disable server operations. In addition, server-side databases can be accessed without authorization (data theft).

To protect against these (and many other) vulnerabilities, firewalls, authentication, encryption, and authorization techniques can be used. Security tests should be designed to probe each of these security technologies in an effort to uncover security holes.

The actual design of security tests requires in-depth knowledge of the inner workings of each security element and a comprehensive understanding of a full range of networking technologies. If the MobileApp or WebApp is business critical, maintains sensitive data, or is a likely target of hackers, it's a good idea to outsource security testing to a vendor who specializes in it.

21.8 PERFORMANCE TESTING

For real-time and embedded systems, software that provides required functionality but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Nothing is more frustrating than a MobileApp that takes minutes to load content when competitive apps download similar content in seconds. Nothing is more aggravating than trying to log on to a WebApp and receiving a "server-busy" message, with the suggestion that you try again later. Nothing is more disconcerting than a MobileApp or WebApp that responds instantly in some situations and then seems to go into an infinite wait state in other situations. All of these occurrences happen on the Web every day, and all of them are performance related.

4 Books by Bell et al. [Bel17], Sullivan and Liu [Sul11], and Cross [Cro07] provide useful information about the subject.

Performance testing is used to uncover performance problems that can result from a lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty or weak operating system capabilities, poorly designed WebApp functionality, and other hardware or software issues that can lead to degraded client-server performance. The intent is twofold: (1) to understand how the system responds as *loading* (i.e., number of users, number of transactions, or overall data volume), and (2) to collect metrics that will lead to design modifications to improve performance.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

Some aspects of MobileApp performance, at least as the end user perceives it, are difficult to test. Network loading, the vagaries of network interfacing hardware, and similar issues are not easily tested at the client or browser level. Mobile performance tests are designed to simulate real-world loading situations. As the number of simultaneous app users grows, or the number of online transactions increases, or the amount of data (downloaded or uploaded) increases, performance testing will help answer the following questions:

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point (in terms of users, transactions, or data loading) does performance become unacceptable?
- What system components are responsible for performance degradation?
- What is the average response time for users under a variety of loading conditions?
- Does performance degradation have an impact on system security?
- Is app reliability or accuracy affected as the load on the system grows?
- What happens when loads that are greater than maximum server capacity are applied?
- Does performance degradation have an impact on company revenues?

To develop answers to these questions, two different performance tests are conducted: (1) *load testing* examines real-world loading at a variety of load levels and in a variety of combinations, and (2) *stress testing* forces loading to be increased to the breaking point to determine how much capacity the app environment can handle.

The intent of load testing is to determine how the WebApp and its server-side environment will respond to various loading conditions. As testing proceeds, permutations to the following variables define a set of test conditions:

N, number of concurrent users

T, number of online transactions per unit of time

D, data load processed by the server per transaction

In every case, these variables are defined within normal operating bounds of the system. As each test condition is run, one or more of the following measures are collected: average user response, average time to download a standardized unit of data, or average time to process a transaction. You should examine these measures to determine whether a precipitous decrease in performance can be traced to a specific combination of *N, T,* and *D*.

Load testing can also be used to assess recommended connection speeds for users of the WebApp. Overall throughput, *P,* is computed in the following manner:

$$P = N \times T \times D$$

As an example, consider a popular sports news site. At a given moment, 20,000 concurrent users submit a request (a transaction, *T*) once every 2 minutes on average. Each transaction requires the WebApp to download a new article that averages 3K bytes in length. Therefore, throughput can be calculated as:

$$P = \frac{20{,}000 \times 0.5 \times 3\text{kb}}{60} = 500 \text{ Kbytes/sec}$$

$$= 4 \text{ megabits per second}$$

The network connection for the server would therefore have to support this data rate and should be tested to ensure that it does.

Stress testing for mobile apps attempts to find errors that will occur under extreme operating conditions. In addition, it provides a mechanism for determining whether the MobileApp will degrade gracefully without compromising security. Among the many actions that might create extreme conditions are: (1) running several mobile apps on the same device, (2) infecting system software with viruses or malware, (3) attempting to take over a device and use it to spread spam, (4) forcing the mobile app to process inordinately large numbers of transactions, and (5) storing inordinately large quantities of data on the device. As these conditions are encountered, the MobileApp is checked to ensure that resource-intensive services (e.g., streaming media) are handled properly.

21.9 REAL-TIME TESTING

The time-dependent, asynchronous nature of many mobile and real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test-case designer have to consider conventional test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error. In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Many MobileApp developers advocate *testing in the wild*, or testing in the users' native environments with the production release versions of the MobileApp resources

TABLE 21.1

Weighted device platform matrix

		OS1	OS2	OS3
	Ranking	3	4	7
Device1	7	N/A	28	49
Device2	3	9	N/A	N/A
Device3	4	12	N/A	N/A
Device4	9	N/A	36	63

[Soa11]. Testing in the wild is designed to be agile and respond to changes as the MobileApp evolves [Ute12].

Some of the characteristics of testing in the wild include adverse and unpredictable environments, outdated browsers and plug-ins, unique hardware, and imperfect connectivity (both Wi-Fi and mobile carrier). To mirror real-world conditions, the demographic characteristics of testers should match those of targeted users, as well as those of their devices. In addition, you should include use cases involving small numbers of users, less-popular browsers, as well as a diverse set of mobile devices. Testing in the wild is always somewhat unpredictable, and test plans must be adapted as testing progresses. For further information, Rooksby and his colleagues have identified themes that are present in successful strategies for testing in the wild [Roo09].

Because MobileApps are often developed for multiple devices and designed to be used in many different contexts and locations, a *weighted device platform matrix* (WDPM) helps ensure that test coverage includes each combination of mobile device and context variables. The WDPM can also be used to help prioritize the device/context combinations so that the most important are tested first.

The steps to build the WDPM (Table 21.1) for several devices and operating systems are: (1) list the important operating system variants as the matrix column labels, (2) list the targeted devices as the matrix row labels, (3) assign a ranking (e.g., 0 to 10) to indicate the relative importance of each operating system and each device, and (4) compute the product of each pair of rankings and enter each product as the cell entry in the matrix (use NA for combinations that are not available).

Testing effort should be adjusted so that the device/platform combinations with the highest ratings receive the most attention for each context variable under consideration.[5] In Table 21.1, **Device4** and **OS3** have the highest rating and would receive high-priority attention during testing.

Actual mobile devices have inherent limitations precipitated by the combination of hardware and firmware delivered in the device. If the range of potential device platforms is large, it is expensive and time consuming to perform MobileApp testing.

5 *Context variables* are variables that are associated with either the current connection or the current transaction that the MobileApp will use to direct its visible-user behavior.

Mobile devices are not designed with testing in mind. Limited processing power and storage capacity may not allow loading of the diagnostic software needed to record the test-case performance. Emulated devices are often easier to manage and allow easier acquisition of test data. Each mobile network (there are hundreds, worldwide) uses its own unique infrastructure to support the mobile Web. Emulators often cannot emulate the effects and timing of network services, and you may not see problems that users will have when the MobileApp runs on an actual device.

Creating test environments in-house is an expensive and error-prone process. Cloud-based testing can offer a standardized infrastructure and preconfigured software images, freeing the MobileApp team from the need to worry about finding servers or purchasing their own licenses for software and testing tools [Goa14]. Cloud service providers give testers access to scalable, ready-to-use virtual laboratories with a library of operating systems, test and execution management tools, and storage necessary for creating a test environment that closely mirrors the real world [Tao17].

Cloud-based testing is not without potential problems: lack of standards, potential security issues, data location and integrity issues, incomplete infrastructure support, improper usages of services, and performance issues are only some of the common challenges that face development teams that use the cloud approach.

Last, it is important to monitor power consumption specifically associated with the use of the MobileApp on a mobile device. Transmitting information from mobile devices consumes more power than monitoring a network for a signal. Processing streaming media consumes more power than loading a Web page or sending a text message. Assessing power consumption accurately must be done in real time on the actual device and in the wild.

21.10 Testing AI Systems

As we discussed in Chapter 13, mobile users expect products like MobileApps, VR systems, and video games to be context aware. Whether the software product is reacting to the user's environment [Abd16], automatically adapting the user interface based on past user behaviors [Par15], or providing a realistic nonplaying character (NPC) in a game situation [Ste16], artificial intelligence (AI) techniques are involved. Often these techniques make use of things like machine learning, data mining, statistics, heuristic programming, or rule-based systems that are outside the scope of this book. There are several problems common to testing these systems that can be addressed with the techniques we have discussed.

AI techniques make use of information that has been obtained from human experts or summarized from large numbers of observations saved in a data store of some kind. These data need to be organized in some way so that they can be accessed and updated efficiently if the software product is to be context aware or self-adaptive. The heuristics for making use of these data to assist decision making in the software are usually described by humans in use cases or formulas obtained from statistical data analysis. Part of what makes these systems hard to test is the large number of data interactions that need to be accounted for by the software, but whose occurrence is hard to predict. Software engineers often need to rely on simulation and model-based techniques to test AI systems.

21.10.1 Static and Dynamic Testing

Static testing is a software verification technique that focuses on review rather than executable testing. It is important to ensure that human experts (stakeholders who understand the application domain) agree with the ways in which the developers have represented the information and its use in the AI system. Like all software verification techniques, it is important to ensure that the program code represents the AI specifications, which means mapping between use case inputs and outputs is reflected in the code.

Dynamic testing for AI systems is a validation technique that exercises the source code with test cases. The intent is to show that the AI system conforms to the behaviors specified by the human experts. In the case of knowledge discovery or data mining, the program may have been designed to discover new relationships unknown to human experts. Human experts must validate these new relationships before they are used in safety-critical software products [Abd16] [Par15].

Many of the real-time testing issues discussed in Section 21.9 apply in dynamic testing of AI systems. Even if automatically generated simulated test cases are used, it is not possible to test every combination of events the software will encounter in the wild. It is often desirable to build in mechanisms to allow the users to specify when they are not happy with the decisions made by the program and collect information on the program state for future corrective action by developers.

21.10.2 Model-Based Testing

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model (in particular the user stories) as the basis for the generation of test cases [DAC03]. In many cases, the model-based testing technique uses formalism like UML state diagrams, an element of the behavioral model (Chapter 8), as the basis for the design of test cases.[6] The MBT technique requires five steps:

1. **Analyze an existing behavioral model for the software, or create one.** Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the steps discussed in Chapter 8: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system (e.g., see Figure 8.8), and (5) review the behavioral model to verify accuracy and consistency.
2. **Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.** The inputs will trigger events that will cause the transition to occur.
3. **Review the behavioral model, and note the expected outputs as the software makes the transition from state to state.** Recall that each state transition is triggered by an event and that as a consequence of the transition

6 Model-based testing can also be used when software requirements are represented with decision tables, grammars, or Markov chains [DAC03].

some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model.

4. **Execute the test cases.** Test cases can be executed manually, or a test script can be created for use by an automated testing tool.
5. **Compare actual and expected results and take corrective action as required.**

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications such as context-aware MobileApps.

21.11 Testing Virtual Environments

It is virtually impossible for a software developer to foresee how the customer will actually use a program. Instructions for use may be misinterpreted; strange combinations of input actions may be used; feedback that seemed clear to the tester may be unintelligible to a user in the field. User experience designers are very aware of the importance of getting feedback from actual users early in the prototyping process to avoid creating software that users dislike.

Acceptance tests are a series of specific tests conducted by the customer in an attempt to uncover product errors before accepting the software from the developer. Conducted by the end user rather than software engineers, acceptance testing can range from an informal "test drive" to a planned and systematically executed series of scripted tests.

When a software product is built for one customer, it is reasonable for that person to conduct a series to validate all requirements. If software is a virtual simulation or game developed as a product to be used by many customers, it is impractical to allow each user to perform formal acceptance tests. Most software product builders use a process called alpha and beta testing to uncover errors that only end users seem able to find.

The *alpha test* is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these at regular intervals. As a result of problems reported during beta tests, the developer makes modifications and then prepares for release of the software product to the entire customer base.

21.11.1 Usability Testing

Usability testing evaluates the degree to which users can interact effectively with the app and the degree to which the app guides users' actions, provides meaningful

feedback, and enforces a consistent interaction approach. Rather than focusing intently on the semantics of some interactive objective, usability reviews and tests are designed to determine the degree to which the app interface makes the user's life easy.[7]

Developers contribute to the design of usability tests, but in general the tests are conducted by end users. Usability testing can occur at a variety of different levels of abstraction: (1) the usability of a specific interface mechanism (e.g., a form) can be assessed, (2) the usability of a complete virtual interface (encompassing interface mechanisms, data objects, and related functions) can be evaluated, or (3) the usability of the complete virtual-world application can be considered.

The first step in usability testing is to identify a set of usability categories and establish testing objectives for each category. The following test categories and objectives (written in the form of a question) illustrate this approach:[8]

Interactivity. Are interaction mechanisms (e.g., pull-down menus, buttons, widgets, inputs) easy to understand and use?

Layout. Are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?

Readability. Is text well written and understandable?[9] Are graphic representations easy to understand?

Aesthetics. Do layout, color, typeface, and related characteristics lead to ease of use? Do users "feel comfortable" with the look and feel of the app?

Display characteristics. Does the app make optimal use of screen size and resolution?

Time sensitivity. Can important features, functions, and content be used or acquired in a timely manner?

Feedback. Do users receive meaningful feedback to their actions? Is the user's work interruptible and recoverable when a system message is displayed?

Personalization. Does the app tailor itself to the specific needs of different user categories or individual users?

Help. Is it easy for users to access help and other support options?

Accessibility. Is the app accessible to people who have disabilities?

Trustworthiness. Are users able to control how personal information is shared? Does the app make use of personal information without user permission?

A series of tests is designed within each of these categories. In some cases, the "test" may be a visual review of the app screen displays. In other cases interface semantics tests may be executed again, but in this instance usability concerns are paramount.

---

7 The term *user-friendliness* has been used in this context. The problem, of course, is that one user's perception of a "friendly" interface may be radically different from another's.

8 For additional information on usability, see Chapter 12.

9 The FOG Readability Index and others may be used to provide a quantitative assessment of readability.

FIGURE 21.3
Qualitative assessment of usability



As an example, we consider usability assessment for interaction and interface mechanisms. The following is a list of interface features that might be reviewed and tested for usability: animations, buttons, color, control, graphics, labels, menus, messages, navigation, selection mechanisms, text, and HUDs[10] (heads-up user displays). As each feature is assessed, it is graded on a qualitative scale by the users who are doing the testing. Figure 21.3 shows a possible set of assessment "grades" that can be selected by users. These grades are applied to each feature individually, to a complete app screen display, or to the app as a whole.

21.11.2 Accessibility Testing

Accessibility testing is the process of verifying the degree to which all people can use a computer system regardless of any user's special need. The special needs most commonly considered for computer system accessibility are: visual, hearing, movement, and cognitive impairments [Zan18]. Many of these special needs evolve as people get older. As a profession, virtual environment development has not done a good job of providing access systems with rich graphical interfaces that rely heavily on touch interactions [Dia14]. The problems merely shift with a switch to voice-activated

10 Mobile apps and games frequently provide graphical displays containing user status, system messages, navigation date, and menu choices as part of the device screen display or HUD.

personal assistants like Alexa® or Siri®. Just imagine trying to operate your smartphone without using all these senses: sight, hearing, touch, or speech.

We discussed guidelines[11] for designing accessible software products in Chapter 13. An effective design strategy should ensure that all important interactions with the user be presented using more than one information channel. A few examples of the areas of focus for accessibility testing follow [Zan18] [Dia14]:

- Ensure that all nontext screen objects are also represented by a text-based description.
- Verify that color is not used exclusively to convey information to the user.
- Demonstrate that high contrast and magnification options are available for elderly or visually challenged users.
- Ensure that speech input alternatives have been implemented to accommodate users that may not be able to manipulate a keyboard, keypad, or mouse.
- Demonstrate that blinking, scrolling, or auto content updating is avoided to accommodate users with reading difficulties.

It is likely that mobile, cloud-based software will come to dominate many things that users need to accomplish on a day-to-day basis (e.g., banking, tax preparation, restaurant reservations, trip planning) and as a consequence, the need for accessible software products will only grow. Along with expert review and automated tools to assess accessibility, a thorough accessibility testing strategy will help to ensure that every user, no matter their challenges, will be accommodated.

21.11.3 Playability Testing

Playability is the degree to which a game or simulation is fun to play and usable by the user/player and was originally conceived as part of the development of video games. Game playability is affected by the quality of the game: usability, storyline, strategy, mechanics, realism, graphics, and sound. With the advent of virtual/augmented reality simulations whose intention is to provide entertainment or learning opportunities (e.g., such as simulated troubleshooting), it makes sense to use playability testing as part of the usability testing for a virtual environment created by MobileApp [Vel16].

Expert review can be used as part of the playability testing, but unless expert users are your target user group you may not get the feedback you need to have your MobileApp succeed in the marketplace. Expert review should be supplemented by playability tests conducted by representative end users, as you might do for a beta or acceptance test. In a typical play test, the user might be given general instructions on using the app and the developers would then step back and observe players use of the game without interruption. The players may be asked to complete a survey on their experience once they are done with the play test [Hus15].

The developers might record the play session or simply take notes on what they observe. The developers are looking for places in the play session where the player does

11 Here is an example of a software accessibility checklist used by the United States Department of Justice: https://www.justice.gov/crt/software-accessibility-checklist.

not know what to do next (this is usually marked by a sudden halt in the player's actions). Developers should note where the player is in the app work flow when this event happens. When the play test has ended, the developers might discuss why the player got stuck and how the player got herself unstuck (it she did). This suggests that playability testing might be helpful in assessing the accessibility of a virtual environment as well.

21.12 Testing Documentation and Help Facilities

The term *usability testing* conjures up images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. But errors in help facilities or online program documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that documentation testing should be an important part of every software test plan.

Documentation testing can be approached in two phases. The first phase, technical review (Chapter 16), examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

INFO



Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document's table of contents and index robust, accurate, and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted, and areas of document ambiguity or weakness are defined for potential rewrite.

21.13 SUMMARY

The goal of MobileApp testing is to exercise each of the many dimensions of software quality for mobile applications with the intent of finding errors or uncovering issues that may lead to quality failures. Testing focuses on the quality elements such as content, function, structure, usability, use of context, navigability, performance, power management, compatibility, interoperability, capacity, and security. It incorporates reviews and usability assessments that occur as the MobileApp is designed, and tests that are conducted once the MobileApp has been implemented and deployed on an actual device.

The MobileApp testing strategy exercises each quality dimension by initially examining "units" of content, functionality, or navigation. Once individual units have been validated, the focus shifts to tests that exercise the MobileApp as a whole. To accomplish this, many tests are derived from the user's perspective and are driven by information contained in use cases. A MobileApp test plan is developed and identifies testing steps, work products (e.g., test cases), and mechanisms for the evaluation of test results. The testing process encompasses several different types of testing.

Content testing (and reviews) focus on various categories of content. The intent is to examine errors that affect the presentation of the content to the end user. The content needs to be examined for performance issues imposed by the mobile device constraints. Interface testing exercises the interaction mechanisms that define the user experience provided by the MobileApp. The intent is to uncover errors that result when the MobileApp does not take device, user, or location context into account.

Navigation testing is based on use cases, derived as part of the modeling activity. The test cases are designed to exercise each usage scenario against the navigation design within the architectural framework used to deploy the MobileApp. Component testing exercises content and functional units within the MobileApp.

Performance testing encompasses a series of tests that are designed to assess MobileApp response time and reliability as demands on server-side resource capacity increase.

Security testing incorporates a series of tests designed to exploit vulnerabilities in the MobileApp or its environment. The intent is to find security holes in either the device operating environment or the Web services being accessed.

Finally, MobileApp testing should address performance issues such as power usage, processing speed, memory limitations, ability to recover from failures, and connectivity issues.

Navigation testing applies use cases, derived as part of the modeling activity, in the design of test cases that exercise each usage scenario against the navigation design. Navigation mechanisms are tested to ensure that any errors impeding completion of a use case are identified and corrected. Component testing exercises content and functional units within the MobileApp.

PROBLEMS AND POINTS TO PONDER

21.1. Are there any situations in which MobileApp testing on actual devices can be disregarded?

21.2. Is it fair to say that the overall mobility testing strategy begins with user-visible elements and moves toward technology elements? Are there exceptions to this strategy?

21.3. Describe the steps associated with user experience testing for an app.

21.4. What is the objective of security testing? Who performs this testing activity?

21.5. Assume that you are developing a MobileApp to access an online pharmacy (`YourCornerPharmacy.com`) that caters to senior citizens. The pharmacy provides typical functions but also maintains a database for each customer so that it can provide drug information and warn of potential drug interactions. Discuss any special usability or accessibility tests for this MobileApp.

21.6. Assume that you have implemented a Web service that provides a drug interaction–checking function for `YourCornerPharmacy.com` (see Problem 21.5). Discuss the types of component-level tests that would have to be conducted on the mobile device to ensure that the MobileApp accesses this function properly.

21.7. Is it possible to test every configuration that a MobileApp is likely to encounter in the production environment? If it is not, how do you select a meaningful set of configuration tests?

21.8. Describe a security test that might need to be conducted for the YourCornerPharmacy MobileApp (Problem 21.5). Who should perform this test?

21.9. What is the difference between testing that is associated with interface mechanisms and testing that addresses interface semantics?

21.10. What is the difference between testing for navigation syntax and navigation semantics?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

22 SOFTWARE CONFIGURATION MANAGEMENT

Change is inevitable when computer software is built and can lead to confusion when you and other members of a software team are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error. Babich [Bab86] suggests an approach that will minimize confusion, improve productivity, and reduce the number of mistakes when he writes: "Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes."

KEY CONCEPTS

baselines 441
change control 448
change management, mobility and agile . . . 453
change management process 447
configuration audit 452
configuration management, elements of . . . 440
configuration objects 441
content management 455
continuous integration 446
identification 452
integration and publishing 455
repository. 453
SCM process 448
software configuration items 438
status reporting. 452
version control. 445



QUICK LOOK

What is it? When you build computer software, change happens. And because it happens, you need to manage it effectively. Software configuration management (SCM), also called change management, is a set of activities designed to manage change.

Who does it? Everyone involved in the software process is involved with change management to some extent, but specialized support positions are sometimes created to manage the SCM process.

Why is it important? If you don't control change, it controls you. And that's never good. It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. As a consequence, software quality suffers and delivery is delayed.

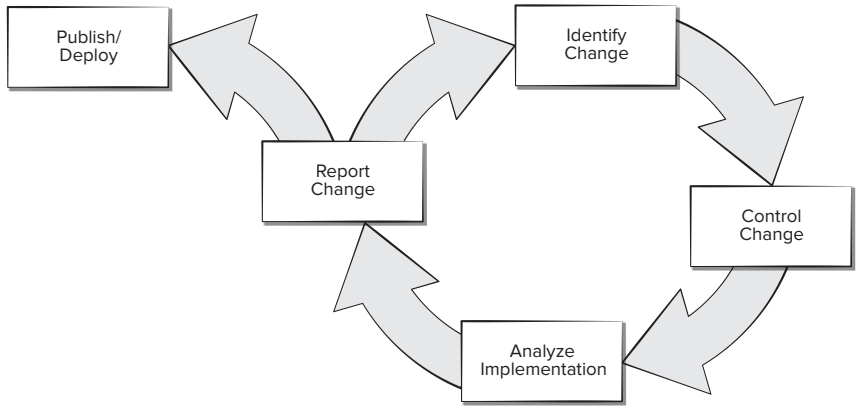
What are the steps? Because many work products are produced when software is built, each must be uniquely identified. Once this is accomplished, mechanisms for version and change control can be established.

What is the work product? A software configuration management plan defines the project strategy for change management. Changes result in updated software products that must be retested and documented, without breaking the project schedule or the production versions of the software products.

How do I ensure that I've done it right? When every work product can be accounted for, traced, controlled, tracked, and analyzed; when everyone who needs to know about a change has been informed—you've done it right.

FIGURE 22.1

Software configuration management work flow



Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Typical SCM work flow is shown in Figure 22.1. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support (Chapter 27) is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that are initiated when a software engineering project begins and terminates only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter, we discuss the specific activities that enable you to manage change.

22.1 SOFTWARE CONFIGURATION MANAGEMENT

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms), (2) work products that describe the computer programs (targeted at various stakeholders), and (3) data or content (contained within the program or external to it). In Web design or game development, managing changes to the multimedia content items can be more demanding than managing the changes to the software or documentation. The items that comprise all information produced as part of the software process are collectively called a *software configuration*.

As software engineering work progresses, a hierarchy of *software configuration items* (SCIs)—a named element of information that can be as small as a single UML diagram or as large as the complete design document—is created. If each SCI simply led to other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change*. Change may occur at any time, for any reason. In fact, the *first law of system*

engineering [Ber80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth or downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we describe major SCM tasks and important concepts that help us to manage change.

22.1.1 An SCM Scenario

This section is extracted from [Dar01].[1]

> A typical configuration management (CM) operational scenario involves several stakeholders: a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product. In the scenario, assume that the product is a small one involving about 15,000 lines of code being developed by an agile team with four developers. (Note that other scenarios of smaller or larger teams are possible, but, in essence, there are generic issues that each of these projects face concerning CM.)

At the operational level, the scenario involves various roles and tasks. For the project manager or team leader, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

The goals of the configuration manager (who on a small team may be the project manager) are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating proposed changes with the development team, and ensuring the changes are acceptable to the product owner. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

---

1 Special permission to reproduce "Spectrum of Functionality in CM Systems" by Susan Dart [Dar01], © 2001 by Carnegie Mellon University is granted by the Software Engineering Institute.

For the software engineers, the goal is to work effectively. There must be a mechanism to ensure that simultaneous changes to the same component are properly tracked, managed, and executed. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting work products. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying one another about tasks required and tasks completed. Changes are propagated across each other's work by merging files. Mechanisms exist to ensure that, for components that undergo simultaneous changes, there is some way of resolving conflicts and merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed. The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

The customer uses the product. Because the product is under CM control, the customer follows formal procedures for requesting changes and for indicating bugs in the product.

Ideally, a CM system used in this scenario should support all these roles and tasks; that is, the roles determine the functionality required of a CM system. The project manager sees CM as an auditing mechanism; the configuration manager sees it as a controlling, tracking, and policy-making mechanism; the software engineer sees it as a changing, building, and access control mechanism; and the customer sees it as a quality assurance mechanism.

22.1.2 Elements of a Configuration Management System

In her comprehensive white paper on software configuration management, Susan Dart [Dar01] identifies four important elements that should exist when a configuration management system is developed:

- **Component elements.** A set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- **Process elements.** A collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- **Construction elements.** A set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- **Human elements.** A set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

These elements (to be discussed in more detail in later sections) are not mutually exclusive. For example, component elements work in conjunction with construction elements as the software process evolves. Process elements guide many human activities that are related to SCM and might therefore be considered human elements as well.

22.1.3 Baselines

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, and how to get it done and still make money). Most software changes are justified, so there's no point in complaining about them. Rather, be certain that you have mechanisms in place to handle them.

A *baseline* is a software configuration management concept that helps you to control change without seriously impeding justifiable change. The IEEE [IEE17] defines a baseline as:

> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review (Chapter 16). For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 22.2.

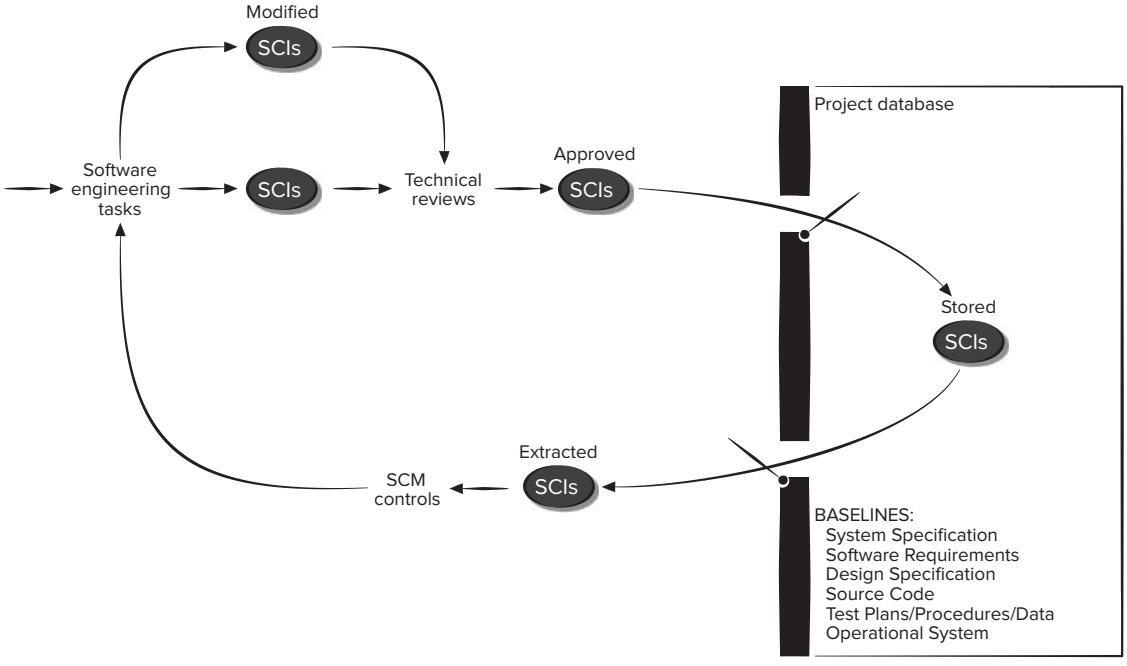
The progression of events that lead to a baseline is also illustrated in Figure 22.2. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository* and discussed in Section 22.5). Be sure that the project database is maintained in a centralized, controlled location. When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private workspace. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 22.2 illustrate the modification path for a baselined SCI.

22.1.4 Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, an SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, a named program component, a multimedia content asset, or a software tool).

In reality, SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A *configuration object* has a

FIGURE 22.2 Baselined SCIs and the project database



name, attributes, and is "connected" to other objects by relationships. Referring to Figure 22.3, the configuration objects, **DesignSpecification, DataModel, ComponentN, SourceCode,** and **TestSpecification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **DataModel** and **ComponentN** are part of the object **DesignSpecification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **SourceCode** object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.[2]

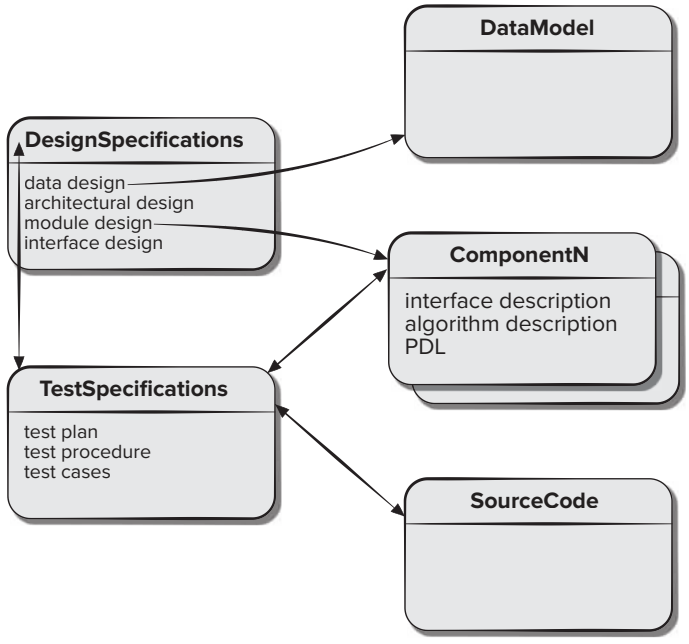
22.1.5 Management of Dependencies and Changes

We introduced the concept of traceability and the use of traceability matrices in Section 7.2.6. The traceability matrix is one way to document dependencies among requirements, architectural decisions (Section 10.5), and defect causes (Section 17.6). These dependencies need to be considered when determining the impact of a proposed change and guiding the selection test cases that should be used for regression testing (Section 20.3). de Sousa and Redmiles write that viewing dependency management as impact management[3] helps developers to focus on how changes made affect their work [Sou08].

2 These relationships are defined within the database. The structure of the database (repository) is discussed in greater detail in Section 22.2.

3 Impact management is discussed further in Section 22.5.2.

FIGURE 22.3
Configuration objects



Impact analysis focuses on organizational behavior as well as individual actions. Impact management involves two complementary aspects: (1) ensuring that software developers employ strategies to minimize the impact of their colleagues' actions on their own work, and (2) encouraging software developers to use practices that minimize the impact of their own work on that of their colleagues. It is important to note that when a developer tries to minimize the impact of her work on others, she is also reducing the work others need to do to minimize the impact of her work on theirs [Sou08].

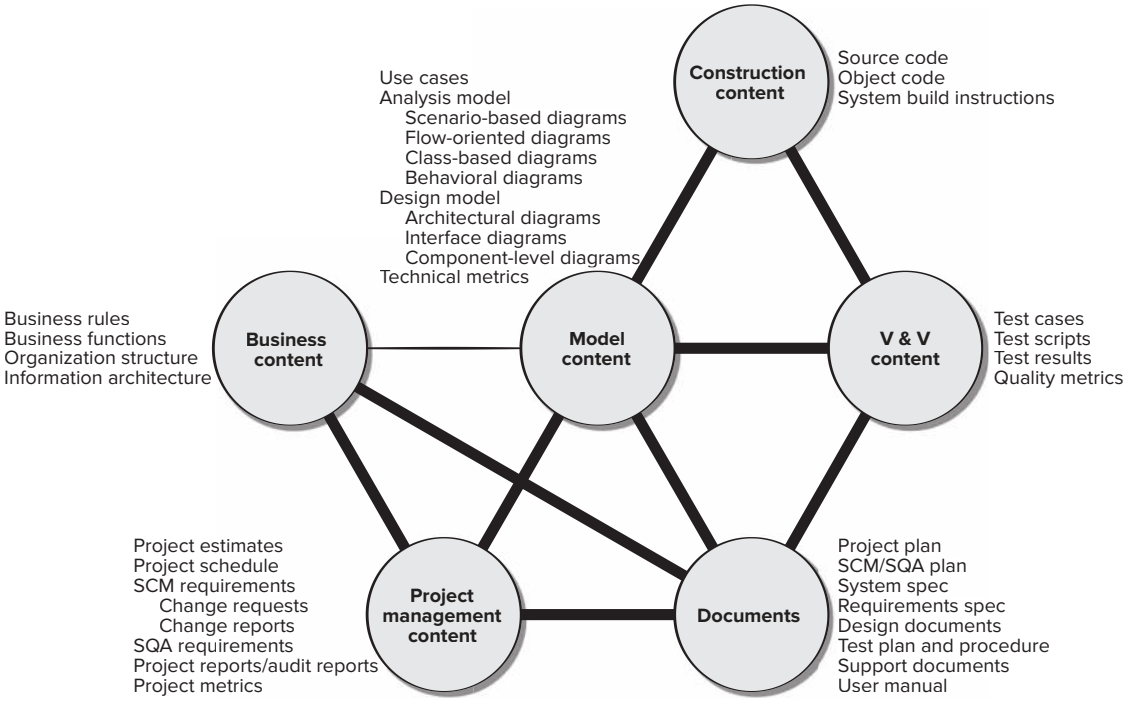
It is important to maintain software work products to ensure that developers are aware of the dependencies among the SCIs. Developers must establish discipline when checking items in and out of the SCM repository and when making approved changes, as discussed in Section 22.2.

22.2 THE SCM REPOSITORY

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.

To achieve these capabilities, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data

FIGURE 22.4 Content of the repository



can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs.

22.2.1 General Features and Content

The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. A detailed breakdown of types of representations, documents, and other work products that are stored in the repository is presented in Figure 22.4.

A robust repository provides two different classes of services: (1) the same types of services that might be expected from any sophisticated database management system and (2) services that are specific to the software engineering environment.

A repository that serves a software engineering team should also (1) integrate with or directly support process management functions, (2) support specific rules that govern the SCM function and the data maintained within the repository, (3) provide an interface to other software engineering tools, and (4) accommodate storage of sophisticated data objects (e.g., text, graphics, video, audio).

22.2.2 SCM Features

To support SCM, the repository must be capable of maintaining SCIs related to many different versions of the software. More important, it must provide the mechanisms

for assembling these SCIs into a version-specific configuration. The repository tool set needs to provide support for the following features.

Versioning. As a project progresses, many versions (Section 22.5.2) of individual work products will be created. The repository must be able to save all these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.

The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects such as screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

Dependency Tracking and Change Management. The repository manages a wide variety of relationships among the data elements stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on. Some of these relationships are merely associations, and some are dependencies or mandatory relationships.

The ability to keep track of all these relationships is crucial to the integrity of the information stored in the repository and to the generation of deliverables based on it, and it is one of the most important contributions of the repository concept to the improvement of the software development process. For example, if a UML class diagram is modified, the repository can detect whether related classes, interface descriptions, and code components also require modification and can bring affected SCIs to the developer's attention.

Requirements Tracing. This special function depends on link management and provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, it provides the ability to identify which requirement generated any given work product (backward tracing).

Configuration Management. A configuration management facility keeps track of a series of configurations representing specific project milestones or production releases.

Audit Trails. An audit trail establishes additional information about when, why, and by whom changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository. A repository trigger mechanism is helpful for prompting the developer or the tool that is being used to initiate entry of audit information (such as the reason for a change) whenever a design element is modified.

22.3 VERSION CONTROL SYSTEMS

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities: (1) a project

database (*repository*) that stores all relevant configuration objects, (2) a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions), (3) a *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software. In addition, version control and change control systems often implement (4) an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

A number of version control systems establish a *change set*—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software. Dart [Dar91] notes that a change set "captures all changes to all files in the configuration along with the reason for changes and details of who made the changes and when."

A number of named change sets can be identified for an application or system. This enables you to construct a version of the software by specifying the change sets (by name) that must be applied to the baseline configuration. To accomplish this, a *system modeling* approach is applied. The system model contains: (1) a *template* that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed, (2) construction rules, and (3) verification rules.[4]

A number of different automated approaches to version control have been proposed over the years.[5] The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

22.4 CONTINUOUS INTEGRATION

Best practices for SCM include: (1) keeping the number of code variants small, (2) test early and often, (3) integrate early and often, and (4) tool use to automate testing, building, and code integration. *Continuous integration* (CI) is important to agile developers following the DevOps workflow (Section 3.5.3). CI also adds value to SCM by ensuring that each change is promptly integrated into the project source code, compiled, and tested automatically. CI offers development teams several concrete advantages [Mol12]:

Accelerated feedback. Notifying developers immediately when integration fails allows fixes to be made while the number of performed changes is small.

Increased quality. Building and integrating software whenever necessary provides confidence into the quality of the developed product.

4 It is also possible to query the system model to assess how a change in one component will impact other components.

5 Github (https://github.com/), Perforce (https://www.perforce.com/), and Apache Subversion also known as SVN (http://subversion.apache.org/) are popular version control systems.

Reduced risk. Integrating components early avoids risking a long integration phase because design failures are discovered and fixed early.

Improved reporting. Providing additional information (e.g., code analysis metrics) allows for more accurate configuration status accounting.

CI is becoming a key technology as software organizations begin their shift to more agile software development processes. CI is best done using specialized tools.[6] CI allows project managers, quality assurance managers, and software engineers to improve software quality by reducing the likelihood of defects escaping outside the development team. Early defect capture always reduces the development costs by allowing cheaper fixes earlier in the software project time line.

22.5 THE CHANGE MANAGEMENT PROCESS

The software change management process defines a series of tasks that have four primary objectives: (1) to identify all items that collectively define the software configuration, (2) to manage changes to one or more of these items, (3) to facilitate the construction of different versions of an application, and (4) to ensure that software quality is maintained as the configuration evolves over time.

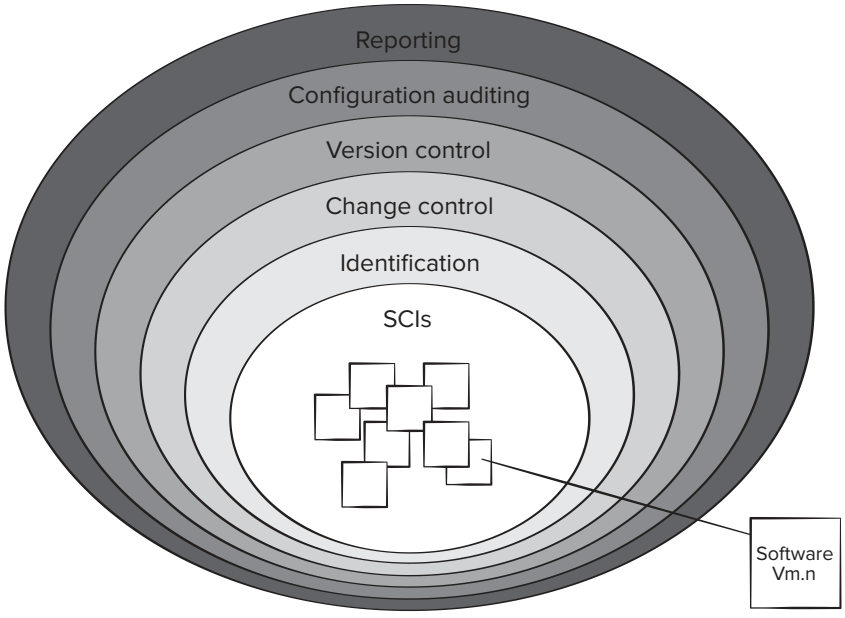
A process that achieves these objectives need not be bureaucratic and ponderous, but it must be characterized in a manner that enables a software team to develop answers to a set of complex questions:

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- How does an organization assess the impact of change and manage the impact effectively?
- Who has responsibility for approving and ranking requested changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead to the definition of five SCM tasks—identification, version control, change control, configuration auditing, and reporting—illustrated in Figure 22.5.

6 Puppet (https://puppet.com/), Jenkins (https://jenkins.io/), and Hudson (http://hudson-ci.org/) are examples of CI tools. Travis-CI (https://travis-ci.org/) is a CI tool designed for sync projects residing on Github.

FIGURE 22.5
Layers of the SCM process



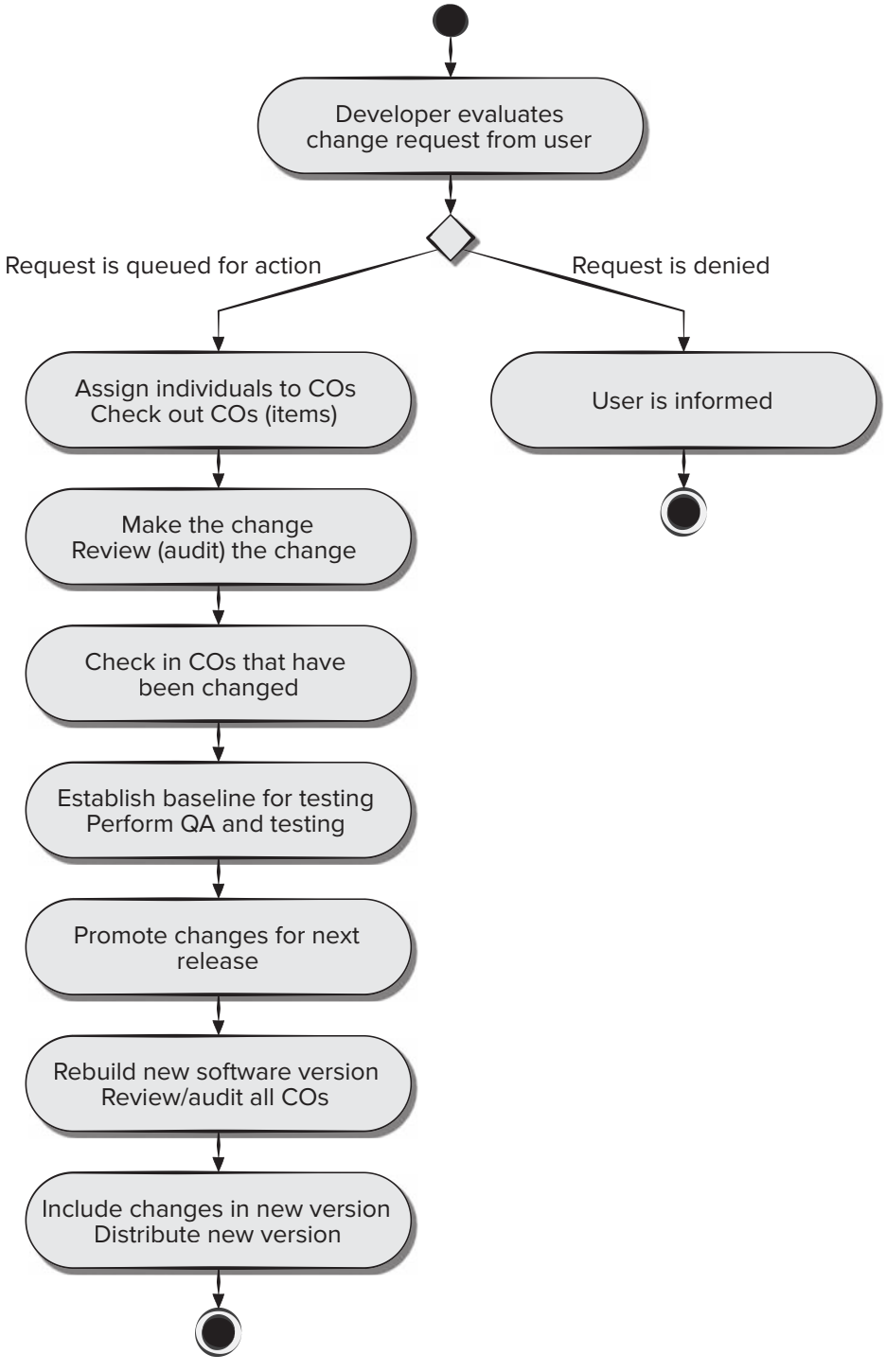
Referring to the figure, SCM tasks can be viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system. As an SCI moves through a layer, the actions implied by each SCM task may or may not be applicable. For example, when a new SCI is created, it must be identified. However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play). A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know. In the sections that follow, we examine each of these SCM process layers in more detail.

22.5.1 Change Control

For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 22.6. A *change request* is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group that makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit.

FIGURE 22.6

The change control process



The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made. As an alternative, the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied. The object(s) is (are) then "checked in" to the database, and appropriate version control mechanisms (Section 22.3) are used to create the next version of the software.

These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

You may feel uncomfortable with the level of bureaucracy implied by the change control process description shown in Figure 22.6. This feeling is not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately, many have none) have created a number of layers of control to help avoid the problems alluded to here.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone technical review and has been approved, a baseline can be created.[7] Once an SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, the developer dispenses with the formal generation of change requests, change reports, and ECOs. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 22.6.

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify customers' perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

7 A baseline can be created for other reasons as well. For example, when "daily builds" are created, all components checked in by a given time become the baseline for the next day's work.

SAFEHOME



SCM Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I know it's early, but we've got to talk about change management.

Vinod (laughing): Hardly. Marketing called this morning with a few "second thoughts." Nothing major, but it's just the beginning.

Jamie: We've been pretty informal about change management on past projects.

Doug: I know, but this is bigger and more visible, and as I recall . . .

Vinod (nodding): We got killed by uncontrolled changes on the home lighting control project . . . remember the delays that . . .

Doug (frowning): A nightmare that I'd prefer not to relive.

Jamie: So what do we do?

Doug: As I see it, three things. First we have to develop—or borrow—a change control process.

Jamie: You mean how people request changes?

Vinod: Yeah, but also how we evaluate the change, decide when to do it (if that's what we decide), and how we keep records of what's affected by the change.

Doug: Second, we've got to get a really good SCM tool for change and version control.

Jamie: We can build a database for all of our work products.

Vinod: They're called SCIs in this context, and most good tools provide some support for that.

Doug: That's a good start, now we have to . . .

Jamie: Uh, Doug, you said there were three things . . .

Doug (smiling): Third—we've all got to commit to follow the change management process and use the tools—no matter what, okay?

22.5.2 Impact Management

A web of software work product interdependencies must be considered every time a change is made. Impact management encompasses the work required to properly understand these interdependencies and control their effects on other SCIs (and the people who are responsible for them).

Impact management is accomplished with three actions [Sou08]. First, an impact network identifies the members of a software team (and other stakeholders) who might effect or be affected by changes that are made to the software. A clear definition of the software architecture (Chapter 10) assists greatly in the creation of an impact network. Next, forward impact management assesses the impact of your own changes on the members of the impact network and then informs members of the impact of those changes. Finally, backward impact management examines changes that are made by other team members and their impact on your work and incorporates mechanisms to mitigate the impact.

22.5.3 Configuration Audit

Identification, version control, and change control help you to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can a software team ensure that the change has been properly implemented? The answer is twofold: (1) technical reviews and (2) the software configuration audit.

The technical review (Chapter 16) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A technical review should be conducted for all but the most trivial changes.

A *software configuration audit* complements the technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a technical review been conducted to assess technical correctness?
3. Has the software process been followed, and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a technical review. However, when SCM is a formal activity, the configuration audit is conducted separately by the quality assurance group. Such formal configuration audits also ensure that the correct SCIs (by version) have been incorporated into a specific build and that all documentation is up to date and consistent with the version that has been built.

22.5.4 Status Reporting

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 22.6. At the very least, develop a "need to know" list for every configuration object and keep it up to date. When a change is made, be sure that everyone on the list is notified. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an online database or website, so that software developers or support staff can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners apprised of important changes.

22.6 MOBILITY AND AGILE CHANGE MANAGEMENT

Earlier in this book, we discussed the special nature of WebApps and MobileApps and the specialized methods[8] that are required to build them. Game developers face similar challenges, as do all agile development teams. Among the many characteristics that differentiate these applications from traditional software is the ubiquitous nature of change.

Mobile developers and game developers often use an iterative, incremental process model that applies many principles derived from agile software development (Chapter 4). Using this approach, an engineering team often develops an increment in a very short time period using a customer-driven approach. Subsequent increments add additional content and functionality, and each is likely to implement changes that lead to enhanced content, better usability, improved aesthetics, better navigation, enhanced performance, and stronger security. Therefore, in the agile world of app and game development, change is viewed somewhat differently.

If you're a member of a software team that builds apps or games, you must embrace change. And yet, a typical agile team eschews all things that appear to be process-heavy, bureaucratic, and formal. Software configuration management is often viewed (albeit incorrectly) to have these characteristics. This seeming contradiction is remedied not by rejecting SCM principles, practices, and tools, but rather by molding them to meet the special needs of mobile projects.

22.6.1 e-Change Control

The work flow associated with change control for conventional software (Section 22.5.1) is generally too ponderous for WebApp and mobile software development. It is unlikely that the change request, change report, and engineering change order sequence can be achieved in an agile manner that is acceptable for many game and app development projects. How then do we manage a continuous stream of changes requested for content and functionality?

To implement effective change management within the "code and go" philosophy that continues to dominate much of game and mobile development, the conventional change control process can be modified. Each change should be categorized into one of four classes:

Class 1. A content or function change that corrects an error or enhances local content or functionality.

Class 2. A content or function change that has an impact on other content objects or functional components.

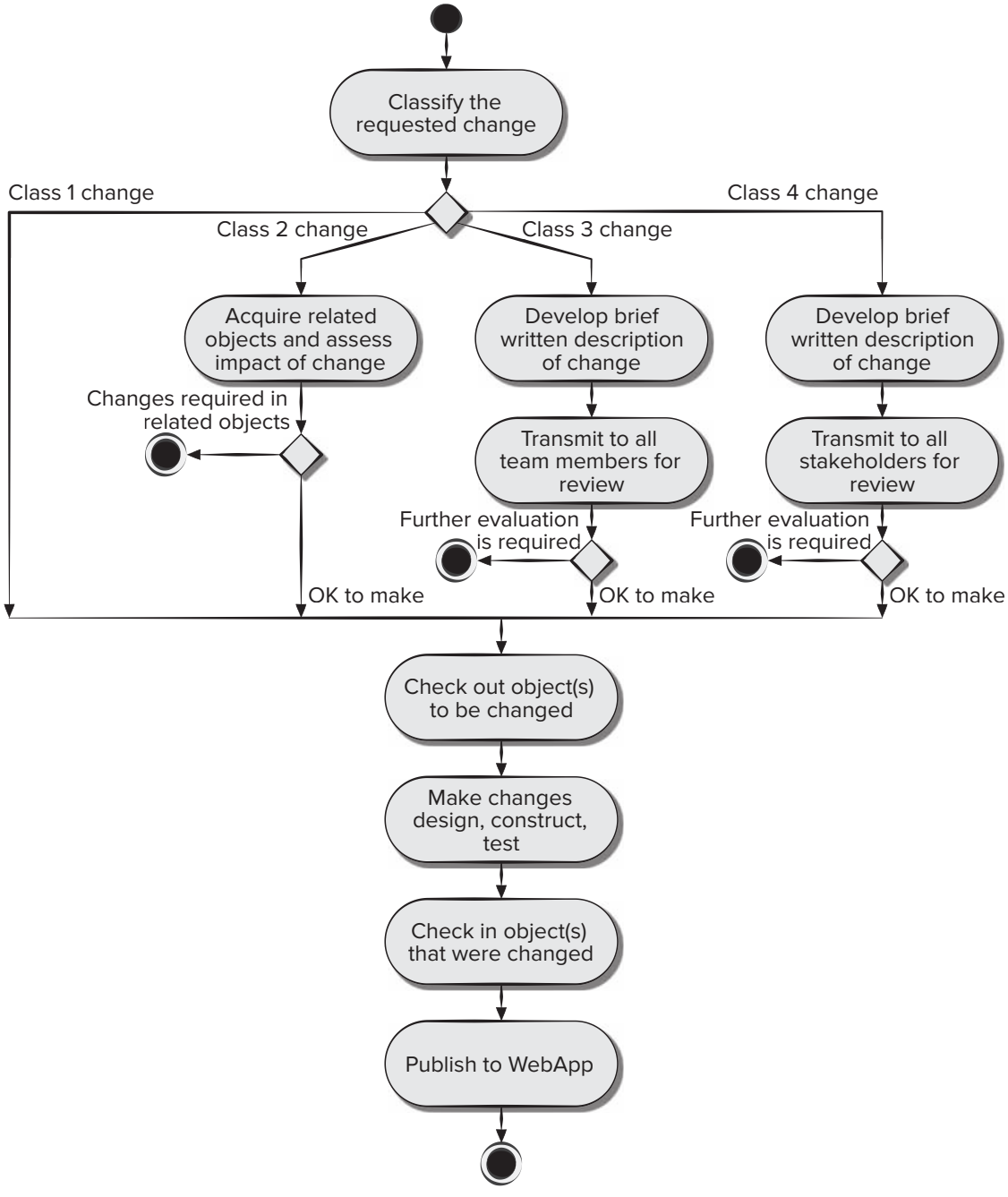
Class 3. A content or function change that has a broad impact across an app (e.g., major extension of functionality, significant enhancement or reduction in content, major required changes in navigation).

Class 4. A major design change (e.g., a change in interface design or navigation approach) that will be immediately noticeable to one or more categories of user.

Once the requested change has been categorized, it can be processed according to the algorithm shown in Figure 22.7 for WebApps but is equally applicable for apps and games.

8 See [Pre08] for a comprehensive discussion of Web engineering methods.

FIGURE 22.7 Managing changes for WebApps



Referring to the figure, class 1 and 2 changes are treated informally and are handled in an agile manner. For a class 1 change, you would evaluate the impact of the change, but no external review or documentation is required. As the change is made, standard check-in and check-out procedures are enforced by configuration repository tools. For class 2 changes, you should review the impact of the change on related objects (or ask other developers responsible for those objects to do so). If the change can be made without requiring significant changes to other objects, modification occurs without additional review or documentation. If substantive changes are required, further evaluation and planning are necessary.

Class 3 and 4 changes are also treated in an agile manner, but some descriptive documentation and more formal review procedures are required. A *change description*—describing the change and providing a brief assessment of the impact of the change—is developed for class 3 changes. The description is distributed to all members of the team who review it to better assess its impact. A change description is also developed for class 4 changes, but in this case all stakeholders conduct the review.

22.6.2 Content Management

Content management is related to configuration management in the sense that a content management system (CMS) establishes a process (supported by appropriate tools) that acquires existing content (from a broad array of app and/or game configuration objects), structures it in a way that enables it to be presented to an end user, and then provides it to the client-side environment for display.

The most common use of a content management system occurs when a dynamic application is built. Apps and games create screen displays "on the fly." That is, the user typically performs an action that the software responds to by changing the information displayed on the screen. The user action may cause the app to query a server-side database; it then formats the information accordingly and presents it to the user.

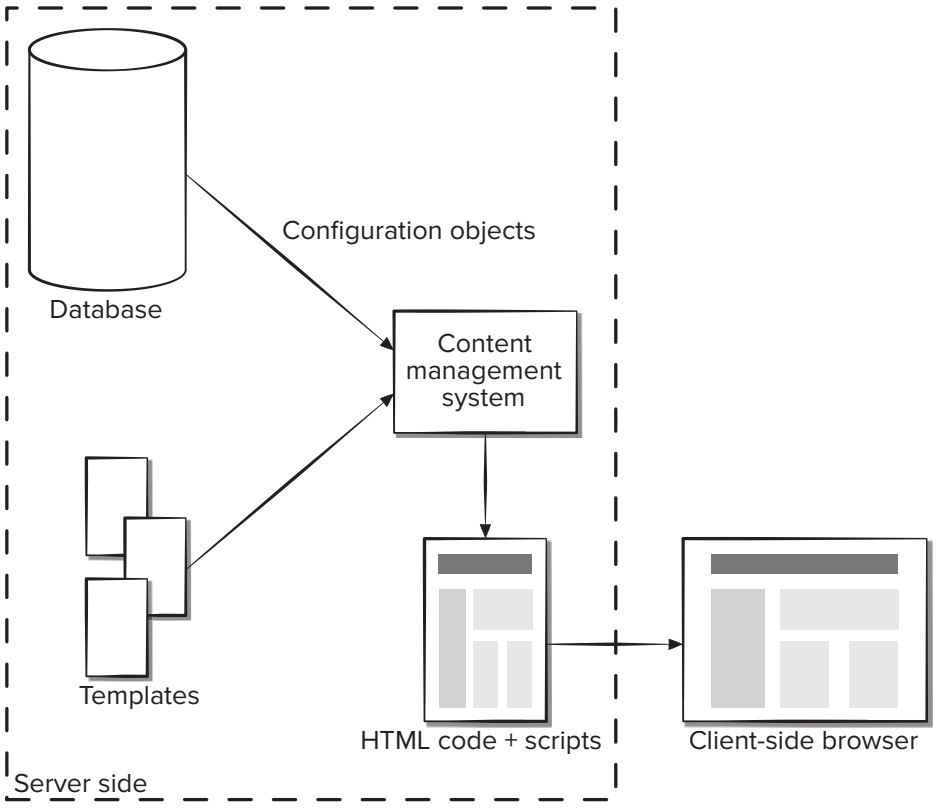
For example, a music store (e.g., Apple iTunes) provides hundreds of thousands of tracks for sale. When a user requests a music track, a database is queried and a variety of information about the artist, the CD (e.g., its cover image or graphics), the musical content, and sample audio are all downloaded and configured into a standard content template. The resultant page is built on the server side and passed to the client side for examination by the end user. A generic representation for WebApps is shown in Figure 22.8.

22.6.3 Integration and Publishing

Content management systems are useful for composing Web services to create context-aware MobileApps and updating game-level scenes at run time, as well as building dynamic Web pages. In the most general sense, a CMS "configures" content for the end user by invoking three integrated subsystems: a collection subsystem, a management subsystem, and a publishing subsystem [Boi04].

The Collection Subsystem. Content is derived from data and information that must be created or acquired by a content developer. The *collection subsystem* encompasses all actions required to create and/or acquire content, and the technical functions that are necessary to (1) convert content into a form that can be represented by a mark-up

FIGURE 22.8
Content management system



language (e.g., HTML, XML), and (2) organize content into screens that can be displayed efficiently on the client side.

Content creation and acquisition (often called *authoring* or *level design* for games) commonly occurs in parallel with other development activities and is often conducted by nontechnical content developers. This activity combines elements of creativity and research and is supported by tools that enable the content author to characterize content in a manner that can be standardized for use within the app or game.

Once content exists, it must be converted to conform to the requirements of a CMS. This implies stripping raw content of any unnecessary information (e.g., redundant graphical representations), formatting the content to conform to the requirements of the CMS, and mapping the results into an information structure that will enable it to be managed and published.

The Management Subsystem. Once content exists, it must be stored in a repository, cataloged for subsequent acquisition and use, and labeled to define (1) current status (e.g., is the content object complete or in development?), (2) the appropriate version of the content object, and (3) related content objects. Configuration management is

performed within this subsystem. Therefore, the *management subsystem* implements a repository that encompasses the following elements:

- **Content database.** The information structure that has been established to store all content objects.
- **Database capabilities.** Functions that enable the CMS to search for specific content objects (or categories of objects), store and retrieve objects, and manage the file structure that has been established for the content.
- **Configuration management functions.** The functional elements and associated workflow that support content object identification, version control, change management, change auditing, and reporting.

In addition to these elements, the management subsystem implements an administration function that encompasses the metadata and rules that control the overall structure of the content and the manner in which it is supported.

The Publishing Subsystem. Content must be extracted from the repository, converted to a form that is amenable to publication and formatted so that it can be transmitted to client-side screen displays. The publishing subsystem accomplishes these tasks using a series of templates. Each *template* is a function that builds a publication using one of three different components [Boi04]:

- **Static elements.** Text, graphics, media, and scripts that require no further processing are transmitted directly to the client side.
- **Publication services.** Function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
- **External services.** Access to external corporate information infrastructure such as enterprise data or "backroom" applications.

A content management system that encompasses each of these subsystems is applicable for major Web and mobile projects. However, the basic philosophy and functionality associated with a CMS are applicable to all dynamic applications.

22.6.4 Version Control

As apps and games evolve through a series of increments, a number of different versions may exist at the same time. One version (the current operational app) is available via the Internet for end users; another version (the next app increment) may be in the final stages of testing prior to deployment; a third version is in development and represents a major update in content, interface aesthetics, and functionality. Configuration objects must be clearly defined so that each can be associated with the appropriate version. Without some type of control, developers and content creators may end up overwriting each other's changes.

It's likely that you've experienced a similar situation. To avoid it, a version control process is required.

1. **A central repository for the app or game project should be established.** The repository will hold current versions of all configuration objects (content, functional components, and others).

2. **Each developer creates his own working folder.** The folder contains those objects that are being created or changed at any given time.
3. **The clocks on all developer workstations should be synchronized.** This is done to avoid overwriting conflicts when two developers make updates that are very close to one another in time.
4. **As new configuration objects are developed or existing objects are changed, they are imported into the central repository.** The version control tool will manage all check-in and check-out functions from the working folders of each developer. The tool should also provide automatic e-mail updates to all interested parties when changes to the repository are made.
5. **As objects are imported or exported from the repository, an automatic, time-stamped log message is made.** This provides useful information for auditing and can become part of an effective reporting scheme.

The version control tool maintains different versions of the app and can revert to an older version if required.

22.6.5 Auditing and Reporting

In the interest of agility, the auditing and reporting functions are deemphasized during the development of games or apps.[9] However, they are not eliminated altogether. All objects that are checked into or out of the repository are recorded in a log that can be reviewed at any point in time. A complete log report can be created so that all members of the team have a chronology of changes over a defined period of time. In addition, an automated e-mail notification (addressed to those developers and stakeholders who have interest) can be sent every time an object is checked in or out of the repository.

22.7 SUMMARY

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All work products created as part of software engineering become part of a software configuration. The configuration is organized in a manner that enables orderly control of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items, that are produced as a result of some software engineering activity. In addition to software engineering work products, the development environment that is used to create software can also be placed under configuration control. All SCIs are stored within a repository that implements a set of mechanisms and data

9 This is beginning to change. There is an increasing emphasis on SCM as one element of application security [Fug14]. By providing a mechanism for tracking and reporting every change made to every application object, a change management tool can provide valuable protection against malicious changes.

structures to ensure data integrity, provide integration support for other software tools, support information sharing among all members of the software team, and implement functions in support of version and change control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Version control is the set of procedures and tools for managing the use of these objects.

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

Configuration management for apps or games is similar in most respects to SCM for conventional software. However, each of the core SCM tasks should be streamlined to make it as lean as possible, and special provisions for content management must be implemented.

Problems and Points to Ponder

22.1. Why is the first law of system engineering true? Provide specific examples for each of the four fundamental reasons for change.

22.2. What are the four elements that exist when an effective SCM system is implemented? Discuss each briefly.

22.3. Assume that you're the manager of a small project. What baselines would you define for the project, and how would you control them?

22.4. Design a project database (repository) system that would enable a software engineer to store, cross-reference, trace, update, change, and so forth all important software configuration items. How would the database handle different versions of the same program? Would source code be handled differently than documentation? How will two developers be precluded from making different changes to the same SCI at the same time?

22.5. Research an existing SCM tool, and describe how it implements control for versions, variants, and configuration objects in general.

22.6. Research an existing SCM tool, and describe how it implements the mechanics of version control. Alternatively, read two or three papers on SCM and describe the different data structures and referencing mechanisms that are used for version control.

22.7. Develop a checklist for use during configuration audits.

22.8. What is the difference between an SCM audit and a technical review? Can their function be folded into one review? What are the pros and cons?

22.9. Briefly describe the differences between SCM for conventional software and SCM for WebApps or MobileApps.

22.10. Describe the value of continuous integration tools to agile software developers.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

23 SOFTWARE METRICS AND ANALYTICS

A key element of any engineering process is measurement. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. You can use measures to better understand the attributes of the models that you create and to assess the quality of the engineered products or systems that you build.

KEY CONCEPTS

data science 461
defect removal efficiency (DRE) 482
goal 485
indicator 462
measure 461
measurement 462
metrics 461
  arguments for 481
  attributes of 462
  design metrics 466
  establishing a program 485
function-oriented 481
LOC-based metrics 481
private and public 480
process 476
productivity 481
project 476
size-oriented 480
software quality 482
source code 473
testing 474
software analytics 462



QUICK LOOK

What is it? Software process and project metrics are quantitative measures that enable you to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Product metrics help software engineers gain insight into the design and construction of the software they build.

Who does it? Software metrics are analyzed and assessed by software managers. Software engineers use product metrics to help them build higher-quality software.

Why is it important? If you don't measure, judgment can only be based on subjective evaluation. You need objective criteria to help guide the design of data, architecture, interfaces, and components. If you measure, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

What are the steps? Derive the process, project, and product measures and metrics that you intend to use. Collect the metrics and then analyze them against historical data. Use the analysis results to gain insight into the process, project, and product.

What is the work product? A set of software metrics that provides insight into the process and understanding of the project.

How do I ensure that I've done it right? Define only a few metrics, and then use them to gain insight into the quality of a software process, project, and product. Apply a consistent, yet simple measurement scheme that is never to be used to assess, reward, or punish individual performance.

Measurement can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Because software measures and metrics are often indirect, they are open to debate.

Within the context of the software process and the projects that are conducted using the process, a software team is concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us plan and estimate more accurately?

Measurement is a management and technical tool. If conducted properly, it provides you with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

In this chapter, we present measures that can be used to assess the quality of the product as it is being engineered. We also present measures that can be used to help manage software projects. These measures provide you with a real-time indication of the effectiveness of your software processes (analysis, design, testing) and the overall quality of the software as it is being built.

23.1 SOFTWARE MEASUREMENT

Data science[1] is concerned with measurement, machine learning, and prediction of future events based on these measures. Measurement assigns numbers or symbols to attributes of entities in the real word. To accomplish this, a measurement model encompassing a consistent set of rules is required. Although the theory of measurement (e.g., [Kyb84]) and its application to computer software (e.g., [Zus97]) are topics that are beyond the scope of this book, it is worthwhile to establish a fundamental framework and a set of basic principles that guide the definition of metrics for software development.

23.1.1 Measures, Metrics, and Indicators

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a *measure* has been established. *Measurement* occurs as the result of the collection of one or more data points (e.g., a number of component reviews and unit tests are investigated to collect measures of the number of errors for each). A *software*

1 Appendix 2 in the book contains an introduction to data science geared toward software engineers.

metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test).

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself.

23.1.2 Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer. Some demand measurement that is too complex; others are so esoteric that few real-world professionals have any hope of understanding them, and others violate the basic intuitive notions of what high-quality software really is. Experience indicates that a metric will be used only if it is intuitive and easy to compute. If dozens of "counts" have to be made, and complex computations are required, it is unlikely that the metric will be widely adopted.

Ejiogu [Eji91] defines a set of attributes that should be encompassed by effective software metrics. It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time. The metric should satisfy the engineer's intuitive notions about the product attribute under consideration (e.g., a metric that measures module cohesion should increase in value as the level of cohesion increases). The metric should always yield results that are unambiguous. The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units. For example, multiplying people on the project teams by programming language variables in the program results in a suspicious mix of units that are not intuitively persuasive. Metrics should be based on the requirements model, the design model, or the structure of the program itself. They should not be dependent on the vagaries of programming language syntax or semantics. Finally, the metric should provide you with information that can lead to a higher-quality end product.

23.2 Software Analytics

There is some confusion about the differences between software metrics and software analytics. Software metrics are used to gauge the quality or performance of a product or process. *Key performance indicators* (KPIs) are metrics that are used to track performance and trigger remedial actions when their values fall in a predetermined range. But how do you know that metrics are meaningful in the first place?

Software analytics is the systematic computational analysis of software engineering data or statistics to provide managers and software engineers with meaningful insights and empower their teams to make better decisions [Bus12]. It is important that the insights provide timely, actionable advice to developers. For example, knowing the number of defects in a software product today is not as important as knowing the number of defects is 5 percent higher than last month. Analytics can help developers predict the number of defects to expect, where to test for them, and how much time it will take to fix them. This allows managers and developers to create incremental schedules that use these predictions to determine expected completion times. The use

of automated tools capable of processing large, dynamic data sets of engineering metrics and measures [Men13] is required to provide real-time insight into large project and product data sets.

Buse and Zimmermann [Bus12] suggest that analytics can help developers make decisions regarding:

- **Targeted testing.** To help focus regression testing and integration testing resources
- **Targeted refactoring.** To help make strategic decisions on how to avoid large technical debt costs
- **Release planning.** To help ensure that market needs as well as technical features in software product are taken into account
- **Understanding customers.** To help developers get actionable information on product use by customers in the field during product engineering
- **Judging stability.** To help managers and developers monitor the state of the evolving prototype and anticipate future maintenance needs
- **Targeting inspection.** To help teams determine the value of individual inspection activities, their frequency, and their scope

The statistical techniques (data mining, machine learning, statistical modeling) required to do software analytic work is beyond the scope of this book. Some of these techniques are discussed briefly in Appendix 2. We will focus on the use of software metrics in the remainder of this chapter.

23.3 PRODUCT METRICS

Over the past four decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton [Fen94] characterizes this research as a search for "the impossible holy grail." Although dozens of complexity measures have been proposed [Zus90], each takes a somewhat different view of what complexity is and what attributes of a system lead to complexity. By analogy, consider a metric for evaluating an attractive car. Some observers might emphasize body design; others might consider mechanical characteristics; still others might tout cost, or performance, or the use of alternative fuels, or the ability to recycle when the car is junked. Because any one of these characteristics may be at odds with others, it is difficult to derive a single value for "attractiveness." The same problem occurs with computer software.

Yet there is a need to measure and control software complexity. And if a single value of this quality metric is difficult to derive, it should be possible to develop measures of different internal program attributes (e.g., effective modularity, functional independence, and other attributes discussed in Chapter 9). These measures and the metrics derived from them can be used as independent indicators of the quality of requirements and design models. But here again, problems arise. Fenton [Fen94] notes this when he states: "The danger of attempting to find measures which characterize so many different attributes is that inevitably the measures have to satisfy conflicting

aims. This is counter to the representational theory of measurement." Although Fenton's statement is correct, many people argue that product measurement conducted during the early stages of the software process provides software engineers with a consistent and objective mechanism for assessing quality.[2]

SAFEHOME



Debating Product Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, and Ed, members of the *SafeHome* software engineering team who are continuing work of component-level design and test-case design.

The conversation:

Vinod: Doug [Doug Miller, software engineering manager] told me that we should all use product metrics, but he was kind of vague. He also said that he wouldn't push the matter . . . that using them was up to us.

Jamie: That's good, 'cause there's no way I have time to start measuring stuff. We're fighting to maintain the schedule as it is.

Ed: I agree with Jamie. We're up against it, here . . . no time.

Vinod: Yeah, I know, but there's probably some merit to using them.

Jamie: I'm not arguing that, Vinod, it's a time thing . . . and I for one don't have any to spare.

Vinod: But what if measuring saves you time?

Ed: Wrong, it takes time and like Jamie said . . .

Vinod: No, wait . . . what if it saves us is time?

Jamie: How?

Vinod: Rework . . . that's how. If a measure we use helps us to avoid one major or even moderate problem, and that saves us from having to rework a part of the system, we save time. No?

Ed: It's possible, I suppose, but can you guarantee that some product metric will help us find a problem?

Vinod: Can you guarantee that it won't?

Jamie: So what are you proposing?

Vinod: I think we should select a few design metrics, probably class-oriented, and use them as part of our review process for every component we develop.

Ed: I'm not real familiar with class-oriented metrics.

Vinod: I'll spend some time checking them out and make a recommendation . . . okay with you guys?

(Ed and Jamie nod without much enthusiasm.)

23.3.1 Metrics for the Requirements Model

Technical work in software engineering begins with the creation of the requirements model. It is at this stage that requirements are derived and a foundation for design is established. Therefore, product metrics that provide insight into the quality of the analysis model are desirable.

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics (e.g., use case points or function points)

2 Although criticism of specific metrics is common in the literature, many critiques focus on esoteric issues and miss the primary objective of metrics in the real world: to help software engineers establish a systematic and objective way to gain insight into their work and to improve product quality as a result.

that are often used for project estimation (Section 25.6) and apply them in this context. These estimation metrics examine the requirements model with the intent of predicting the "size" of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort. By measuring characteristics of the requirements model, it is possible to gain quantitative insight into its specificity and completeness.

Conventional Software. Davis and his colleagues [Dav93] propose a list of characteristics that can be used to assess the quality of the requirements model and the corresponding requirements specification: *specificity* (lack of ambiguity), *completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision,* and *reusability.* In addition, the authors note that high-quality specifications are electronically stored; executable or at least interpretable; annotated by relative importance; and stable, versioned, organized, cross-referenced, and specified at the right level of detail.

Although many of these characteristics appear to be qualitative in nature, each can be represented using one or more metrics [Dav93]. For example, we assume that there are n_r requirements in a specification, such that

$$n_r = n_f + n_{\text{nf}}$$

where n_f is the number of functional requirements and n_{nf} is the number of nonfunctional (e.g., performance) requirements.

To determine the *specificity* (lack of ambiguity) of requirements, Davis and colleagues suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = \frac{n_{ui}}{n_r}$$

where n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification. Other characteristics are computed in a similar manner.

Mobile Software. The objective of all mobile projects is to deliver a combination of content and functionality to the end user. Measures and metrics used for traditional software engineering projects are difficult to translate directly to MobileApps. Yet, it is possible to develop measures that can be determined during the requirements gathering activities that can serve as the basis for creating MobileApp metrics. Among the measures that can be collected are the following:

Number of static screen displays. These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

Number of dynamic screen displays. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

Number of persistent data objects. As the number of persistent data objects (e.g., a database or data file) grows, the complexity of the MobileApp also grows and the effort to implement it increases proportionally.

Number of external systems interfaced. As the requirement for interfacing grows, system complexity and development effort also increase.

Number of static content objects. These objects represent low relative complexity and generally require less effort to construct than dynamic pages.

Number of dynamic content objects. These objects represent higher relative complexity and require more effort to construct than static pages.

Number of executable functions. As the number of executable functions (e.g., a script or applet) increases, modeling and construction effort also increase.

For example with these measures, you can define a metric that reflects the degree of end-user customization that is required for the MobileApp and correlate it to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted. To accomplish this, you define

N_{sp} = number of static screen displays
N_{dp} = number of dynamic screen displays

Then,

$$\text{Customization index, } C = \frac{N_{dp}}{N_{dp} + N_{sp}}$$

The value of C ranges from 0 to 1. As C grows larger, the level of app customization becomes a significant technical issue.

Similar metrics can be computed and correlated with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. If the values of these metrics are stored in a database with project measures (after a number of projects have been completed), the relationships between the app requirement measures and project measures will provide indicators that can aid in project assessment tasks.

23.3.2 Design Metrics for Conventional Software

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them as indicators to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Architectural design metrics focus on characteristics of the program architecture (Chapter 10) with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. These metrics are "black box" in the sense that they do not require any knowledge of the inner workings of a particular software component. Metrics can provide insight into structural data and system complexity associated with architectural design.

Card and Glass [Car90] define three software design complexity measures: structural complexity, data complexity, and system complexity.

For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module i is defined in the following manner:

$$S(i) = f_{\text{out}}^2(i)$$

where $f_{\text{out}}(i)$ is the fan-out[3] of module *i.*

Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

where $v(i)$ is the number of input and output variables that are passed to and from module *i.*

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i)$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

Fenton [Fen91] suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to the call-and-return architecture in Figure 23.1, the following metrics can be defined:

$$\text{Size} = n + a$$

where n is the number of nodes and a is the number of arcs. For the architecture shown in Figure 23.1,

$$\text{Size} = 17 + 18 = 35$$

Depth = longest path from the root (top) node to a leaf node. For the architecture shown in Figure 23.1, depth = 4.

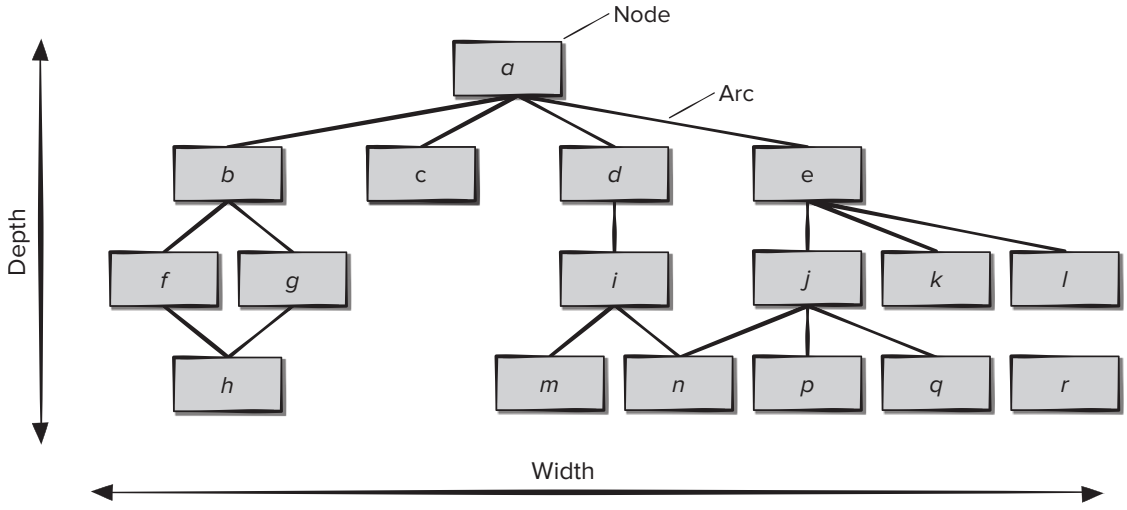
Width = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 23.1, width = 6.

The arc-to-node ratio, $r = a/n$, measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in Figure 23.1, $r = 18/17 = 1.06$.

The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-2005 [IEE05], the Air Force uses information obtained from data and architectural design

3 *Fan-out* is defined as the number of modules immediately subordinate to module *I,* that is, the number of modules that are directly invoked by module *i.*

FIGURE 23.1 Morphology metrics



to derive a *design structure quality index* (DSQI) that ranges from 0 to 1 (see: [USA87] and [Cha89] for details).

23.3.3 Design Metrics for Object-Oriented Software

There is much about object-oriented design that is subjective—an experienced designer "knows" how to characterize an OO system so that it will effectively implement customer requirements. But, as an OO design model grows in size and complexity, a more objective view of the characteristics of the design can benefit both the experienced designer (who gains additional insight) and the novice (who obtains an indication of quality that would otherwise be unavailable).

In a detailed treatment of software metrics for OO systems, Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design. *Size* is defined by taking a static count of OO entities such as classes or operations, coupled with the depth of an inheritance tree. *Complexity* is defined in terms of structural characteristics by examining how classes of an OO design are interrelated to one another. *Coupling* is measured by counting physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects). *Sufficiency* is "the degree to which an abstraction [class] possesses the features required of it . . ." [Whi97]. *Completeness* determines whether a class delivers the set of properties that fully reflect the needs of the problem domain. *Cohesion* is determined be examining whether all operations work together to achieve a single, well-defined purpose. *Primitiveness* is the degree to which an operation is atomic—that is, the operation cannot be constructed out of a sequence of other operations contained within a class. Similarity determines the degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose. Volatility measures the likelihood that a change will occur.

In reality, product metrics for OO systems can be applied not only to the design model, but also to the requirements model. In the remainder of this section, we discuss metrics that provide an indication of quality at the OO class level and the operation level. In addition, metrics applicable for project management and testing are also explored.

Chidamber and Kemerer (CK) have proposed one of the most widely referenced sets of OO software metrics [Chi94].[4] Often referred to as the *CK metrics suite,* the authors have proposed six class-based design metrics for OO systems.[5]

Weighted Methods per Class (WMC). Assume that n methods of complexity c_1, c_2, . . . , c_n are defined for a class **C.** The specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on a value of 1.0.

$$\text{WMC} = \Sigma c_i$$

for $i = 1$ to n. The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class. In addition, the larger the number of methods, the more complex is the inheritance tree (all subclasses inherit the methods of their parents). Finally, as the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse. For all of these reasons, WMC should be kept as low as is reasonable.

Depth of the Inheritance Tree (DIT). This metric is "the maximum length from the node to the root of the tree" [Chi94]. Referring to Figure 23.2, the value of DIT for the class hierarchy shown is 4. As DIT grows, it is likely that lower-level classes will inherit many methods. This leads to potential difficulties when attempting to predict the behavior of a class. A deep class hierarchy (DIT is large) also leads to greater design complexity. On the positive side, large DIT values imply that many methods may be reused.

Number of Children (NOC). The subclasses that are immediately subordinate to a class in the class hierarchy are termed its children. Referring to Figure 23.2, class $\mathbf{C}_2$ has three children—subclasses $\mathbf{C}_{21}$, $\mathbf{C}_{22}$, and $\mathbf{C}_{23}$. As the number of children grows, reuse increases, but also, as NOC increases, the abstraction represented by the parent class can be diluted if some of the children are not appropriate members of the parent class. As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

Coupling Between Object Classes (CBO). The CRC model (Chapter 8) may be used to determine the value for CBO. In essence, CBO is the number of collaborations listed for a class on its CRC index card.[6] As CBO increases, it is likely that the

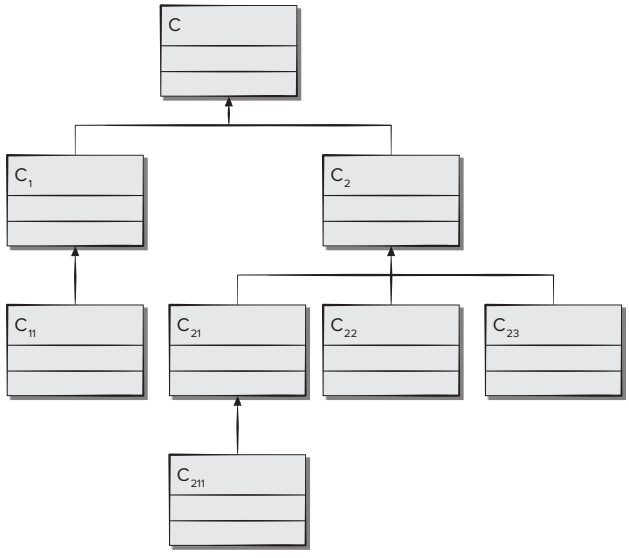
4 An alternative suite of OO metrics has been proposed by Harrison, Counsell, and Nithi [Har98b]. Interested readers are urged to examine their work.

5 Chidamber and Kemerer use the term *methods* rather than *operations*. Their usage of the term is reflected in this section.

6 If CRC index cards are developed manually, completeness and consistency must be assessed before CBO can be determined reliably.

FIGURE 23.2

A class hierarchy



reusability of a class will decrease. High values of CBO also complicate modifications and the testing that ensues when modifications are made. In general, the CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.

Response for a Class (RFC). The response set of a class is "a set of methods that can potentially be executed in response to a message received by an object of that class" [Chi94]. RFC is the number of methods in the response set. As RFC increases, the effort required for testing also increases because the test sequence (Chapter 20) grows. It also follows that, as RFC increases, the overall design complexity of the class increases.

Lack of Cohesion in Methods (LCOM). Each method within a class **C** accesses one or more attributes (also called instance variables). LCOM is the number of methods that access one or more of the same attributes.[7] If no methods access the same attributes, then LCOM = 0. To illustrate the case where LCOM ≠ 0, consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes). Therefore, LCOM = 4. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high, that is, keep LCOM low.[8]

7 The formal definition is a bit more complex. See [Chi94] for details.

8 The LCOM metric provides useful insight in some situations, but it can be misleading in others. For example, keeping coupling encapsulated within a class increases the cohesion of the system as a whole. Therefore, in at least one important sense, higher LCOM actually suggests that a class may have higher cohesion, not lower.

SAFEHOME



Applying CK Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, Shakira, and Ed, members of the *SafeHome* software engineering team who are continuing to work on component-level design and test-case design.

The conversation:

Vinod: Did you guys get a chance to read the description of the CK metrics suite I sent you on Wednesday and make those measurements?

Shakira: Wasn't too complicated. I went back to my UML class and sequence diagrams, like you suggested, and got rough counts for DIT, RFC, and LCOM. I couldn't find the CRC model, so I didn't count CBO.

Jamie (smiling): You couldn't find the CRC model because I had it.

Shakira: That's what I love about this team, superb communication.

Vinod: I did my counts . . . did you guys develop numbers for the CK metrics?

(Jamie and Ed nod in the affirmative.)

Jamie: Since I had the CRC cards, I took a look at CBO, and it looked pretty uniform across most of the classes. There was one exception, which I noted.

Ed: There are a few classes where RFC is pretty high, compared with the averages . . . maybe we should take a look at simplifying them.

Jamie: Maybe yes, maybe no. I'm still concerned about time, and I don't want to fix stuff that isn't really broken.

Vinod: I agree with that. Maybe we should look for classes that have bad numbers in at least two or more of the CK metrics. Kind of two strikes and you're modified.

Shakira (looking over Ed's list of classes with high RFC): Look, see this class. It's got a high LCOM as well as a high RFC. Two strikes?

Vinod: Yeah I think so . . . it'll be difficult to implement because of complexity and difficult to test for the same reason. Probably worth designing two separate classes to achieve the same behavior.

Jamie: You think modifying it'll save us time?

Vinod: Over the long haul, yes.

23.3.4 User Interface Design Metrics

Although there is significant literature on the design of human-computer interfaces (Chapter 12), relatively little information has been published on metrics that would provide insight into the quality and usability of the interface. Although UI metrics may be useful in some cases, the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [Nie94] report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated."

In the paragraphs that follow, we present a representative sampling of design metrics that may have application for websites, browser-based applications, and mobile applications. Many of these metrics are applicable to all user interfaces. It is important to note, however, that many of these metrics have not as yet been validated and should be used judiciously.

Interface Metrics. For WebApps, the following interface measures can be considered:

Suggested Metric	*Description*
Layout appropriateness	The relative position of entities within the interface
Layout complexity	Number of distinct regions[9] defined for an interface
Layout region complexity	Average number of distinct links per region
Recognition complexity	Average number of distinct items the user must look at before making a navigation or data input decision
Recognition time	Average time (in seconds) that it takes a user to select the appropriate action for a given task
Typing effort	Average number of keystrokes required for a specific function
Mouse pick effort	Average number of mouse picks per function
Selection complexity	Average number of links that can be selected per page
Content acquisition time	Average number of words of text per Web page
Memory load	Average number of distinct data items that the user must remember to achieve a specific objective

Aesthetic (Graphic Design) Metrics. By its nature, aesthetic design relies on qualitative judgment and is not generally amenable to measurement and metrics. However, Ivory and her colleagues [Ivo01] propose a set of measures that may be useful in assessing the impact of aesthetic design:

Suggested Metric	*Description*
Word count	Total number of words that appear on a page
Body text percentage	Percentage of words that are body versus display text (e.g., headers)
Emphasized body text percentage	Portion of body text that is emphasized (e.g., bold, capitalized)
Text positioning count	Changes in text position from flush left
Text cluster count	Text areas highlighted with color, bordered regions, rules, or lists
Link count	Total links on a page
Page size	Total bytes for the page as well as elements, graphics, and style sheets
Graphic percentage	Percentage of page bytes that are for graphics
Graphics count	Total graphics on a page (not including graphics specified in scripts, applets, and objects)
Color count	Total colors employed
Font count	Total fonts employed (i.e., face + size + bold + italic)

9 A distinct region is an area within the layout display that accomplishes some specific set of related functions (e.g., a menu bar, a static graphical display, a content area, an animated display).

Content Metrics. Metrics in this category focus on content complexity and on clusters of content objects that are organized into pages [Men01].

Suggested Metric	*Description*
Page wait	Average time required for a page to download at different connection speeds
Page complexity	Average number of different types of media used on page, not including text
Graphic complexity	Average number of graphics media per page
Audio complexity	Average number of audio media per page
Video complexity	Average number of video media per page
Animation complexity	Average number of animations per page
Scanned image complexity	Average number of scanned images per page

Navigation Metrics. Metrics in this category address the complexity of the navigational flow [Men01]. In general, they are applicable only for static Web applications, which don't include dynamically generated links and pages.

Suggested Metric	*Description*
Page-linking complexity	Number of links per page
Connectivity	Total number of internal links, not including dynamically generated links
Connectivity density	Connectivity divided by page count

Using a subset of the metrics suggested, it may be possible to derive empirical relations that allow a WebApp development team to assess technical quality and predict effort based on projected estimates of complexity. Further work remains to be accomplished in this area.

23.3.5 Metrics for Source Code

Halstead's theory of "software science" [Hal77] proposed the first analytical "laws" for computer software.[10] Halstead assigned quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. The measures are:

n_1 = number of distinct operators that appear in a program

n_2 = number of distinct operands that appear in a program

N_1 = total number of operator occurrences

N_2 = total number of operand occurrences

10 It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed (e.g., [Fel89]).

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software.

Halstead shows that length N can be estimated as

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined as

$$V = N \log_2 (n_1 + n_2)$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

Halstead's work is amenable to experimental verification, and a large body of research has been conducted to investigate software science. A discussion of this work is beyond the scope of this book. For further information, see [Zus90], [Fen91], and [Zus97].

23.4 METRICS FOR TESTING

Testing metrics fall into two broad categories: (1) metrics that attempt to predict the likely number of tests required at various testing levels, and (2) metrics that focus on test coverage for a given component. The majority of metrics proposed for testing focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

Architectural design metrics provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing, a test-case design method presented in Chapter 19. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, you should expend above-average effort to uncover errors in such modules before they are integrated in a system.

Testing effort can be estimated using metrics derived from Halstead measures (Section 23.3.5). Using the definitions for program volume V and program level PL, Halstead effort e can be computed as

$$PL = \frac{1}{(n_1/2)(N_2/n_2)}$$

$$e = \frac{V}{PL}$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\text{Percentage of testing effort } (k) = \frac{e(k)}{\Sigma\, e(i)}$$

where $e(k)$ is computed for module k and the summation in the denominator is the sum of Halstead effort across all modules of the system.

OO testing can be quite complex. Metrics can assist you in targeting testing resources at threads, scenarios, and packages of classes that are "suspect" based on measured characteristics. The OO design metrics noted in Section 23.3.3 provide an indication of design quality. They also provide a general indication of the amount of testing effort required to exercise an OO system.

Binder [Bin94b] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system. The metrics consider aspects of encapsulation and inheritance.

Lack of cohesion in methods (LCOM).[11] The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

Percent public and protected (PAP). Public attributes are inherited from other classes and therefore are visible to those classes. Protected attributes are accessible to methods in subclasses. This metric indicates the percentage of class attributes that are public or protected. High values for PAP increase the likelihood of side effects among classes because public and protected attributes lead to high potential for coupling.[12] Tests must be designed to ensure that such side effects are uncovered.

Public access to data members (PAD). This metric indicates the number of classes (or methods) that can access another class's attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

Number of root classes (NOR). This metric is a count of the distinct class hierarchies that are described in the design model. Test suites for each root class and the corresponding class hierarchy must be developed. As NOR increases, testing effort also increases.

Fan-in (FIN). When used in the OO context, fan-in in the inheritance hierarchy is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

Number of children (NOC) and depth of the inheritance tree (DIT).[13] As we mentioned in Chapter 18, superclass methods will have to be retested for each subclass.

11 See Section 23.3.3 for a description of LCOM.

12 Some people promote designs with none of the attributes being public or private, that is, PAP = 0. This implies that all attributes must be accessed in other classes via methods.

13 See Section 23.3.3 for a description of NOC and DIT.

23.5 METRICS FOR MAINTENANCE

All the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-2005 [IEE05] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_T = number of modules in the current release

F_c = number of modules in the current release that have been changed

F_a = number of modules in the current release that have been added

F_d = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

23.6 PROCESS AND PROJECT METRICS

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement (Chapter 28). *Project metrics* enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domains.

Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement (Chapter 28). But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of

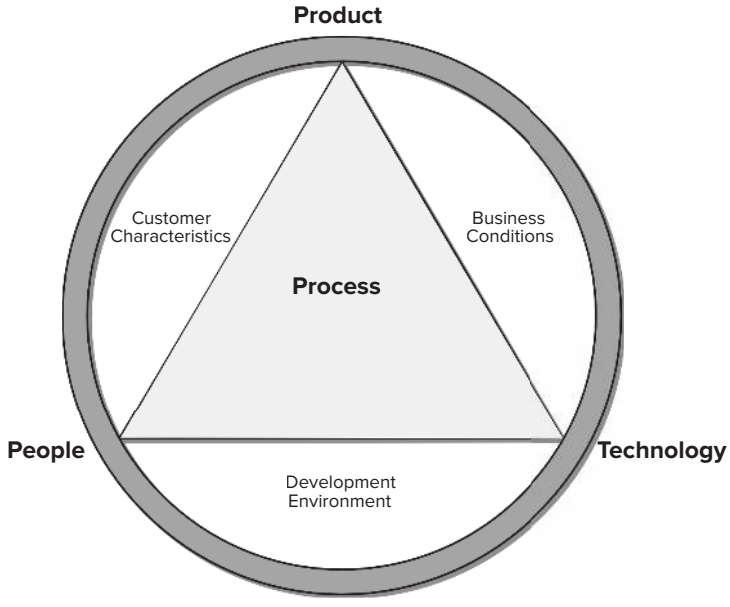


FIGURE 23.3
Determinants for software quality and organizational effectiveness

"controllable factors in improving software quality and organizational performance" [Pau94].

Referring to Figure 23.3, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people have been shown [Boe81] to be the most influential factors in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time used, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities described in Chapter 1.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds,

measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

Software process metrics can provide significant benefits as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [Gra92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects[14] encountered as an application, system, or product is developed and used.

14 In this book, an *error* is defined as some flaw in a software engineering work product that is uncovered *before* the software is delivered to the end user. A *defect* is a flaw that is uncovered *after* delivery to the end user. It should be noted that others do not make this distinction.

SAFEHOME



Establishing a Metrics Approach

The scene: Doug Miller's office as the *SafeHome* software project is about to begin.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: Before we start work on this project, I'd like you guys to define and collect a set of simple metrics. To start, you'll have to define your goals.

Vinod (frowning): We've never done that before, and . . .

Jamie (interrupting): And based on the time line management has been talking about, we'll never have the time. What good are metrics anyway?

Doug (raising his hand to stop the onslaught): Slow down and take a breath, guys. The fact that we've never done it before is all the more reason to start now, and the metrics work I'm talking about shouldn't take much time at all . . . in fact, it just might save us time.

Vinod: How?

Doug: Look, we're going to be doing a lot more in-house software engineering as our products get more intelligent, become context aware, mobile, all that . . . and we need to understand the process we use to build software . . . and improve it so we can build software better. The only way to do that is to measure.

Jamie: But we're under time pressure, Doug. I'm not in favor of more paper pushing . . . we need the time to do our work, not collect data.

Doug (calmly): Jamie, an engineer's work involves collecting data, evaluating it, and using the results to improve the product and the process. Am I wrong?

Jamie: No, but . . .

Doug: What if we hold the number of measures we collect to no more than five or six and focus on quality?

Vinod: No one can argue against high quality . . .

Jamie: True . . . but, I don't know. I still think this isn't necessary.

Doug: I'm going to ask you to humor me on this one. How much do you guys know about software metrics?

Jamie (looking at Vinod): Not much.

Doug: Here are some Web refs . . . spend a few hours getting up to speed.

Jamie (smiling): I thought you said this wouldn't take any time.

Doug: Time you spend learning is never wasted . . . go do it, and then we'll establish some goals, ask a few questions, and define the metrics we need to collect.

23.7 SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" that are discussed in Chapter 15.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have partitioned the software metrics domain into process, project, and product metrics and noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 23.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 23.4) for project *alpha*: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project *alpha* indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

To develop metrics that can be assimilated with similar metrics from other projects, you can choose lines of code as a normalization value. From the rudimentary data

FIGURE 23.4 Size-oriented metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	Category
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per KLOC
- Pages of documentation per KLOC

In addition, other interesting metrics can be computed:

- Errors per person-month
- KLOC per person-month
- $ per page of documentation

Size-oriented metrics are not universally accepted as the best way to measure the software process. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter programs; that they cannot easily accommodate nonprocedural languages; and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

Similar arguments, pro and con, can be made for function-oriented metrics such as function points (FP) or use case points (both are discussed in Chapter 25). Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Computation of a function-oriented metric is based on characteristics of the software's information domain and complexity.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language–independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, to use LOC and FP for estimation (Chapter 25), an historical baseline of information must be established. It is this historical data that over time will let you judge the value of a particular metric on future projects.

Size-oriented measures (e.g., LOC) and function-oriented measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answer to these questions is an emphatic no! The reason

for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Within the context of process and project metrics, you should be concerned primarily with productivity and quality—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced.

For process improvement and project planning purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

23.8 Metrics for Software Quality

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. Software is a complex entity. Therefore, errors are to be expected as work products are developed. Process metrics are intended to improve the software process so that errors are uncovered in the most effective manner.

You can use measurement to assess the quality of the requirements and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time assessment, you apply product metrics to evaluate the quality of software engineering work products in objective rather than subjective ways.

A project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are combined to provide project-level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed later in this section.

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [Gil88] suggests definitions and measures for each.

> **Correctness.** Correctness is the degree to which the software performs its required function. Defects (lack of correctness) are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.

Maintainability. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is *mean time to change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

Integrity. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as:

$$\text{Integrity} = \Sigma[1 - (\text{threat} \times (1 - \text{security}))]$$

For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high). If, on the other hand, the threat probability is 0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

Usability. Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics presented in Chapter 12.

These four factors are only a sampling of those that have been proposed as measures for software quality.

A quality metric that provides benefit at both the project and process level is *defect removal efficiency* (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = \frac{E}{E + D}$$

where E is the number of errors found before delivery of the software to the end user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task.

For example, requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found). When used in this context, we redefine DRE as

$$\text{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$

where E_i is the number of errors found during software engineering action i and E_{i+1} is the number of errors found during software engineering action $i + 1$ that are traceable to errors that were not discovered in software engineering action i.

A quality objective for a software team (or an individual software engineer) is to achieve a DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity or action. If DRE is low as you move through analysis and design, spend some time improving the way you conduct formal technical reviews.

SAFEHOME



A Metrics-Based Quality Approach

The scene: Doug Miller's office 2 days after initial meeting on software metrics.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: Did you both have a chance to learn a little about process and project metrics?

Vinod and Jamie: (Both nod.)

Doug: It's always a good idea to establish goals when you adopt any metrics. What are yours?

Vinod: Our metrics should focus on quality. In fact, our overall goal is to keep the number of errors we pass on from one software engineering activity to the next to an absolute minimum.

Doug: And be very sure you keep the number of defects released with the product to as close to zero as possible.

Vinod (nodding): Of course.

Jamie: I like DRE as a metric, and I think we can use it for the entire project, but also as we move from one framework activity to the next. It'll encourage us to find errors at each step.

Vinod: I'd also like to collect the number of hours we spend on reviews.

Jamie: And the overall effort we spend on each software engineering task.

Doug: You can compute a review-to-development ratio . . . might be interesting.

Jamie: I'd like to track some use case data as well. Like the amount of effort required to develop a use case, the amount of effort required to build software to implement a use case, and . . .

Doug (smiling): I thought we were going to keep this simple.

Vinod: We should, but once you get into this metrics stuff, there's a lot of interesting things to look at.

Doug: I agree, but let's walk before we run and stick to our goal. Limit data to be collected to five or six items, and we're ready to go.

23.9 Establishing Software Metrics Programs

The Software Engineering Institute has developed a comprehensive guidebook [Par96b] for establishing a "goal-driven" software metrics program. The guidebook suggests the following steps: (1) identify your business goals, (2) identify what you want to know or learn, (3) identify your subgoals, (4) identify the entities and attributes related to your subgoals, (5) formalize your measurement goals, (6) identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals, (7) identify the data elements that you will collect to construct the indicators, (8) identify the measures to be used, and make these definitions operational, (9) identify the actions that you will take to implement the measures, and (10) prepare a plan for implementing the measures. A detailed discussion of these steps is best left to the SEI's guidebook. However, a brief overview of key points is illustrated by the following example.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider the *SafeHome* product. Working as a team, software engineering and business managers develop a list of prioritized business goals:

1. Improve our customers' satisfaction with our products.
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

The software organization examines each business goal and asks: "What activities do we manage, execute, or support and what do we want to improve within these activities?" To answer these questions, the SEI recommends the creation of an "entity-question list" in which all things (entities) within the software process that are managed or influenced by the software organization are noted. Examples of entities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process.

Consider the fourth goal: "Make support for our products easier." The following list of questions might be derived for this goal [Par96b]:

- Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?
- How large is the change request backlog?
- Is our response time for fixing bugs acceptable based on customer need?
- Is our change control process (Chapter 22) followed?
- Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following subgoal: *Improve the performance of the change management process.* The software process entities and attributes that are relevant to the subgoal are identified, and the measurement goals associated with them are delineated.

The SEI [Par96b] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, you refine measurement goals into questions that are further refined into entities and attributes that are then refined into metrics.

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it is reasonable to suggest that software organizations[15] of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce.

A small organization can begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires improvement. For example, "reduce the time to evaluate and implement change requests." A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, t_{queue}.
- Effort (person-hours) to perform the evaluation, W_{eval}.
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, t_{eval}.
- Effort (person-hours) required to make the change, W_{change}.
- Time required (hours or days) to make the change, t_{change}.
- Errors uncovered during work to make change, E_{change}.
- Defects uncovered after change is released to the customer base, D_{change}.

Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data, E_{change} and D_{change}. The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce t_{queue}, W_{eval}, t_{eval}, W_{change}, and/or E_{change}. In addition, the defect removal efficiency can be computed as

$$\text{DRE} = \frac{E_{change}}{E_{change} + D_{change}}$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted previously in this chapter, the problem is cultural.

15 This discussion is equally relevant to software teams that have adopted an agile software development process (Chapter 3).

Attempting to collect measures where none have been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner. Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If you do not measure, there is no real way of determining whether you are improving. And if you are not improving, you are lost.

23.10 SUMMARY

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of software projects; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project work flow and technical approach in a real-time manner.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business.

Both size- and function-oriented metrics are used throughout the industry. Size-oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. Few product metrics have been proposed for direct use in software testing and maintenance. However, many other product metrics can be used to guide the testing process and as a mechanism for assessing the maintainability of a computer program.

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling you to assess quality before the product is built. Metrics provide the insight necessary to create effective requirements and design models, solid code, and thorough tests.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

To be useful in a real-world context, a software metric must be simple and computable, persuasive, consistent, and objective. It should be programming language independent and provide you with effective feedback.

PROBLEMS AND POINTS TO PONDER

23.1. Software for System X has 24 individual functional requirements and 14 nonfunctional requirements. What is the specificity of the requirements? The completeness?

23.2. A major information system has 1140 modules. There are 96 modules that perform control and coordination functions and 490 modules whose function depends on prior processing. The system processes approximately 220 data objects that each has an average of three attributes.

There are 140 unique database items and 90 different database segments. Finally, 600 modules have single entry and exit points. Compute the DSQI for this system.

23.3. A class **X** has 12 operations. Cyclomatic complexity has been computed for all operations in the OO system, and the average value of module complexity is 4. For class **X,** the complexity for operations 1 to 12 is 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectively. Compute the weighted methods per class.

23.4. A legacy system has 940 modules. The latest release required that 90 of these modules be changed. In addition, 40 new modules were added and 12 old modules were removed. Compute the software maturity index for the system.

23.5. Why should some software metrics be kept "private"? Provide examples of three metrics that should be private. Provide examples of three metrics that should be public.

23.6. Team A found 342 errors during the software engineering process prior to release. Team B found 184 errors. What additional measures would have to be made for projects A and B to determine which of the teams eliminated errors more efficiently? What metrics would you propose to help in making the determination? What historical data might be useful?

23.7. A Web engineering team has built an e-commerce WebApp that contains 145 individual pages. Of these pages, 65 are dynamic; that is, they are internally generated based on end-user input. What is the customization index for this application?

23.8. A WebApp and its support environment have not been fully fortified against attack. Web engineers estimate that the likelihood of repelling an attack is only 30 percent. The system does not contain sensitive or controversial information, so the threat probability is 25 percent. What is the integrity of the WebApp?

23.9. At the conclusion of a project, it has been determined that 30 errors were found during the modeling phase and 12 errors were found during the construction phase that were traceable to errors not discovered in the modeling phase. What is the DRE for these two phases?

23.10. A software team delivers a software increment to end users. The users uncover eight defects during the first month of use. Prior to delivery, the software team found 242 errors during formal technical reviews and all testing tasks. What is the overall DRE for the project after 1 month's usage?

Design element: Quick Look icon magnifying glass: © Roger Pressman

PART

Four

MANAGING SOFTWARE PROJECTS

In this part of *Software Engineering: A Practitioner's Approach,* you'll learn the management techniques required to plan, organize, monitor, and control software projects. These questions are addressed in the chapters that follow:

- How must people, process, and problem be managed during a software project?
- How can software metrics be used to manage a software project and the software process?
- How does a software team generate reliable estimates of effort, cost, and project duration?
- What techniques can be used to systematically assess the risks that can have an impact on project success?
- How does a software project manager select the set of software engineering work tasks?
- How is a project schedule created?
- Why are maintenance and support so important for both software engineering managers and practitioners?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product constrained by the available resources.

CHAPTER

24 PROJECT MANAGEMENT CONCEPTS

In the preface to his book on software project management, Meiler Page-Jones [Pag85] comments on software projects that are not going well, "I've watched in horror as . . . managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time."

KEY CONCEPTS

agile teams 495
coordination and communication 496
critical practices 502
people 491
problem decomposition 497
product. 491
project 492
software scope 497
software team 494
stakeholders 493
team leaders 493
W^{5}HH principle 501

QUICK LOOK

What is it? Although many of us (in our darker moments) take Dilbert's[1] view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and coordinating of people, processes, and events that occur as software evolves from a preliminary concept to full operational deployment.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies among people involved in a software project.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four Ps—people, product, process, and project. People must be organized to perform software work effectively. Product scope and requirements must be understood. A process that is appropriate for the people and the product should be selected. The project must be planned by estimating effort and calendar time to accomplish work tasks. This is true even for agile projects management.

What is the work product? A project plan is created and evolves as project activities commence. The plan is a living document that defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until the team has delivered a high-quality product on time and within budget. However, a team leader does it right when she encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

1 Try searching for the term *management* on the Dilbert website: http://dilbert.com/.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a postmortem were to be conducted for every project, it is very likely that a consistent theme would be encountered: project management was weak or nonexistent.

In this chapter and Chapters 25 through 27, we'll present the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 25 discusses the techniques that are used to estimate costs and create realistic (but flexible) schedules. Chapter 26 presents the management activities that lead to effective risk monitoring, mitigation, and management. Chapter 27 considers product support concerns and discusses the management issues that you'll encounter when dealing with maintenance of deployed systems. Finally, Chapter 28 discusses techniques for studying and improving your team's software engineering processes.

24.1 The Management Spectrum

Effective software project management focuses on the four Ps: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who begins work without a solid plan jeopardizes the success of the project. The manager who is not ready to revise the plan when changes arise is doomed to fail.

24.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives" [Cur09].

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, and team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

24.1.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering (Chapter 7). Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. These often take the form of user stories and formal use cases. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

24.1.3 The Process

A software process (Chapters 2 through 4) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. Even agile developers follow a change-friendly process (Chapter 3) to impose some discipline on their software engineering work. A number of task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

24.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.[2]

To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project [Gha14]. Each of these issues is discussed in Section 24.5 and in the chapters that follow.

---

2 Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, we think, is that a substantial number of these "failed" projects are ill conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

24.2 PEOPLE

People build computer software, and projects succeed because well-trained, motivated people get things done. All of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue that people are primary, but their actions sometimes belie their words. In this section, we examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering.

24.2.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. **Senior managers (product owners)** who define the business issues that often have a significant influence on the project.
2. **Project (technical) managers (Scrum masters or team leads)** who must plan, motivate, organize, and coordinate the practitioners who do software work.
3. **Practitioners** who deliver the technical skills that are necessary to engineer a product or application.
4. **Customers** who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. **End users** who interact with the software once it is released for production use.

Every software project is populated by people who fall within this taxonomy.[3] To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

24.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers" [Edg95]. Shared leadership often helps teams perform better, but team leaders often monopolize decision-making authority and fail to provide team members with the levels of autonomy needed to complete their tasks [Hoe16].

James Kouzes has been writing about effective leadership in various technical areas for many years. He lists five practices found in exemplary technology leaders [Kou14]:

> **Model the way.** Leaders must practice what they preach. They demonstrate commitment to the team and project through shared sacrifice (e.g., by being the last one to go home each night or taking the time to become an expert on the software application).
>
> **Inspire and shared vision.** Leaders recognize that they cannot lead without followers. It is important to motivate team members to tie their personal

3 When WebApps, MobileApps, or games are developed, other nontechnical people may be involved in content creation.

aspirations to the team goals. This means involving stakeholders early in the goal-setting process.

Challenge the process. Leaders must take the initiative to look for innovative ways to improve their own work and the work of their teams. Encourage team members to experiment and take risks by helping them generate frequent small successes while learning from their failures.

Enable others to act. Foster the team's collaborative abilities by building trust and facilitating relationships. Increase the team's sense of competence through sharing decision making and goal setting.

Encourage the heart. Celebrate the accomplishments of individuals. Build community (team) spirit by celebrating shared goals and victories, both inside and outside the team.

Another way of looking at successful project leaders might be to suggest that they adopt a problem-solving management style. A software project manager should concentrate on understanding the problem to be solved, coordinate the flow of ideas from stakeholders, and let everyone on the team know (by words and, far more important, by actions) that quality begins with each one of them and that their input and contributions are valued.

24.2.3 The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concerns with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes seven project factors that should be considered when planning the structure of software engineering teams: (1) difficulty of the problem to be solved, (2) "size" of the resultant program(s) in lines of code or function points, (3) time that the team will stay together (team lifetime), (4) degree to which the problem can be modularized, (5) quality and reliability of the system to be built, (6) rigidity of the delivery date, and (7) degree of sociability (communication) required for the project.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book *Peopleware,* DeMarco and Lister [DeM98] look for teams that "jell." They write:

> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

But not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built and the people doing the work and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[4] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

Many software organizations advocate agile software development (Chapter 3) as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.

The small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems [Hoe16]. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write:

> If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job . . .

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing.*

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time.

To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day. Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

4 Technical reviews are discussed in detail in Chapter 16.

24.2.4 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

SAFEHOME



Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold on that for a moment. I'd like to talk about how we are going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and the fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

Doug: And, we're going to use UML as our modeling approach.

Vinod: But keep extraneous documentation to an absolute minimum.

Doug: Who is the liaison with me?

Jamie: We decided that Vinod will be the tech lead—he's got the most experience, so Vinod is your liaison, but feel free to talk to any of us.

Doug (laughing): Don't worry, I will.

24.3 PRODUCT

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide information necessary for estimates, but analysis often takes weeks or even months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed now!

Like it or not, you must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

24.3.1 Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described. Even in the most fluid situations, the number of prototypes needs to be considered and the scope of the first prototype needs to be set.

24.3.2 Problem Decomposition

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapters 7 and 8). During the scoping activity, no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality and content (information) that must be delivered and (2) the process that will be used to deliver it. This can be accomplished using a list of functions or with use cases or for agile work, user stories.

Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 25). Because both cost and schedule estimates are functionally oriented,

some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software.

24.4 PROCESS

The framework activities (Chapter 1) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by your project team. The recommended process model in Chapter 4 may be a good starting point for many project teams to consider.

Your team must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities, must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 25.

24.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization. The process framework establishes a skeleton for project planning. It is adapted by allocating a task set that is appropriate to the project. Assume that the organization has adopted the generic framework activities—communication, planning, modeling, construction, and deployment—discussed in Chapter 1.

The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 24.1 is created. Each major product function (the figure lists functions for the fitness app software discussed in Chapter 2) or user story is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.[5] The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 25.

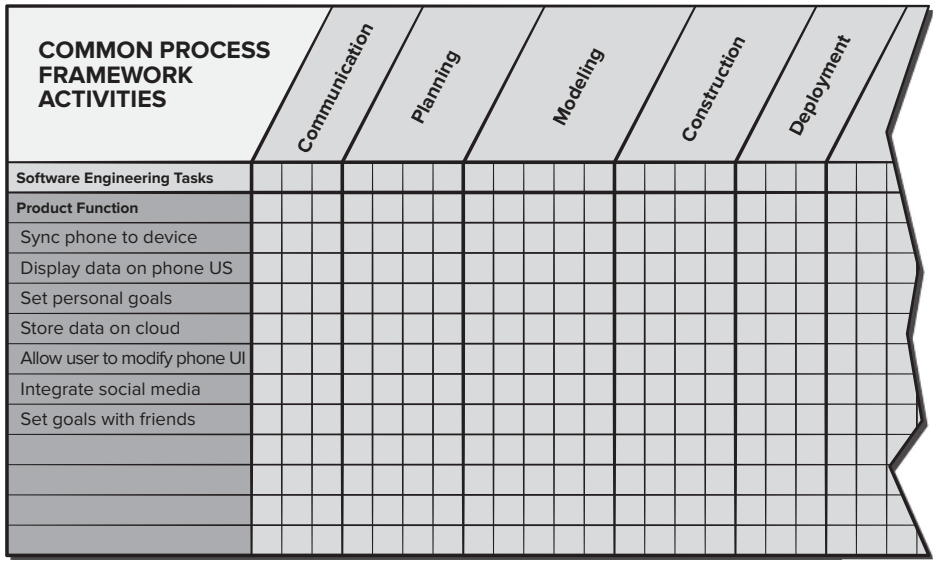
24.4.2 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that

5 It should be noted that work tasks must be adapted to the specific needs of the project based on a number of adaptation criteria.

FIGURE 24.1

Melding the problem and the process



populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using a single sprint approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, or potential for significant component reuse) will lead to the selection of other process models.[6]

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?" For example, a small, relatively simple project might require the following work tasks for the communication activity:

1. Develop a list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope by listing the user stories.
4. Review the statement of scope with all concerned, and determine the importance of each user story to the stakeholders.
5. Modify the statement of scope as required.

6 Recall that project characteristics also have a strong bearing on the structure of the software team (Section 24.2.3).

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the **communication:**

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software. This is often done by developing use cases that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the collection of use cases with all concerned, and determine their relative importance to all stakeholders.
10. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call **communication,** but the first project team performs half as many software engineering work tasks as the second.

24.5 PROJECT

To manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines signs that indicate that an information systems project is in jeopardy. In some cases, software people don't understand their customer's needs. This leads to a project with a poorly defined scope. In other projects, changes are managed poorly. Sometimes the chosen technology changes or business needs shift and management sponsorship is lost. Management can set unrealistic deadlines or end users can be resistant to the new system. There are cases in which the project team simply does not have the requisite skills. And finally, there are developers who never seem to learn from their mistakes.

Jaded industry professionals often refer to the "90–90 rule" when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time [Zah94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding paragraph.

But enough negativity! What are the characteristics of successful software projects? Ghazi [Gha14] and her colleagues note several characteristics that are present in successful software projects and also found in most well-designed process models.

1. Clear and well-understood requirements accepted by all stakeholders
2. Active and continuous participation of users throughout the development process
3. A project manager with required leadership skills who is able to share project vision with the team
4. A project plan and schedule developed with stakeholder participation to achieve user goals
5. Skilled and engaged team members
6. Development team members with compatible personalities who enjoy working in a collaborative environment
7. Realistic schedule and budget estimates which are monitored and maintained
8. Customer needs that are understood and satisfied
9. Team members who experience a high degree of job satisfaction
10. A working product that reflects desired scope and quality

24.6 THE W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [Boe96] states: "[Y]ou need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the *W^{5}HH Principle,* after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

Why is the system being developed? All stakeholders should assess the validity of business reasons for the software work. Does the business purpose justify the expenditure of people, time, and money?

What will be done? The task set required for the project is defined.

When will it be done? The team establishes a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

Who is responsible for a function? The role and responsibility of each member of the software team is defined.

Where are they located organizationally? Not all roles and responsibilities reside within software practitioners. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 25) based on answers to earlier questions.

Boehm's W^{5}HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide you and your team with an excellent planning outline.

24.7 CRITICAL PRACTICES

The Airlie Council[7] has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [Air99]. These practices are still applicable to modern performance-based management of all software projects [All14].

Critical practices[8] include: metric-based project management (Chapter 23), empirical cost and schedule estimation (Chapter 25), earned value tracking (Chapter 25), defect tracking against quality targets (Chapters 19 through 21), and people-oriented management (Chapter 24). Each of these critical practices is addressed throughout Part Four of this book.

24.8 SUMMARY

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the modeling, construction, and deployment of computer software.

Four Ps have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. Product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, technical reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. Each of these topics is considered in the chapters that follow.

---

7 The Airlie Council was comprised of a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering.

8 Only those critical practices associated with "project integrity" are noted here.

PROBLEMS AND POINTS TO PONDER

24.1. Based on information contained in this chapter and your own experience, develop "10 commandments" for empowering software engineers. That is, make a list of 10 guidelines that will lead to software people who work to their full potential.

24.2. The Software Engineering Institute's People Capability Maturity Model (People-CMM) takes an organized look at "key practice areas" (KPAs) that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

24.3. Describe three real-life situations in which the customer and the end user are the same. Describe three situations in which they are different.

24.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

24.5. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

24.6. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team structure would you choose and why? What software process model(s) would you choose and why?

24.7. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next-generation version of its widely used mobile fitness app. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

24.8. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal is to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

24.9. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

24.10. What, in your opinion, is the most important aspect of people management for a software project?

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

25 CREATING A VIABLE SOFTWARE PLAN

Software project management begins with a set of activities that are collectively called *project planning*. Before the project can begin, the software team estimates the work to be done, the resources that will be required, and the time that will elapse from start to finish. Once these activities are accomplished, the software team should establish a project schedule that defines software engineering tasks and milestones, identifies who is responsible for conducting each task, and specifies the intertask dependencies that may have a strong bearing on progress.

KEY CONCEPTS

agile development 519
critical path 520
effort 509
estimation,
  agile projects 519
  decomposition techniques 510
  empirical models 510
  FP-based 514
  problem-based 512
  process-based 515
  reconciliation 518
  techniques 511
  use case points (UCPs) 517
people and effort 522
principles 521
project planning 504
resources 507
software scope 507
software sizing 511
task network 525
time-boxing 528
time-line charts 526
tracking 528
work breakdown 526

QUICK LOOK

What is it? Software project planning encompasses five major activities—estimation, scheduling, risk analysis, quality management planning, and change management planning.

Who does it? Software project managers and other members of the software team.

Why is it important? You need to assess the tasks to perform, and the time line for the work to be conducted. Many software engineering tasks must occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without creating a schedule.

What are the steps? Software engineering activities and tasks are refined to accommodate the functions and constraints imposed by project scope. The problem is decomposed, and estimation, risk analysis, and scheduling occur.

What is the work product? An adaptable plan containing a simple table delineating the tasks to be performed, the functions to be implemented, and the cost, effort, and time involved for each is generated. A project schedule is also created based on this information.

How do I ensure that I've done it right? That's hard, because you won't really know until the project has been completed. However, if you use a systematic planning approach, you can feel confident that you've given it your best shot.

There was once a bright-eyed young engineer who was chosen to develop some code for an automated manufacturing application. The reason for his selection was simple. He was the only person in his group who knew the ins and outs of the manufacturing controller, but at the time he knew nothing about software engineering and even less about project scheduling and tracking.

His boss informed the young engineer that the project had to be completed in 2 months. He considered his approach and began writing code. After 2 weeks, the boss called him into his office and asked how things were going.

"Really great," said the young engineer with youthful enthusiasm. "This is much simpler than I thought. I'm probably close to 75 percent finished."

The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time.

A week later the boss called the engineer into his office and asked, "Where are we?"

"Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon."

"How does the deadline look?" the boss asked.

"No problem," said the engineer. "I'm close to 90 percent complete."

If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer[1] stayed 90 percent complete for the entire project duration and finished (with the help of others) only 1 month late.

This story has been repeated hundreds of thousands of times by software developers during the past five decades. The big question is why.

25.1 Comments on Estimation

Planning requires you to make an initial commitment, even though it's likely that this "commitment" will be proven wrong. Whenever estimates are made, you have to look into the future and accept some degree of uncertainty as a matter of course.

Estimating is as much art as it is science, and it should not be conducted in a haphazard manner. Because estimation lays a foundation for all other project planning actions, and project planning provides the road map for successful software engineering, we would be ill-advised to embark without it.

Estimation of resources, cost, and schedule for software development requires experience, access to good historical information (e.g., process and product metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk,[2] and this risk leads to uncertainty. Project complexity, project size, and the degree of structural uncertainty all affect the reliability of estimates.

Project complexity has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past efforts. The first-time developer of a sophisticated e-commerce application might

1 In case you were wondering, this story is autobiographical (RSP).

2 Systematic techniques for risk analysis are presented in Chapter 26.

consider it to be exceedingly complex. However, a Web engineering team developing its tenth e-commerce WebApp would consider such work run of the mill. A number of quantitative software complexity measures have been proposed [Zus97], but they are rarely used in real-world projects. However, other, more subjective assessments of complexity (e.g., function point complexity adjustment factors described in Section 25.6) can be established early in the planning process.

Project size is another important factor that can affect the accuracy and efficacy of estimates. As size increases, the interdependency among various elements of the software grows rapidly.[3] Problem decomposition, an important approach to estimating, becomes more difficult because the refinement of problem elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail.

The *degree of structural uncertainty* also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information that must be processed.

The availability of historical information has a strong influence on estimation risk. By looking back, you can emulate things that worked and improve areas where problems arose. When comprehensive software metrics (Chapter 23) are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high. As a planner, you and the customer should recognize that variability in software requirements means instability in cost and schedule.

However, you should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible to revisit estimates (as more information is known) and revise them when stakeholders make changes to requirements or schedules.

25.2 THE PROJECT PLANNING PROCESS

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of a project planning task set. Therefore, the plan must be adapted and updated as the project proceeds. In the following sections, each of the activities associated with the software project planning task set is discussed.

3 Size often increases due to "scope creep" that occurs when problem requirements change. Increases in project size can have a geometric impact on project cost and schedule (Michael Mah, personal communication).

TASK SET



Task Set for Project Planning

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks (Chapter 26).
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.
5. Estimate cost and effort.
   a. Decompose the problem.
   b. Develop two or more estimates using size, function points, process tasks, or use cases.
   c. Reconcile the estimates.
6. Develop an initial project schedule (Section 25.11).
   a. Establish a meaningful task set.
   b. Define a task network.
   c. Use scheduling tools to develop a time-line chart.
   d. Define schedule tracking mechanisms.
7. Repeat steps 1 to 6 to create a detailed schedule for each prototype as the scope of each prototype is defined.

25.3 SOFTWARE SCOPE AND FEASIBILITY

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that *bound* the system. Scope can be defined by developing a set of use cases[4] that is developed with the end users.

Functions described in the use cases are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations often constrain processing and response-time requirements.

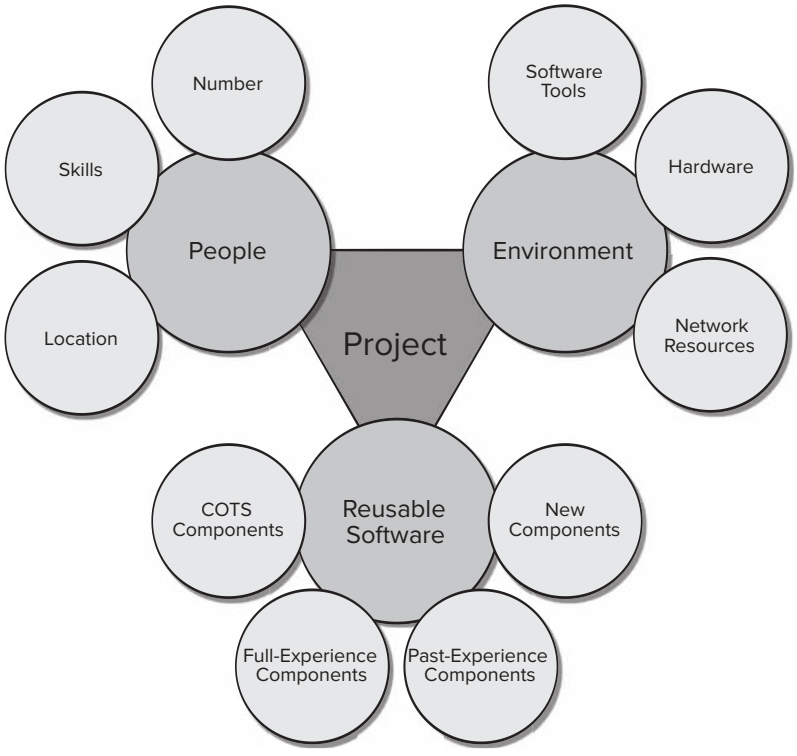
Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?" All too often, software engineers rush past these questions (or are pushed past them by impatient managers or other stakeholders), only to become mired in a project that is doomed from the onset. You must try to determine if the system can be created using available technology, dollars, time, and other resources. Project feasibility is important, but a consideration of business need is even more important. It does no good to build a high-tech system or product that no one wants.

25.4 RESOURCES

Once scope is defined, you must estimate the resources required to build software that will implement the set of use cases that describe software features and functions. Figure 25.1 depicts the three major categories of software engineering resources—people,

4 Use cases have been discussed in detail throughout Part Two of this book. A use case is a scenario-based description of the user's interaction with the software from the user's point of view.

FIGURE 25.1
Project resources



reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a *time window*. Availability of the resource for a specified window must be established at the earliest practical time.

25.4.1 Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, e-commerce) are specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified.

The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

25.4.2 Reusable Software Resources

Component-based software engineering (CBSE)[5] emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur. It is also important to consider whether it would be less costly to buy an existing software product (assuming it satisfies all stakeholder needs) than building a custom software product from scratch.

25.4.3 Environmental Resources

The environment that supports a software project, often called the *software engineering environment* (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.[6] Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.

25.5 Data Analytics and Software Project Estimation

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).

5 CBSE was considered briefly in Chapter 11.

6 Other hardware—the *target environment*—is the computer on which the software will execute when it has been released to the end user.

2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion.

An *empirical estimation model* for computer software uses formulas derived from existing project data to predict effort as a function of things like LOC or FP.[7] Values for LOC or FP are estimated using the approach described in Sections 25.6.3 and 25.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model [Whi15].

A typical empirical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

$$E = A + B \times (e_v)^C \tag{25.1}$$

where A, B, and C are empirically derived constants, E is effort in person-months, and e_v is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (25.1), the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).

Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form

$$d = f(v_i)$$

where d is one of a number of estimated values (e.g., effort, cost, project duration) and v_i are selected independent parameters (e.g., estimated lines of code). The empirical data that support most software estimation models are derived from a limited

7 An empirical model using use cases as the independent variable is suggested in Section 25.6.6. However, relatively few have appeared in the literature to date.

sample of projects.[8] For this reason, no estimation model is appropriate for all classes of software and in all development environments.

Ideally, any estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

Each of the software estimation methods is only as good as the historical data used to seed the estimate. If no historical data exist, the estimates rest on a very shaky foundation. Therefore, you should use the results obtained from such models judiciously. In Chapter 23, we examined the characteristics of some of the software metrics or data analytics that provide the basis for historical estimation data. Software data analytics concepts are discussed briefly in Appendix 2 of this book.

25.6 DECOMPOSITION AND ESTIMATION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

In Chapter 24, the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

25.6.1 Software Sizing

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which you have properly estimated the size of the product to be built, (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects), (3) the degree to which the project plan reflects the abilities of the software team, and (4) the stability of product requirements and the environment that supports the software engineering effort.

Because a project estimate is only as good as the estimate of the size of the work to be accomplished, *software sizing* represents your first major challenge as a planner. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP). Size can be estimated by considering the type of project and its application domain, the functionality delivered (i.e., the number of function points), the number of components (or use cases) to be delivered, and the degree to which a set of existing components must be modified for the new system.

8 As an example, the COCOMO (Constructive Cost Model) was originally developed in 1981, with updated versions, COCOMO II and COCOMO III, released in later years. A presentation on the genesis of the COCOMO model can be downloaded from: http://www.psmsc.com/UG2016/Presentations/p10-Clark-COCOMO%20III%20Presentation%20v1.pdf.

25.6.2 Problem-Based Estimation

In Chapter 23, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm)[9] are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project. When collecting productivity metrics for projects, be sure to establish a taxonomy of project types. This will enable you to compute domain-specific averages, making estimation more accurate. Many modern applications reside on a network or are part of a client-server architecture. Therefore, be sure that your estimates include the effort required to develop "infrastructure" software.

25.6.3 An Example of LOC-Based Estimation

As an example of an LOC estimation technique, we consider a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on a notebook computer. A preliminary statement of software scope can be developed:

> The mechanical CAD software will accept two- and three-dimensional geometric data from a designer. The designer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of devices. The software will be designed to control and interact with peripheral devices that include a touchpad, scanner, laser printer, and large-bed digital plotter.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin, the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

For our purposes, assume that further refinement has occurred and that the major software functions listed in Figure 25.2 are identified. Following the decomposition technique for LOC, an estimation table (Figure 25.2) is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC;

9 The acronym *pm* means person-month of effort.

FIGURE 25.2

Estimation table for the LOC methods

Function	Estimated LOC
User interface and control facilities (UICF)	2300
Two-dimensional geometric analysis (2DGA)	5300
Three-dimensional geometric analysis (3DGA)	6800
Database management (DBM)	3350
Computer graphics display facilities (GCDF)	4950
Peripheral control function (PCF)	2100
Design analysis modules (DAM)	8400
Estimated lines of code	33200

and pessimistic, 8600 LOC. Applying Equation (25.1), the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33200 lines of code is established for the CAD system.

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of $8,000 per month, the cost per line of code is approximately $13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $431,000 and the estimated effort is 54 person-months.[10] Do not succumb to the temptation to use this result as your project estimate. You should derive another result using a different approach.

SAFEHOME



Estimating

The scene: Doug Miller's office as project planning begins.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: We need to develop an effort estimate for the project, and then we've got to define a micro schedule for the first increment and a macro schedule for the remaining increments.

Vinod (nodding): Okay, but we haven't defined any increments yet.

Doug: True, but that's why we need to estimate.

Jamie (frowning): You want to know how long it's going to take us?

Doug: Here's what I need. First, we need to functionally decompose the *SafeHome* software . . . at a high level . . . then we've got to estimate the number of lines of code that each function will take . . . then . . .

10 Estimates are rounded to the nearest $1,000 and person-month. Further precision is unnecessary and unrealistic, given the limitations of estimation accuracy.

Jamie: Whoa! How are we supposed to do that?

Vinod: I've done it on past projects. You begin with use cases, determine the functionality required to implement each, then guesstimate the LOC count for each piece of the function. The best approach is to have everyone do it independently and then compare results.

Doug: Or you can do a functional decomposition for the entire project.

Jamie: But that'll take forever and we've got to get started.

Vinod: No . . . it can be done in a few hours . . . this morning, in fact.

Doug: I agree . . . we can't expect exactitude, just a ballpark idea of what the size of *SafeHome* will be.

Jamie: I think we should just estimate effort . . . that's all.

Doug: We'll do that too. Then use both estimates as a cross-check.

Vinod: Let's go do it . . .

25.6.4 An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to Table 25.1, you would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. To compute the *count total* needed in the FP equation:

$$\text{FP}_{\text{estimated}} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)]$$

For the purposes of this estimate, the complexity weighting factor is assumed to be average. Table 25.1 presents the results of this estimate, and the FP count total is 320.

To compute a value for $\Sigma(F_i)$, each of the 14 complexity weighting factors listed in Table 25.2 is scored with a value between 0 (not important) and 5 (very important).

The sum of these ratings for the complexity factors $\Sigma(F_i)$ is 52. So the value of the adjustment factor is 1.17:

$$[0.65 + 0.01 \times \Sigma(F_i)] = 1.17$$

Finally, the estimated number of FP is derived:

$$\text{FP}_{\text{estimated}} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] = 375$$

TABLE 25.1

Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	96 (24 × 4 = 96)
Number of external outputs	12	14	22	14	5	70 (14 × 5 = 70)
Number of external inquiries	16	20	28	20	5	100 (20 × 5 = 100)
Number of internal logical files	4	4	5	4	10	40 (4 × 10 = 40)
Number of external interface files	2	2	3	2	7	14 (2 × 7 = 14)
Count total						*320*

TABLE 25.2
Estimating information domain values

Complexity Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of $8,000 per month, the cost per FP is approximately $1,230. Based on the FP estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

25.6.5 An Example of Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of activities, actions, and tasks and the effort required to accomplish each is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities[11] may be represented as part of a table similar to the one presented in Figure 25.3.

Once problem functions and process activities are melded, you estimate the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 25.3. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity.

11 The framework activities chosen for this project differ somewhat from the generic activities discussed in Chapter 2. They are customer communication (CC), planning, risk analysis, engineering, and construction/release.

FIGURE 25.3 Process-based estimation table

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

To illustrate the use of process-based estimation, we again consider the CAD software introduced in Section 25.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

Referring to the completed process-based table shown in Figure 25.3, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that approximately 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of $8,000 per month, the total estimated project cost is $368,000 and the estimated effort is 46 person-months. If desired, labor rates could be associated with each framework activity or software engineering task and computed separately.

25.6.6 An Example of Estimation Using Use Case Points

As we have noted throughout Part Two of this book, use cases provide a software team with insight into software scope and requirements. Once use cases have been developed, they can be used to estimate the projected "size" of a software project. Use cases do not address the complexity of the functions and features that are described, and they can describe complex behavior (e.g., interactions) that involve many functions and features. Even with these constraints, it is possible to compute *use case points* (UCPs) in a manner that is analogous to the computation of functions points (Section 25.6).

Cohn [Coh05] indicates that the computation of use case points must take the following characteristics into account:

- The number and complexity of the use cases in the system.
- The number and complexity of the actors on the system.
- Various nonfunctional requirements (such as portability, performance, maintainability) that are not written as use cases.
- The environment in which the project will be developed (e.g., the programming language, the software team's motivation).

To begin, each use case is assessed to determine its relative complexity. A simple use case indicates a simple user interface, a single database, and three or fewer transactions and five or fewer class implementations. An average use case indicates a more complex UI, two or three databases, and four to seven transactions with 5 to 10 classes. Finally, a complex use case implies a complex UI with multiple databases, using eight or more transactions and 11 or more classes. Each use case is assessed using these criteria and the count of each type is weighted by a factor of 5, 10, and 15, respectively. A total *unadjusted use case weight* (UUCW) is the sum of all weighted counts [Nun11].

Next, each actor is assessed. Simple actors are automatons (another system, a machine or device) that communicate through an API. Average actors are automatons that communicate through a protocol or a data store, and complex actors are humans who communicate through a GUI or other human interface. Each actor is assessed using these criteria, and the count of each type is weighted by a factor of 1, 2, and 3, respectively. The total *unadjusted actor weight* (UAW) is the sum of all weighted counts.

These unadjusted values are modified by considering technical complexity factors (TCFs) and environment complexity factors (ECFs). Thirteen factors contribute to an assessment of the final TCF, and eight factors contribute to the computation of the final ECF [Coh05]. Once these values have been determined, the final UCP value is computed in the following manner:

$$\text{UCP} = (\text{UUCW} + \text{UAW}) \times \text{TCF} \times \text{ECF} \tag{25.2}$$

The CAD software introduced in Section 25.6.3 is composed of three subsystem groups: user interface subsystem (includes UICF), engineering subsystem group (includes the 2DGA, 3DGA, and DAM subsystems), and infrastructure subsystem group (includes CGDF and PCF subsystems). Sixteen complex use cases describe the

user interface subsystem. The engineering subsystem group is described by 14 average use cases and 8 simple use cases. And the infrastructure subsystem is described with 10 simple use cases. Therefore,

$$\text{UUCW} = (16 \text{ use cases} \times 15) + [(14 \text{ use cases} \times 10) + (8 \text{ use cases} \times 5)] + (10 \text{ use cases} \times 5) = 470$$

Analysis of the use cases indicates that there are 8 simple actors, 12 average actors, and 4 complex actors. Therefore,

$$\text{UAW} = (8 \text{ actors} \times 1) + (12 \text{ actors} \times 2) + (4 \text{ actors} \times 3) = 44$$

After evaluation of the technology and the environment,

$$\text{TCF} = 1.04$$

$$\text{ECF} = 0.96$$

Using Equation (25.2),

$$\text{UCP} = (470 + 44) \times 1.04 \times 0.96 = 513$$

Using past project data as a guide, the development group has produced 85 LOC per UCP. Therefore, an estimate of the overall size of the CAD project is 43600 LOC. Similar computations can be made for applied effort or project duration.

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of $8,000 per month, the cost per line of code is approximately $13. Based on the use case estimate and the historical productivity data, the total estimated project cost is $552,000 and the estimated effort is about 70 person-months.

25.6.7 Reconciling Estimates

Any estimation technique, no matter how sophisticated, must be checked by computing at least one other estimate using a different approach. If you have created two or three estimates independently, you now have two or three estimates for cost and effort that need to be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

When your estimates are far apart, you need to reevaluate the information used to make the estimates. Widely divergent estimates can often be traced to one of two causes: (1) the scope of the project is not adequately understood or has been misinterpreted by the planner, or (2) productivity data used for problem-based estimation techniques is inappropriate for the application or has been misapplied. You should determine the cause of divergence and then recompute these estimates.

The estimation techniques discussed in the preceding sections resulted in multiple estimates that should be reconciled to produce a single estimate of effort, project duration, or cost. The total estimated effort for the CAD software (Section 25.6.3) ranges from a low of 46 person-months (derived using a process-based estimation approach) to a high of 68 person-months (derived with use case estimation). The simple average of all four estimates is 56 person-months. But is this the best approach when the high and low estimates are 21 person-months apart?

One approach may be to compute a weighted average, based on calling a high estimate a pessimistic estimate, a low estimate an optimistic estimate, and an in-between value a most likely value. A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates. For example,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6} \tag{25.3}$$

gives heaviest credence to the "most likely" estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical productivity data should be examined. Do our estimates seem correct? The only reasonable answer to this question is, we can't be sure. Even then, common sense and experience must prevail.

25.6.8 Estimation for Agile Development

Because the requirements for an agile project (Chapter 3) are defined by a set of user stories, it is possible to develop an estimation approach that is informal, reasonably disciplined, and meaningful within the context of project planning for each software increment. Estimation for agile projects uses a decomposition approach that encompasses the following steps:

1. Each user story (the equivalent of a mini use case created at the very start of a project by end users or other stakeholders) is considered separately for estimation purposes.

2. The user story is decomposed into the set of software engineering tasks that will be required to develop it.

3a. Each task is estimated separately. Note: Estimation can be based on historical data, an empirical model, or "experience" (e.g., using a technique like planning poker, Section 7.2.3).

3b. Alternatively, the "volume" of the user story can be estimated in LOC, FP, or some other volume-oriented measure (e.g., use case count).

4a. Estimates for each task are summed to create an estimate for the user story.

4b. Alternatively, the volume estimate for the user story is translated into effort using historical data.

5. The effort estimates for all user stories that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

Because the project duration required for the development of a software increment is quite short (typically 3 to 6 weeks), this estimation approach serves two purposes: (1) to be certain that the number of scenarios to be included in the increment conforms to the available resources, and (2) to establish a basis for allocating effort as the increment is developed.

25.7 PROJECT SCHEDULING

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled.

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes:

- An unrealistic deadline established by someone outside the software team and forced on managers and practitioners on the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Aggressive (read "unrealistic") deadlines are an unpleasant fact in the software business. Sometimes such deadlines are demanded for reasons that are legitimate, from the point of view of the person who sets the deadline. But common sense says that legitimacy must also be perceived by the people doing the work.

The estimation methods discussed in this chapter and the scheduling techniques described in this section are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should inform management and all stakeholders of her findings and suggest alternatives to mitigate the damage of missing the deadline.

The reality of a technical project (whether it involves building a virtual world for a video game or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about the impact on the project completion date. Other tasks lie on the *critical path*. If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

As a project manager, your objective is to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress to ensure that delay is recognized "one day at a time." To accomplish this, you must have a schedule that has been defined at a degree of resolution that allows progress to be monitored and the project to be controlled. The tasks required to achieve a project manager's needs to build a schedule

and track progress should not be performed manually. There are many excellent scheduling tools. A good manager uses them.

25.7.1 Basic Principles

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort).[12] On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved.

Each of these principles is applied as the project schedule evolves.

12 In reality, less than 3 person-days of effort are available because of unrelated meetings, sickness, vacation, and a variety of other reasons. For our purposes, however, we assume 100 percent availability.

25.7.2 The Relationship Between People and Effort

There is a common myth that is still believed by many managers who are responsible for software development work: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.

In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time. If you must add people to a late project, be sure that you've assigned them work that is highly compartmentalized.

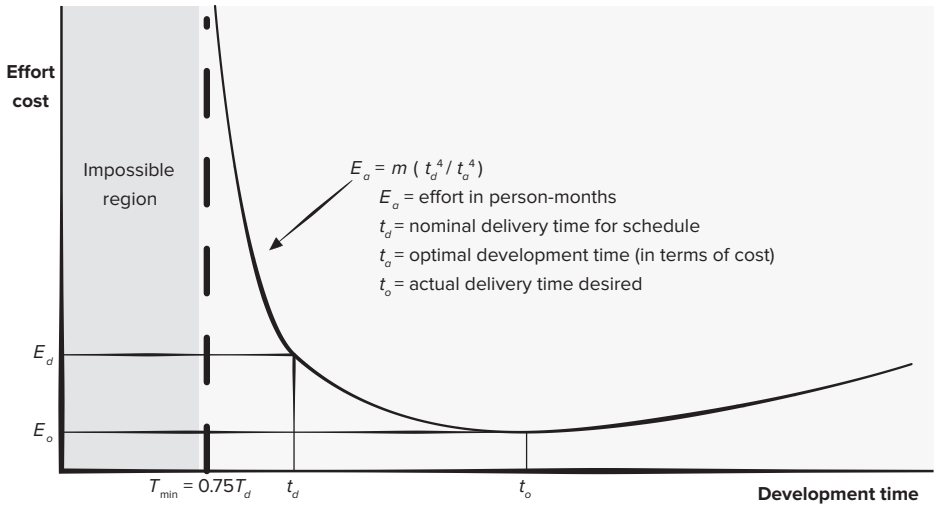
Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources).

The *Putnam-Norden-Rayleigh (PNR) curve*[13] provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 25.4. The curve indicates a minimum value t_o that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of t_o (i.e., as we try to accelerate delivery), the curve rises nonlinearly.

As an example, we assume that a project team has estimated a level of effort E_d will be required to achieve a nominal delivery time t_d that is optimal in terms of

FIGURE 25.4

The relationship between effort and delivery time



13 Original research can be found in [Nor70] and [Put78].

schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of t_d. In fact, the PNR curve indicates the project delivery time cannot be compressed much beyond $0.75t_d$. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option $t_o = 2t_d$. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.

The *software equation* [Put92] introduced is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), *L,* is related to effort and development time by the equation:

$$L = P \times E^{1/3} t^{4/3} \tag{25.4}$$

where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that leads to high-quality software engineering work (typical values for P range between 2000 and 12000), and t is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort E:

$$E = \frac{L^3}{P^3 t^4} \tag{25.5}$$

where E is the effort expended (in person-years) over the entire life cycle for software development and maintenance and t is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

This leads to some interesting results. As a project deadline becomes tighter and tighter, you reach a point at which the work cannot be completed on schedule, regardless of the number of people doing the work. Face reality and define a new delivery date.

Consider also a complex, real-time software project estimated at 33000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (25.5) yields:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ person-years}$$

This implies that, by extending the end date by 6 months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: Benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

25.8 Defining a Project Task Set

Regardless of the process model that is chosen, the work that a software team performs is achieved through a set of tasks that enable you to define, develop, and ultimately support computer software. No single task set is appropriate for all projects. The set

of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product. Therefore, an effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

As we noted in Chapter 2, a task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project. The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.

To develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work. Many factors influence the task set to be chosen. These include [Pre05]: size of the project, number of potential users, mission criticality, application longevity, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, embedded and nonembedded characteristics, project staff, and reengineering factors. When taken in combination, these factors provide an indication of the *degree of rigor* with which the software process should be applied.

25.8.1 A Task Set Example

A *concept development project* is initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following task set:

1.1 **Concept scoping** determines the overall scope of the project.

1.2 **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.

1.3 **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.

1.4 **Proof of concept** demonstrates the viability of a new technology in the software context.

1.5 **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

1.6 **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these tasks should yield few surprises. In fact, the software engineering flow for concept development projects (and for all other types of projects as well) is little more than common sense.

25.8.2 Refinement of Major Tasks

The major tasks (i.e., software engineering actions) described in the preceding section may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by

taking each major task and decomposing it into a set of subtasks (with related work products and milestones).

As an example of task decomposition, consider Task 1.1, Concept Scoping. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping activity:

```
Task definition: Task 1.1 Concept Scoping
1.1.1  Identify need, benefits and potential customers;
1.1.2  Define desired output/control and input events that drive the
       application;
       Begin Task 1.1.2
       1.1.2.1 TR: Review written description of need[14]
       1.1.2.2 Derive a list of customer visible outputs/inputs
       1.1.2.3 TR: Review outputs/inputs with customer and revise as
               required;
       endtask Task 1.1.2
1.1.3  Define the functionality/behavior for each major function;
       Begin Task 1.1.3
       1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
       1.1.3.2 Derive a model of functions/behaviors;
       1.1.3.3 TR: Review functions/behaviors with customer and revise as
               required;
       endtask Task 1.1.3
1.1.4  Isolate those elements of the technology to be implemented in software;
1.1.5  Research availability of existing software;
1.1.6  Define technical feasibility;
1.1.7  Make quick estimate of size;
1.1.8  Create a Scope Definition;
endtask definition: Task 1.1
```

The tasks and subtasks noted in the process design language refinement form the basis for a detailed schedule for the concept scoping activity.

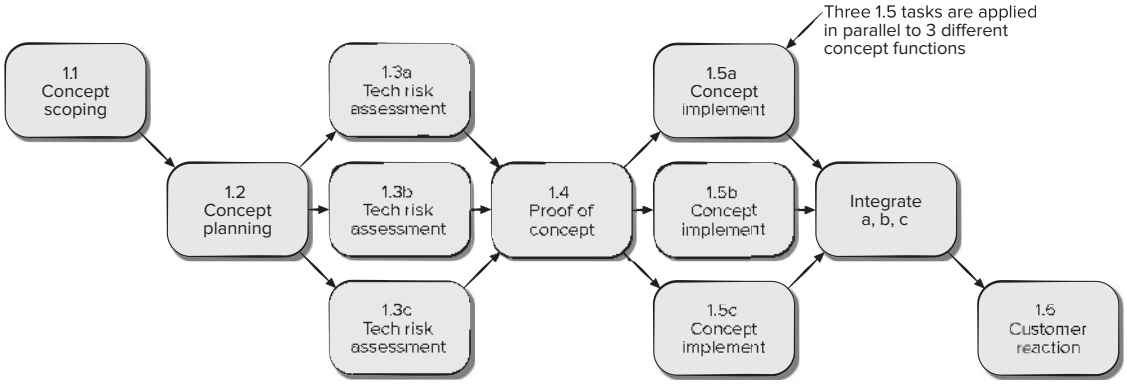
25.9 DEFINING A TASK NETWORK

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project. The task network is a useful mechanism for depicting intertask dependencies and determining the critical path. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering tasks. Figure 25.5 shows a schematic task network for a concept development project.

14 TR indicates that a technical review (Chapter 16) is to be conducted.

FIGURE 25.5 A task network for concept development



The concurrent nature of software engineering activities leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, you should determine intertask dependencies to ensure continuous progress toward completion. In addition, you should be aware of those tasks that lie on the *critical path.* That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule. These issues are discussed in more detail later in this chapter.

It is important to note that the task network shown in Figure 25.5 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each activity shown in the figure would be expanded. For example, Task 1.1 would be expanded to show all tasks detailed in the refinement of Tasks 1.1 shown in Section 25.8.2.

25.10 SCHEDULING

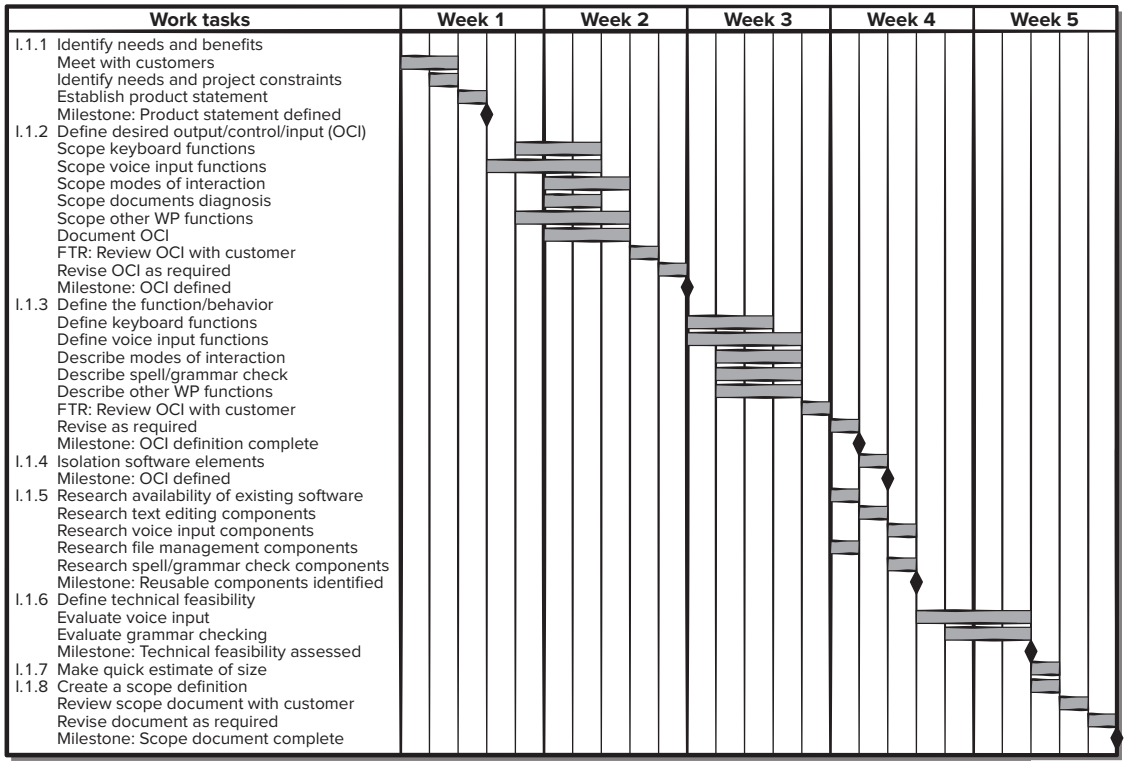
Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects [Fer14]. Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Project scheduling tools allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish "most likely" time estimates for individual tasks by applying statistical models, and (3) calculate "boundary times" that define a time "window" for a particular task [Ker17].

25.10.1 Time-Line Charts

When creating a software project schedule, you begin with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as a

FIGURE 25.6 An example time-line chart



task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.

As a consequence of this input, a *time-line chart,* also called a *Gantt chart,* is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project [Toc18].

Figure 25.6 illustrates the format of a time-line chart. It depicts a part of a software project schedule that emphasizes the **concept scoping** task for a word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information (Figure 25.7). Used in conjunction with the time-line chart, project tables enable you to track progress.

FIGURE 25.7 An example project table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope documents diagnosis	wk2, d1		wk2, d2		BLS	1.5 pd	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the Function/behavior							

25.10.2 Tracking the Schedule

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones (the diamonds shown in Figure 25.7) have been accomplished by the scheduled date.
- Comparing the actual start date to the planned start date for each project task listed in the resource table (Figure 25.8).
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Tracking the project velocity, which is a way of seeing how quickly the development team is clearing the user story backlog (Section 3.5).

In reality, all these tracking techniques are used by experienced project managers.

A software project manager employs control to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur, you must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [Jal04]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

Time-boxing is often associated with agile incremental process models (Chapter 4), and a schedule is derived for each incremental delivery. These tasks become part of the increment schedule and are allocated over the increment development schedule. They can be input to scheduling software (e.g., Microsoft *Project*) and used for tracking and control.

The initial reaction to the time-boxing approach is often negative: "If the work isn't finished, how can we proceed?" The answer lies in the way work is accomplished. By the time the time-box boundary is encountered, it is likely that 90 percent of the task has been completed.[15] The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

SAFEHOME



Tracking the Schedule

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug (glancing at a PowerPoint slide): The schedule for the first *SafeHome* increment seems reasonable, but we're going to have trouble tracking progress.

Vinod (a concerned look on his face): Why? We have tasks scheduled on a daily basis, plenty of work products, and we've been sure that we're not over allocating resources.

Doug: All good, but how do we know when the requirements model for the first increment is complete?

Jamie: Things are iterative, so that's difficult.

Doug: I understand that, but . . . well, for instance, take "analysis classes defined." You indicated that as a milestone.

Vinod: We have.

Doug: Who makes that determination?

Jamie (aggravated): They're done when they're done.

Doug: That's not good enough, Jamie. We have to schedule TRs [technical reviews, Chapter 16], and you haven't done that. The successful completion of a review on the analysis model, for instance, is a reasonable milestone. Understand?

Jamie (frowning): Okay, back to the drawing board.

Doug: It shouldn't take more than an hour to make the corrections . . . everyone else can get started now.

15 A cynic might recall the saying: "The first 90 percent of the system takes 90 percent of the time; the remaining 10 percent of the system takes 90 percent of the time."

25.11 SUMMARY

A software project planner must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved. In addition, the planner must predict the resources (hardware and software) that will be required and the risk involved.

The statement of scope helps the planner to develop estimates using one or more techniques that fall into two broad categories: decomposition and empirical modeling. Decomposition techniques require a delineation of major software functions, followed by estimates of either (1) the number of LOC, (2) selected values within the information domain, (3) the number of use cases, (4) the number of person-months required to implement each function, or (5) the number of person-months required for each software engineering activity. Empirical techniques use empirically derived expressions for effort and time to predict these project quantities. Automated tools can be used to implement a specific empirical model.

Accurate project estimates generally use at least two of the three techniques just noted. By comparing and reconciling estimates developed using different techniques, the planner is more likely to derive an accurate estimate. Software project estimation can never be an exact science, but a combination of good historical data and systematic techniques can improve estimation accuracy.

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical path, a time-line chart, and a variety of project information. Using the schedule as a guide, you can track and control each step in the software process.

PROBLEMS AND POINTS TO PONDER

25.1. Assume that you are the project manager for a company that builds software for household robots. You have been contracted to build the software for a robot that mows the lawn for a homeowner. Write a statement of scope that describes the software. Be sure your statement of scope is bounded. If you're unfamiliar with robots, do a bit of research before you begin writing. Also, state your assumptions about the hardware that will be required. Alternate: Replace the lawn-mowing robot with another problem that is of interest to you.

25.2. Do a functional decomposition of the robot software you described in Problem 25.1. Estimate the size of each function in LOC. Assuming that your organization produces 450 LOC/pm with a burdened labor rate of $7,000 per person-month, estimate the effort and cost required to build the software using the LOC-based estimation technique described in this chapter.

25.3. Develop a spreadsheet model that implements one or more of the estimation techniques described in this chapter. Alternatively, acquire one or more online models for estimation from Web-based sources.

25.4. It seems odd that cost and schedule estimates are developed during software project planning—before detailed software requirements analysis or design has been conducted. Why do you think this is done? Are there circumstances when it should not be done?

25.5. What is the difference between a macroscopic schedule and a detailed schedule? Is it possible to manage a project if only a macroscopic schedule is developed? Why?

25.6. The relationship between people and time is highly nonlinear. Using Putnam's software equation (described in Section 25.8.2), develop a table that relates number of people to project duration for a software project requiring 50000 LOC and 15 person-years of effort (the productivity parameter is 5000 and $B = 0.37$). Assume that the software must be delivered in 24 months plus or minus 12 months.

25.7. Assume that a university has contracted you to develop an online course registration system (OLCRS). First, act as the customer (if you're a student, that should be easy) and specify the characteristics of a good system. (Alternatively, your instructor will provide you with a set of preliminary requirements for the system.) Using the estimation methods discussed in this chapter, develop an effort and duration estimate for OLCRS. Suggest how you would:

a. Define parallel work activities during the OLCRS project.
b. Distribute effort throughout the project.
c. Establish milestones for the project.

25.8. Select an appropriate task set for the OLCRS project.

25.9. Define a task network for OLCRS described in Problem 25.8, or alternatively, for another software project that interests you. Be sure to show tasks and milestones and to attach effort and duration estimates to each task. If possible, use an automated scheduling tool to perform this work.

25.10. Using a scheduling tool (if available) or paper and pencil (if necessary), develop a time-line chart for the OLCRS project.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

26 RISK MANAGEMENT

In his book on risk analysis and management, Robert Charette [Cha89] notes that "risk concerns future happenings." He correctly points out that concern about today and yesterday should not be the focus, but rather poses the pivotal question: ". . . by changing our actions today, [can we] create an opportunity for a different and hopefully better situation for ourselves tomorrow." The implication is that "risk involves choice," and that introduces uncertainty in our actions.

KEY CONCEPTS

assessment 545
fuzzy logic 539
gamification 544
identification 535
proactive strategies 533
projection 538
reactive strategies 533
refinement 542
risk categories 534
risk exposure 540
risk item checklist 546
risk table 538
RMMM 546
safety and hazards 545
technical debt 533

QUICK LOOK

What is it? A risk is a potential problem for a software project—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan.

Who does it? Everyone involved in the software process—managers, software engineers, and other stakeholders—participates in risk analysis and management.

Why is it important? Think about the Scout motto: "Be prepared." Software is a difficult undertaking. Lots of things can go wrong, and frankly, many often do. Understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software project management. A project without a risk management plan can find itself in serious trouble that could have been avoided if the project team had addressed its development risks in a more systematic manner and followed its plans.

What are the steps? Recognizing what can go wrong is the first step, called "risk identification." Next, each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur. Once this information is established, risks are ranked, by probability and impact. Finally, a plan is developed to manage those risks that have high probability and high impact.

What is the work product? A risk mitigation, monitoring, and management (RMMM) plan or a set of risk information sheets is produced.

How do I ensure that I've done it right? The risks that are analyzed and managed should be derived from a thorough study of the people, the product, the process, and the project. The RMMM should be revisited as the project proceeds to ensure that risks are kept up to date. Contingency plans for risk management should be realistic.

When you consider risk in the context of software engineering, Charette's conceptual underpinnings are always in evidence. The future is your concern—what risks might cause the software project to go awry? Change is your concern—how will changes in customer requirements, development technologies, target environments, and all other entities connected to the project affect timeliness and overall success? Last, you must grapple with choices—what methods and tools should you use, how many people should be involved, how much emphasis on quality is "enough"?

Technical debt is the term used to describe costs associated with putting off activities like software documentation and refactoring. Technical debt that is not paid can result in a delivered software product with inadequate functionality, erratic behavior, poor quality, insufficient documentation, and unnecessary complexity. Technical debt implies that the costs (effort, time, and resources) of dealing with technical issues can be reduced if problems are dealt with earlier rather than later during project development. Like financial interest, technical debt increases with time because unrecognized problems introduced early compound. Accumulating technical debt is mortgaging a project's future [Fai17].

Simply moving to agile software development does not remove the need to do intentional risk management. Elbanna and Sarker [Elb16] conducted a survey of several organizations that made extensive use of agile software development practices. They found several development risks that seemed to go unmanaged in agile projects. Technical debt was likely to accumulate as developers pushed for more new code and at the same time often forgot to spend time reducing this debt. In addition, inexperienced agile teams tend to produce more defects than they might have following a more controlled development model. Agile teams may be more likely to use nonstandardized project management and testing tools. These autonomous teams may not be documenting their decision-making processes adequately. That can doom developers to repeat past errors on future projects. None of these risks is impossible to control, as long as software developers are aware of them and make plans to manage them during their time-boxed sprints.

26.1 Reactive Versus Proactive Risk Strategies

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [Tho92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would react in some heroic way.

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, most software teams continue to rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action trying to correct the problem rapidly. This is often called a *fire-fighting mode.* When this fails, "crisis management" [Cha92] takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A *proactive* risk strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Proactive risk management is one of the software engineering tools that can be used to reduce technical debt. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management.

26.2 SOFTWARE RISKS

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics: *uncertainty*—the risk may or may not happen; that is, there are no 100 percent probable risks[1]—and *loss*—if the risk becomes a reality, unwanted consequences or losses will occur [Hig95]. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. In Chapter 25, project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology can also be risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

It is extremely important to note that simple risk categorization won't always work. Some risks are simply unpredictable in advance.

1 A risk that is 100 percent probable is a constraint on the software project.

Another general categorization of risks has been proposed by Charette [Cha89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

INFO



Seven Principles of Risk Management

The Software Engineering Institute (SEI) (**www.sei.cmu.edu**) identifies seven principles that "provide a framework to accomplish effective risk management." They are:

Maintain a global perspective. View software risks within the context of a system in which it is a component and the business problem that it is intended to solve.

Take a forward-looking view. Think about the risks that may arise in the future (e.g., due to changes in the software); establish contingency plans so that future events are manageable.

Encourage open communication. If someone states a potential risk, don't discount it. If a risk is proposed in an informal manner, consider it. Encourage all stakeholders and users to suggest risks at any time.

Integrate. A consideration of risk must be integrated into the software process.

Emphasize a continuous process. The team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.

Develop a shared product vision. If all stakeholders share the same vision of the software, it is likely that better risk identification and assessment will occur.

Encourage teamwork. The talents, skills, and knowledge of all stakeholders should be pooled when risk management activities are conducted.

26.3 RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that have been presented in Section 26.2. *Generic risks* are a potential threat to every software project. *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. Although generic risks are important to consider, it's the product-specific risks that cause the most headaches. Be certain to invest the time to identify as many product-specific risks as possible. To identify product-specific risks, you

should begin by examining the project plan and the software statement of scope and then develop an answer to the question, "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- **Product size.** Risks associated with the overall size of the software to be built or modified.
- **Business impact.** Risks associated with constraints imposed by management or the marketplace.
- **Stakeholder characteristics.** Risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
- **Process definition.** Risks associated with the degree to which the software process has been defined and is followed by the development organization.
- **Development environment.** Risks associated with the availability and quality of the tools to be used to build the product.
- **Technology to be built.** Risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- **Staff size and experience.** Risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow you to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" [AFC88] are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

Several comprehensive checklists for software project risk are available on the Web (e.g., [Arn11], [NAS07]). You can use these checklists to gain insight into generic risks for software projects. In addition to the use of checklists, *risk patterns* ([Mil04], [San17]) have been proposed as a systematic approach to risk identification.

26.3.1 Assessing Overall Project Risk

So how can we determine if the software project we're working on is at serious risk? The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world [Kei98]. The questions are ordered by their relative importance to the success of a project.

1. Have top software and customer managers formally committed to support the project?

2. Are end users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and its customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end users have realistic expectations?
6. Is the project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

26.3.2 Risk Components and Drivers

The U.S. Air Force [AFC88] has published a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components—performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

- **Performance risk.** The degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- **Cost risk.** The degree of uncertainty that the project budget will be maintained.
- **Support risk.** The degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- **Schedule risk.** The degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Boehm [Boe89] suggests that negligible impact results only in inconvenience. The additional cost required to mitigate the impact is very low. Marginal impact might affect secondary mission objectives or requirements but would not impact overall mission success. Cost would be somewhat higher but manageable. Critical impact would directly affect system performance and some or all requirements and would bring mission success into question. Cost to mitigate would be high. Finally, catastrophic impact would result in mission failure. The cost of mitigation would be unacceptable.

26.4 Risk Projection

Risk projection, also called *risk estimation,* attempts to rate each risk in two ways—(1) the likelihood or probability that the risk is real and will occur and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the perceived likelihood of a risk.
2. Delineate the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.

26.4.1 Developing a Risk Table

A risk table provides you with a simple technique for risk projection.[2] A sample risk table is illustrated in Figure 26.1.

You begin by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists referenced

FIGURE 26.1 Sample risk table prior to sorting

Risk	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet exceptions	TR	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	

Impact values:
1 – catastrophic
2 – critical
3 – marginal
4 – negligible

2 The risk table can be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries.

in Section 26.3. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge.

Another possible starting point would be to look at the risk assessment table for a previous project and see which risks apply to your current project, which do not, and identify risks that should be added. Risk assessment does not always need to be re-created from scratch. Agile developers often work on similar projects and can realize the cost savings by maintaining lists of risk management procedures shared companywide.

Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented in Figure 26.1, and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged[3] to determine an overall impact value.

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

A project manager can then establish a cutoff line at some row in the table (Figure 26.1). All risks that lie above the cutoff line should be managed. The column labeled RMMM contains a pointer into a *risk mitigation, monitoring, and management plan* or, alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 26.5 and 26.6.

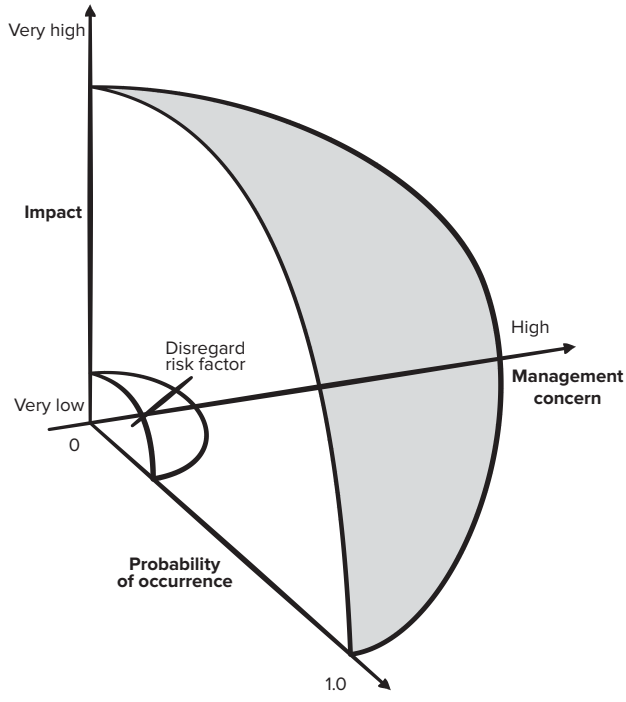
Risk probability can be determined by making individual estimates and then developing a single consensus value. Although this approach is workable, more sophisticated techniques for determining risk probability have been developed (e.g., [McC09]). Recent work makes use of *fuzzy logic*[4] to determine the characteristics that make software projects that are prone to failure. These projects often have multiple interrelated risk factors that are affected by imprecision or uncertainty and require use of expert knowledge and fuzzy logic to better understand the nature of their combined risk impact [Rod16].

Referring to Figure 26.2, risk impact and probability have a distinct influence on management concern. A risk factor that is classified as high impact but has a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

3 A weighted average can be used if one risk component has more significance for a project.

4 Fuzzy logic is a type of logic that recognizes more than simple true and false values (unlike the propositional logic you may have studied in a discrete mathematics class). With fuzzy logic, propositions can be represented with degrees of truthfulness and falsehood. For example, the statement, any animal with stripes is a tiger might only be 50% likely to be true. Fuzzy logic has proved to be particularly useful in artificial intelligence applications that involve making decisions involving incomplete or uncertain information.

FIGURE 26.2
Risk and management concern



26.4.2 Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, you want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], you can apply the following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) using the discussion presented in Section 26.3.2, determine the impact for each component based on the criteria shown, and (3) complete the risk table and analyze the results as described in the preceding sections.

The overall *risk exposure* (RE), is determined using the following relationship [Hal98]:

$$\text{RE} = P \times C$$

where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. Eighty percent (likely).

Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Because the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be $18 \times 100 \times 14 = \$25{,}200$.

Risk exposure. $RE = 0.80 \times 25{,}200 \sim \$20{,}200$.

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques described in Sections 26.4.1 and 26.4.2 are applied iteratively as the software project proceeds.[5] The project team should revisit the risk table at regular intervals, reevaluating each risk to determine when new circumstances cause its probability and impact to change. After completing this activity, the team may decide to add new risks to the table, remove some risks that are no longer relevant, or change the relative positions of still others. The team should compare the RE for all risks to their project cost estimate. If the total RE is greater than 50 percent of the project cost, the viability of the project must be questioned.

SAFEHOME



Risk Analysis

The scene: Doug Miller's office, prior to the initiation of the *SafeHome* software project.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I'd like to spend some time brainstorming risks for the *SafeHome* project.

Jamie: As in what can go wrong?

Doug: Yep. Here are a few categories where things can go wrong.

5 If you have further interest, a more mathematical treatment of the cost of risk is presented in [Ben10b].

[He shows everyone the categories noted in the introduction to Section 26.3.]

Vinod: Umm . . . do you want us to just call them out, or . . .

Doug: No, here's what I thought we'd do. Everyone make a list of risks . . . right now . . .

[Ten minutes pass, everyone is writing.]

Doug: Okay, stop.

Jamie: But I'm not done!

Doug: That's okay. We'll revisit the list again. Now, for each item on your list, assign a percent likelihood that the risk will occur. Then, assign an impact to the project on a scale of 1 (minor) to 5 (catastrophic).

Vinod: So if I think that the risk is a coin flip, I specify a 50 percent likelihood, and if I think it'll have a moderate project impact, I specify a 3, right?

Doug: Exactly.

[Five minutes pass, everyone is writing.]

Doug: Okay, stop. Now we'll make a group list on the whiteboard. I'll do the writing; we'll call out one entry from your list in round-robin format.

[Fifteen minutes pass; the list is created.]

Jamie (pointing at the board and laughing): Vinod, that risk (pointing toward an entry on the board) is ridiculous. There's a higher likelihood that we'll all get hit by lightning. We should remove it.

Doug: No, let's leave it for now. We consider all risks, no matter how weird. Later we'll winnow the list.

Jamie: But we already have over 40 risks . . . how on earth can we manage them all?

Doug: We can't. That's why we'll define a cut-off after we sort these guys. I'll do that off-line and we'll meet again tomorrow. For now, get back to work . . . and in your spare time, think about any risks that we've missed.

26.5 Risk Refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format [Glu94]. That is, the risk is stated in the following form:

> Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 26.4.2, you could write:

> Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

26.6 RISK MITIGATION, MONITORING, AND MANAGEMENT

All the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.* For example, assume that high staff turnover is noted as a project risk r_1. Based on its history and management intuition, the likelihood l_1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x_1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, *risk-monitoring* activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more likely or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called

for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well under way and several people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents or Wikis," and/or meeting with other team members who will remain on the project.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technology costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. You need to perform a classic cost-benefit analysis. If RE for a specific risk is less than the cost of risk mitigation, don't try to mitigate the risk but continue to monitor it. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may require significant resources to manage. For this reason, you should adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help you to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

Gamification[6] has been suggested as an approach to encouraging software developers to follow process compliance procedures in the areas like quality and risk

6 Deterding et al. defined gamification as the use of game design elements in nongame settings to increase motivation and attention on a task [Det11]. It is important to note that this definition does not refer to the *playing* of games, but rather the use of game design elements in other contexts.

management [Ped14]. A typical gamification approach might award points, badges, or other nonmonetary awards to each developer and make use of a public leader board showing each person's ranking within the development group. If such an approach can be implemented based on automatic data collection (e.g., tracking the number of commitments to the software repository), it can be a cost-effective way of ensuring all team members are monitoring the risk factor measures intended to trigger the mitigation steps required to prevent the risk from becoming a disaster [Baj11]. A word of caution: You must ensure that members of the team are *not* rewarded for doing things like injecting problems into the process, so they can earn badges for mitigating the problems they caused [Bri13].

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

Software safety and hazard analysis (e.g., [Fir15], [Har12a], [Lev12]) are software quality assurance activities (Chapter 17) that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

SAFEHOME



Gamification and Risk Management

The scene: Doug Miller's office, prior to the initiation of the *SafeHome* software project.

The players: Doug Miller, manager of the *SafeHome* software engineering team, and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I'd like to spend some time brainstorming ideas for getting all developers on board with monitoring and mitigating the risks we identified for the *SafeHome* project.

Jamie: I thought it was your job to keep track of the project?

Doug: It is, but the development team members are able to see potential problems much quicker than I can, since they're working in the trenches.

Vinod: Working is right. We already have a lot to do to keep this project moving forward . . .

Doug: Exactly, we don't have time to fix problems that could have been prevented by addressing risks early.

Jamie: Okay . . . so what are you thinking?

Doug: I'm thinking since you guys like games, that may be a way to make risk monitoring more like a game than a project task.

Jamie: You know, my buddy—he works at another company—told me about something called "gamification." Sounds like what you're talking about, Doug.

Doug: Yeah, gamification is growing in popularity as a compliance tool in software process areas like quality assurance and risk management.

Vinod: So how can we do this?

Doug: I was thinking that most of the triggers for our risk mitigation tasks are based on the metrics we already have in the project management dashboard. It might just be a question of encouraging developers to check the dashboard on a regular basis.

Jamie: Are we going to use "snoopware" to track the number of times people check?

Doug: Maybe not. Perhaps we can reward the first person who reports a trigger value that indicates something's wrong.

Jamie: My buddy mentioned that they use leader boards to encourage competition. We can do a simple leader board as a Google doc spreadsheet.

Vinod: Some games have badges. We might create badges for the leader board.

Doug: Would people work for badges?

Jamie: They might if they could cash badges in for rewards.

Doug: What kind of rewards?

Jamie: Maybe something like getting first pick on a user story to develop in the current sprint. Or a cash reward if a person's actions save the company a lot of rework costs.

Doug: Let me think some more on this and let you know what I come up with for rewards and a point system for the leader board. We'll meet again tomorrow. For now, think about the most important risks we should be monitoring when we start our first sprint.

26.7 The RMMM Plan

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate *risk mitigation, monitoring, and management plan.* The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS) [Wil97]. In most cases, the RIS is maintained using a cloud database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. This approach might lend itself to supporting gamification of the risk management process. It would also facilitate the sharing of the risk information sheets to all the company software teams. The format of the RIS is illustrated in Figure 26.3.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin [what risk(s) caused which problems throughout the project].

FIGURE 26.3 Risk information sheet

Risk information sheet

Risk ID: P02-4-32	Date: 5 / 9 / 19	Prob: 80%	Impact: high

Description:
Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Refinement/context:
Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.

Mitigation/monitoring:
1. Contact third party to determine conformance with design standards.
2. Press for interface standards completion; consider component structure when deciding on interface protocol.
3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.

Management/contingency plan/trigger:
RE computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly.
Trigger: Mitigation steps unproductive as of 7 / 1 / 19.

Current status:
5 / 12 / 19: Mitigation steps initiated.

Originator: D. Gagne	Assigned: B. Laster

26.8 SUMMARY

Whenever a lot is riding on a software project, common sense dictates risk analysis. And yet, most software project managers do it informally and superficially, if they do it at all. The time spent identifying, analyzing, and managing risk pays itself back in many ways—less upheaval during the project, a greater ability to track and control a project, and the confidence that comes with planning for problems before they occur.

Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it. To quote Sun Tzu, a Chinese general who lived 2,500 years ago, "If you know the enemy and know yourself, you need not fear the result of a hundred battles." For the software project manager, the enemy is risk.

PROBLEMS AND POINTS TO PONDER

26.1. Provide five examples from other fields that illustrate the problems associated with a reactive risk strategy.

26.2. Describe the difference between "known risks" and "predictable risks."

26.3. You're the project manager for a major software company. You have been asked to lead a team seeking to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Create a risk table for the project.

26.4. Develop a risk mitigation strategy and specific risk mitigation activities for three of the risks noted in Figure 26.1.

26.5. Develop a risk monitoring strategy and specific risk monitoring activities for three of the risks noted in Figure 26.1. Be sure to identify the factors that you'll be monitoring to determine whether the risk is becoming more likely or less likely to occur.

26.6. Develop a risk management strategy and specific risk management activities for three of the risks noted in Figure 26.1.

26.7. Attempt to refine three of the risks noted in Figure 26.1, and then create risk information sheets for each.

26.8. Recompute the risk exposure discussed in Section 26.4.2 when cost/LOC is $16 and the probability is 60 percent.

26.9. Can you think of a situation in which a high-probability, high-impact risk would not be considered as part of your RMMM plan?

26.10. Describe five software application areas in which software safety and hazard analysis would be a major concern.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

27 A Strategy for Software Support

Regardless of its application domain, its size, or its complexity, computer software will evolve over time. Change drives this process. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when customers request new features or functions, and when the application is reengineered to provide benefit in a modern context. Software support actually begins when the developers involve stakeholders in the requirements gathering and prototype evolution process (Figure 27.1). Software support ends with the decision to retire the system from active use.

Key Concepts

document restructuring 564
forward engineering 565
inventory analysis 563
maintainability 553
maintenance tasks 554
refactoring 560
  architecture 561
  code 561
  data 561
release management 550
reverse engineering 555
  data 555
  processing 556
  user interfaces 557
software evolution 562
software maintenance 552
software reengineering 562
software support 550
supportability 552

Quick Look

What is it? Software support encompasses a set of activities that correct bugs, adapt the software to changes in its environment, enhance the software based on stakeholder requests, and reengineer the software to achieve better functionality and performance. During these activities, quality must be ensured and change must be controlled.

Who does it? At an organizational level, support staff from the software engineering organization perform all support activities. User training, bug report management, performing warranty repairs, and continuing to manage customer relations may be handled by other specialized teams.

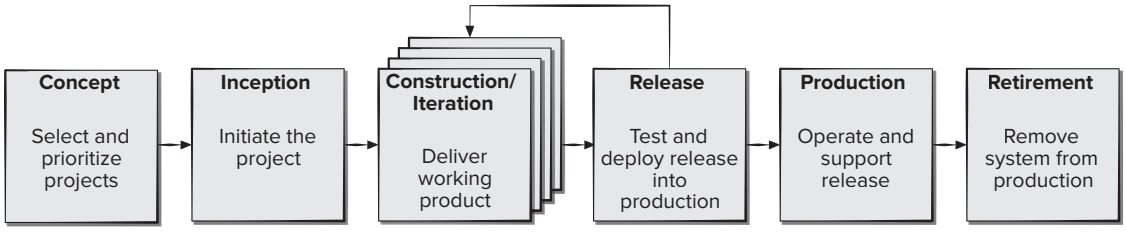
Why is it important? Software exists within a rapidly changing technology and business environment. That's why software must be maintained continually, and at the appropriate time, reengineered to keep pace.

What are the steps? Software support incorporates a maintenance function that corrects defects, adapts the software to meet a changing environment, and enhances functionality to meet the evolving needs of customers. At a strategic level, the support team works with stakeholders to examine the existing business goals for the software product and creates a revised software product to better meet the revised business goals. Software evolution using reengineering creates versions of existing programs that exhibit higher quality and better maintainability.

What is the work product? A variety of maintenance and reengineering work products (e.g., use cases, analysis and design models, test procedures) are produced. The final output is upgraded software that is easier to maintain and better meets the needs of its users.

How do I ensure that I've done it right? Use the same software quality and change management practices that are applied in every software engineering process.

FIGURE 27.1 Software prototype evolution process model



Over the past 40 years, Manny Lehman (e.g., [Leh97a]) and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution.* The details of this work are beyond the scope of this book, but a brief mention of some of these laws [Leh97b] is worthwhile:

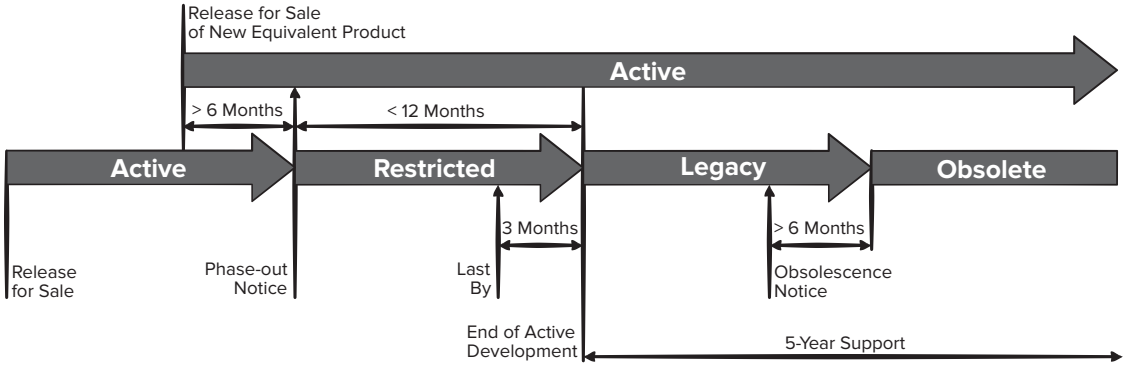
> **Law of continuing change (1974).** Software that has been implemented in a real-world computing context and will therefore evolve over time (called *E-type systems*) must be continually adapted else they become progressively less satisfactory.
>
> **Law of increasing complexity (1974).** As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
>
> **Law of conservation of familiarity (1980).** As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain knowledge of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that knowledge. Hence the average incremental growth remains invariant as the system evolves.
>
> **Law of continuing growth (1980).** The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
>
> **Law of declining quality (1996).** The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality. In this chapter, we'll discuss the challenges of software support including maintenance and evolution activities that are required to extend the effective life of legacy systems.

27.1 SOFTWARE SUPPORT

Software support can be considered an umbrella activity that includes many activities we have already discussed in this book: change management (Chapter 22), proactive risk management (Chapter 26), process management (Chapter 25), configuration management (Chapter 22), quality assurance (Chapter 17), and release management (Chapter 4). *Release management* is the process that brings high-quality code changes from a developer's workspace to the end user, encompassing code change integration, continuous integration, build system specifications, infrastructure-as-code, and

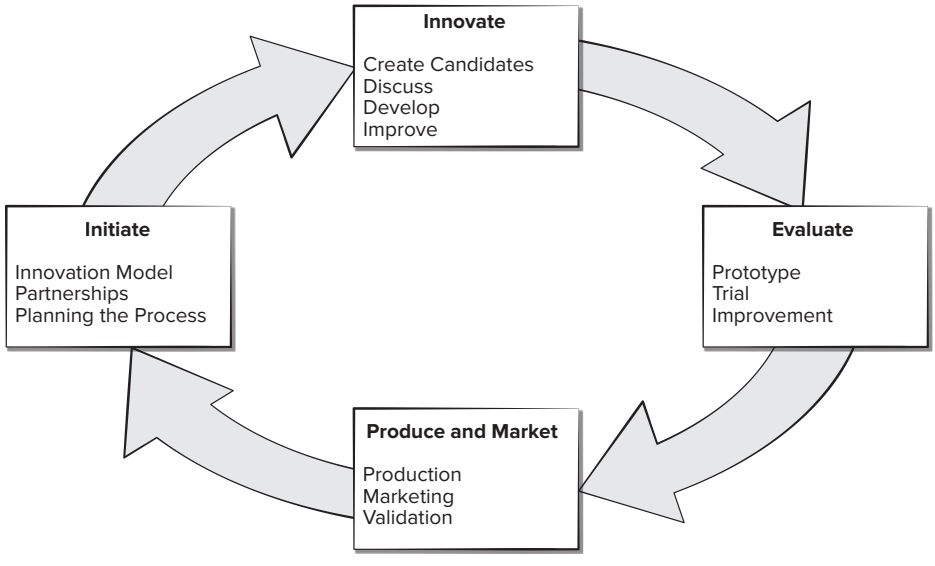
FIGURE 27.2 Software release and retirement example



deployment and release [Ada16]. Ultimately software needs to be retired. Figure 27.2 provides an example of a time line showing the release and retirement of a software product.

To effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life of the software. Figure 27.3 shows one model for supporting software after it is released.

FIGURE 27.3
Iterative software support model



A reasonable definition of *software supportability* is

> . . . the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, labor, or any other resource required to maintain the software operational and capable of satisfying its function. [SSO08]

In essence, supportability is one of many quality factors that should be considered during the analysis and design actions that are part of the software process. It should be addressed as part of the requirements model (or specification) and considered as the design evolves and construction commences. There should be some consideration for how long the software will be maintained before it is replaced by a new product.

For example, the need to "antibug" software at the component and code level has been discussed previously in this book. The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects *will* be encountered). In addition, support personnel should have access to a database that contains records of all defects that have already been encountered—their characteristics, cause, and cure. This will enable support personnel to examine "similar" defects and may provide a means for more rapid diagnosis and correction.

Although defects encountered in an application are a critical support issue, supportability also demands that resources be provided to support day-to-day end-user issues. The job of end-user support personnel is to answer user queries about the installation, operation, and use of the application.

27.2 SOFTWARE MAINTENANCE

Maintenance begins almost immediately. Software is released to end users, and within days, bug reports filter back to the software engineering organization. Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment. And within months, another corporate group that wanted nothing to do with the software when it was released now recognizes that it may provide unexpected benefit. They'll need a few enhancements to make it work in their world.

The challenge of software maintenance has begun. You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has grown long, and the work it implies threatens to overwhelm the available resources. As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications. In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance for products that have been in active use for several years.

As we noted in Chapter 22, the ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, you must develop mechanisms for evaluating, controlling, and making modifications.

Throughout this book, we've emphasized the importance of understanding the problem (analysis) and developing a well-structured solution (design). In fact, Part 2 of the book is dedicated to the mechanics of these software engineering actions, and Part 3 focuses on the techniques required to be sure you've done them correctly. Both analysis and design lead to an important software characteristic that we call maintainability. In essence, *maintainability* is a qualitative indication[1] of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability.

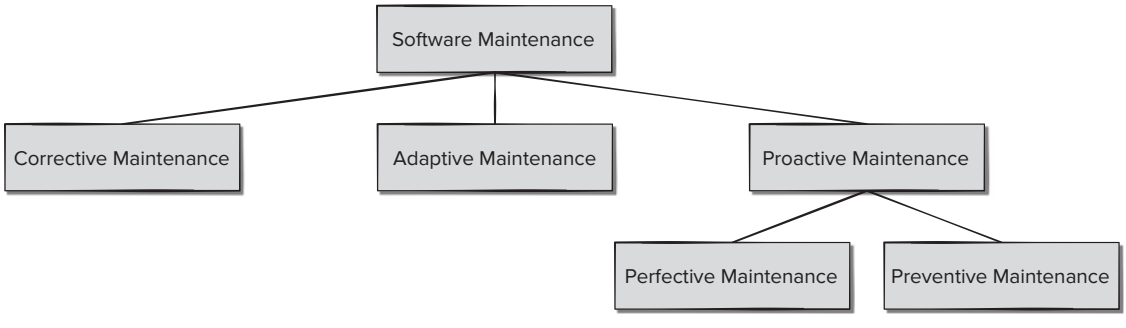
But what is maintainability? Maintainable software exhibits effective modularity (Chapter 9). It makes use of design patterns (Chapter 14) that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. It has undergone a variety of quality assurance techniques (Part 3 of this book) that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they may not be around when changes must be made. Therefore, the design and implementation of the software must "assist" the person who is making the change.

27.2.1 Maintenance Types

In Chapter 4 we discussed the four types of maintenance shown in Figure 27.4. It is clear that corrective and adaptive maintenance do not add new functionality. It is likely that new functionality will be added to the software during perfective maintenance and possibly during preventative maintenance as well.

In this chapter, we will discuss three broad classes of software maintenance that are relevant to the software support process: reverse engineering, software refactoring, and software evolution or reengineering. Reverse engineering is the process of analyzing a software system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction (Section 27.2.2). Often reverse engineering is used to rediscover system

FIGURE 27.4 Types of software maintenance



1 There are many quantitative measures that provide an indirect indication of maintainability (e.g., [Sch99], [SEI02]).

design elements and redocument them prior to modifying the system source code. Refactoring is the process of changing a software system in such a way that it does not alter its external behavior but improves its internal structure. Refactoring is often used to improve the quality of a software product and make it easier to understand and easier to maintain (Section 27.4). Reengineering (evolution) of software is the process of taking an existing software system and generating a new system from it that has the same quality as software created using modern software engineering practices [Osb90]. Reengineering and software evolution are discussed in Section 27.5.

27.2.2 Maintenance Tasks

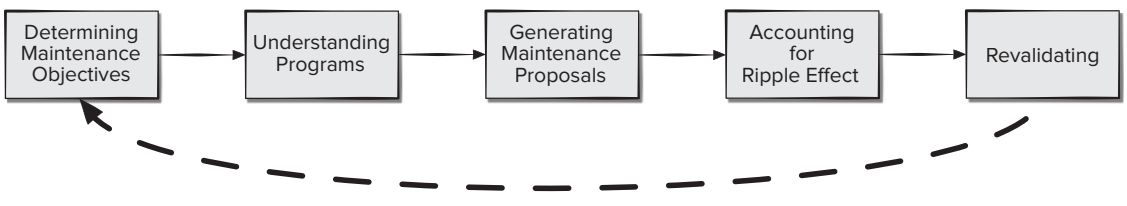
The scenario is all too common: An application has served the business needs of a company for 10 or 15 years. During this time period it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the urgency of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software evolution and reengineering (Section 27.5) has been spawned by software maintenance problems that have been building for almost half a century. Figure 27.5 shows a set of generic tasks that should be completed as part of a controlled software maintenance process.

Agile process models similar to the one we described in Chapter 4 deliver incremental prototypes in 4-week sprints. It can be argued that agile developers are in perpetual software support mode as they add new stakeholder requested features in every software increment. However, it is important to realize that software development is not maintenance. It is advisable for separate groups of engineers to handle these two tasks. Heeager and Rose [Hee15] suggest nine heuristics to help the maintenance become more agile.

1. Use sprints to organize the maintenance work. You should balance the goal of keeping customers happy with the technical needs of the developers.
2. Allow for urgent customer requests to interrupt scheduled maintenance sprints, by including time for them during maintenance sprint planning.
3. Facilitate team learning by ensuring that more experienced developers are able to mentor less experienced team members even when working on their own discrete tasks.

FIGURE 27.5 Software maintenance tasks



4. Allow multiple team members to accept customer requests as they arise and coordinate their processing with the other maintenance team members.
5. Balance the use of written documentation with face-to-face communication to ensure planning meeting time is used wisely.
6. Write informal use cases to supplement the other documentation being used for communications with stakeholders.
7. Have developers test each other's work (both defect repairs and new feature implementations). This allows for shared learning and improves the feelings of product ownership by the team members.
8. Make sure developers are empowered to share knowledge with one another. This can motivate people to improve the skills and knowledge (allows developers to learn new things, improves their professional skills, and distributes tasks more evenly).
9. Keep planning meetings short, frequent, and focused.

27.2.3 Reverse Engineering

The first task that needs to be completed by software engineers before responding to any maintenance request is to understand the system that needs to be modified. Sadly, the system being maintained often has low quality and lacks reasonable documentation. This is what technical debt is all about. Technical debt is often caused by developers adding features without documenting them or considering their impact on the larger software system.

Reverse engineering can be used to extract design information from source code, but the abstraction level of this information, the completeness of the documentation, and the degree to which human analysts work comfortably with the available tools, are highly variable. Reverse engineering conjures an image of the "magic slot." You feed a haphazardly designed, undocumented source file into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist.

Reverse engineering requires developers to evaluate the old software system by examining its (often undocumented) source code, developing a meaningful specification of the processing being performed, the user interface that was used, and the program data structures or associated database.

Reverse Engineering to Understand Data. This occurs at different levels of abstraction and is often the first reengineering task. In some cases, the first reverse engineering activity attempts to construct a UML class diagram. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new systemwide database.

Internal Data Structures. Reverse engineering techniques for internal program data focus on the definition of object classes. This is accomplished by examining the

program code with the intent of grouping related program variables together. In many cases, the data organization within the code suggests several abstract data types. For example, record structures, files, lists, and other data structures often provide initial suggestions for possible classes.

Database Structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [Pre94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys), (3) refine the tentative classes, (4) define generalizations, and (5) discover associations using techniques that are analogous to the CRC approach. Once information defined in the preceding steps is known, a series of transformations [Pre94] can be applied to map the old database structure into a new database structure.

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within a larger system. Each of the programs that make up the system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system, program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[2]

Things become more complex when the code inside a component is considered. You should look for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, you can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools can be used to help you understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

2 Often, specifications written early in the life history of a program are never updated. As changes are made, the code no longer conforms to the specification.

Reverse engineering to understand user interfaces may need to be done as part of the maintenance task. Sophisticated GUIs have become *de rigueur* for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [Mer93] suggest three basic questions that must be answered as reverse engineering of the UI commences:

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a "replacement," or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation (Chapter 9) can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioral model must be extracted from the code.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop a new interaction metaphor. For example, an old UI that prompts the user to provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a touch-screen gesture to accomplish the same function.

27.3 PROACTIVE SOFTWARE SUPPORT

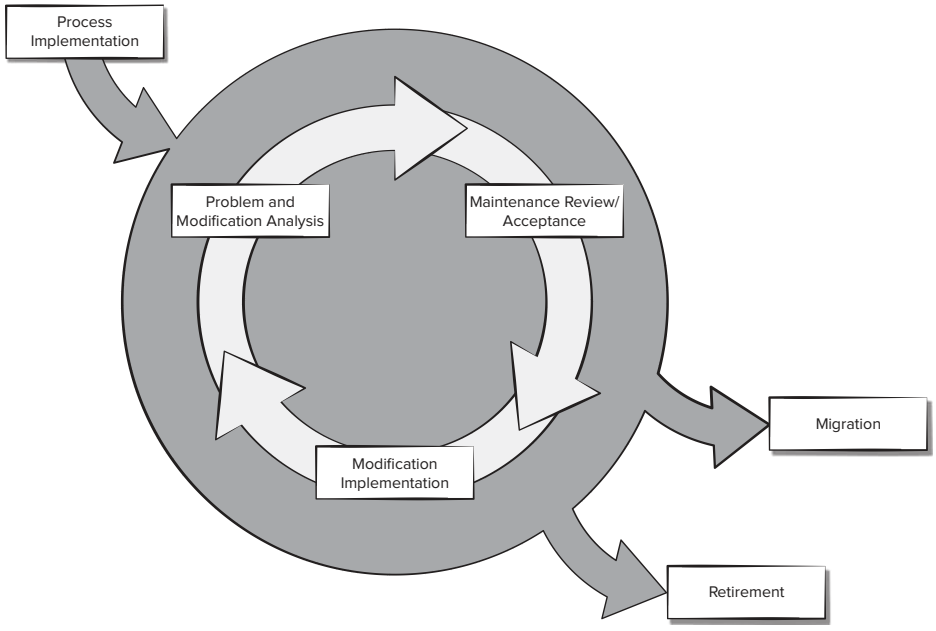
We described the differences between reactive and proactive risk management in Chapter 26. We also described preventative maintenance and perfective maintenance as being proactive maintenance activities (Section 27.2.1). If the goal of software engineering is to deliver high-quality products that meet customer needs in a timely and cost-effective manner, then software support, like other software engineering activities, requires the use of managed workflow processes so that unnecessary rework is avoided.

Supporting software means adapting it to meet changing customer demands and repairing defects reported by end users. This work may be required by law, by contractual agreement, or via product warranty. Repairing software can be a time-consuming and costly process, and it's important to anticipate problems and schedule the work required to respond to customer concerns before they become emergencies. *Proactive software support* requires software engineers to create tools and processes that can help them identify and resolve software issues before they become problems. A generic process for proactive software maintenance and support is shown in Figure 27.6.

The proactive support process is similar to risk monitoring and mitigation (Section 26.6). Developers need to search for indicators that suggest their software

FIGURE 27.6

Software maintenance and support process



product may have quality problems. Sometimes these problems can be addressed by evolving the product and migrating it to a newer version (Section 27.5). Sometimes the software can be restructured or refactored (Section 27.4) to improve its quality and make it easier to maintain. In some cases, the problems are so severe that the developer will need to make plans to retire the product and begin creating a replacement product before the customers abandon it.

27.3.1 Use of Software Analytics

There are currently three dominant uses for artificial intelligence methods in software engineering work [Har12b]: probabilistic reasoning, machine learning and prediction, and search-based software engineering. Probabilistic reasoning techniques can be used to model software reliability (Section 17.7.2). Machine learning can be used to automate the process of discovering root causes of software failures before they occur by predicting the presence of defects likely to cause these failures (Section 15.4.3). Search-based software engineering may be used to assist developers in identifying useful test cases to make regression testing more effective (Section 20.3). All of these AI applications make use of software analytics similar to those we discussed in Chapter 23.

For analytics to be useful they must be actionable, which means expending the effort to determine which measures are worth collecting because of their predictive value and which are not. Port and Tabor [Por18] suggest that analytics can be used to estimate: defect discovery rates based on estimates of yet undiscovered defects present in the product, time between defect discoveries during operation, and effort required to repair defects. Having an understanding of these can allow for better

planning in terms of the cost and time that should be allocated for maintaining the system once it is released to the end users for active use. It is important to keep in mind that even the best estimates contain elements of guessing, so unanticipated failures may still occur.

Zhang and her colleagues [Zha13] report several lessons learned when using software analytics for proactive maintenance tasks.

1. Be sure you are using analytics to identify meaningful development problems, or you will get no buy-in from the software engineers.
2. The analytics must make use of application domain knowledge to be useful to developers (this implies the use of experts to validate the analytics).
3. Developing analytics requires iterative and timely feedback from the intended users.
4. Make sure the analytics are scalable to larger problems and customizable to incorporate new discoveries made over time.
5. Evaluation criteria used needs to be correlated to real software engineering practices.

Mining of the historical information housed in software repositories is a popular way of obtaining the training information needed for the AI techniques mentioned earlier [Sun15]. Using this discovered knowledge helps developers to target their software support actions. Additional discussion of the use of analytics and data science appears in Appendix 2 to this book.

27.3.2 Role of Social Media

Many online stores such as Google Play or Apple's App Store allow users to provide feedback on the apps by posting ratings or comments. The feedback found in these reviews may contain usage scenarios, bug reports, or feature requests. Mining these reports using natural language processing and machine learning techniques can help developers identify potential maintenance and software evolution tasks [Sun15]. However, much of this information is unstructured, and there is so much of it that it is hard to make sense of it without the use of automated statistical tools to reduce the sheer volume of information and create actionable analytics to guide support decisions.

Many companies maintain Facebook pages or Twitter feeds to support their user communities. Some companies encourage their software product users to send program crash information for analysis by the support team members. Still other companies pursue the questionable practice of tracking how and where their products are being used by their customers without their knowledge. It is very easy to collect a lot of user information automatically. Software engineers must resist the temptation to make use of this information in unethical ways.

27.3.3 Cost of Support

In a perfect world, every unmaintainable program would be retired immediately, to be replaced by high-quality, reengineered applications developed using modern software engineering practices. But we live in a world of limited resources. Software evolution and maintenance tasks drain resources that can be used for other business

purposes. Therefore, before an organization attempts to modify or replace an existing application, it should perform a cost-benefit analysis.

A cost-benefit analysis model for reengineering has been proposed by Sneed [Sne95]. Nine parameters are defined:

P_1 = current annual maintenance cost for an application
P_2 = current annual operations cost for an application
P_3 = current annual business value of an application
P_4 = predicted annual maintenance cost after reengineering
P_5 = predicted annual operations cost after reengineering
P_6 = predicted annual business value after reengineering
P_7 = estimated reengineering costs
P_8 = estimated reengineering calendar time
P_9 = reengineering risk factor ($P_9 = 1.0$ is nominal)
L = expected life of the system

The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times L \tag{27.1}$$

The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{27.2}$$

Using the costs presented in Equations (27.1) and (27.2), the overall benefit of reengineering can be computed as

$$\text{Cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \tag{27.3}$$

The cost-benefit analysis presented in these equations can be performed for all high-priority applications identified as candidates to evolve or retire (Section 27.5). Those applications that show the highest cost-benefit can be targeted for proactive maintenance or evolution, while work on other applications can be postponed until resources are available.

27.4 REFACTORING

Software refactoring (also known as restructuring) modifies source code and/or data in an effort to make it amenable to future changes. In general, refactoring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the refactoring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering (Section 27.5).

Refactoring occurs when the basic architecture of an application is solid, even though technical internals need work. It is initiated when major parts of the software are serviceable and only a subset of all modules and data need extensive modification.[3]

3 It is sometimes difficult to make a distinction between extensive refactoring and evolution. Both are reengineering.

27.4.1 Data Refactoring

Before data refactoring can begin, a reverse engineering activity called *source code analysis* should be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis*.

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When refactoring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

27.4.2 Code Refactoring

Code refactoring is performed to yield a design that produces the same function but with higher quality than the original program. The objective is to take "spaghetti-bowl" code and derive a design that conforms to the quality factors discussed in Chapters 15 and 17.

Other restructuring techniques have also been proposed for use with refactoring tools. One approach might rely on the use of anti-patterns (Section 14.5), both to identify bad code design practices and suggest possible solutions to reduce coupling and improve cohesion [Bro98]. Although code refactoring can alleviate immediate problems associated with debugging or small changes, it is not reengineering. Real benefit is achieved only when data and architecture are refactored as well.

27.4.3 Architecture Refactoring

We made the point in Chapter 10 that making architectural changes to a software product already in production can be a costly and time-consuming process. However, when a program with control flow that looks like the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements, it may be desirable to consider architectural refactoring as one of the design trade-offs. In general, for a messy program like this you have the following options:

1. You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.
2. You can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. You can revise (redesign, recode, and test) those portions of the software that require modification, applying a meaningful software engineering approach to all revised segments.

4. You can completely redo (redesign, recode, and test) the complete program, using reengineering tools to assist in understanding the current design.

There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied. We will discuss software evolution and reengineering in Section 27.5.

27.5 SOFTWARE EVOLUTION

At first glance, the suggestion that you redevelop a large program when a working version already exists may seem quite extravagant. Reengineering takes time, it has significant cost, and it absorbs resources that might be otherwise occupied on immediate concerns. For all these reasons, reengineering is not accomplished in a few months or even a few years.

Reengineering of software systems is an activity that will absorb software engineering resources for many years. That's why every organization needs a pragmatic strategy for software reengineering. If time and resources are in short supply, you might consider applying the Pareto principle to the software that is to be reengineered and apply the reengineering process to the 20 percent of the software that accounts for 80 percent of the problems.

Before passing judgment, consider the following arguments. The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line. In addition, redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance. Automated tools for reengineering or software evolution will make some part of the job easier. Because a prototype of the software already exists, development productivity should be much higher than average. The user now has experience with the software. Therefore, new requirements and the direction of change can be ascertained with greater ease. At the end of this evolutionary preventive maintenance, the developers will end up with a complete software configuration (documents, programs, and data).

Reengineering is a rebuilding activity. To better understand it, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).

- Before you start rebuilding, be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the evolving computer-based systems and applications.

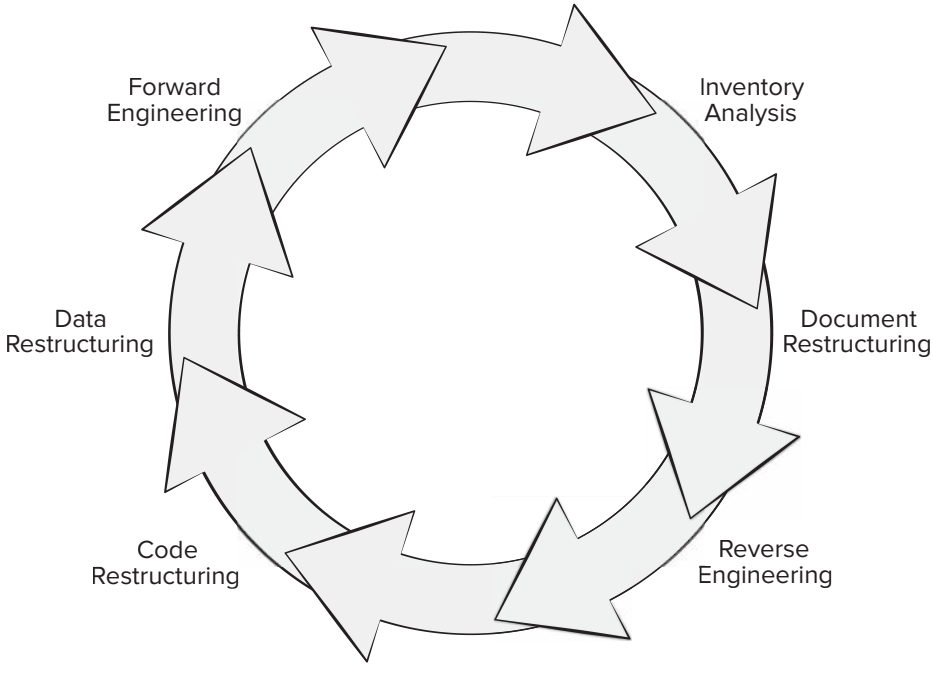
To implement these principles, you can use a cyclical process model for reengineering like the one shown in Figure 27.7. This model defines six activities. Because it is cyclical, each of the activities presented may be revisited as often as needed. For any particular cycle, the process can terminate after any one of these activities.

27.5.1 Inventory Analysis

Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity,

FIGURE 27.7

A software reengineering process model



current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular basis. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

27.5.2 Document Restructuring

Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options? In some cases, creating documentation when none exists is simply too costly. If the software works, let it be! In other cases, some documentation must be created, but only when changes are made. If a modification occurs, document it. Finally, there are situations in which a critical system must be fully documented, but even here, documents should achieve an essential minimum. Your software organization must choose the documentation option that is most appropriate for each case.

27.5.3 Reverse Engineering

Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of *design recovery*. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

27.5.4 Code Refactoring

The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is *code refactoring*. Some legacy systems have a solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of good design practices are noted, and code is then refactored or even rewritten in a more modern programming language. The resultant refactored code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

27.5.5 Data Refactoring

A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

27.5.6 Forward Engineering

In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software re-creates the function of the existing system and also adds new functions and/or improves overall performance. In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

27.6 SUMMARY

Software support is an ongoing activity that occurs throughout the life cycle of an application. During support, maintenance actions are initiated, defects are corrected, applications are adapted to a changing operational or business environment, enhancements are implemented at the request of stakeholders. In addition, users are supported as they integrate an application into their personal or business work flow.

Software maintenance and support activities need to be proactive in their nature. It is better to anticipate problems and remove their root causes before the customers find them and become dissatisfied with the software product. The use of software analytics may help software developers identify potential defects and maintenance issues before they become problematic.

At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality. Software evolution or reengineering encompasses a series of activities that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability—programs that will be viable well into the twenty-first century.

The cost-benefit of reengineering can be determined quantitatively. The cost of the status quo, that is, the cost associated with ongoing support and maintenance of an existing application, is compared to the projected costs of reengineering and the resultant reduction in maintenance and support costs. In almost every case in which a program has a long life and currently exhibits poor maintainability or supportability, reengineering represents a cost-effective business strategy.

PROBLEMS AND POINTS TO PONDER

27.1. How does software support differ from software maintenance?

27.2. Your instructor will select one of the programs that everyone in the class has developed during this course. Exchange your program randomly with someone else in the class. Do not explain or walk through the program. Now, implement an enhancement (specified by your instructor) in the program you have received.

a. Perform all software engineering tasks including a brief walkthrough (but not with the author of the program).
b. Keep careful track of all errors encountered during testing.
c. Discuss your experiences in class.

27.3. Create a software reengineering inventory analysis checklist, and propose a quantitative software rating system that could be applied to existing programs in an effort to pick candidate programs for reengineering. Your system should extend beyond the economic analysis presented in Section 27.3.

27.4. Suggest alternatives to paper and ink or conventional electronic documentation that could serve as the basis for document restructuring. (Hint: Think of new descriptive technologies that could be used to communicate the intent of the software.)

27.5. Some people believe that artificial intelligence technology will increase the abstraction level of the reverse engineering process. Do some research on this subject (i.e., the use of AI for reverse engineering), and write a brief paper that takes a stand on this point.

27.6. Why is completeness difficult to achieve as abstraction level increases?

27.7. Why is proactive software support preferable to reactive defect repair?

27.8. Using information obtained via the Web, present characteristics of three reverse engineering tools to your class.

27.9. There is a subtle difference between refactoring and forward engineering. What is it?

27.10. How would you determine P_4 through P_7 in the cost-benefit model presented in Section 27.3.3?

Design element: Quick Look icon magnifying glass: © Roger Pressman

PART

Five

ADVANCED TOPICS

In this part of *Software Engineering: A Practitioner's Approach,* we consider several advanced topics that will extend your understanding of software engineering. The following questions are addressed in the chapters that follow:

- What is software process improvement and how can it be used to improve the state of software engineering practice?
- What emerging trends can be expected to have a significant influence on software engineering practice in the next decade?
- What is the road ahead for software engineers?

Once these questions are answered, you'll understand topics that may have a profound impact on software engineering in the years to come.

CHAPTER

28 SOFTWARE PROCESS IMPROVEMENT

Long before the phrase "software process improvement" was widely used, RSP worked with major corporations to help them improve the state of their software engineering practices. Based on his experiences, he wrote a book titled *Making Software Engineering Happen* [Pre88]. In the preface of that book he made the following comment:

> For the past ten years I have had the opportunity to help a number of large companies implement software engineering practices. The job is difficult and rarely goes as smoothly as one might like—but when it succeeds, the results are profound . . .
>
> But all is not sweetness and light. Many companies attempt to implement software engineering practice and give up in frustration. Others do it half-way and never see the benefits noted above. Still others do it in a heavy-handed fashion that results in open rebellion among technical staff and managers and subsequent loss of morale.

KEY CONCEPTS

assessment 572
CMMI 576
education and training 573
evaluation 575
installation/migration 574
justification 573
maturity models 570
People CMM 577
return on investment. 580
risk management 575
selection 573
software process improvement (SPI)
  applicability 571
  definition of 569
  frameworks 569
  process. 571



QUICK LOOK

What is it? Software process improvement (SPI) encompasses a set of activities that will lead to a better software process and higher-quality software delivered in a timely manner.

Who does it? Technical managers, software engineers, and individuals who have quality assurance responsibility.

Why is it important? As an organization works to improve its software engineering practices, it must address and correct weaknesses in its existing software process.

What are the steps? The approach to SPI is iterative and continuous, but it can be viewed in five steps: (1) assessment; (2) education and training; (3) selection and justification of process and technology; (4) implementation of the SPI plan; and (5) evaluation and tuning of the results.

What is the work product? An improved software process that leads to higher-quality software.

How do I ensure that I've done it right? The software your organization produces will be delivered with fewer defects, rework at each stage of the software process will be reduced, and on-time delivery will become much more likely.

Even though these words were written over three decades ago, they remain equally true today.

Over the years, virtually every major software engineering organization has attempted to "make software engineering happen." Some have implemented individual practices that have helped to improve the quality of the product they build and the timeliness of their delivery. Others have established a "mature" software process that guides their technical and project management activities. But others continue to struggle. Their practices are hit-and-miss, and their process is ad hoc. Occasionally, their work is spectacular, but in the main, each project is an adventure, and no one knows whether it will end badly or well.

So, which of these two cohorts needs software process improvement? The answer (which may surprise you) is *both.* Those that have succeeded in making software engineering happen cannot become complacent. They must work continually to improve their approach to software engineering. And those that struggle must begin their journey down the road toward improvement.

28.1 What Is SPI?

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Because SPI is not free, it must deliver a return on investment. The effort and time that is required to implement an SPI strategy must pay for itself in some measurable way. To do this, the results of improved process and practice must lead to a reduction in software "problems" that cost time and money. It must reduce the number of defects that are delivered to end users, reduce the amount of rework due to quality problems, reduce the costs associated with software maintenance and support (Chapter 27), and reduce the indirect costs that occur when software is delivered late.

28.1.1 Approaches to SPI

Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of SPI frameworks. An *SPI framework* defines (1) a set of characteristics that must be present if an effective software process is to be achieved, (2) a method for assessing whether those characteristics are present, (3) a mechanism for summarizing the results of any assessment, and (4) a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

An SPI framework assesses the "maturity" of an organization's software process and provides a qualitative indication of a maturity level. In fact, the term *maturity model* (Section 28.1.2) is often applied. The SPI framework encompasses a maturity model that in turn incorporates a set of process quality indicators that provide an overall measure of the process quality that will lead to product quality.

FIGURE 28.1

Elements of an SPI framework
Source: Adapted from Rout, Terry, "Software Process Assessment—Part 1: Concepts and Introductory Guide," Spice, 2002.

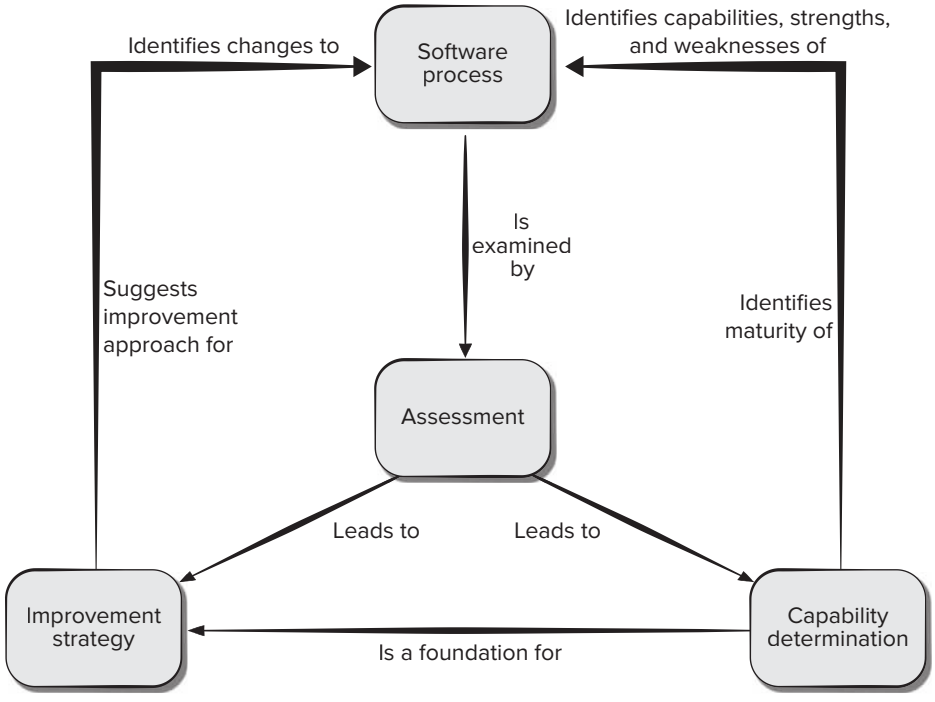


Figure 28.1 provides an overview of a typical SPI framework. The key elements of the framework and their relationship to one another are shown.

You should note that there is no universal SPI framework. In fact, the SPI framework that is chosen by an organization reflects the people in the organization that are championing the SPI effort. As an SPI framework is applied, an organization must establish mechanisms to: (1) support technology transition, (2) determine the degree to which an organization is ready to absorb process changes that are proposed, and (3) measure the degree to which changes have been adopted.

28.1.2 Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization, that is, an indication of the quality of the software process, the degree to which practitioners understand and apply the process, and the general state of software engineering practice. This is accomplished using some type of ordinal scale.

For example, the Software Engineering Institute's original *Capability Maturity Model* (Section 28.3) suggests five levels of maturity ranging from *initial* (rudimentary software process) to *optimized* (a process that leads to best practices).[1]

1 The original CMM has been updated and is discussed in Section 28.3.

The overriding question is whether maturity scales, such as the one proposed as part of the Capability Maturity Model Integration (CMMI) process framework, provide any real benefit. We think that they do. A maturity scale provides an easily understood snapshot of process quality that can be used by practitioners and managers as a benchmark from which improvement strategies can be planned.

28.1.3 Is SPI for Everyone?

For many years, SPI was viewed as a "corporate" activity—a euphemism for something that only large companies perform. But today, a significant percentage of all software development is being performed by companies that employ fewer than 100 people (or in the case of startups less than 24 people). Can a small company initiate SPI activities and do it successfully?

There are substantial cultural differences between large software development organizations and small ones. It should come as no surprise that small organizations are more informal, apply fewer standard practices, and tend to be self-organizing. They also tend to pride themselves on the "creativity" of individual members of the software organization, and initially view an SPI framework as overly bureaucratic and ponderous. Yet, process improvement is as important for a small organization as it is for a large one.

Within small organizations the implementation of an SPI framework requires resources that may be in short supply. Managers must allocate people and money to make software engineering happen. Therefore, regardless of the size of the software organization, it's reasonable to consider the business motivation for SPI. You always need to look at the process activities that are being proposed. If a specific process model or SPI approach feels like overkill for your organization, it probably is.

Quantitative analyses of many projects have shown that the agile project methodologies favored by smaller organizations can lead to greater process efficiency and increase customer satisfaction [Ser15]. It is still likely SPI will be approved and implemented only after its proponents demonstrate *financial leverage* [Bir98]. Financial leverage is demonstrated by examining technical benefits (e.g., fewer defects delivered to the field, reduced rework, lower maintenance costs, or more rapid time to market) and translating them into dollars. You must show a realistic return on investment (Section 28.6) to justify SPI costs.

28.2 The SPI Process

The hard part of SPI isn't the definition of characteristics that define a high-quality software process or the creation of a process maturity model. Those things are relatively easy. Rather, the hard part is establishing a consensus for initiating SPI and defining an ongoing strategy for implementing it across a software organization.

The Software Engineering Institute has developed IDEAL—"an organizational improvement model that serves as a road map for initiating, planning, and implementing improvement actions" [SEI08]. IDEAL is representative of many process models for SPI, defining five distinct activities—initiating, diagnosing, establishing, acting, and learning—that guide an organization through SPI activities.

In this book, we present a somewhat different road map for SPI, based on the process model for SPI originally proposed in [Pre88]. It applies a commonsense philosophy that requires an organization to (1) look in the mirror, (2) then get smarter so it can make intelligent choices, (3) select the process model (and related technology elements) that best meets its needs, (4) instantiate the model into its operating environment and its culture, and (5) evaluate what has been done. These five activities (discussed in the subsections[2] that follow) are applied in an iterative (cyclical) manner that fosters continuous process improvement.

28.2.1 Assessment and Gap Analysis

Any attempt to improve your current software process without first assessing the efficacy of current framework activities and associated software engineering practices would be like starting on a long journey to a new location with no idea where you are starting from. You'd depart with great flourish, wander around trying to get your bearings, expend lots of energy and endure large doses of frustration, and likely, decide you really didn't want to travel anyway. Stated simply, before you begin any journey, it's a good idea to know precisely where you are.

The first road map activity, called *assessment,* allows you to get your bearings. The intent of assessment is to uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process.

Assessment examines a wide range of actions and tasks that will lead to a high-quality process. For example, regardless of the process model that is chosen, the software organization must establish generic mechanisms such as defined approaches for customer communication; established methods for representing user requirements; a project management framework that includes scoping, estimation, scheduling, and project tracking; risk analysis methods; change management procedures; quality assurance and control activities including reviews; and many others. Each is considered within the context of the framework activities (Chapter 2) that have been established, and each is assessed to determine whether all the following questions have been addressed:

- Is the objective of the activity clearly defined?
- Are work products required as input and produced as output identified and described?
- Are the work tasks to be performed clearly described?
- Are the people who must perform the activity identified by role?
- Have entry and exit criteria been established?
- Have metrics for the activity been established?
- Are tools available to support the activity?
- Is there an explicit training program that addresses the activity?
- Is the activity performed uniformly for all projects?

2 Some of the content in these sections has been adapted from [Pre88] with permission.

Although the questions noted imply a *yes* or *no* answer, the role of assessment is to look behind the answer to determine whether the activity in question is being performed in a manner that would conform to best practice.

As the process assessment is conducted, you (or those who have been hired to perform the assessment) should also focus on the following issues:

Consistency. Are important activities, actions, and tasks applied consistently across all software projects and by all software teams?

Sophistication. Are management and technical actions performed with a level of sophistication that implies a thorough understanding of best practice?

Acceptance. Is the software process and software engineering practice widely accepted by management and technical staff?

Commitment. Has management committed the resources required to achieve consistency, sophistication, and acceptance?

The difference between local application and best practice represents a "gap" that offers opportunities for improvement. The degree to which consistency, sophistication, acceptance, and commitment are achieved indicates the amount of cultural change that will be required to achieve meaningful improvement.

28.2.2 Education and Training

Although few software people question the benefit of an agile, organized software process or solid software engineering practices, many practitioners and managers do not know enough about either subject.[3] As a consequence, inaccurate perceptions of process and practice lead to inappropriate decisions when an SPI framework is introduced. It follows that a key element of any SPI strategy is education and training for practitioners, technical managers, and more senior managers who have direct contact with the software organization. It is wise to try to provide "just-in-time" training targeted to the real needs of a software team. Three types of education and training should be conducted: generic software engineering concepts and methods, specific technology and tools, and communication and quality-oriented topics. In a modern context, education and training can be delivered in a variety of different ways. Everything from podcasts, to short YouTube videos, to more comprehensive Internet-based training such as Coursera,[4] to e-books, to classroom courses can be offered as part of an SPI strategy.

28.2.3 Selection and Justification

Once the initial assessment activity[5] has been completed and education has begun, a software organization should begin to make choices. These choices occur during a *selection and justification activity* in which process characteristics and specific software engineering methods and tools are chosen to populate the software process.

---

3 If you've spent time reading this book, you won't be one of them!

4 See https://www.coursera.org/.

5 In actuality, assessment is an ongoing activity. It is conducted periodically in an effort to determine whether the SPI strategy has achieved its immediate goals and to set the stage for future improvement.

First, you should choose the process model (Chapters 2 through 4) that best fits your organization, its stakeholders, and the software that you build. You should decide which of the set of framework activities will be applied, the major work products that will be produced, and the quality assurance checkpoints that will enable your team to assess progress. If the SPI assessment activity indicates that you have specific weaknesses (e.g., you have no formal SQA functions), you should focus attention on process characteristics that will address these weaknesses directly.

Next, develop an adaptable work breakdown for each framework activity (e.g., modeling), defining the task set that would be applied for a typical project. You should also consider the software engineering methods that can be applied to achieve these tasks. As choices are made, education and training should be coordinated to ensure that understanding is reinforced.

As you make your choices, be sure to consider the culture of your organization and the level of acceptance that each choice will likely elicit. Ideally, everyone works together to select various process and technology elements and moves smoothly toward the installation or migration activity (Section 28.2.4). In reality, selection can be a rocky road. It is often difficult to achieve consensus among different constituencies. If the criteria for selection are established by committee, people may argue endlessly about whether the criteria are appropriate and whether a choice truly meets the criteria that have been established.

It is true that a bad choice can do more harm than good, but "paralysis by analysis" means that little if any progress occurs and process problems remain. As long as the process characteristic or technology element has a good chance at meeting an organization's needs, it's sometimes better to pull the trigger and make a choice, rather than waiting for the perfect solution.

28.2.4 Installation/Migration

Installation is the first point at which a software organization feels the effects of changes triggered by following the SPI road map. In some cases, an entirely new process is recommended for an organization. Framework activities, software engineering actions, and individual work tasks must be defined and installed as part of a new software engineering culture. Such changes represent a substantial organizational and technological transition and must be managed very carefully.

In other cases, changes associated with SPI are relatively minor, representing small, but meaningful, modifications to an existing process model. Such changes are often referred to as *process migration.* Today, many software organizations have a "process" in place. The problem is that it doesn't work in an effective manner. Therefore, an incremental *migration* from one process (that doesn't work as well as desired) to another process is a more effective strategy.

Installation and migration are *software process redesign* (SPR) activities. Scacchi [Sca00] states that "SPR is concerned with identification, application, and refinement of new ways to dramatically improve and transform software processes." When a formal approach to SPR is initiated, three different process models are considered: (1) the existing ("as is") process, (2) a transitional ("here to there") process, and the target ("to be") process. If the target process is significantly different from the existing process, the only rational approach to installation is an incremental strategy in which the transitional process is implemented in steps. The transitional process

provides a series of waypoints that enable the software organization's culture to adapt to small changes over time.

28.2.5 Evaluation

Although it is listed as the last activity in the SPI road map, *evaluation* occurs throughout SPI. The evaluation activity assesses the degree to which changes have been instantiated and adopted, the degree to which such changes result in better software quality or other tangible process benefits, and the overall status of the process and the organizational culture as SPI activities proceed.

Both qualitative factors and quantitative metrics are considered during the evaluation activity. From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes. Quantitative metrics (Chapter 23) are collected from projects that have used the transitional or "to be" process and compared with similar metrics that were collected for projects that were conducted under the "as is" process.

28.2.6 Risk Management for SPI

SPI is a risky undertaking. SPI often fails because risks were not properly thought through and no contingency planning occurred. In fact, more than half of all SPI efforts end in failure. The reasons for failure vary greatly and are organizationally specific. Among the most common risks are: a lack of management support, cultural resistance by technical staff, a poorly planned SPI strategy, an overly formal approach to SPI, selection of an inappropriate process, a lack of buy-in by key stakeholders, an inadequate budget, a lack of staff training, and organizational instability, but there are a myriad of other factors. The role of those chartered with the responsibility for SPI is to analyze likely risks and develop an internal strategy for mitigating them [Dut15].

A software organization should manage risk at three key points in the SPI process [Ive04]: prior to the initiation of the SPI road map, during the execution of SPI activities (assessment, education, selection, installation), and during the evaluation activity that follows the instantiation of some process characteristic. In general, the following categories [Ive04] can be identified for SPI risk factors: budget and cost, content and deliverables, culture, maintenance of SPI deliverables, mission and goals, organizational management, organizational stability, process stakeholders, schedule for SPI development, SPI development environment, SPI development process, SPI project management, and SPI staff.

Within each category, several generic risk factors can be identified. For example, the organizational culture has a strong bearing on risk. The following generic risk factors[6] can be defined for the culture category [Ive04]:

- Attitude toward change, based on prior efforts to change
- Experience with quality programs, level of success
- Action orientation for solving problems versus political struggles
- Use of facts to manage the organization and business

6 Risk factors for each of the risk categories noted in this section can be found in [Ive04].

- Patience with change; ability to spend time socializing
- Tools orientation—expectation that tools can solve the problems
- Level of "planfulness"—ability of organization to plan
- Ability of organization members to participate with various levels of organization openly at meetings
- Ability of organization members to manage meetings effectively
- Level of experience in organization with defined processes

Using the risk factors and generic attributes as a guide, a risk table (Chapter 26) can be developed to isolate those risks that warrant further management attention.

28.3 The CMMI

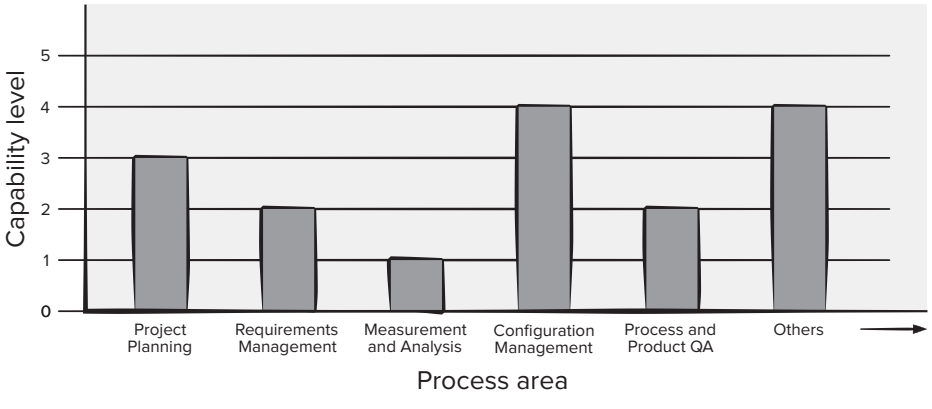
The original Capability Maturity Model (CMM) was developed and upgraded by the Software Engineering Institute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability Maturity Model Integration* (CMMI) [CMM18], a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure 28.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

Level 0: *Incomplete*. The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

FIGURE 28.2

CMMI process area capability profile
Source: Phillips, Mike, "CMMI V1.1 Tutorial," April 9, 2002.



Level 1: *Performed*. All the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: *Managed*. All capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM18].

Level 3: *Defined*. All capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" [CMM18].

Level 4: *Quantitatively managed*. All capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM18].

Level 5: *Optimized*. All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity levels and process areas is shown in Figure 28.3.

A software process, no matter how well conceived, will not succeed without talented, motivated software people. The *People CMM* suggests practices that improve the workforce competence and culture [CMM18a]. The goal of the People CMM is to encourage continuous improvement of generic workforce knowledge (called "core competencies"), specific software engineering and project management skills (called "workforce competencies"), and process-related abilities. Like the CMMI, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes.

FIGURE 28.3

Process areas required to achieve a maturity level

Level	Focus	Process Area
Optimizing	Continuous Process Improvement	Causal analysis and resolution Organizational innovation and deployment
Quantitatively Managed	Quantitative Management	Quantitative project management Organizational process performance
Defined	Process Standardization	Technical solution Verification Organizational training Integrated project management Integrated teaming Requirements development Validation Decision analysis and resolution Organizational environment for integration Product integration Organizational process definition Integrated supplier management Risk management Organizational process focus
Managed	Basic Product Management	Supplier agreement management Process and product quality assurance Project planning Requirements management Configuration management Measurement and analysis Project monitoring and control
Performed		

INFO



The CMMI—Should We, or Shouldn't We?

The CMMI is a process meta-model. It defines (in 700+ pages) the process characteristics that should exist if an organization wants to establish a software process that is complete. The question that has been debated for almost two decades is: "Is the CMMI overkill?" Like most things in life (and in software), the answer is not a simple yes or no.

The spirit of the CMMI should always be adopted. At the risk of oversimplification, it argues that software development must be taken seriously—it must be planned thoroughly, it must be controlled uniformly, it must be tracked accurately, and it must be conducted professionally. It must focus on the needs of project stakeholders, the skills of the software engineers, and the quality of the end product. No one would argue with these ideas.

The detailed requirements of the CMMI should be seriously considered if an organization builds large, complex systems that involve dozens or hundreds of people over many months or years. It may be that the CMMI is "just right" in such situations, if the organizational culture is amenable to standard process models and management is committed to making it a success. However, in other situations, the CMMI may simply be too much for an organization to successfully assimilate. Does this mean that the CMMI is "bad" or "overly bureaucratic" or "old-fashioned"? No . . . it does not. It simply means that what is right for one organizational culture may not be right for another.

The CMMI is a significant achievement in software engineering. It provides a comprehensive discussion of the activities and actions that should be present when an organization builds computer software. Even if a software organization chooses not to adopt its details, every software team should embrace its spirit and gain insight from its discussion of software engineering process and practice.

28.4 Other SPI Frameworks

Although the SEI's CMM and CMMI are the most widely applied SPI frameworks, several alternatives[7] have been proposed and are in use. We provide a brief overview of two of these frameworks.[8]

28.4.1 SPICE

The SPICE (*Software Process Improvement and Capability dEtermination*) model [SPI99] provides an SPI assessment framework that is compliant with ISO 15504-5:2015 and ISO 12207:2017. The purpose of SPICE was to provide a framework to assess a process and provide information on the strengths, weaknesses, and capabilities to help an organization achieve its goals. [Kar12l presents an overview of the SPI framework, including a model for process management, guidelines for conducting an assessment, and rating the process under consideration.

28.4.2 TickIT Plus

The TickIT auditing method [Tic18] ensures compliance with *ISO 9001:2015 for Software*—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

ISO 9001:2015 has adopted a "plan-do-check-act" cycle that is applied to the quality management elements of a software project. Within a software context, "plan" establishes the process objectives, activities, and tasks necessary to achieve high-quality software and resultant customer satisfaction. "Do" implements the software process (including both framework and umbrella activities). "Act" initiates software process improvement activities that continually work to improve the process. TickIT can be used throughout the "plan-do-check-act" cycle to ensure that SPI progress is being made. TickIT auditors assess the application of the cycle as a precursor to ISO 9001:2015 certification. For a detailed discussion of ISO 9001:2015 and TickIT, you should examine [Tic18] and [ISO15].

7 It's reasonable to argue that some of these frameworks are not so much "alternatives" as they are complementary approaches to SPI. A comprehensive table of many more SPI frameworks can be found at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.4787&rep=rep1&type=pdf.

8 If you have further interest, a wide array of print and Web-based resources is available for each.

28.5 SPI Return on Investment

SPI is hard work and requires substantial investment of dollars and people. Managers who approve the budget and resources for SPI will invariably ask the question: "How do I know that we'll achieve a reasonable return for the money we're spending?"

At a qualitative level, proponents of SPI argue that an improved software process will lead to improved software quality. They contend that an improved process will result in the implementation of better-quality filters (resulting in fewer propagated defects), better control of change (resulting in less project chaos), and less technical rework (resulting in lower cost and better time to market). But can these qualitative benefits be translated into quantitative results? The classic return on investment (ROI) equation is:

$$\text{ROI} = \frac{\Sigma(\text{benefits}) - \Sigma(\text{costs})}{\Sigma(\text{costs})} \times 100\%$$

where

benefits include the cost savings associated with higher product quality (fewer defects), less rework, reduced effort associated with changes, and the income that accrues from shorter time to market.

costs include both direct SPI costs (e.g., training, measurement) and indirect costs associated with greater emphasis on quality control and change management activities and more rigorous application of software engineering methods (e.g., the creation of a design model).

In the real world, these quantitative benefits and costs are sometimes difficult to measure with accuracy, and all are open to interpretation. But that doesn't mean that a software organization should conduct an SPI program without careful analysis of the costs and benefits that accrue. Even very small software organizations benefit from software process improvement, but they examine the ROI of the SPI activities they choose to employ [Lar16]. A comprehensive treatment of ROI for SPI can be found in a unique book by David Rico [Ric04].

28.6 SPI Trends

Over the past 35 years, many companies have attempted to improve their software engineering practices by applying an SPI framework to effect organizational change and technology transition. As we noted earlier in this chapter, over half fail in this endeavor. Regardless of success or failure, all spend significant amounts of money. David Rico [Ric04] reports that a typical application of an SPI framework such as the SEI CMMI can cost between $25,000 and $70,000 per person and take years to complete! It should come as no surprise that the future of SPI should emphasize a less costly and time-consuming approach.

To be effective in the twenty-first-century world of software development, future SPI frameworks must become significantly more agile [Bjø16]. Rather than an organizational focus (which can take years to complete successfully), contemporary SPI efforts should focus on the project level, working to improve a team process in weeks,

not months or years [Bjø16]. To achieve meaningful results (even at the project level) in a short time frame, complex framework models may give way to simpler models [Lar16]. Rather than dozens of key practices and hundreds of supplementary practices, an agile SPI framework should emphasize only a few pivotal practices (e.g., analogous to the framework activities discussed throughout this book) [Din16].

Any attempt at SPI demands a knowledgeable workforce, but education and training can be expensive and should be minimized (and streamlined). Rather than classroom courses (expensive and time consuming), future SPI efforts should rely on Web-based training that is targeted at pivotal practices. Rather than far-reaching attempts to change organizational culture (with all the political perils that ensue), cultural change should occur as it does in the real world, one small group at a time until a tipping point is reached. Jovanovic and his colleagues suggest using retrospective gaming as part of the Scrum retrospective as a means of educating and engaging agile development team members in process improvement [Jov15].

The SPI work of the past three decades has significant merit. The frameworks and models that have been developed represent substantial intellectual assets for the software engineering community. But like all things, these assets guide future attempts at SPI not by becoming a recurring dogma, but by serving as the basis for better, simpler, and more agile SPI models.

28.7 SUMMARY

A software process improvement framework defines the characteristics that must be present if an effective software process is to be achieved, an assessment method that helps determine whether those characteristics are present, and a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing. Regardless of who in the organization sponsors SPI, the goal is to improve process quality and to improve software quality and timeliness.

A process maturity model provides an overall indication of the "process maturity" exhibited by a software organization. It provides a qualitative feel for the relative effectiveness of the software process that is currently being used.

The SPI road map begins with assessment—a series of evaluation activities that uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process. Assessment is the tool that allows a software organization to develop an overall SPI plan.

One of the key elements of any SPI plan is education and training, an activity that focuses on improving the knowledge level of managers and practitioners. Once staff becomes well versed in current software technologies, selection and justification commence. These tasks lead to choices about the architecture of the software process, the methods that populate it, and the tools that support it. Installation and evaluation are SPI activities that instantiate process changes and assess their efficacy and impact.

To successfully improve its software process, an organization must exhibit the following characteristics: management commitment and support for SPI, staff involvement throughout the SPI process, process integration into the overall organizational culture, an SPI strategy that has been customized for local needs, and solid management of the SPI project.

Several SPI frameworks are in use today. The SEI's CMM and CMMI are widely used. The People CMM has been customized to assess the quality of the organizational culture and the people who populate it. SPICE and TickIT are additional frameworks that can lead to effective SPI.

SPI is hard work that requires substantial investment of dollars and people. To ensure that a reasonable return on investment is achieved, an organization must measure the costs associated with SPI and the benefits that can be directly attributed to it.

Problems and Points to Ponder

28.1. Why is it that software organizations often struggle when they embark on an effort to improve local software process?

28.2. Describe the concept of "process maturity" in your own words.

28.3. Do some research (check the SEI website), and determine the process maturity distribution for software organizations in the United States and worldwide.

28.4. You work for a very small software organization—only 11 people are involved in developing software. Is SPI for you? Explain your answer.

28.5. Assessment is analogous to an annual physical exam. Using a physical exam as a metaphor, describe the SPI assessment activity.

28.6. What is the difference between an "as is" process, a "here to there" process, and a "to be" process?

28.7. How is risk management applied within the context of SPI?

28.8. Do some research on the key factors for predicting success in software process improvement efforts. Pick one of them, and write a paper describing how it might be achieved in a small software development organization.

28.9. Do some research to try to determine how CMMI can be used with agile process frameworks.

28.10. Select one of the SPI frameworks discussed in Section 28.5, and write a brief paper describing it in more detail.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

29 EMERGING TRENDS IN SOFTWARE ENGINEERING

Throughout the relatively brief history of software engineering, practitioners and researchers have developed an array of process models, technical methods, and automated tools in an effort to foster fundamental change in the way we build computer software. Even though past experience indicates otherwise, there is a tacit desire to find the "silver bullet"—the magic process or transcendent technology that will allow us to build large, complex, software-based systems easily, without confusion, without mistakes, without delay—without the many problems that continue to plague software work.

KEY CONCEPTS

building blocks 591
collaborative development 595
complexity 588
crowd sourcing 596
emergent requirements 590
hype cycle 586
innovation life cycle 584
model-driven software development 596
open source 592
open-world software 589
postmodern design 601
requirements engineering 596
search-based software engineering 597
soft trends 587
technology directions 593
technology evolution 584
test-driven development 598
tools 599
variability intensive systems 590

QUICK LOOK

What is it? No one can predict the future with absolute certainty. But it is possible to assess trends in the software engineering area and from those trends to suggest possible directions for the technology. That's what we attempt to do in this chapter.

Who does it? Anyone who is willing to spend the time to stay abreast of software engineering issues can try to predict the future direction of the technology.

Why is it important? Why did ancient kings hire soothsayers? Why do major multinational corporations hire consulting firms and think tanks to prepare forecasts? Why does a substantial percentage of the public read horoscopes? We want to know what's coming so we can ready ourselves.

What are the steps? There is no formula for predicting the road ahead. We attempt to do this by collecting data, organizing it to provide useful information, examining subtle associations to extract knowledge, and from this knowledge to suggest probable trends that predict how things will be at some future time.

What is the work product? A view of the near-term future that may or may not be correct.

How do I ensure that I've done it right? Predicting the road ahead is an art, not a science. In fact, it's quite rare when a serious prediction about the future is absolutely right or unequivocally wrong (with the exception, thankfully, of predictions of the end of the world). We look for trends and try to extrapolate them. We can assess the correctness of the extrapolation only as time passes.

But history indicates that our quest for the silver bullet appears doomed to failure. New technologies are introduced regularly, hyped as "solutions" to many of the problems software engineers face, and incorporated into projects large and small. Industry pundits stress the importance of these "new" software technologies, the cognoscenti of the software community adopt them with enthusiasm, and ultimately, they do play a role in the software engineering world. But they tend not to meet their promise, and as a consequence, the quest continues.

In past editions of this book (approaching four decades), we have discussed emerging technologies and their projected impact on software engineering. Some have been widely adopted, but others never reached their potential. Our conclusion: Technologies come and go; the real trends that we should explore are softer. By this we mean that progress in software engineering will be guided by business, organizational, market, and cultural trends. Those trends lead to technology innovation.

In this chapter, we'll mention a few software engineering technology trends, but our primary emphasis will be on some of the business, organizational, market, and cultural trends that may have an important influence on software engineering technology over the next 10 or 20 years.

29.1 TECHNOLOGY EVOLUTION

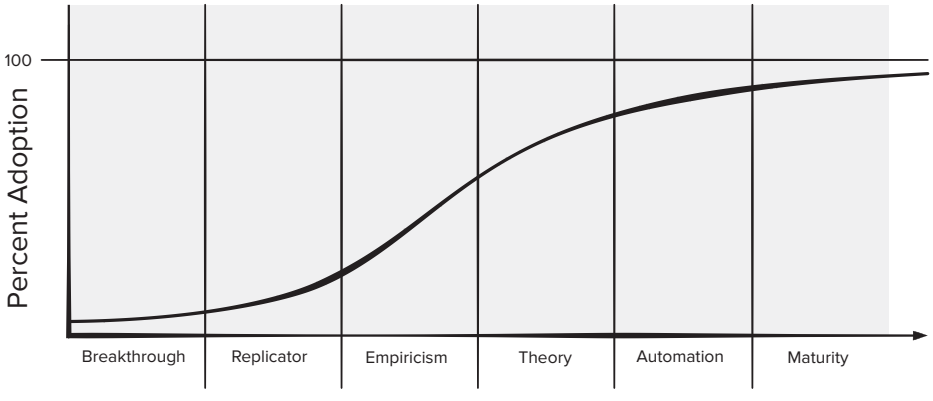
In a fascinating book that provides a compelling look at how computing (and other related) technologies will evolve, Ray Kurzweil [Kur05] argued that technological evolution is similar to biological evolution, but at a rate that is orders of magnitude faster. Evolution (whether biological or technological) occurs as a result of positive feedback—"the more capable methods resulting from one stage of evolutionary progress are used to create the next stage" [Kur05].

The big questions for the twenty-first century are: (1) How rapidly does a technology evolve? (2) How significant are the effects of positive feedback? (3) How profound will the resultant changes be?

When a successful new technology is introduced, the initial concept moves through a reasonably predictable "innovation life cycle" [Gai95], as illustrated in Figure 29.1.

FIGURE 29.1

A technology innovation cycle



In the *breakthrough* phase, a problem is recognized and repeated attempts at a viable solution are attempted. At some point, a solution shows promise. The initial breakthrough work is reproduced in the *replicator* phase and gains wider usage. *Empiricism* leads to the creation of empirical rules that govern the use of the technology, and repeated success leads to a broader *theory* of usage that is followed by the creation of automated tools during the *automation* phase. Finally, the technology matures and is used widely.

You should note that many research and technology trends never reach maturity. In fact, the vast majority of "promising" technologies in the software engineering domain receive widespread interest for a few years and then fall into niche usage by a dedicated band of adherents. This is not to say that these technologies lack merit, but rather to emphasize that the journey through the innovation life cycle is long and hard.

Computing technology is evolving at an exponential rate, and its growth may soon become explosive. Kurzweil [Kur05] agrees that computing technologies evolve through an "S-curve" that exhibits relatively slow growth during the technology's formative years, rapid acceleration during its growth period, and then a leveling-off period as the technology reaches its limits. Today, we are at the knee of the S-curve for modern computing technologies—at the transition between early growth and the explosive growth that is to follow. The implication is that over the next 20 to 40 years, we will see dramatic (even mindboggling) changes in computing capability. He suggests that within 20 years, technology evolution will accelerate at an increasingly rapid pace, ultimately leading to an era of nonbiological intelligence that will merge with and extend human intelligence in ways that are fascinating to contemplate.

And all of this, no matter how it evolves, will require software and systems that make our current efforts look infantile by comparison. By the year 2040, a combination of extreme computation, artificial intelligence and machine learning, nanotechnology, massively high bandwidth ubiquitous networks, and robotics will lead us into a different world.[1] Software—possibly in forms we cannot yet comprehend—will continue to reside at the core of this new world. Software engineering will not go away.

29.2 SOFTWARE ENGINEERING AS A DISCIPLINE

For almost 50 years, many academic researchers and industry professionals have clamored for a true engineering discipline for software. In an important follow-on to her classic 1990 paper on the subject, Mary Shaw [Sha09] comments on this continuing quest:

> Engineering disciplines typically evolve from craft practices of a technology, sufficient for local or ad hoc use. When the technology becomes economically significant, it requires stable production techniques and management control. The resulting commercial market is based on experience, rather than a deep understanding of the technology . . . an engineering profession emerges when . . . science becomes sufficiently mature to support purposeful practice and design evolution with predictable outcomes.

1 Kurzweil [Kur05] presents a reasoned technical argument that predicts a strong artificial intelligence (that will pass the Turing Test) by 2029 and suggests that the evolution of humans and machines will begin to merge by 2045. The vast majority of readers of this book will live to see whether this, in fact, takes place.

We would argue that the industry has achieved "purposeful practice," but that "predictable outcomes" have remained elusive.

As mobility begins to dominate the software landscape, Shaw identifies challenges that "emerge from the deep interdependencies between very complex systems and their users" [Sha09]. She argues that the knowledge base that leads to "purposeful practice" has been democratized by the specialized social networks that now populate the Web. For example, rather than referencing a centrally controlled software engineering handbook, a software developer can pose a problem on an appropriate forum and obtain a crowd-sourced solution that draws from the experience of many other developers. The proposed solution if often critiqued in real time, with alternatives and adaptations offered as options.

But this is not the level of discipline that many demand. As Shaw states: "[P]roblems facing software engineers are increasingly situated in complex social contexts and delineating the problem's boundaries is increasingly difficult" [Sha09]. As a consequence, isolating the scientific underpinnings of a discipline remains a challenge. At this point in the history of our field, it is reasonable to state that "the discovery of new software engineering ideas is, by now, naturally incremental and evolutionary" [Erd10].

29.3 Observing Software Engineering Trends

Barry Boehm [Boe08] suggests that "software engineers [will] face the formidable challenges of dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence, but they also have opportunities to make significant contributions that will make a difference for the better." But what are the trends that will enable you to face these challenges in the years ahead?

In the introduction to this chapter, we noted that "soft trends" have a significant impact on the overall direction of software engineering. But other ("harder") research- and technology-oriented trends remain important. Research trends "are driven by general perceptions of the state of the art and the state of the practice, by researcher perceptions of practitioner needs, by national funding programs that rally around specific strategic goals, and by sheer technical interest" [Mil00b]. Technology trends occur when research trends are extrapolated to meet industry needs and are shaped by market-driven demand.

In Section 29.1, we discussed the S-curve model for technology evolution. The S-curve is appropriate for considering the long-term effects of core technologies as they evolve. But what of more modest, short-term innovations, tools, and methods? The Gartner Group [Gar08]—a consultancy that studies technology trends across many industries—has developed a *hype cycle for emerging technologies,* represented in Figure 29.2. The "hype cycle" presents a realistic view of short-term technology integration. The long-term trend, however, is exponential. Not every software engineering technology makes it all the way through the hype cycle. In some cases, the disillusionment is justified, and the technology is relegated to obscurity.

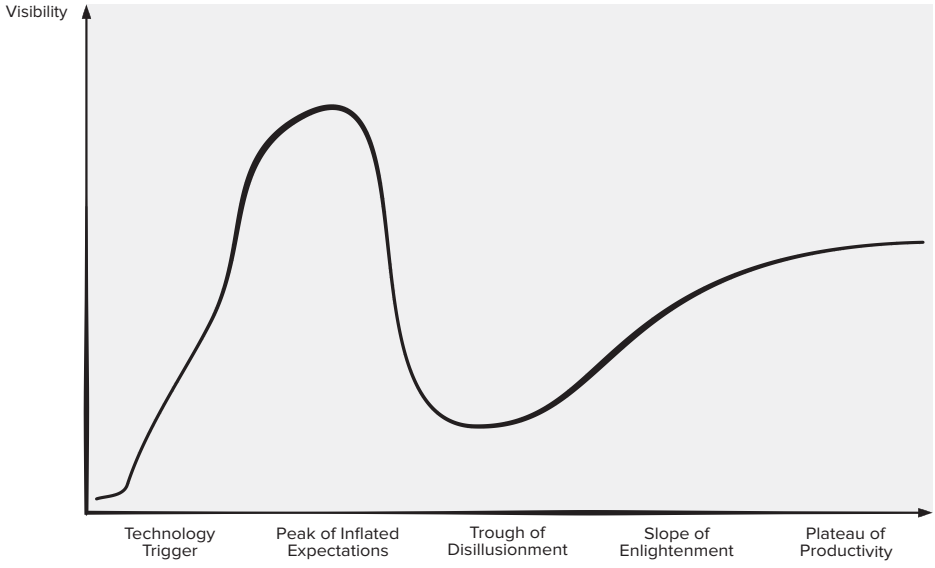


FIGURE 29.2

The Gartner Group's hype cycle for emerging technologies
Source: Linden, Alexander, Fenn, Jackie, "Understanding Gartner's Hype Cycles," Strategic Analysis Report, Gartner, Inc., May 30, 2003, 5.

29.4 IDENTIFYING "SOFT TRENDS"

Each nation with a substantial IT industry has a set of unique characteristics that define the manner in which business is conducted, the organizational dynamics that arise within a company, the distinct marketing issues that apply to local customers, and the overriding culture that dictates all human interaction. However, some trends in each of these areas are universal and have as much to do with sociology, anthropology, and group psychology (often referred to as the "soft sciences") as they do with academic or industrial research.

Connectivity and collaboration (enabled by high-bandwidth communication) has already led to software teams that do not occupy the same physical space (telecommuting and part-time employment in a local context). One team collaborates with other teams that are separated by time zones, primary language, and culture. Software engineering must respond with an overarching process model for "distributed global engineering teams" that is agile enough to meet the demands of immediacy but disciplined enough to coordinate disparate groups.

Globalization leads to a diverse workforce (in terms of language, culture, problem resolution, management philosophy, communication priorities, and person-to-person interaction). This, in turn, demands a flexible organizational structure. Different teams (in different countries) must respond to engineering problems in a way that best accommodates their unique needs, while at the same time fostering a level of uniformity that allows an overall global project to proceed. This type of organization suggests fewer levels of management and a greater emphasis

on team-level decision making. It can lead to greater agility, but only if communication mechanisms have been established so that every team can understand project and technical status (via networked groupware) at any time. Software engineering methods and tools can help achieve some level of uniformity (teams speak the same "language" implemented through specific methods and tools). Software process can provide the framework for the instantiation of these methods and tools.

In some world regions (the United States and Europe are examples), the population is aging. This undeniable demographic (and cultural trend) implies that many experienced software engineers and managers will be leaving the field over the coming decades. The software engineering community must respond with viable mechanisms that capture the knowledge of these aging managers and technologists [e.g., the use of *patterns* (Chapter 14) is a step in the right direction], so that it will be available to future generations of software workers. In other regions of the world, the number of young people available to the software industry is exploding. This provides an opportunity to mold a software engineering culture without the burden of 50 years of "old-school" prejudices.

It is estimated that over 1 billion new consumers will enter the worldwide marketplace over the next decade. Consumer spending in emerging economies is projected to grow to about $8 trillion by 2022 [Jai18]. The digitally influenced component of that spending will top $4 trillion. The implication—an increasing demand for new software. Can new software engineering technologies be developed to meet this worldwide demand? Modern market trends are often driven by the supply side.[2] In other cases, demand-side requirements drive the market. In either case, a cycle of innovation and demand progresses in a way that sometimes makes it difficult to determine which came first!

Finally, human culture itself will impact the direction of software engineering. Every generation establishes its own imprint on local culture, and yours will be no different. Faith Popcorn [Pop08], a well-known consultant who specializes in cultural trends, characterizes them in the following manner: "Our Trends are not fads. Our Trends endure. Our Trends evolve. They represent underlying forces, first causes, basic human needs, attitudes, aspirations. They help us navigate the world, understand what's happening and why, and prepare for what is yet to come." A detailed discussion of how modern cultural trends will have an impact on software engineering is best left to those who specialize in the "soft sciences."

29.4.1 Managing Complexity

When the first edition of this book was written (1982), digital consumer products as we now know them today didn't exist, and mainframe-based systems containing a million lines of source code (LOC) were considered to be quite large. Today, small digital devices typically house between 60,000 to 200,000 lines of custom software, coupled with a few million LOC for operating system features. Modern computer-based

2 Supply side adopts a "build it and they will come" approach to markets. Unique technologies are created, and consumers flock to adopt them—sometimes!

systems containing 10 to 50 million lines of code are common.[3] In the relatively near future, systems[4] requiring over 1 billion LOC will begin to emerge.[5]

Think about that for a moment!

Consider the interfaces for a billion LOC system, both to the outside world, to other interoperable systems, to the Internet (or its successor), and to the millions of internal components that must all work together to make this computing monster operate successfully. Is there a reliable way to ensure that all these connections will allow information to flow properly?

Consider the project itself. How do we manage the work flow and track progress? Will conventional approaches scale upward by orders of magnitude?

Consider the number of people (and their locations) who will be doing the work, the coordination of people and technology, the unrelenting flow of changes, the likelihood of a multiplatform, multioperating system environment. Is there a way to manage and coordinate people who are working on a monster project?

Consider the engineering challenge. How can we analyze tens of thousands of requirements, constraints, and restrictions in a way that ensures that inconsistency and ambiguity, omissions, and outright errors are uncovered and corrected? How can we create a design architecture that is robust enough to handle a system of this size? How can software engineers establish a change management system that will have to handle hundreds of thousands of changes?

Consider the challenge of quality assurance. How can we perform verification and validation in a meaningful way? How do you test a 1-billion-LOC system?

In the early days, software engineers attempted to manage complexity in what can only be described as an ad-hoc fashion. Today, we use process, methods, and tools to keep complexity under control. But tomorrow? Is our current approach up to the task?

In the future, we are likely to see wider use of artificial intelligence techniques to help software engineers manage these levels of complexity [Har12b], [Xie18]. Machine learning is one such technique that can help with testing and bug fixing [Mei18]. Data science techniques can be used to help make sense of the vast amount of software engineering data generated by these large projects [Kim16b]. Mining of these repositories is becoming an accepted research technique in the software engineering communities [Dye15].

29.4.2 Open-World Software

Concepts such as ambient intelligence,[6] context-aware applications, and pervasive/ubiquitous computing all focus on integrating software-based systems into an

3 For example, modern laptop operating systems (e.g., Linux, macOS, and Windows) have between 30 and 60 million LOC. Operating system software for mobile devices can exceed 2 million LOC.

4 In reality, this "system" will actually be a system of systems—hundreds of interoperable applications working together to achieve some overall objective, and many will be cloud based.

5 Not all complex systems are large. A relatively small application (say, less than 100,000 LOC) can still be exceedingly complex.

6 A worthwhile and quite detailed introduction to *ambient intelligence* can be found at https://www.researchgate.net/publication/220737998_Ambient_Intelligence_Basic_Concepts_and_Applications.

environment far broader than a laptop, a mobile computing device, or any other digital device. These separate visions of the near-term future of computing collectively suggest "open-world software"—software that is designed to adapt to a continually changing environment "by self-organizing its structure and self-adapting its behavior" [Bar06b].

To help illustrate the challenges that software engineers will face in the near future, consider the notion of *ambient intelligence* (amI). Ducatel [Duc01] defines amI in the following way: "People are surrounded by intelligent, intuitive interfaces that are embedded in all kinds of objects. The ambient intelligence environment is capable of recognizing and responding to the presence of different individuals [while working] in a seamless, unobtrusive way."

With the widespread use of low-cost, yet increasingly powerful smartphones, we are well on our way to ubiquitous amI systems. The challenge for software engineers is to develop apps that provide ever-increasing functionality in products of all types—functionality that adapts to user needs while at the same time protecting privacy and providing security. The rise of digital assistants and intelligent chat bots are indicative of the types of applications ahead.

The engineering of *variability intensive systems* focuses on software that needs to accommodate different usage and deployment scenarios, as well as intentional and unintentional variability in functionality or quality attributes (e.g., performance). This includes meeting the challenges posed by context-aware apps, autonomous agents, and pervasive computing as well as creating product-line software.[7] These systems can be highly variable during all software engineering activities (e.g., dynamic run-time conditions, rapidly changing requirements, configuration management), and we need to improve our understanding of how to design and manage them in a cost-effective manner [Gal17].

29.4.3 Emergent Requirements

At the beginning of a software project, there's a truism that applies equally to every stakeholder involved: "You don't know what you don't know." That means that customers rarely define "stable" requirements. It also means that software engineers cannot always foresee where ambiguities and inconsistencies lie. Requirements change—but that's nothing new.

As systems become more complex, it follows that even a rudimentary attempt to state comprehensive requirements is doomed to failure. A statement of overall goals may be possible, delineation of intermediate objectives can be accomplished, but stable requirements—not a chance! Requirements will emerge as everyone involved in the engineering and construction of a complex system learns more about it, the environment in which it is to reside, and the users who will interact with it.

This reality implies a number of software engineering trends. First, process models must be designed to embrace change and adopt the basic tenets of the agile philosophy (Chapter 3). Next, methods that yield engineering models (e.g., requirements and

7 Product-line software is a set of application programs that are built from a common set of reusable software modules designed to be easily adaptable when creating in new software products.

design models) must be used judiciously because those models will change repeatedly as more knowledge about the system is acquired. Finally, tools that support both process and methods must make adaptation and change easy.

But there is another aspect to emergent requirements. The vast majority of software developed to date assumes that the boundary between the software-based system and its external environment is stable. The boundary may change, but it will do so in a controlled manner, allowing the software to be adapted as part of a regular software maintenance cycle. This assumption is beginning to change. The growth of variability intensive systems (Section 29.4.2) demands that software "adapt and react to changes dynamically, even if they're unanticipated" [Bar06b].

By their nature, emergent requirements lead to change. How do we control the evolution of a widely used application or system over its lifetime, and what effect does this have on the way we design software?

As the number of changes grows, the likelihood of unintended side effects also grows. This should be a cause for concern as complex systems with emergent requirements become the norm. The software engineering community must develop methods that help software teams predict the impact of change across an entire system, thereby mitigating unintended side effects. Today, our ability to accomplish this is severely limited.

29.4.4 The Talent Mix

As software-based systems become more complex, as communication and collaboration among global teams becomes commonplace, as emergent requirements (with the resultant flow of changes) become the norm, the very nature of a software engineering team may change. Each software team must bring a variety of creative talent and technical skills to its part of a complex system, and the overall process must allow the output of these islands of talent to merge effectively. Using data mining for knowledge discovery in the human aspects of software engineering may help managers select the right development team before beginning a project [Gup15].

Alexandra Weber-Morales [Mor05] suggests the talent mix of a "software dream team." The *Brain* is a chief architect who is able to navigate the demands of stakeholders and map them into a technology framework that is both extensible and implementable. The *Data Grrl* is a database and data structures guru who "blasts through rows and columns with profound understanding of predicate logic and set theory as it pertains to the relational model." The *Blocker* is a technical leader (manager) who allows the team to work free of interference from other teams while at the same time ensuring that collaboration is occurring. The *Hacker* is a consummate programmer who is at home with patterns and languages and can use both effectively. The *Gatherer* "deftly discovers system requirements with . . . anthropological insight" and accurately expresses them with clarity.

29.4.5 Software Building Blocks

All of us who have fostered a software engineering philosophy have emphasized the need for reuse—of source code, object-oriented classes, components, patterns, and libraries. Although the software engineering community has made progress as it attempts to capture past knowledge and reuse proven solutions, a significant percentage

of the software that is built today continues to be built "from scratch." Part of the reason for this is a continuing desire (by stakeholders and software engineering practitioners) for "unique solutions."

In the hardware world, original equipment manufacturers (OEMs) of digital devices use application-specific standard products (ASSPs) produced by silicon vendors almost exclusively. This "merchant hardware" provides the building blocks necessary to implement everything from a smartphone to a wearable computing device. Increasingly, the same OEMs are using "merchant software"—software building blocks designed specifically for a unique application domain [e.g., Voice over Internet Protocol (VoIP) devices]. Michael Ward [War07] comments:

> One advantage of the use of software components is that the OEM can leverage the functionality provided by the software without having to develop in-house expertise in the specific functions or invest developer time on the effort to implement and validate the components. Other advantages include the ability to acquire and deploy only the specific set of functionalities that are needed for the system, as well as the ability to integrate these components into an already-existing architecture.

In addition to components packaged as merchant software, there is an increasing tendency to adopt *software platform solutions* that "incorporate collections of related functionalities, typically provided within an integrated software framework" [War07]. A software platform frees an OEM from the work associated with developing base functionality and instead allows the OEM to dedicate software effort on those features that differentiate its product.

29.4.6 Changing Perceptions of "Value"

During the last quarter of the twentieth century, the operative question that businesspeople asked when discussing software was: Why does it cost so much? That question is rarely asked today and has been replaced by: Why can't we get it (software and/or the software-based product) sooner?

When computer software is considered, the modern perception of value is changing from business value (cost and profitability) to customer values that include: speed of delivery, richness of functionality, and overall product quality.

29.4.7 Open Source

Who owns the software you or your organization uses? Increasingly, the answer is "everyone." The "open source" movement has been described in the following manner [OSO12]: "Open source is a development method for software that harnesses the power of distributed peer review and transparency of process. The promise of open source is better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in." The term *open source,* when applied to computer software, implies that software engineering work products (models, source code, test suites) are open to the public and can be reviewed and extended (with controls) by anyone with interest and permission.

If you have further interest, Weber [Web05] provides a worthwhile introduction, Feller and his colleagues [Fel07] have edited a comprehensive and objective anthology that considers the benefits and problems associated with open source, and Brown [Bro12] provides a more technical discussion.

29.5 TECHNOLOGY DIRECTIONS

We always seem to think that software engineering will change more rapidly than it does. A new "hyped" technology (it could be a new process, a unique method, or an exciting tool) is introduced, and pundits suggest that "everything" will change. But software engineering is about far more than technology—it's about people and their ability to communicate their needs and innovate to make those needs a reality. Whenever people are involved, change occurs slowly in fits and starts. It's only when a "tipping point" [Gla02] is reached that a technology cascades across the software engineering community and broad-based change truly does occur.

In this section we'll examine a few trends in process, methods, and tools that are likely to have some influence on software engineering over the next decade. Will they lead to a tipping point? We'll just have to wait and see.

29.5.1 Process Trends

It can be argued that all the business, organizational, and cultural trends discussed in Section 29.4 reinforce the need for process. But do the frameworks discussed in Chapter 28 provide a road map into the future? Will process frameworks evolve to find a better balance between discipline and creativity? Will the software process adapt to the differing needs of stakeholders who procure software, those who build it, and those who use it? Can it provide a means for reducing risk for all three constituencies at the same time?

These and many other questions remain open. In the paragraphs that follow, we have adapted six ideas proposed by Conradi and Fuggetta [Con02] to suggest possible process trends.

1. **As SPI frameworks evolve, they will emphasize "strategies that focus on goal orientation and product innovation"** [Con02]. In the fast-paced world of software development, long-term SPI strategies rarely survive in a dynamic business environment. Too much changes too quickly. This means that a stable, step-by-step road map for SPI may have to be replaced with a framework that emphasizes short-term goals that have a product orientation.
2. **Because software engineers have a good sense of where the process is weak, process changes should generally be driven by their needs and should start from the bottom up.** Conradi and Fuggetta [Con02] suggest that future SPI activities should "use a simple and focused scorecard to start with, not a large assessment." By focusing SPI efforts narrowly and working from the bottom up, practitioners will begin to see substantive changes early—changes that make a real difference in the way that software engineering work is conducted.
3. **Automated software process technology (SPT) will move away from global process management (broad-based support of the entire software process) to focus on those aspects of the software process that can best benefit from automation.** No one is against tools and automation, but in many instances, SPI has not met its promise (see Section 29.3). To be most effective, it should focus on umbrella activities (Chapter 1)—the most stable elements of the software process.

4. **Greater emphasis will be placed on the return on investment of SPI activities.** In Chapter 28 you learned that return on investment (ROI) can be defined as:

$$\text{ROI} = \frac{\Sigma(\text{benefits}) - \Sigma(\text{costs})}{\Sigma(\text{costs})} \times 100\%$$

   To date, software organizations have struggled to clearly delineate "benefits" in a quantitative manner. It can be argued [Con02] that "we therefore need a standardized market-value model . . . to account for software improvement initiatives."

5. **As time passes, the software community may come to understand that expertise in sociology and anthropology may have as much or more to do with successful SPI as other, more technical disciplines.** More than anything else SPI changes organizational culture, and cultural change involves individuals and groups of people. Conradi and Fuggetta [Con02] correctly note that "software developers are knowledge workers. They tend to respond negatively to top-level dictates on how to do work or change processes." Much can be learned by examining the sociology of groups to better understand effective ways to introduce change.

6. **New modes of learning may facilitate the transition to a more effective software process.** In this context, "learning" implies learning from successes and mistakes. A software organization that collects metrics (Chapter 23) allows itself to understand how elements of a process affect the quality of the end product.

29.5.2 The Grand Challenge

There is one trend that is undeniable—software-based systems will undoubtedly become bigger and more complex as time passes. It is the engineering of these large, complex systems, regardless of delivery platform or application domain, that poses the "grand challenge" [Bro06] for software engineers. Manfred Broy [Bro06] suggests that software engineers can meet "the daunting challenge of complex software systems development" by creating new approaches to understanding system models and using those models as a basis for the construction of high-quality next-generation software. Techniques currently being studied for variability intensive systems (continuous delivery, self-adaptive software, value-based software engineering, content aware computing) may benefit the developers of all types of software products [Gal17].

As the software engineering community develops new model-driven approaches (discussed briefly later in this section) to the representation of system requirements and design, the following characteristics [Bro06] must be addressed:

- **Multifunctionality.** As digital devices evolve, they have begun to deliver a rich set of sometimes unrelated functions. The mobile phone, once considered a straightforward communication device, has become a powerful pocket computer that performs a wide spectrum of functions that are arguably more important than making a phone call. As Broy [Bro06] notes, "[E]ngineers must describe the detailed context in which the functions will be delivered

and, most important, must identify the potentially harmful interactions between the system's different features."

- **Reactivity and timeliness.** Digital devices increasingly interact with the real world and must react to external stimuli in a timely manner. They must interface with a broad array of sensors and must respond in a period that is appropriate to the task at hand. New methods must be developed that (1) help software engineers predict the timing of various reactive features and (2) implement those features in a way that makes the feature less machine dependent and more portable.
- **New modes of user interaction.** Open-world trends for software mean that new modes of interaction must be modeled and implemented. Whether these new approaches use multitouch interfaces, voice recognition, or direct mind interfaces, new generations of software for digital devices must accommodate them.
- **Complex architectures.** A luxury automobile has over 2,000 functions controlled by software residing within a complex hardware architecture that includes multiple processors, a sophisticated bus structure, actuators, sensors, an increasingly sophisticated human interface, and many safety-rated components. Even more complex systems (e.g., autonomous vehicles) are on the immediate horizon, presenting significant challenges for software designers.
- **Heterogeneous, distributed systems.** The real-time components of any modern embedded system can be connected via an internal bus, a wireless network, or across the Internet (or all three).
- **Criticality.** Software has become the pivotal component in virtually all business-critical systems and in most safety-critical systems. Yet, the software engineering community has only begun to apply even the most basic principles of software safety.
- **Maintenance variability.** The life of software within a digital device rarely lasts beyond 3 to 5 years, but the complex avionics systems within an aircraft has a useful life of at least 20 years. Automobile software falls somewhere in between. Should this have an impact on design?

Broy [Bro06] argues that these and other software characteristics can be managed only if the software engineering community develops a more effective distributed and collaborative software engineering philosophy, better requirements engineering approaches, a more robust approach to model-driven development, and better software tools. In the sections that follow, we'll explore each of these areas briefly.

29.5.3 Collaborative Development

It seems too obvious to state, but we'll do so anyway: Software engineering is an information technology. From the onset of any software project, every stakeholder must share information—about basic business goals and objectives, about specific system requirements, about architectural design issues, about almost every aspect of the software to be built. Collaboration involves the timely dissemination of information and an effective process for communication and decision making.

Today, software engineers collaborate across time zones and international boundaries. Every one of them must share information. The same holds for open-source

projects in which hundreds or thousands of software developers work to build an open-source app. Crowd sourcing has been suggested as a means of enhancing coverage test cases generated by automated testing tools [Mao17]. Coordination of such large testing communities will be challenging. Information must be disseminated so that open collaboration can occur.

29.5.4 Requirements Engineering

Basic requirements engineering actions—elicitation, elaboration, negotiation, specification, and validation—were presented in Chapters 7 and 8. The success or failure of these actions has a very strong influence on the success or failure of the entire software engineering process. And yet, requirements engineering (RE) has been compared to "trying to put a hose clamp around jello" [Gon04]. As we've noted in many places throughout this book, software requirements have a tendency to keep changing, and with the advent of open-world systems, emergent requirements (and near-continuous change) may become the norm.

Today, most "informal" requirements engineering approaches begin with the creation of user scenarios (e.g., use cases). More formal approaches create one or more requirements models and use these as a basis for design. Formal methods enable a software engineer to represent requirements using a verifiable mathematical notation. All can work reasonably well when requirements are stable, but do not readily solve the problem of dynamic or emergent requirements.

There are a number of distinct requirements engineering research directions including natural language processing from translated textual descriptions into more structured representations (e.g., analysis classes), greater reliance on databases for structuring and understanding software requirements, the use of RE patterns to describe typical problems and solutions when requirements engineering tasks are conducted, and goal-oriented requirements engineering. However, at the industry level, RE actions remain relatively informal and surprisingly basic. To improve the manner in which requirements are defined, the software engineering community will likely implement three distinct subprocesses as RE is conducted [Gli07]: (1) improved knowledge acquisition and knowledge sharing that allows more complete understanding of application domain constraints and stakeholder needs, (2) greater emphasis on iteration as requirements are defined, and (3) more effective communication and coordination tools that enable all stakeholders to collaborate effectively.

The RE subprocesses noted in the preceding paragraph will only succeed if they are properly integrated into an evolving approach to software engineering. As pattern-based problem solving and component-based solutions begin to dominate many application domains, RE must accommodate the desire for agility (rapid incremental delivery) and the inherent emergent requirements that result. The notion of a static "software specification" is beginning to disappear, to be replaced by "value-driven requirements" [Som05] derived as stakeholders respond to features and functions delivered in early software increments.

29.5.5 Model-Driven Software Development

Software engineers grapple with abstraction at virtually every step in the software engineering process. As design commences, architectural and component-level

abstractions are represented and assessed. They must then be translated into a programming language representation that transforms the design (a relatively high level of abstraction) into an operable system with a specific computing environment (a low level of abstraction). *Model-driven software development*[8] couples domain-specific modeling languages with transformation engines and generators in a way that facilitates the representation of abstraction at high levels and then transforms it into lower levels [Sch06]. Model-driven approaches address a continuing challenge for all software developers—how to represent software at a higher level of abstraction than code.

Domain-specific modeling languages (DSMLs) represent "application structure, behavior and requirements within particular application domains" and are described with meta-models that "define the relationships among concepts in the domain and precisely specify the key semantics and constraints associated with these domain concepts" [Sch06]. The key difference between a DSML and a general-purpose modeling language such as UML (Appendix 1) is that the DSML is tuned to design concepts inherent in the application domain and can therefore represent relationships and constraints among design elements in an efficient manner.

29.5.6 Search-Based Software Engineering

Many activities in software engineering can be stated as optimization problems. *Search-based software engineering* (SBSE) applies metaheuristic search techniques such as genetic algorithms[9] to software engineering problems. Lionel Briand [Bri09] believes that evolutionary and other search techniques are more easily scaled to industrial-size problems than model-driven techniques and that there are opportunities for synergy between the two. Search-based software engineering was devised on the premise that it is often easier to check that a candidate solution solves a problem than it is to construct a solution from scratch [Kul13].

Search-based software engineering techniques can be used as the basis for genetic improvement to grow software products by grafting on new functional and nonfunctional features to existing software product line [Har14]. Genetic improvement of software has already resulted in dramatic performance improvements (e.g., execution time, energy usage, and memory consumption) in existing software products [Pet18]. Successful software products evolve continually; however, evolution, if not properly managed, may weaken the software quality and may need to be refactored to remain viable.

Search-based software engineering techniques have been used to generate and repair sequences of refactoring recommendations. It is time consuming to create refactoring recommendations manually. Using a dynamic, interactive approach to generate refactoring recommendations can improve software quality while minimizing deviations from the original design [Ali18]. Search-based software engineering techniques have been used to design test cases to evaluate developer fixes to the software following crashes [Als18]. It is possible that these techniques may lead to software systems capable of repairing themselves.

---

8 The term *model-driven engineering* (MDE) is also used.

9 Genetic algorithms can be used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover, and selection to evolve potential solutions from a population of potential solutions.

29.5.7 Test-Driven Development

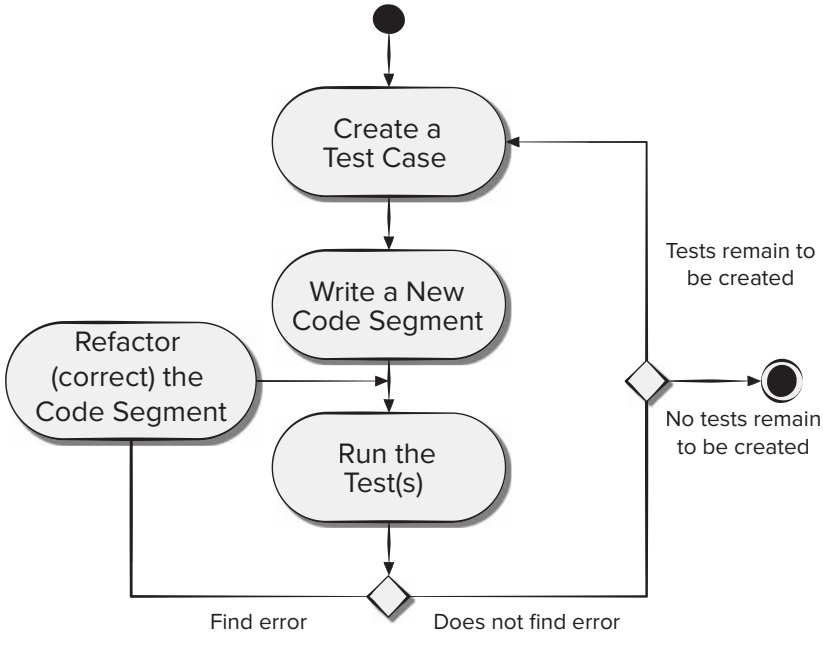
Requirements drive design, and design establishes a foundation for construction. This simple software engineering reality works reasonably well and is essential as a software architecture is created. However, a subtle change can provide significant benefit when component-level design and construction are considered.

In *test-driven development* (TDD), requirements for a software component serve as the basis for the creation of a series of test cases that exercise the interface and attempt to find errors in the data structures and functionality delivered by the component. TDD is not really a new technology but rather a trend that emphasizes the design of test cases *before* the creation of source code.[10]

The TDD process follows the simple procedural flow illustrated in Figure 29.3. Before the first small segment of code is created, a software engineer creates a test to exercise the code (to try to make the code fail). The code is then written to satisfy the test. If it passes, a new test is created for the next segment of code to be developed. The process continues until the component is fully coded and all tests execute without error. However, if any test succeeds in finding an error, the existing code is refactored (corrected) and all tests created to that point are executed again. This iterative flow continues until there are no tests left to be created, implying that the component meets all requirements defined for it.

FIGURE 29.3

Test-driven development process flow



10 Recall that Extreme Programming (Chapter 3) emphasizes this approach as part of its agile process model.

During TDD, code is developed in very small increments (one subfunction at a time), and no code is written until a test exists to exercise it. You should note that each iteration results in one or more new tests that are added to a regression test suite that is run with every change. This is done to ensure that the new code has not generated side effects that cause errors in the older code. If you have further interest in TDD, see [Bec04b], [Ste10], or [Whi12].

29.6 TOOLS-RELATED TRENDS

Hundreds of industry-grade software engineering tools are introduced each year. The majority are provided by tools vendors who claim that their tool will improve project management, or requirements analysis, or design modeling, or code generation, or testing, or change management, or any of the many software engineering activities, actions, and tasks discussed throughout this book. Other tools have been developed as open-source offerings. The majority of open-source tools focus on "programming" activities with a specific emphasis on the construction activity (particularly code generation). Still other tools grow out of research efforts at universities and government labs. Although they have appeal in very limited applications, the majority are not ready for broad industry application.

At the industry level, the most comprehensive tools packages form *software engineering environments* (SEEs)[11] that integrate a collection of individual tools around a central database (repository). When considered as a whole, a SEE integrates information across the software process and assists in the collaboration that is required for many large, complex software-based systems. But current environments are not easily extensible (it's difficult to integrate a COTS tool that is not part of the package) and tend to be general purpose (i.e., they are not application domain specific). There is also a substantial time lag between the introduction of new technology solutions (e.g., model-driven software development) and the availability of viable SEEs that support the new technology.

In the past, software tools followed two distinct paths—a *human-focused path* that responds to some of the "soft trends" discussed in Section 29.4, and a technology-centered path that addresses new technologies (Section 25.5) as they are introduced and adopted. Moving forward, software engineers are beginning to build tools that focus on the interaction of humans and technology. Machine-generated solutions do not always apply to every problem. Humans are still needed to make the decision to accept a machine recommendation or not.

The soft trends discussed in Section 29.4—the need to manage complexity, accommodate emergent requirements, establish process models that embrace change, coordinate global teams with a changing talent mix, among others—suggest a new era in which tools support for stakeholder collaboration will become as important as tools support for technology.

Agility in software engineering (Chapter 3) is achieved when stakeholders work as a team. Therefore, the trend toward collaborative SEEs will provide benefits even

11 The term *integrated development environment* (IDE) is also used.

when software is developed locally. But what of the technology tools that complement the system and components that empower better collaboration?

One of the dominant trends in technology tools is the creation of a tool set that supports model-driven development (Section 29.5.5) with an emphasis on architecture-driven design. Oren Novotny [Nov04] suggests that the model rather than the source code becomes the central software engineering focus:

> Platform independent models are created in UML and then undergo various levels of transformation to eventually wind up as source code for a specific platform. It makes sense then, that the model, not the file, should become the new unit of output. A model has many different views at different levels of abstraction. At the highest level, platform independent components can be specified in analysis; at the lowest level there is a platform specific implementation that reduces to a set of classes in code.

Novotny argues that a new generation of tools will work in conjunction with a repository to create models at all necessary levels of abstraction, establish relationships between the various models, translate models at one level of abstraction to another level (e.g., translate a design model into source code), manage changes and versions, and coordinate quality control and assurance actions against the software models. Marouane Kessentini has deployed industry-grade tools at companies like eBay and SEMA that are designed to reduce technical debt problems by automatically detecting software defects [Man17] and recommending refactoring solutions to resolve them [Ali18]. This work is showing great promise.

In addition to complete software engineering environments, point-solution tools that address everything from requirements gathering to design/code refactoring to testing will continue to evolve and become more functionally capable. In some instances, modeling and testing tools targeted at a specific application domain will provide enhanced benefit when compared to their generic equivalents. Mark Harman's group at Facebook has announced deployment of a tool that automatically designs test cases and tests developers fixes following software crashes [Als18] with the hope that production software may be able to repair itself someday.

29.7 SUMMARY

The trends that have an effect on software engineering technology often come from business, organizational, market, and cultural arenas. These "soft trends" can guide the direction of research and the technology that is derived as a consequence of research. It is likely that artificial intelligence and data science methods will continue to impact all aspects of software engineering.

As a new technology is introduced, it moves through a life cycle that does not always lead to widespread adoption, even though original expectations are high. The degree to which any software engineering technology gains widespread adoption is tied to its ability to address the problems posed by both soft and hard trends. Digital personal assistants and social media seem to be influencing the activities of individuals in many aspects of everyday life. With their rise have come concerns about the importance of both security and privacy in the development of software products.

Soft trends—the growing need for connectivity and collaboration, global projects, knowledge transfer, the impact of emerging economies, and the influence of human

culture itself—lead to a set of challenges that spans managing complexity and emergent requirements, to juggling an ever-changing talent mix among geographically dispersed software teams. Global engineering is likely here to stay.

Hard trends—the ever-accelerating pace of technology change—flow out of soft trends and affect the structure of the software and scope of the process and the manner in which a process framework is characterized. Collaborative development, new forms of requirements engineering, model-based and test-driven development, and postmodern design will change the methods landscape. Tools environments will respond to a growing need for communication and collaboration and at the same time integrate domain-specific point solutions that may change the nature of current software engineering tasks. Machine learning is likely to be one approach to automating many important software engineering tasks.

PROBLEMS AND POINTS TO PONDER

29.1. Get a copy of the best-selling book *The Tipping Point* by Malcolm Gladwell (available via Google Book Search), and discuss how his theories apply to the adoption of new software engineering technologies.

29.2. Why does open-world software present a challenge to conventional software engineering approaches?

29.3. Review the Gartner Group's *hype cycle for emerging technologies.* Select a well-known technology product, and present a brief history that illustrates how it traveled along the curve. Select another well-known technology product that did not follow the path suggested by the hype curve.

29.4. What is a "soft trend"?

29.5. You're faced with an extremely complex problem that will require a lengthy solution. How would you go about addressing the complexity and crafting a solution?

29.6. What are "emergent requirements," and why do they present a challenge to software engineers?

29.7. Select an open-source development effort (other than Linux), and present a brief history of its evolution and relative success.

29.8. Describe how you think the software process will change over the next decade.

29.9. You're based in Los Angeles and are working on a global software engineering team. You and colleagues in London, Mumbai, Hong Kong, and Sydney must edit a 245-page requirements specification for a large system. The first editing pass must be completed in three days. Describe the ideal online tool set that would enable you to collaborate effectively.

29.10. Describe model-driven software development in your own words. Do the same for test-driven development.

Design element: Quick Look icon magnifying glass: © Roger Pressman

CHAPTER

30 CONCLUDING COMMENTS

In the 29 chapters that have preceded this one, we've explored a process for software engineering that encompasses management procedures and technical methods, basic concepts and principles, specialized techniques, people-oriented activities and tasks that are amenable to automation, paper-and-pencil notation, and software tools. We have argued that measurement, discipline, and an overriding focus on agility and quality will result in software that meets the customer's needs, software that is reliable, software that is supportable, software that is better. Yet, we have never promised that software engineering is a panacea.

KEY CONCEPTS

artificial intelligence 606
change 603
communication 604
ethics 607
future 606
genetic algorithms 606
information spectrum 605
knowledge 605
knowledge discovery 605
machine learning 606
people 603
responsibility 607
software revisited 603



QUICK LOOK

What is it? As we come to the end of a relatively long journey through software engineering, it's time to put things into perspective and make a few concluding comments.

Who does it? Authors like us. When you come to the end of a long and challenging book, it's nice to wrap things up in a meaningful way.

Why is it important? It's always worthwhile to remember where we've been and to consider where we're going.

What are the steps? We'll consider where we've been and address some of the core issues and some directions for the future.

What is the work product? A discussion that will help you understand the big picture.

How do I ensure that I've done it right? That's difficult to accomplish in real time. It's only after a number of years that any of us can tell whether the software engineering concepts, principles, methods, and techniques discussed in this book have helped you to become a better software engineer.

Software and systems technologies remain a challenge for every software professional and every company that builds computer-based systems. Although he wrote these words with a twentieth-century outlook, Max Hopper [Hop90] accurately describes the current state of affairs:

> Because changes in information technology are becoming so rapid and unforgiving, and the consequences of falling behind are so irreversible, companies will either master the technology or die . . . Think of it as a technology treadmill. Companies will have to run harder and harder just to stay in place.

Changes in software engineering technology are indeed "rapid and unforgiving," but at the same time real progress is often quite slow. By the time a decision is made to adopt a new process, method, or tool; conduct the training necessary to understand its application; and introduce the technology into the software development culture, something newer (and even better) has come along, and the process begins anew.

One thing we've learned over our years in this field is that software engineering practitioners are "fashion conscious." The road ahead will be littered with the carcasses of exciting new technologies (the latest fashion) that never really made it (despite the hype). It will be shaped by more modest technologies that somehow modify the direction and width of the thoroughfare. We discussed a few of those in Chapter 29.

In this concluding chapter, we'll take a broader view and consider where we've been and where we're going from a more philosophical perspective.

30.1 The Importance of Software—Revisited

The importance of computer software can be stated in many ways. In Chapter 1, software was characterized as a differentiator. The function delivered by software differentiates products, systems, and services and provides competitive advantage in the marketplace. But software is more than a differentiator. When taken as a whole, software engineering work products generate the most important commodity that any individual, business, or government can acquire—information.

In Chapter 29, we briefly discussed open-world computing—a technology that is fundamentally changing our perception of computers, the things that we do with them (and they do for us), and our perception of information as a guide, a commodity, and a necessity. We also noted that software required to support open-world computing will present dramatic new challenges for software engineers. But far more important, the growing pervasiveness of computer software will present even more dramatic challenges for society as a whole. Whenever a technology has a broad impact—an impact that can save lives or endanger them, build businesses or destroy them, inform government leaders or mislead them—it must be handled with care.

30.2 People and the Way They Build Systems

The software required for high-technology systems becomes more complex with each passing year, and the size of resultant programs increases proportionally. The rapid growth in the size of the "average" program would present us with few problems if it wasn't for one simple fact: As program size increases, the number of people who must work on the program must also increase.

Experience indicates that as the number of people on a software project team increases, the overall productivity of the group may suffer. One way around this problem is to create a number of software engineering teams, thereby compartmentalizing people into individual working groups. However, as the number of software engineering teams grows, communication between them becomes as difficult and time consuming. Worse, communication (between individuals or teams) tends to be inefficient—that is, too much time is spent transferring too little information content, and all too often, important information "falls into the cracks."

If the software engineering community is to deal effectively with the communication dilemma, the road ahead for software engineers must include radical changes in the way individuals and teams communicate with one another. In Chapter 27, we discussed the use of social media in supporting software release management that may provide dramatic improvements in the ways developers and customers communicate.

Finally, communication is the transfer of knowledge, and the acquisition (and transfer) of knowledge is changing in profound ways. As search engines become increasingly sophisticated, social networking and crowd-sourcing morph into development tools, and mobile applications provide better synergy, the speed and quality of knowledge transfer will grow exponentially.

If past history is any indication, it is fair to say that people themselves will not change. However, the ways in which they communicate, the environment in which they work, the manner in which they acquire knowledge, the methods and tools that they use, the discipline that they apply, and therefore, the overall culture for software development will change in significant and even profound ways.

SAFEHOME



Conclusion?

The scene: Doug Miller's office.

The players: Doug Miller, manager of the *SafeHome* software engineering group, and Vinod Raman, a member of the product software engineering team.

The conversation:

Doug: I'm really pleased that we got it done without too much drama.

Vinod (sighing and leaning back in his chair): Yeah, but the project grew, didn't it?

Doug: And you're surprised? When we started *SafeHome*, marketing thought a desktop app would do the trick, and then . . .

Vinod (smiling): And then, context aware mobility took over and we dabbled with VR.

Doug: But we all learned a lot.

Vinod: We did. The tech stuff was interesting, but the software engineering stuff is probably what allowed us to get it done close to schedule.

Doug: Yeah, that and hard work by all of you guys. What are you seeing from customer support? How's quality in the field?

Vinod: There are a few issues, but nothing really serious. We're on it. In fact, I gotta meet with Jamie on one of them in 5 minutes.

Doug: Before you go . . .

Vinod (on his way out the door): I know, more work, right?

Doug: Engineering has developed a new sensor . . . very high tech . . . we'll need to integrate it in *SafeHome II.*

Vinod: *SafeHome II?*

Doug: Yeah, *SafeHome II.* We'll begin planning next week.

30.3 KNOWLEDGE DISCOVERY

Over the history of computing, a subtle transition has occurred in the terminology that is used to describe software development work performed for the business community. Fifty years ago, the term *data processing* was the operative phrase for describing the use of computers in a business context. Today, data processing has given way to another phrase—*information technology*—that implies the same thing but presents a subtle shift in focus. The emphasis is not merely to process large quantities of data but rather to extract meaningful information from this data. Obviously, this was always the intent, but the shift in terminology reflects a far more important shift in management philosophy.

When software applications are discussed today, the words *data, information,* and *content* occur repeatedly. We encounter the word *knowledge* in many artificial intelligence applications. Virtually no one discusses *wisdom* in the context of software applications.

Data is raw information—collections of facts that must be processed to be meaningful. Information is derived by associating facts within a given context. Knowledge relates information obtained in one context with other information obtained in a different context. Finally, wisdom occurs when generalized principles are derived from disparate bits of knowledge.

To date, the vast majority of all software has been built to process data or information. Software engineers are now equally concerned with systems that process knowledge.[1] Information collected on a variety of related and unrelated topics is connected to form a body of facts that we call *knowledge*. The key is our ability to associate information from a variety of different sources that may not have any obvious connection and combine it in a way that provides us with some distinct benefit.[2]

To illustrate the progression from data to knowledge, consider census data indicating that the birthrate in 1996 in the United States was 4.9 million. This number represents a data value. Relating this piece of data with birthrates for the preceding 40 years, we can derive a useful piece of information—aging baby boomers of the 1950s and early 1960s made a last-gasp effort to have children prior to the end of their child-bearing years. In addition, gen-Xers began their childbearing years. The census data can then be connected to other seemingly unrelated pieces of information, for example, the current number of elementary school teachers who will retire during the next decade, the number of college students graduating with degrees in primary and secondary education, and the pressure on politicians to hold down taxes and therefore limit pay increases for teachers.

All these pieces of information can be combined to formulate a representation of knowledge—there will be significant pressure on the education system in the United States in the early twenty-first century, and this pressure will continue for a number of decades. Using this knowledge, a business opportunity may emerge. There may be significant opportunity to develop new modes of learning that are more effective and less costly than current approaches.

1 The rapid growth of data mining and data warehousing technologies reflect this growing trend.

2 The semantic Web (Web 2.0) allows the creation of "mashups" that may provide a facile mechanism for achieving this.

The road ahead for software leads to systems that both discover and process knowledge. We have been processing data using computers for more than 70 years and extracting information for more than three decades. One of the most significant challenges facing the software engineering community is to build systems that take the next step along the spectrum—systems that extract knowledge from data and information in a way that is practical and beneficial. *Knowledge discovery* is an interdisciplinary area focusing upon methodologies for extracting useful relationships from data.

Mark Harman (currently Facebook's manager for software engineering research) was one of the first people to recognize the value of using data mining and machine learning to solve difficult software engineering problems [Har12b]. The availability of several public software engineering data repositories (e.g., *Bugzilla, GitHub, SourceForge*) make it possible to use search-based software engineering techniques to discover insights into software development artifacts and processes [Dye15] [Gup15]. This is not an easy task and suggests that it may be desirable to include data scientists as members of large software engineering projects [Kim16b]. The knowledge discovered while mining public repositories may suggest practice improvements to software engineers working on smaller proprietary projects or may yield techniques that can be applied on their own software engineering data repositories.

Machine learning has been used in many areas of software engineering ranging from behavior extraction, design pattern recognition, program generation, test-case generation, and defect detection [Mei18]. This work cannot be done without access to large sets of software engineering data and access to domain experts to help shape the concepts learned by machines. *Genetic algorithms*[3] make use of automated search and can be used to grow an improved software product or process by heuristically combining elements of existing software products and processes. Genetics have been used to improve software performance for a diverse set of properties such as execution time, memory consumption, as well as defect repair and existing system functionality extensions [Pet18].

Intelligent software engineering is emerging as an academic area of study that combines artificial intelligence (AI) and software engineering. Intelligent software engineering techniques explore software engineering solutions to improve the productivity of developing AI software and the dependability of AI software. It also seeks to address some of the problems encountered when attempting to automate software engineering processes [Xie18]. As AI techniques become more powerful and easier to use, they are increasingly deployed as key components of modern software systems. Although this allows the creation of products better able to adapt to user needs, it also creates additional problems for software engineers and exposes companies to new risks [Fel18].

30.4 The Long View

In Section 30.3, we suggested that the road ahead leads to systems that "discover knowledge." But the future of computing in general and software-based systems in particular may lead to events that are considerably more profound.

3 Genetic algorithms are used to search a population of computer-generated potential solutions to a problem with the intent of finding the best solution while maintaining the diversity of the set of candidate solutions.

In a fascinating book that is must reading for every person involved in computing technologies, Ray Kurzweil [Kur05] suggests that we have reached a time when "the pace of technological change will be so rapid, its impact so deep, that human life will be irreversibly transformed." Kurzweil[4] makes a compelling argument that we are currently at the "knee" of an exponential growth curve that will lead to enormous increases in computing capacity over the next few decades. When coupled with equivalent advances in nanotechnology, genetics, and robotics, we may approach a time in the middle part of this century when the distinction between humans (as we know them today) and machines begins to blur—a time when human evolution accelerates in ways that are both frightening (to some) and spectacular (to others).

Kurzweil argues that sometime in the coming decade computing capacity and the requisite software will be sufficient to model every aspect of the human brain, including all the physical connections, analog processes, and chemical overlays [Kur13]. When this occurs, human beings will take the first step toward achieving "strong AI (artificial intelligence)," and as a consequence, machines that truly do think (using today's conventional use of the word). But there will be a fundamental difference. Human brain processes are exceedingly complex and only loosely connected to external informal sources. They are also computationally slow, even in comparison to today's computing technology. When full human brain emulation occurs, "thought" will occur at speeds thousands of times more rapid than its human counterpart with intimate connections to a sea of information (think of the present-day Web as a primitive example). The result is . . . well . . . so fantastical that it's best left to Kurzweil to describe.

It's important to note that not everyone believes that the future Kurzweil describes is a good thing. In a now-famous essay titled "The Future Doesn't Need Us," Bill Joy [Joy00], one of the founders of Sun Microsystems, argues that "robotics, genetic engineering, and nanotech are threatening to make humans an endangered species." His arguments, along with commentary by luminaries such Bill Gates, Elon Musk, and the late Steven Hawking, predicting a potential technology dystopia represent a counterpoint to Kurzweil's predicted utopian future. Both should be seriously considered as software engineers take one of the lead roles in defining the long view for the human race.

30.5 THE SOFTWARE ENGINEER'S RESPONSIBILITY

Software engineering has evolved into a respected, worldwide profession. As professionals, software engineers should abide by a code of ethics that guides the work they do and the products they produce. An ACM/IEEE-CS Joint Task Force has produced

4 It's important to note that Kurzweil is not a run-of-the mill science fiction writer or a futurist without portfolio. He is a serious technologist who (from Wikipedia) "has been a pioneer in the fields of optical character recognition (OCR), text-to-speech synthesis, speech recognition technology, and electronic keyboard instruments."

a *Software Engineering Code of Ethics and Professional Practices* (Version 5.1). The code [ACM12] states:

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
>
> 1. PUBLIC—Software engineers shall act consistently with the public interest.
> 2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
> 3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
> 4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.
> 5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
> 6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
> 7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.
> 8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Although each of these eight principles is equally important, an overriding theme appears: A software engineer should work in the public interest. On a personal level, a software engineer should abide by the following rules:

- Never steal data for personal gain.
- Never distribute or sell proprietary information obtained as part of your work on a software project.
- Never maliciously destroy or modify another person's programs, files, or data.
- Never violate the privacy of an individual, a group, or an organization.
- Never hack into a system for sport or profit.
- Never create or promulgate a computer virus or worm.
- Never use computing technology to facilitate discrimination or harassment.

Over the past decade, certain members of the software industry have lobbied for protective legislation that (1) allows companies to release software without disclosing known defects, (2) exempts developers from liability for any damages resulting from these known defects, (3) constrains others from disclosing defects without permission from the original developer, (4) allows the incorporation of "self-help" software within a product that can disable (via remote command) the operation of the product, and (5) exempts developers of software with "self-help" from damages should the software be disabled by a third party.

Like all legislation, debate often centers on issues that are political, not technological. However, many people (including us) feel that protective legislation, if

improperly drafted, conflicts with the software engineering code of ethics by indirectly exempting software engineers from their responsibility to produce high-quality software. Given the sizeable social media data breaches that occurred in 2018, there will be more demands for enhanced security protections by companies storing large amounts of confidential customer data.

The growing decision-making capabilities of autonomous systems and the influence of AI in our daily lives make us want to consider the values embedded in these systems [Vak18]. The software engineering profession needs to examine ways to measure bias in search algorithms and social networks [Pit18]. The revised ACM Code of Ethics and Professional Conduct [ACM18] adopts several new principles that address issues in specific computing technologies such as AI, machine learning, and autonomous machines making ethically significant decisions [Got18]. It is likely that the ACM and IEEE will consider revising their Software Engineering Code of Ethics to reflect these new areas as well.

30.6 A Final Comment from RSP

It has been almost four decades since work on the first edition of this book began. I [RSP] can still recall sitting at my desk as a young professor, writing the manuscript for a book on a subject that few people cared about and even fewer really understood. I remember the rejection letters from publishers, who argued (politely, but firmly) that there would never be a market for a book on "software engineering." Luckily, McGraw-Hill decided to try it,[5] and the rest, as they say, is history.

Since the first edition, this book has changed dramatically—in scope, in size, in style, and in content. Like software engineering, it has grown and matured over the years.

An engineering approach to the development of computer software is now conventional wisdom. Debate continues on the "right paradigm," the importance of agility, the degree of automation, and the most effective methods. But the underlying principles of software engineering are now accepted throughout the industry. Why, then, have we seen their broad adoption only recently?

The answer, I think, lies in the difficulty of technology transition and the cultural change that accompanies it. Even though most of us appreciate the need for an engineering discipline for software, we struggle against the inertia of past practice and face new application domains (and the developers who work in them) that appear ready to repeat the mistakes of the past. To ease the transition, we need many things—an agile, adaptable, and sensible software process; more effective methods; more powerful tools; better acceptance by practitioners and support from managers; and no small dose of education.

You may not agree with every approach described in this book. Some of the techniques and opinions are controversial; others must be tuned to work well in different

5 Actually, credit should go to Peter Freeman and Eric Munson, who convinced McGraw-Hill that it was worth a shot, and 3 million copies later, it's fair to say they made a good decision.

software development environments. It is my sincere hope, however, that *Software Engineering: A Practitioner's Approach* has delineated the problems we face, demonstrated the strength of software engineering concepts, and provided a framework of methods and tools.

As we move further into the twenty-first century, software continues to be the most important product and the most important industry on the world stage. Its impact and importance have come a long, long way. And yet, a new generation of software developers must meet many of the same challenges faced by earlier generations. Let us hope that the people who meet the challenge—software engineers—will have the wisdom to develop systems that improve the human condition.

Design element: Quick Look icon magnifying glass: © Roger Pressman

APPENDIX

1 AN INTRODUCTION TO UML[1]

KEY CONCEPTS

activity diagram 622
class diagram 612
communication diagram 621
dependency 614
deployment diagram 615
generalization 613
interaction frames 620
multiplicity 614
sequence diagram 618
state diagram 625
stereotype 613
swimlanes 624
use-case diagram 616

The *Unified Modeling Language* (UML) is "a standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system" [Boo05]. In other words, just as building architects create blueprints to be used by a construction company, software architects create UML diagrams to help software developers build the software. If you understand the vocabulary of UML (the diagrams' pictorial elements and their meanings), you can much more easily understand and specify a system and explain the design of that system to others.

Grady Booch, Jim Rumbaugh, and Ivar Jacobson developed UML in the mid-1990s with much feedback from the software development community. UML merged a number of competing modeling notations that were in use by the software industry at the time. In 1997, UML 1.0 was submitted to the Object Management Group, a nonprofit consortium involved in maintaining specifications for use by the computer industry. UML 1.0 was revised to UML 1.1 and adopted later that year. The current standard is UML 2.5.1[2] and is now an ISO standard. Because this standard is new, many older references, such as [Gam95] do not use UML notation.

UML 2.5.1 provides 13 different diagrams for use in software modeling. In this appendix, we will discuss only *class, deployment, use-case, sequence, communication, activity,* and *state diagrams.* These diagrams are used in this edition of *Software Engineering: A Practitioner's Approach.*

You should note that there are many optional features in UML diagrams. The UML language provides these (sometimes arcane) options so that you can express all the important aspects of a system. At the same time, you have the flexibility to suppress those parts of the diagram that are not relevant to the aspect being modeled to avoid cluttering the diagram with irrelevant details. Therefore, the omission of a feature does not mean that the feature is absent; it may mean that the feature was suppressed. In

1 This appendix has been contributed by Dale Skrien and has been adapted from his book, *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). All content is used with permission.

2 See https://www.omg.org/spec/UML/2.5.1/. This document contains the current specification for the UML 2.5.1 modeling language.

this appendix, we will not present exhaustive coverage of all the features of the UML diagrams. Instead, we focus on the standard options, especially those options that have been used in this book.

Class Diagrams

To model classes, including their attributes, operations, and their relationships and associations with other classes, UML provides a *class diagram.* A class diagram provides a static or structural view of the system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram.

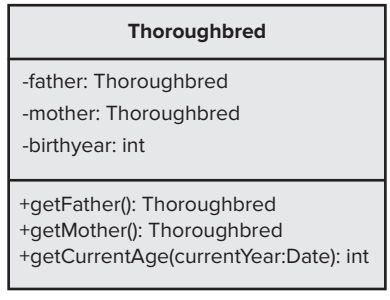
The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. Attributes can be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed. For example, an object might always know the current time and be able to return it to you whenever you ask, in which case it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An *operation* refers to what objects of the class can do. It is usually implemented as a *method* of the class.

Figure A1.1 presents a simple example of a **Thoroughbred** class that models thoroughbred horses. It has three attributes displayed—`mother`, `father`, and `birth year`. The diagram also shows three operations: *getCurrentAge(), getFather(),* and *getMother()*. There may be other suppressed attributes and operations not shown in the diagram.

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding −, #, ~, or +, indicating, respectively, *private, protected, package,* or *public* visibility. In Figure A1.1, all attributes have private visibility, as indicated by the leading minus sign (−). You can also specify that

FIGURE A1.1

A class diagram for a Thoroughbred class



an attribute is a static or class attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type.

An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the **Horse** class in Figure A1.2, for an example. An interface is indicated by adding the phrase "«interface»" (called a *stereotype*) above the name. See the **OwnedObject** interface in Figure A1.2. An interface can also be represented graphically by a hollow circle.

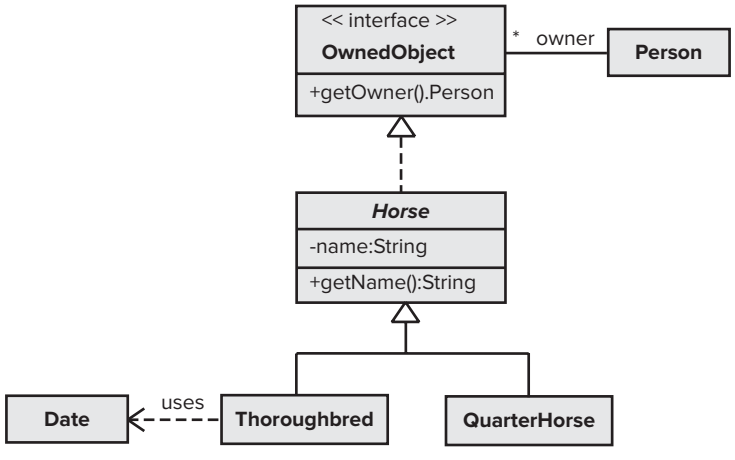
It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This section is particularly useful when transitioning from CRC cards (Chapter 8) to class diagrams in that the responsibilities listed on the CRC cards can be added to this fourth section in the class box in the UML diagram before creating the attributes and operations that carry out these responsibilities. This fourth section is not shown in any of the figures in this appendix.

Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a *generalization*. For example, in Figure A1.2, the **Thoroughbred** and **QuarterHorse** classes are shown to be subclasses of the **Horse** abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a *realization*. For example, in Figure A1.2, the **Horse** class implements or realizes the **OwnedObject** interface.

An *association* between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A1.2, there is an association between **OwnedObject** and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity

FIGURE A1.2

A class diagram regarding horses



are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class.

An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the first class is aware of the second class, but the second class is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A1.2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A1.2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type.

A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them. However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A1.2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses."

The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a **Person** class and a **SocialSecurityNumber** class in a class diagram. A multiplicity specified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*" means zero or more. An * was used as the multiplicity on the **OwnedObject** end of the association with class **Person** in Figure A1.2 because a **Person** can own zero or more objects.

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An *aggregation* is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which

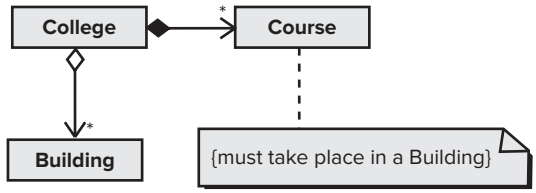


FIGURE A1.3
The relationship between Colleges, Courses, and Buildings

the arrow points is considered a "part" of the class at the diamond end of the association. A *composition* is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner. See Figure A1.3 for examples of aggregation and composition.

A **College** has an aggregation of **Building** objects, which represent the buildings making up the campus. The college also has a collection of courses. If the college were to fold, the buildings would still exist (assuming the college wasn't physically destroyed) and could be used for other things, but a **Course** object has no use outside of the college at which it is being offered. If the college were to cease to exist as a business entity, the **Course** object would no longer be useful and so it would also cease to exist.

Another common element of a class diagram is a *note*, which is represented by a box with a dog-eared corner and is connected to other icons by a dashed line. It can have arbitrary content (text and graphics) and is similar in a programming language comment. It might contain information about the role of a class or constraints that all objects of that class must satisfy. If the contents are a constraint, braces surround the contents. Note the constraint attached to the **Course** class in Figure A1.3.

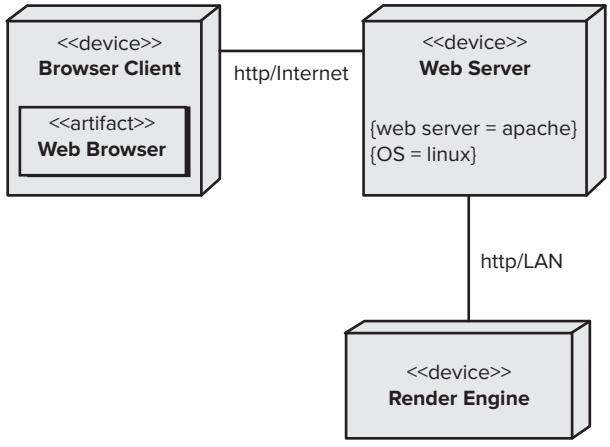
DEPLOYMENT DIAGRAMS

A UML *deployment diagram* focuses on the structure of the software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments. Suppose, for example, you are developing a Web-based graphics-rendering package. Users of your package will use their Web browser to go to your website and enter rendering information. Your website would render a graphical image according to the user's specification and send it back to the user. Because graphics rendering can be computationally expensive, you decide to move the rendering itself off the Web server and onto a separate platform. Therefore, there will be three hardware devices involved in your system: the Web client (the users' computer running a browser), the computer hosting the Web server, and the computer hosting the rendering engine.

Figure A1.4 shows the deployment diagram for such a package. In such a diagram, hardware components are drawn in boxes labeled with "«device»". Communication paths between hardware components are drawn with lines with optional labels. In Figure A1.4, the paths are labeled with the communication protocol and the type of network used to connect the devices.

FIGURE A1.4

A deployment diagram



Each node in a deployment diagram can also be annotated with details about the device. For example, in Figure A1.4, the browser client is depicted to show that it contains an artifact consisting of the Web browser software. An artifact is typically a file containing software running on a device. You can also specify tagged values, as is shown in Figure A1.4 in the Web server node. These values define the vendor of the Web server and the operating system used by the server.

Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label "«execution environment»". These nodes represent systems, such as operating systems, that can host other software programs.

USE-CASE DIAGRAMS

Use cases (Chapters 7 and 8) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we'll create some for a software application for managing an online digital music store. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD).

Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

In this diagram, the stick figure represents an *actor* (Chapter 8) that is associated with one category of user (or other interaction element). Complex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the actors are outside the system.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod or smartphone. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity and then let the other

FIGURE A1.5

A use-case diagram for the music system

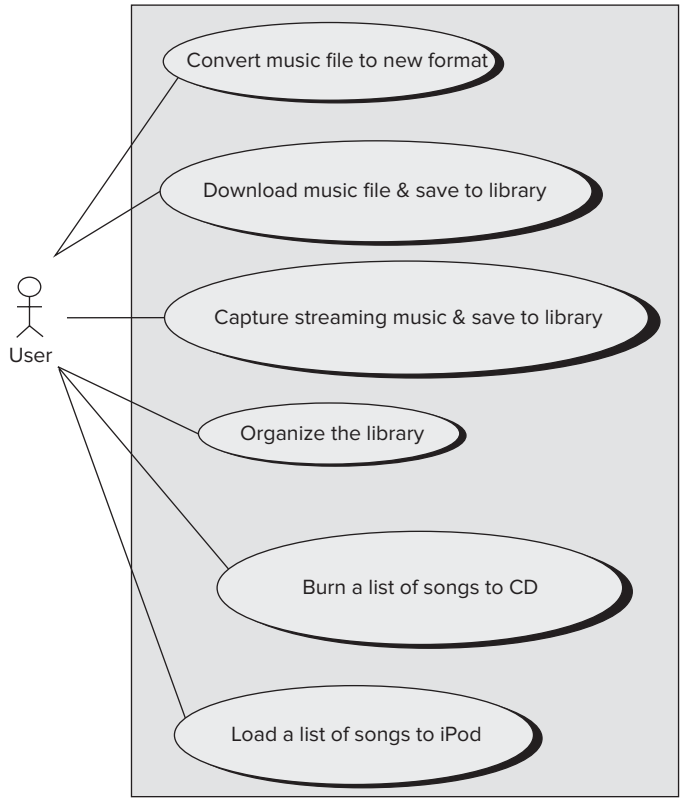
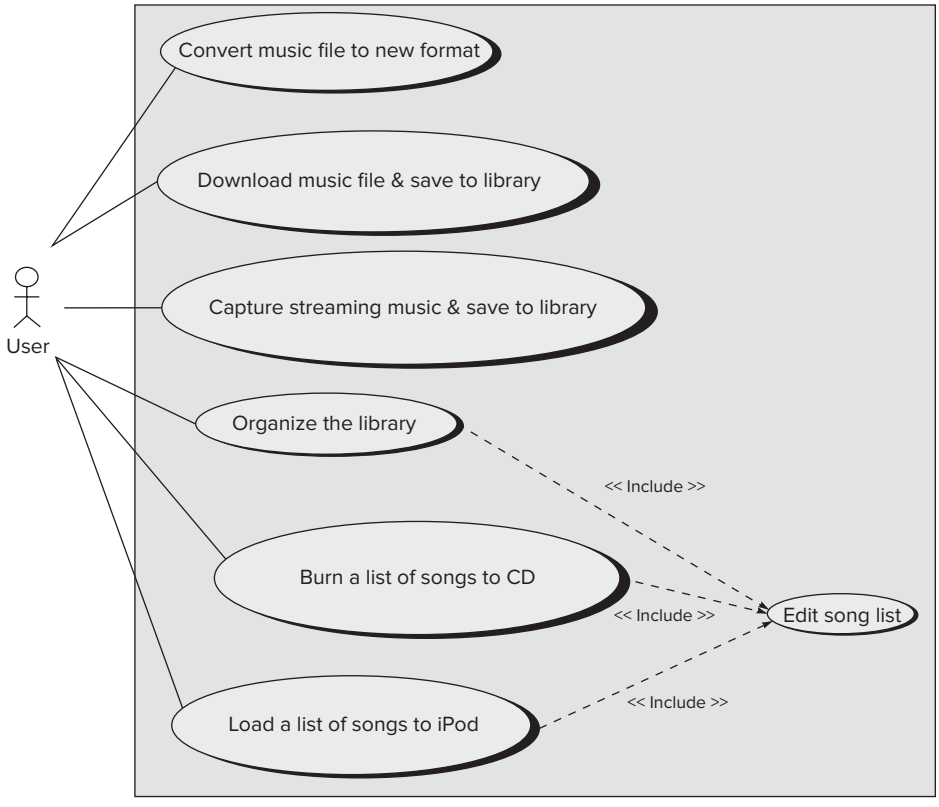


FIGURE A1.6

A use-case diagram with included use cases



use cases include this new use case as one of their steps. Such inclusion is indicated in use-case diagrams, as in Figure A1.6, by means of a dashed arrow labeled «include» connecting a use case with an included use case.

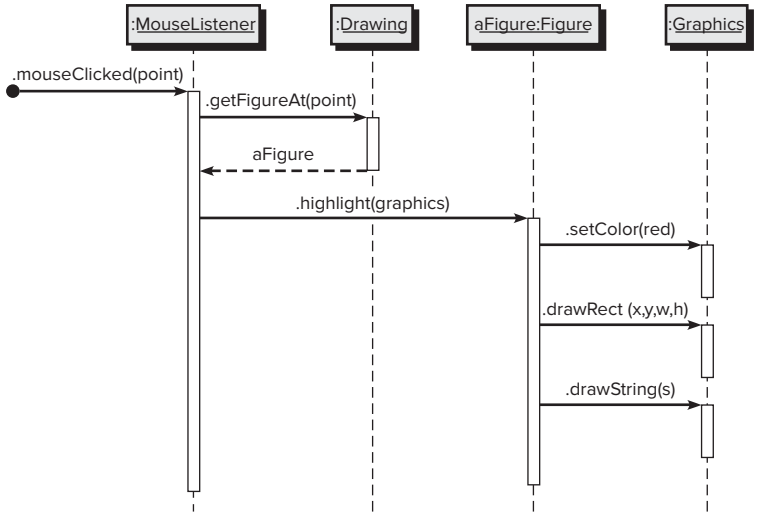
A use-case diagram, because it displays all use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In our digital music organizer, we would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most valuable contribution of use cases to the software development process is the textual description of each use case, not the overall use-case diagram [Fow04]. It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

SEQUENCE DIAGRAMS

In contrast to class diagrams and deployment diagrams, which show the static structure of a software component, a *sequence diagram* is used to show the dynamic communications between objects during execution of a task. It shows the temporal order in which messages are sent between the objects to accomplish that task. One might use a sequence diagram to show the interactions in one use case or in one scenario of the software system.

FIGURE A1.7

A sample sequence diagram



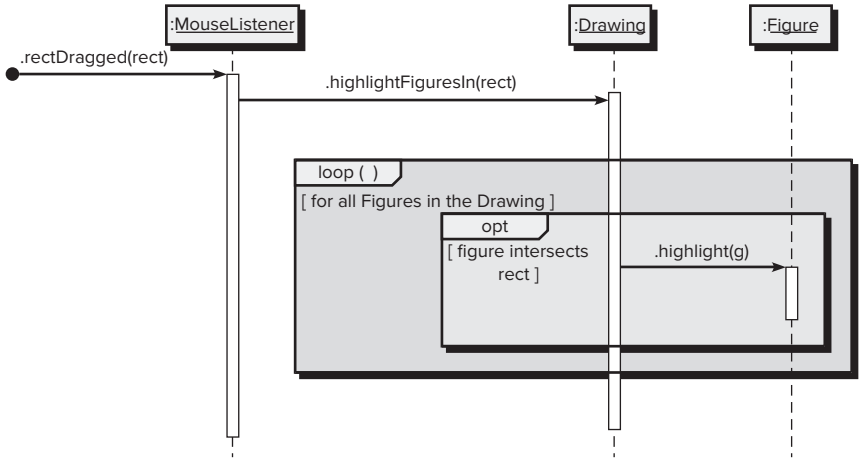
In Figure A1.7, you see a sequence diagram for a drawing program. The diagram shows the steps involved in highlighting a figure in the drawing when it has been clicked. Each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes. If the box represents an object (as is the case in all our examples), then inside the box you can optionally state the type of the object preceded by the colon. You can also precede the colon and type by a name for the object, as shown in the third box in Figure A1.7. Below each box there is a dashed line called the *lifeline* of the object. The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.

A sequence diagram shows method calls using horizontal arrows from the *caller* to the *callee*, labeled with the method name and optionally including its parameters, their types, and the return type. For example, in Figure A1.7, the **MouseListener** calls the **Drawing**'s *getFigureAt()* method. When an object is executing a method (that is, when it has an activation frame on the stack), you can optionally display a white bar, called an *activation bar*, down the object's lifeline. In Figure A1.7, activation bars are drawn for all method calls. The diagram can also optionally show the return from a method call with a dashed arrow and an optional label. In Figure A1.7, the *getFigureAt()* method call's return is shown labeled with the name of the object that was returned. A common practice, as we have done in Figure A1.7, is to leave off the return arrow when a void method has been called, since it clutters up the diagram while providing little information of importance. A black circle with an arrow coming from it indicates a *found message* whose source is unknown or irrelevant.

You should now be able to understand the task that Figure A1.7 is displaying. An unknown source calls the *mouseClicked()* method of a **MouseListener**, passing in the point where the click occurred as the argument. The **MouseListener** in turn calls the *getFigureAt()* method of a **Drawing**, which returns a **Figure.** The **MouseListener** then calls the highlight method of **Figure**, passing in a **Graphics** object as an argument. In response, **Figure** calls three methods of the **Graphics** object to draw the figure in red.

FIGURE A1.8

A sequence diagram with two interaction frames



The diagram in Figure A1.7 is very straightforward and contains no conditionals or loops. If logical control structures are required, it is probably best to draw a separate sequence diagram for each case. That is, if the message flow can take two different paths depending on a condition, then draw two separate sequence diagrams, one for each possibility.

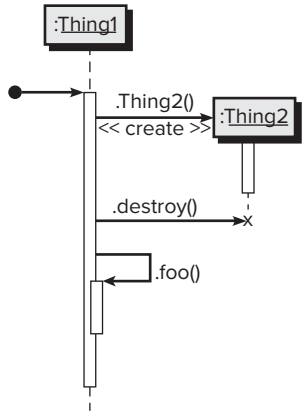
If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use *interaction frames*, which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent. Figure A1.8 illustrates this, showing the process involved in highlighting all figures inside a given rectangle. The **MouseListener** is sent the `rectDragged` message. The **MouseListener** then tells the drawing to highlight all figures in the rectangle by calling the method *highlightFiguresIn()*, passing the rectangle as the argument. The method loops through all **Figure** objects in the **Drawing** object and, if the **Figure** intersects the rectangle, the **Figure** is asked to highlight itself. The phrases in square brackets are called *guards*, which are Boolean conditions that must be true if the action inside the interaction frame is to continue.

There are many other special features that can be included in a sequence diagram. For example:

1. You can distinguish between synchronous and asynchronous messages. Synchronous messages are shown with solid arrowheads, while asynchronous messages are shown with stick arrowheads.
2. You can show an object sending itself a message with an arrow going out from the object, turning downward, and then pointing back to the same object.
3. You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label) to an object's box. In this case, the box will appear lower in the diagram than the boxes corresponding to objects already in existence when the action begins.

FIGURE A1.9

Creation, destruction, and loops in sequence diagrams



4. You can show object destruction by a big X at the end of the object's lifeline. Other objects can destroy an object, in which case an arrow points from the other object to the X. An X is also useful for indicating that an object is no longer usable and so is ready for garbage collection.

The last three features are all shown in the sequence diagram in Figure A1.9.

COMMUNICATION DIAGRAMS

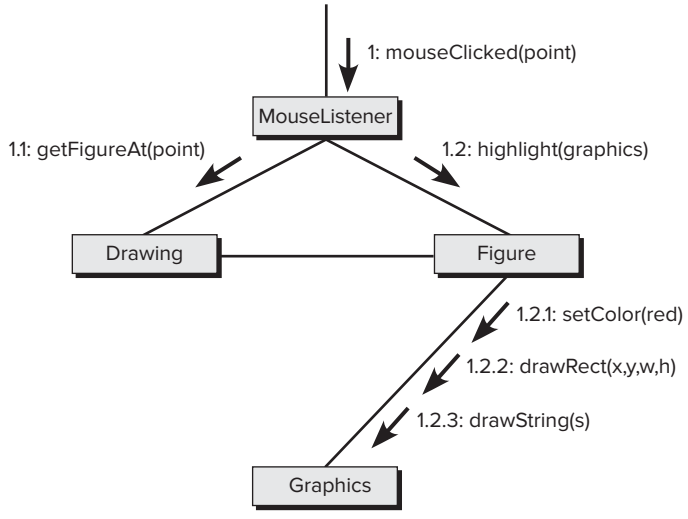
The UML *communication diagram* (known as a "collaboration diagram" in UML 1.X) provides another indication of the temporal order of the communications, but emphasizes the relationships among the objects and classes instead of the temporal order. A communication diagram is illustrated in Figure A1.10, which displays the same actions shown in the sequence diagram in Figure A1.7.

In a communication diagram the interacting objects are represented by rectangles. Associations between objects are represented by lines connecting the rectangles. There is typically an incoming arrow to one object in the diagram that starts the sequence of message passing. That arrow is labeled with a number and a message name. If the incoming message is labeled with the number 1 and if it causes the receiving object to invoke other messages on other objects, then those messages are represented by arrows from the sender to the receiver along an association line and are given numbers 1.1, 1.2, and so forth, in the order they are called. If those messages in turn invoke other messages, another decimal point and number are added to the number labeling these messages, to indicate further nesting of the message passing.

In Figure A1.10, you see that the **mouseClicked** message invokes the methods *getFigureAt()* and then *highlight()*. The *highlight()* message invokes three other messages: *setColor(), drawRect(),* and *drawString()*. The numbering in each label shows the nesting as well as the sequential nature of each message.

There are many optional features that can be added to the arrow labels. For example, you can precede the number with a letter. An incoming arrow could be labeled `A1: mouseClicked(point)`. indicating an execution thread, A. If other messages are

FIGURE A1.10
A UML communication diagram



executed in other threads, their label would be preceded by a different letter. For example, if the *mouseClicked()* method is executed in thread A but it creates a new thread B and invokes *highlight()* in that thread, then the arrow from **MouseListener** to **Figure** would be labeled `1.B2: highlight(graphics)`.

If you are interested in showing the relationships among the objects in addition to the messages being sent between them, the communication diagram is probably a better option than the sequence diagram. If you are more interested in the temporal order of the message passing, then a sequence diagram is probably better.

ACTIVITY DIAGRAMS

A UML *activity diagram* depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.

The main component of an activity diagram is an *action* node, represented by a rounded rectangle, which corresponds to a task performed by the software system. Arrows from one action node to another indicate the flow of control. That is, an arrow between two action nodes means that after the first action is complete the second action begins. A solid black dot forms the *initial node* that indicates the starting point of the activity. A black dot surrounded by a black circle is the *final node,* indicating the end of the activity.

A *fork* represents the separation of activities into two or more concurrent activities. It is drawn as a horizontal black bar with one arrow pointing to it and two or more arrows pointing out from it. Each outgoing arrow represents a flow of control that can be executed concurrently with the flows corresponding to the other outgoing arrows. These concurrent activities can be performed on a computer using different threads or even using different computers.

FIGURE A1.11

A UML activity diagram showing how to bake a cake

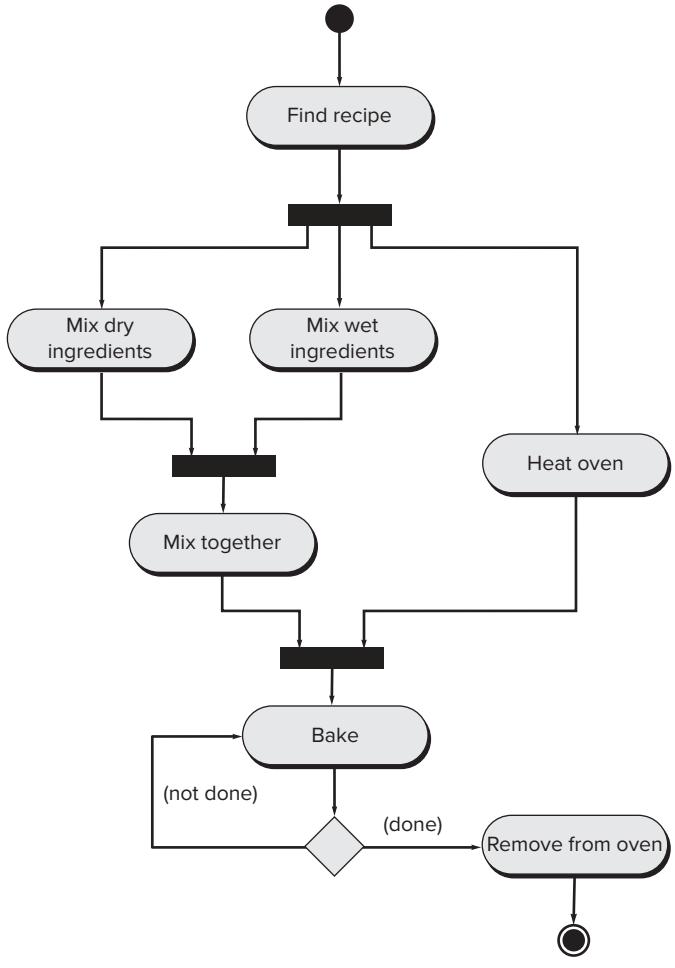


Figure A1.11 shows a sample activity diagram involving baking a cake. The first step is finding the recipe. Once the recipe has been found, the dry ingredients and wet ingredients can be measured and mixed and the oven can be preheated. The mixing of the dry ingredients can be done in parallel with the mixing of the wet ingredients and the preheating of the oven.

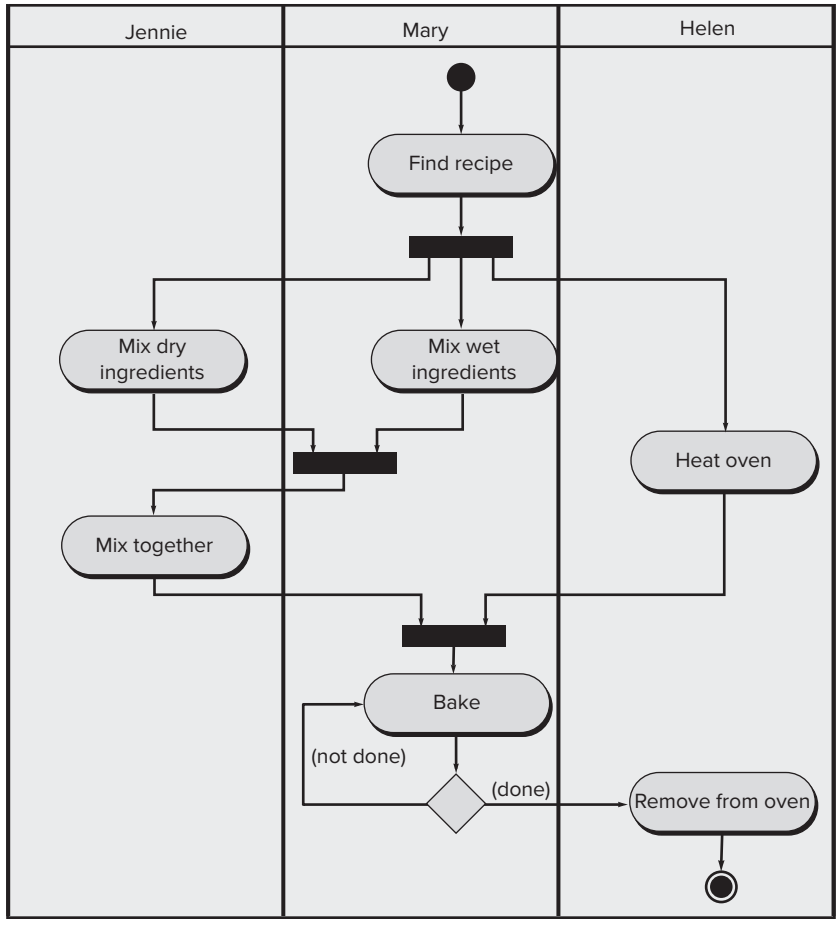
A *join* is a way of synchronizing concurrent flows of control. It is represented by a horizontal black bar with two or more incoming arrows and one outgoing arrow. The flow of control represented by the outgoing arrow cannot begin execution until all flows represented by incoming arrows have been completed. In Figure A1.11, we have a join before the action of mixing together the wet and dry ingredients. This join indicates that all dry ingredients must be mixed and all wet ingredients must be mixed before the two mixtures can be combined. The second join in the figure indicates that, before the baking of the cake can begin, all ingredients must be mixed together, and the oven must be at the right temperature.

A *decision* node corresponds to a branch in the flow of control based on a condition. Such a node is displayed as a white triangle with an incoming arrow and two or more outgoing arrows. Each outgoing arrow is labeled with a guard (a condition inside square brackets). The flow of control follows the outgoing arrow whose guard is true. It is advisable to make sure that the conditions cover all possibilities so that exactly one of them is true every time a decision node is reached. Figure A1.11 shows a decision node following the baking of the cake. If the cake is done, then it is removed from the oven. Otherwise, it is baked for a while longer.

One of the things the activity diagram in Figure A1.11 does not tell you is who or what does each of the actions. Often, the exact division of labor does not matter. But if you do want to indicate how the actions are divided among the participants, you can decorate the activity diagram with swimlanes, as shown in Figure A1.12. *Swimlanes*, as the name implies, are formed by dividing the diagram into strips or "lanes," each of which corresponds to one of the participants. All actions in one lane are done by the corresponding participant. In Figure A1.12, Jennie is responsible for

FIGURE A1.12

The cake-baking activity diagram with swimlanes added



mixing the dry ingredients and then mixing the dry and wet ingredients together, Helen is responsible for heating the oven and taking the cake out, and Mary is responsible for everything else.

STATE DIAGRAMS

The behavior of an object at a particular point in time often depends on the state of the object, that is, the values of its variables at that time. As a trivial example, consider an object with a Boolean instance variable. When asked to perform an operation, the object might do one thing if that variable is *true* and do something else if it is *false*.

A UML *state diagram* models an object's states, the actions that are performed depending on those states, and the transitions between the states of the object.

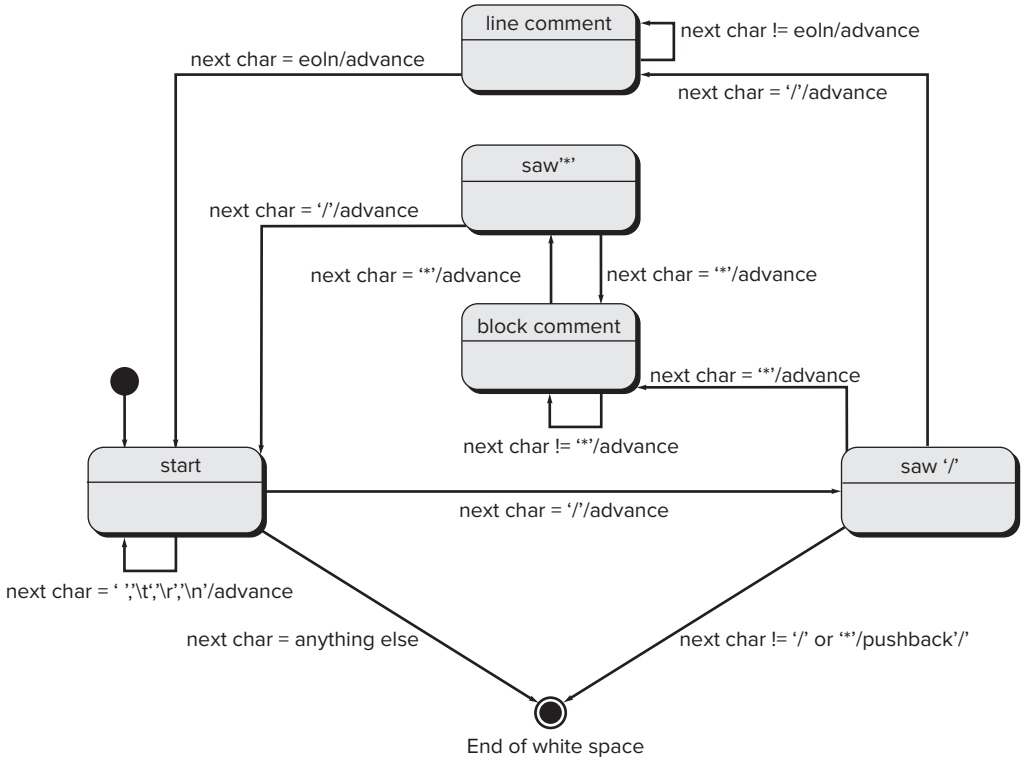
As an example, consider the state diagram for a part of a Java compiler. The input to the compiler is a text file, which can be thought of as a long string of characters. The compiler reads characters one at a time and from them determines the structure of the program. One small part of this process of reading the characters involves ignoring "white-space" characters (e.g., the *space, tab, newline,* and *return* characters) and characters inside a comment.

Suppose that the compiler delegates to a **WhiteSpaceAndCommentEliminator** the job of advancing over white-space characters and characters in comments. That is, this object's job is to read input characters until all white-space and comment characters have been read, at which point it returns control to the compiler to read and process non-white-space and noncomment characters. Think about how the **WhiteSpaceAndCommentEliminator** object reads in characters and determines whether the next character is white space or part of a comment. The object can check for white space by testing the next character against " ", "\t", "\n", and "\r". But how does it determine whether the next character is part of a comment? For example, when it sees a "/" for the first time, it doesn't yet know whether that character represents a division operator, part of the /= operator, or the beginning of a line or block comment. To make this determination, **WhiteSpaceAndCommentEliminator** needs to make a note of the fact that it saw a "/" and then move on to the next character. If the character following the "/" is another "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that it is now reading a comment and can advance to the end of the comment without processing or saving any characters. If the character following the first "/" is anything other than a "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that the "/" represents the division operator or part of the /= operator and so it stops advancing over characters.

In summary, as **WhiteSpaceAndCommentEliminator** reads in characters, it needs to keep track of several things, including whether the current character is white space, whether the previous character it read was a "/", whether it is currently reading characters in a comment, whether it has reached the end of comment, and so forth. These all correspond to different states of the **WhiteSpaceAndCommentEliminator** object. In each of these states, **WhiteSpaceAndCommentEliminator** behaves differently with regard to the next character read in.

To help you visualize all the states of this object and how it changes state, you can use a UML state diagram as shown in Figure A1.13. A state diagram displays states using rounded rectangles, each of which has a name in its upper half. There is also a

FIGURE A1.13 **A state diagram for advancing past white space and comments in Java**



black circle called the "initial pseudostate," which isn't really a state and instead just points to the initial state. In Figure A1.13, the **start** state is the initial state. Arrows from one state to another state indicate transitions or changes in the state of the object. Each transition is labeled with a trigger event, a slash (/), and an activity. All parts of the transition labels are optional in state diagrams. If the object is in one state and the trigger event for one of its transitions occurs, then that transition's activity is performed and the object takes on the new state indicated by the transition. For example, in Figure A1.13, if the **WhiteSpaceAndCommentEliminator** object is in the **start** state and the next character is "/", then **WhiteSpaceAndCommentEliminator** advances past that character and changes to the **saw '/'** state. If the character after the "/" is another "/", then the object advances to the **line comment** state and stays there until it reads an end-of-line character. If instead the next character after the "/""is a "*", then the object advances to the **block comment** state and stays there until it sees another "*" followed by a "/", which indicates the end of the block comment. Study the diagram to make sure you understand it. Note that, after advancing past white space or a comment, **WhiteSpaceAndCommentEliminator** goes back to the **start** state and starts over. That behavior is necessary since there might be several successive comments or white-space characters before any other characters in the Java source code.

An object may transition to a final state, indicated by a black circle with a white circle around it, which indicates there are no more transitions. In Figure A1.13, the

WhiteSpaceAndCommentEliminator object is finished when the next character is not white space or part of a comment. Note that all transitions except the two transitions leading to the final state have activities consisting of advancing to the next character. The two transitions to the final state do not advance over the next character because the next character is part of a word or symbol of interest to the compiler. Note that if the object is in the **saw '/'** state but the next character is not "/" or "*", then the "/" is a division operator or part of the /= operator and so we don't want to advance. In fact, we want to back up one character to make the "/" into the next character, so that the "/" can be used by the compiler. In Figure A1.13, this activity of backing up is labeled as pushback '/'.

A state diagram will help you to uncover missed or unexpected situations. That is, with a state diagram, it is relatively easy to ensure that all possible trigger events for all possible states have been accounted for. For example, in Figure A1.13, you can easily verify that every state has included transitions for all possible characters.

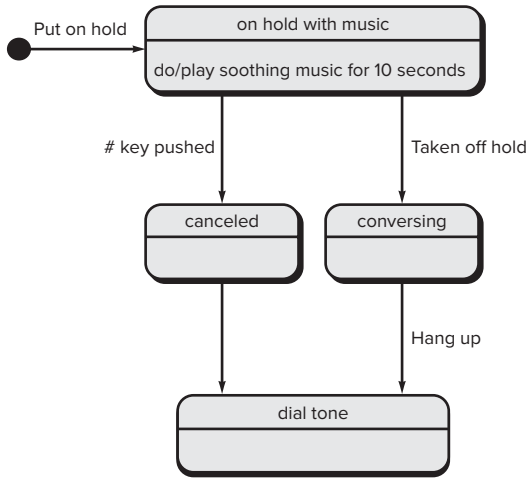
UML state diagrams can contain many other features not included in Figure A1.13. For example, when an object is in a state, it usually does nothing but sit and wait for a trigger event to occur. However, there is a special kind of state, called an *activity state*, in which the object performs some activity, called a *do-activity*, while it is in that state. To indicate that a state is an activity state in the state diagram, you include in the bottom half of the state's rounded rectangle the phrase "do/" followed by the activity that is to be done while in that state. The do-activity may finish before any state transitions occur, after which the activity state behaves like a normal waiting state. If a transition out of the activity state occurs before the do-activity is finished, then the do-activity is interrupted.

Because a trigger event is optional when a transition occurs, it is possible that no trigger event may be listed as part of a transition's label. In such cases for normal waiting states, the object will immediately transition from that state to the new state. For activity states, such a transition is taken as soon as the do-activity finishes.

Figure A1.14 illustrates this situation using the states for a business telephone. When a caller is placed on hold, the call goes into the **On hold with music** state

FIGURE A1.14

A state diagram with an activity state and a triggerless transition



(soothing music is played for 10 seconds). After 10 seconds, the do-activity of the state is completed and the state behaves like a normal nonactivity state. If the caller pushes the # key when the call is in the **On hold with music** state, the call transitions to the **Canceled** state and then transitions immediately to the **dial tone** state. If the # key is pushed before the 10 seconds of soothing music has completed, the do-activity is interrupted and the music stops immediately.

APPENDIX

DATA SCIENCE FOR SOFTWARE ENGINEERS

2

Contributed by: William Grosky and Terry Ruas[1]

KEY CONCEPTS

classification problems 634
computational intelligence 638
data science 629
dimensional reduction 637
machine learning 631
regression problems 634
search-based software engineering 638
statistical models 633

DATA SCIENCE—THE BIG PICTURE

Data science incorporates the work of many different disciplines to transform raw data into information, knowledge, and, hopefully, into wisdom. Data science has a long history that incorporates concepts from computer science, mathematics, statistics, data visualization, along with algorithms and their implementations. It is beyond the scope of this appendix to exhaustively detail all concepts under the rubric of "data science." Instead, we hope to provide a concise summary of the most important topics while connecting them to software engineering.

Figure A2.1 indicates that data science is the intersection of three major areas: computer science, mathematics and statistics, and domain knowledge [Con10].

A data scientist must be interested in more than the data itself. Using knowledge of mathematics and statistics along with domain-specific knowledge, the data scientist develops necessary skills to evaluate whether data, experiments, and evaluation are properly designed for a given problem. However, to bring these capabilities to different scenarios requires a certain flexibility, and good computing skills can be the way to accomplish it.

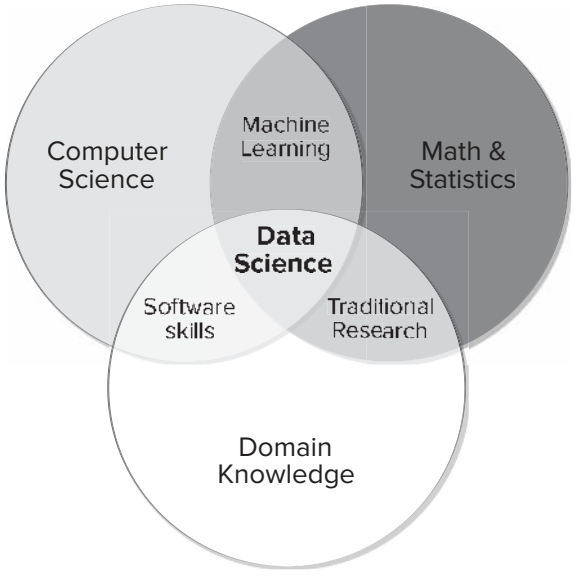
Popular Languages, APIs, and Tools

One of the great things about data science is that you can use it in virtually any environment that allows you to manipulate data. But to accomplish this, you'll need programming languages, APIs, and tools to make your lives easier. We'll provide an overview of these in the following sections.

Languages When it comes to programming languages, we all have our biases toward the one we like the most. For practitioners in data science, this is no different. However, you should keep in mind that one size does not fit all, and the proper approach to language selection for data science applications is to choose the right tool for the right job, considering its constraints, contexts, and goals.

1 Computer and Information Science Department, University of Michigan, Dearborn.

FIGURE A2.1
Data science Venn diagram



For data science application, fast prototyping is a strongly desired characteristic, one that allows us to produce interesting projects with a simplified and intuitive syntax. The resources available with respect to purpose and performance also play a crucial role in the adoption of a programming language. Thus, the less used a language is, the less attractive it will be for our daily tasks. Data science is closely related with *data munging*,[2] which is "the process of transforming and mapping data from one 'raw' data form into another format with the intent of making it more appropriate and valuable for a variety of downstream purposes, such as analytics." Data munging is a time-consuming activity and community support through well-documented APIs and libraries can make a significant difference, especially when looking for use examples, details about specific methods, constraints, and other technical aspects. Thus, a programming language used for data science applications should be broadly adopted, supported, and documented.

Therefore, it should come as no surprise that Python is widely used in the data science community. Other promising programming languages rising among data scientists are Scala and Julia, both more concerned with high performance and scalability. R is another interesting choice for data manipulation, specialized in statistical functions and data visualization libraries. Because its architecture is focused mainly on statistical analysis, data cleaning, and data visualization, R should not be your first choice for general-purpose programming. In other words, R is highly effective if used to solve the right problems.

Java's popularity is indisputable in data science and many other areas of software development. Following the recent trends in data science and big data, Java also has

2 See https://en.wikipedia.org/wiki/Data_wrangling.

dedicated frameworks, such as Hive,[3] Spark,[4] and Hadoop.[5] Considering its nonspecific architecture and verbosity, Java should not be the first option for advanced statistical analysis or data munging, especially for machine-learning algorithms. For these cases, Python and R offer dynamic scripting and huge dedicated libraries that might be more interesting. Other strong programming languages, but not that popular among data scientists, are C/C++, F#, SQL.

Libraries and Tools

It is impossible to talk about data science without referring to the artifacts that assist you in the process of extracting knowledge out of data. In this section, we'll note some of the most popular ones offered in Python [Van16].

NumPy is designed especially to efficiently manipulate *n*-dimensional arrays and handle scientific tasks. You can reshape the number of rows and columns, slice matrices, perform linear algebra operations, sort, search, and perform many other useful tasks. NumPy is used by a vast number of other libraries, and it is part of the SciPy stack.

There are two kinds of SciPy, the library itself and the scientific stack, composed of several open-source ecosystems, including the former. The sub-libraries that form the scientific stack are: NumPy, SciPy, Matplotlib, IPython, Sympy, and Pandas. The library, which is built on top of NumPy, is designed to provide efficient methods to deal with optimization, integration, and several other useful operations [Nun17], [Sci18].

As part of the scientific ecosystem, Pandas helps you with data structures and analysis through easy manipulations. It allows you to shape your data intuitively, providing easy adaptability from unstructured data to structured DataFrames. Some useful functions in Pandas include: indexing, labeling, fixing missing data records, and easy integration with different data structures [McK17], [Num18].

Particularly important for data science, Python also has a large portfolio of machine-learning libraries, in which Scikit-learn, TensorFlow, and Keras have a special place in the spotlight. Scikit-learn is probably one of the most known off-the-shelf machine-learning libraries in Python, featuring several algorithm types, such as clustering, regression, classification, and dimensionality reduction [Ger17]. TensorFlow, originally developed by the Google Brain team (part of Google's AI division), proposes an open-source machine-learning and deep-learning framework for everyone.

Aside from the presented libraries, Python also has several other specialized tools that are used in data science. Examples of visualization tools are Matplotlib, Seaborn, Bokeh, and Plotly; examples of natural language processing (NLP) tools are Natural Language Toolkit (NLTK), Gensim, spaCy, and Scrapy [Act18].

Data Science and Machine Learning

Data science is an umbrella term for a collection of data-driven approaches for finding approximate solutions to very difficult problems. The main enabling technology of data science is *machine learning*, a suite of statistics-based techniques that use an

3 https://hive.apache.org/.
4 https://spark.apache.org/.
5 https://hadoop.apache.org/.

inductive approach that attempts to generalize from a set of known exemplars to unknown exemplars.

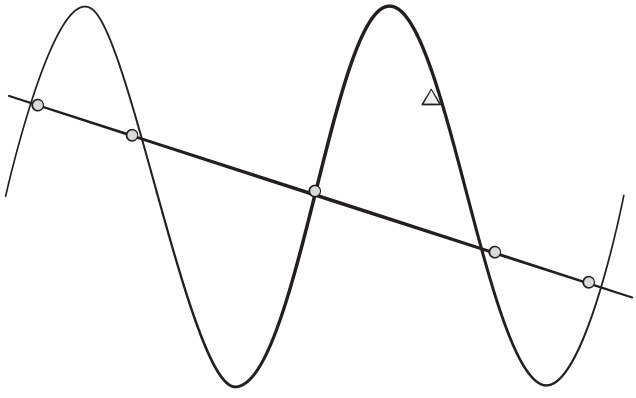
For example, our environment can consist of a set of multiple readings of various meteorological conditions, such as high temperature, low temperature, humidity, as well as several other values. We then have a small subset of these examples, our known exemplars, that are labeled; that is, for each example in this small set, the system is told whether it rained the following day or not. From this *training set* of known exemplars, the system then constructs a mathematical model to categorize an unknown daily example as to whether or not it will rain the following day. Another example from software engineering would be to predict whether an individual piece of code has a fault, based on a training set of faulty and nonfaulty programs. Alternatively, we might try to predict the cost of developing a new version of a program, based on the history of costs of previous versions of that same program.

In model building, there is an inherent tension between trying to construct a model that generalizes the training data, along with following the commonly held principle of Occam's razor, which is that the model should be as simple as possible to explain the current training set data. Figure A2.2 illustrates this conundrum. If the training set consists of only the circles, Occam's razor would choose the linear model, but with the addition of the triangle into the training set, perhaps Occam's razor would choose the sine curve. So, which is the appropriate model: straight line, sine curve, or some, as yet, unknown curve?

In the meteorological example, the system tries to discover commonalities and, at the same time, differences among the meteorological readings, both in the training set for the days it rained the following day and for the days it didn't rain the following day. Then it uses this metadata to determine whether or not it will rain tomorrow. Similarly, in the first software engineering example, the system tries to discover commonalities, in the training set, among programs with faults and those with no faults, as well as how faulty and nonfaulty programs differ, eventually being able to determine whether an unknown program is faulty or not. In the second software engineering example, the system tries to find a general rule that connects the cost of succeeding versions of programs in the training set, using this rule to predict the cost of succeeding versions not in the training set.

FIGURE A2.2

Linear versus nonlinear versus nonlinear model



The process which is followed during a data analytics project consists of (1) collecting the appropriate data, (2) *cleaning* the data, (3) *transforming* the data, (4) *analyzing* the data, and then (5) *fabricating* a training set.

1. **Data collection.** Thinking about what data to collect is quite important, as it depends on the goal of your project. Questions that should be answered include what type of data is needed and how much data should be collected. For example, for software engineering data collection, what type of artifacts are needed? Do we need source code, object code, bug logs? What volume of data do we need to do the appropriate analytics?
2. **Data cleaning.** Once collected, the data must be cleaned. This is a process that consists of eliminating problems in the data that would cause problems in further processing. For example, missing data should be filled in, and corrupt data should be found and corrected.
3. **Data transformation.** After cleaning, data should be transformed so they are more suitable for the downstream analytics tasks. This process is called *data munging* or *data wrangling*. An example of this activity might be changing the format in which the data appears, eliminating punctuation in a text data file, and doing parts-of-speech analysis for text data.
4. **Data analysis.** After all this, the data are ready to be analyzed and processed by various data analytics tools. But, before this happens, we generally use visualization tools for various tasks. For example, it may help us determine which features to use to predict the value of other features. It is only after this that we can determine the best analytical approach to be used for predictive or inferential purposes.
5. **Training set fabrication.** Choosing an appropriate training set is important. The generalizations produced from different training sets might differ among themselves, but the hope is that the downstream answers produced are still correct. It is important not to *overfit* the training set, which means that the approach predicts items in the training set with close to 100 percent accuracy but largely fails to predict the correct results for unknown items. This can happen quite easily if one is not careful to try to avoid this outcome. All of this is determined by testing the derived statistical model on a *test set* of data to determine its error rate.

For the above process to work, objects must be represented by some mathematical structure that can be manipulated easily and compared among themselves. A common way of associating a mathematical structure with an individual object is to use *feature vectors*. A *feature* is a given property of an object. A feature vector is a vector of values for multiple features of an object class, so that the feature vectors for objects in the same object class have the same feature ordering. For example, the meteorological features of a day may have the following structure: low-temperature, high-temperature, low-humidity, high-humidity, prevailing-cloud-type, overall-wind-strength. The variables of low-temperature, high-temperature, low-humidity, high humidity are *continuous* variables, while prevailing-cloud-type is an *unordered categorical variable* and overall-wind-strength is an *ordered categorical variable* (assuming the possible values are weak, average, strong).

In the software engineering environment, a feature vector corresponding to a piece of code could be a vector of values of several software metrics, such as number of lines of code, average program execution time, cohesion, and coupling. It is often challenging to choose the appropriate feature vector, and a new area of study, *feature engineering,* has evolved to help guide the process.

Machine-Learning Approaches

Machine learning is an integral part of data science. The process discussed earlier in this section establishes the data set that is used to drive learning. *Supervised learning* consists of approaches where the user is in the loop and interacts with the learning system, mainly by providing certain types of meta-information, such as labeled data for training sets. *Unsupervised learning* does not have the user in the loop to provide categorical information. These techniques are purely data driven and find ways of labeling the data from the data itself.

Within supervised learning approaches, there are two main types of problems: *classification* problems and *regression* problems.

Classification problems are those whose aim is to determine which of several classes an entity belongs; in other words, to predict a class label. A problem with two possible labels is called a *binary* classification problem, while a problem with more than two classes is called a *multiclass* classification problem. If an entity can fall into several classes, we have a *multilabel* classification problem. In this case, it often happens that the membership of an entity in an individual class is associated with a number between 0 and 1. This number can be interpreted as the strength of membership or the probability of membership. In this case, for a given entity, the sum of all its associated membership strengths or probabilities is equal to 1. A classic example of a binary classification problem is to classify e-mail as *spam* or *nonspam*. An example of a multiclass classification problem would be to classify the contents of an e-mail to various topic classes.

Regression problems are those whose aim is to predict the value of an output variable given the values of several input variables. The value predicted can be real valued or discrete valued. Suppose one had many feature vectors consisting of vita-related information for a prospective new hire. An example of a regression problem would be to predict the length of time that person will stay with your company before looking for a new job.

The boundary between classification problems and regression problems is imprecise. A regression problem where the values predicted are from a finite set can be couched as a classification problem where each class corresponds to a given value in the finite set of predicted values. Similarly, a classification problem can be couched as a regression problem where the output values predicted correspond to the set of class labels.

Popular techniques used for supervised learning include *linear regression*, *logistic regression*, *linear discriminant analysis*, *decision trees*, *k-nearest neighbor*, and *neural networks*. Popular techniques for unsupervised learning approaches include *neural networks*, *clustering*, and *dimensional reduction*. We consider only a small sampling of these techniques in this appendix.

Decision Trees Decision tree learning is a predictive technique that uses data-derived observations contained in the branches of the tree to develop conclusions about a target value contained in the leaves of the tree. Based on the input variable

values, a set of hierarchical decisions are made. The output variable value is found by following a tree from the root to a leaf, based on answers to questions asked along the way.

In general, decision trees can be binary or nonbinary and the questions asked can be arbitrary, as long as they conform to the number of children at an individual node. For this appendix, we will consider only binary trees with Boolean questions of the form $x < a$ or $x \leq b$, for some input variable x and constants a and b. If the answer to a question is TRUE, we would choose the left child to continue our walk down the tree, while if it is FALSE, we would choose the right child.

Given a training set of input and output variable values, we construct the tree choosing the type of Boolean question to ask at each internal node. This is usually done in a greedy fashion, simply asking, at a given node, which decision minimizes the sum of the squared errors. For visualization purposes, let us assume we have two input variables, x_1 and x_2, both of which are continuous. Suppose the training set is of the form (y, x_1, x_2) and is $t_1 = (5.7, 2.3, 9.6)$, $t_2 = (3.5, 1.1, 10)$, $t_3 = (0.55, 3.6, 17.5)$. We first must decide whether the first split will be on x_1 or x_2. We choose the input variable giving us the lowest error.

Now, each node of the tree is associated with a subset of the training set. For example, the root is associated with the entire training set. If the question at the root is $x_1 < 1$, then the root's left child is associated with the empty set and the root's right child is associated with the entire training set. However, if the question at the root is $x_1 < 1.8$, the root's left child is associated with t_2 and the root's right child is associated with the training tuples t_1 and t_3. Note that if we change 1.8 to 2.2, we would have the same association. However, even if the tree is the same, the choice of the split point would affect the results for input value pairs which are not in the training set.

So, what is the error produced by the $x_1 < 1.8$ split? If we stopped at this point, the tree would be used as follows. For an input value pair (c, d), if $c < 1.8$, we would predict the output value of 3.5, while if $c \geq 1.8$, we would predict the output value of 3.125, the average of 5.7 and 0.55. For this example, the squared error produced from the left child is 0, while the squared error produced from the right child is $(3.125 - 5.7)^2 + (3.125 - 0.55)^2 \cong 13.26$. We could stop here, or perform a further refinement on the right-hand side, producing a division of the plane into three regions, each associated with a single training set tuple. See Figure A2.3 for an illustration of the trees, associated regions, and errors for two different splits.

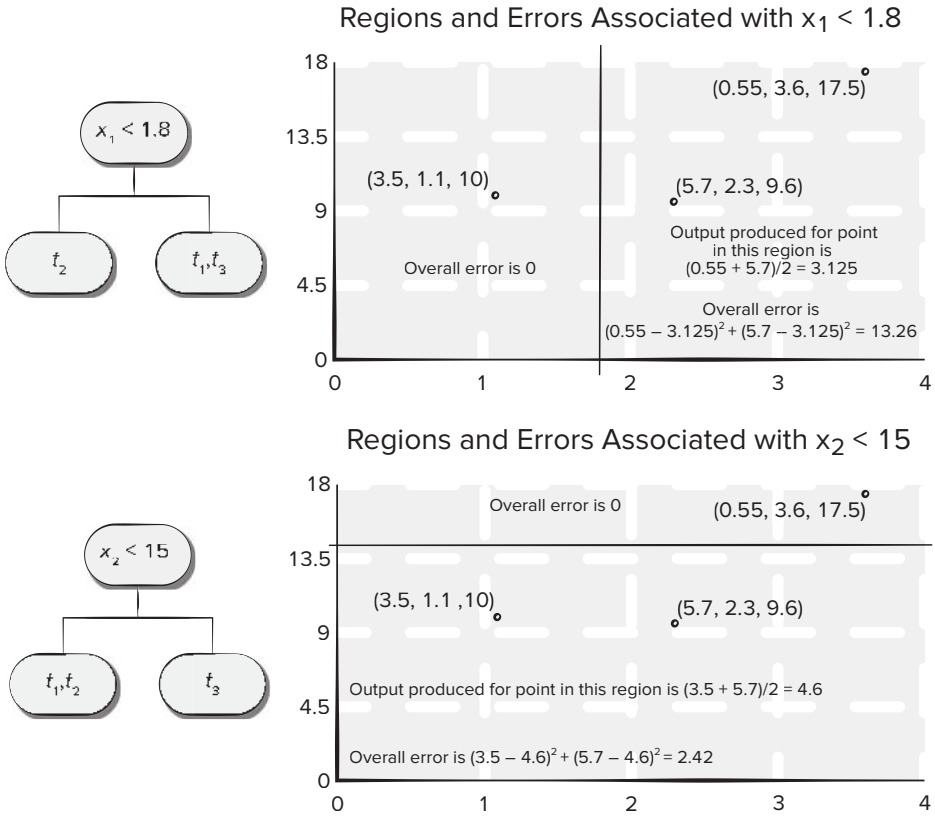
There are many efficient approaches to finding the best tree, which include when and how a region should be split and when to cease splitting a region containing more than one training set tuple. Some of these approaches can be found in [Jam13].

In Young et al. [You18], the use of decision trees in software engineering research is illustrated for the problem of just-in-time defect prediction. This technique predicts defects at small granularities. Code changes are predicted which are more likely to introduce defects. In this paper, it is demonstrated that the decision tree methodology is better than many other learning techniques for this problem. Decision trees are used in an *ensemble learning* environment. This type of learning combines many parallel learners in such a way that the final results are much improved.

Nearest Neighbor The technique of k-nearest neighbor is an approach to estimate the probability of membership of an unclassified input variable tuple, v, in a finite set

FIGURE A2.3

Decision trees examples for different splits (a) Decision tree split $x_1 < 1.8$ (b) Decision tree split $x_2 < 15$



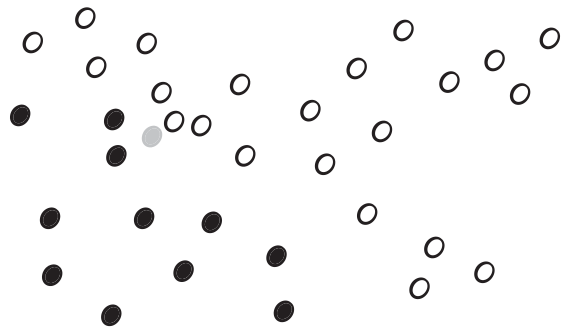
of classes. It is quite simple conceptually but can be very powerful. One finds the closest k points in the training set to the given point v. For a given class, c, suppose there are n points among these closest k training set points that belong to class c. Then the probability that v belongs to c would be n/k. If we had to label v with a single class, it would be the class with the highest probability. The value of k certainly affects the results. It has been found that values of k that are too small or too large don't perform well. As k increases to its sweet spot, the error decreases, but as k further increases, the error gets larger. See Figure A2.4 for an example: with 1-nearest neighbor, the gray point is classified as black; with 3-nearest neighbors, it is classified as white; and with 5-nearest neighbors, it is classified as white.

In Huang et al. [Hua17], nearest neighbor is used to develop an improved approach for missing data in the software quality area. Missing data causes many problems for machine learning. There are many approaches to intelligently estimate values for this data.

Neural Networks Neural networks embody *connectionism*, an architecture consisting of connections of multiple simple processors (i.e., brain cells) together in a massively parallel environment, supporting many concurrent processes, that can be used

FIGURE A2.4

***k*-Nearest neighbor example for $k = 1,3,5$**



to solve many problems. The power of neural networks results from the fact that this approach models the output variable as a nonlinear function of several linear combinations of the input variables. The more powerful neural networks have some form of feedback. To specify a neural network, we must specify the connectivity of the nodes, the way that a given node transforms all its inputs into an output, and how the final output is generated. In general, a neural network uses feedback through what is called a backpropagation technique (steepest descent or following the gradient direction) to train the network (estimate the parameters needed to carry out the regression). A weakness of neural networks is that it is hard to determine how particular parameters correlate with the parameters of the problem, leading to a weakness in explanatory power as to why a neural net has made a given decision.

Clustering Clustering is a general data-driven approach to find groups of any entities that are similar in some sense. A group of *clusters* are found, each cluster containing a set of entities. The idea is that two entities in the same cluster are highly similar, while two entities, each in different clusters, are not as similar. It is generally up to the investigator to figure out what exactly is meant by similarity. In the context of this appendix, the entities will be feature vectors and similarity is defined using a distance function between vectors. Vectors having a smaller distance between them will be more similar. There are hundreds of clustering algorithms, and there is no guarantee that you will get the same clustering from each of them. Clusters can have different shapes, and some algorithms work better with convex shapes, while others can relax this condition. Some algorithms are specifically designed for high-dimensional spaces, while others aren't.

Dimensional Reduction By dimensional reduction, we mean decreasing the lengths of the input feature vectors. In many important environments, the size of these vectors can get quite large. In natural language processing, for example, each vocabulary word has its own position in the vector. It is quite common, therefore, for these vectors to have from 5,000 to 50,000 elements. Reducing the dimensionality would thus speed up the learning process. Initially, in several disciplines of computer science, this was the sole reason for dimensionality reduction. However, it was soon discovered that reducing the dimensionality of the input feature vectors also improved the performance of many of the underlying algorithms used in downstream applications.

Computational Intelligence and Search-Based Software Engineering

Computational intelligence generally refers to the ability of a system (hardware and software) to learn a specific task from a set of data collected about that task. Some classify computational computing as a combination of granular computing (fuzzy sets, rough sets, probabilistic reasoning), neuro-computing (neural nets), evolutionary computing (genetic programming, genetic algorithms, and swarm intelligence), and artificial life (artificial immune systems). These approaches can be used for optimization, classification, search, and regression.

In search-based software engineering, computational intelligence–based techniques have been used for various sorts of optimizations. For example, there are papers [Oun17] using genetic algorithms for multi-objective optimization that quickly search the space of all possible code refactorings and recommend the best options, each option illustrating some particular trade-offs among the input variables, but still locally optimal. It has been shown that these approaches can also be used to build predictive models [Mal17].

The merger of data science, machine learning, and search-based software engineering may lead to exciting breakthroughs in the way software is specified, designed, coded, and tested. Time will tell.

References

[Abb83] Abbott, R., "Program Design by Informal English Descriptions," *CACM,* vol. 26, no. 11, November 1983, pp. 892–894.

[Abd16] Abdessalem, R., et al., "Testing Advanced Driver Assistance Systems Using Multi-Objective Search and Neural Networks," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering,* ACM, 2016, pp. 63–74.

[Abr17] Abreu, L., "UX Design Patterns for Mobile Apps: Which and Why," 2017, available at https://www.raywenderlich.com/167174/design-patterns-mobile-apps-which-why.

[ACM12] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice,* 2012, available at https://ethics.acm.org/code-of-ethics/software-engineering-code.

[ACM18] *ACM Code of Ethics and Professional Conduct,* 2018, available at https://ethics.acm.org/.

[Act18] ActiveWizards, "Top 20 Python Libraries for Data Science in 2018," February 13, 2018. Retrieved October 2018 from ActiveWizards: https://activewizards.com/blog/top-20-python-libraries-for-data-science-in-2018/.

[Ada16] Adams, B., and S. McIntosh, "Modern Release Engineering in a Nutshell—Why Researchers Should Care," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* March 2016, pp. 78–90.

[AFC88] Software Risk Abatement, AFCS/AFLC Pamphlet 800-45, U.S. Air Force, September 30, 1988.

[Agi17] Agile Alliance home page, available at https://www.agilealliance.org.

[Air99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.

[Alb10] Alberts, C., et al., "Integrated Measurement and Analysis Framework for Software Security," CMU/SEI-2010-TN-025. Software Engineering Institute, Carnegie Mellon University, 2010, available at http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9369.

[Ale02] Alexander, I., "Misuse Cases Help to Elicit Non-Functional Requirements," 2002, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.9670&rep=rep1&type=pdf.

[Ale03] Alexander, I., "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software,* vol. 20, no. 1, 2003, pp. 58–66.

[Ale11] Alexander, I., "Gore, Sore, or What?" *IEEE Software,* vol. 28, no. 1, January–February 2011, pp. 8–10.

[Ale17] Alebrahim, A., "Phase 1: Context Elicitation & Problem Analysis," *Bridging the Gap between Requirements Engineering and Software Architecture,* Springer Vieweg, Wiesbaden, 2017, available at https://doi.org/10.1007/978-3-658-17694-5_4.

[Ale77] Alexander, C., *A Pattern Language,* Oxford University Press, 1977.

[Ale79] Alexander, C., *The Timeless Way of Building,* Oxford University Press, 1979.

[Alh13] Alhusain, S., "Towards Machine Learning Based Design Pattern Recognition," *Proceedings of 13th UK Workshop on Computational Intelligence,* September 2013, pp. 244–251.

[Ali14] Alice, G., and Mead, N., "Using Malware Analysis to Tailor SQUARE for Mobile Platforms," CMU/SEI-2014-TN-018. Software Engineering Institute, Carnegie Mellon University, 2014, available at http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=425994.

[Ali18] Alizadeh, V., M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations," *IEEE Transactions on Software Engineering,* 2018, available at https://ieeexplore.ieee.org/document/8477161.

[All08] Allen, J. H., et al., *Software Security Engineering: A Guide for Project Managers,* Addison-Wesley Professional, 2008.

[All14] Alleman, G., *Performance-Based Project Management: Increasing the Probability of Project Success,* AMACOM, 2014.

[Als18] Alshahwan, N., et al., "Deploying Search Based Software Engineering with Sapienz at Facebook," *Search-Based Software Engineering SSBSE 2018.* Lecture Notes in Computer Science, T. Colanzi and P. McMinn (eds.), vol. 11036, Springer, 2018.

[Amb02] Ambler, S., and R. Jeffries, *Agile Modeling,* Wiley, 2002.

[Amb04] Ambler, S., "Examining the Cost of Change Curve," in *The Object Primer,* 3rd ed., Cambridge University Press, 2004.

[Amb95] Ambler, S., "Using Use-Cases," *Software Development,* July 1995, pp. 53–61.

[Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology,* Cambridge University Press/SIGS Books, 1998.

[Amp13] Ampatzoglou, A., et al., "Building and Mining a Repository of Design Pattern Instances: Practical and Research Benefits," *Entertainment Computing,* vol. 4, April 2013, pp. 131–142.

[And05] Andreou, A., et al., "Key Issues for the Design and Development of Mobile Commerce Services and Applications," *International Journal of Mobile Communications,* vol. 3, no. 3, March 2005, pp. 303–323.

[And06] Andrews, M., and J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services,* Addison-Wesley, 2006.

[And16] Anderson, D., and A. Carmichael, *Essential Kanban Condensed,* Lean Kanban University Press, 2016, available at leankanban.com/guide.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology,* 1987.

[App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," February 2000, available at www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[App13] Apple Computer, *Accessibility,* 2013, available at www.apple.com/accessibility/.

[Arl02] Arlow, J., and I. Neustadt, *UML and the Unified Process,* Addison-Wesley, 2002.

[Arn11] Arnuphaptrairong, T., "Top Ten Lists of Software Project Risks: Evidence from the Literature Survey," *Proceedings International Multi-Conference of Engineers and Computer Scientists*, vol. I, IMECS 2011, March 2011.

[ISO17] ISO/IEC/IEEE 24765:2017(E), ISO/IEC/IEEE International Standard: Systems and Software Engineering—Vocabulary, available at https://standards.ieee.org/findstds/standard/24765-2017.html.

[Ast04] Astels, D., *Test Driven Development: A Practical Guide,* Prentice Hall, 2004.

[Baa10] Baaz, A., et al., "Appreciating Lessons Learned," *IEEE Software,* vol. 27, no. 4, July–August, 2010, pp. 72–79.

[Bab09] Babar, M., and I. Groton, "Software Architecture Review: The State of Practice," *IEEE Computer,* vol. 42, no. 6, June 2009, pp. 1–8.

[Bab86] Babich, W. A., *Software Configuration Management,* Addison-Wesley, 1986.

[Bae98] Baetjer, Jr., H., *Software as Capital,* IEEE Computer Society Press, 1998, p. 85.

[Baj11] Bajdor, P., and L. Dragolea, "The Gamification as a Tool to Improve Risk Management in the Enterprise," *Annales Universitatis Apulensis Series Oeconomica,* vol. 2, no. 13, 2011, available at http://www.oeconomica.uab.ro/upload/lucrari/1320112/38.pdf.

[Bar06] Baresi, L., E. DiNitto, and C. Ghezzi, "Toward Open-World Software: Issues and Challenges," *IEEE Computer,* vol. 39, no. 10, October 2006, pp. 36–43.

[Bas12] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice,* 3rd ed., Addison-Wesley, 2012.

[Bat18] Batarseh, F., and A. Gonzalez, "Predicting Failures in Agile Software Development through Data Analytics," *A Software Quality Journal,* vol. 26, no. 1, March 2018, pp. 49–66.

[Bec99] Beck, K., *Extreme Programming Explained: Embrace Change,* Addison-Wesley, 1999.

[Bec01] Beck, K., et al., "Manifesto for Agile Software Development," 2001, available at www.agilemanifesto.org/.

[Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change,* 2nd ed., Addison-Wesley, 2004.

[Bec04b] Beck, K., *Test-Driven Development: By Example,* 2nd ed., Addison-Wesley, 2004.

[Bee99] Beedle, M., et al., "SCRUM: An Extension Pattern Language for Hyperproductive Software Development," included in *Pattern Languages of Program Design 4,* Addison-Wesley Longman, 1999, available at http://jeffsutherland.com/scrum/scrum_plop.pdf.

[Beg10] Begel, A., R. DeLine, and T. Zimmermann, "Social Media for Software Engineering," *Proceedings FoSER 2010,* ACM, November 2010.

[Bei90] Beizer, B., *Software Testing Techniques,* 2nd ed., Van Nostrand-Reinhold, 1990.

[Bei95] Beizer, B., *Black-Box Testing,* Wiley, 1995.

[Bel17] Bell, L., et al., *Agile Application Security,* O'Reilly Media, 2017.

[Bel14] Bellomo, S., P. Krutchen, R. Nord, and I. Ozkaya, "How to Agilely Architect an Agile Architecture," *Cutter IT Journal,* February 2014, pp. 10–15.

[Ben10a] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Analysis and Design Using UML,* 4th ed., McGraw-Hill, 2010.

[Ben10b] Benaroch, M., and A. Appari, "Financial Pricing of Software Development Risk Factors," *IEEE Software,* vol. 27, no. 3, September–October 2010, pp. 65–73.

[Ber80] Bersoff, E., V. Henderson, and S. Siegel, *Software Configuration Management,* Prentice Hall, 1980.

[Ber93] Berard, E., *Essays on Object-Oriented Software Engineering,* vol. 1, Addison-Wesley, 1993.

[Bin94] Binder, R., "Testing Object-Oriented Systems: A Status Report," *American Programmer,* vol. 7, no. 4, April 1994, pp. 23–28.

[Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools,* Addison-Wesley, 1999.

[Bir98] Biró, M., and T. Remzsö, "Business Motivations for Software Process Improvement," *ERCIM News,* no. 32, January 1998, available at www.ercim.org/publication/Ercim_News/enw32/biro.html.

[Bis02] Bishop, M., *Computer Security: Art and Science*, Addison-Wesley Professional, 2002.

[Bjø16] Bjørnson, F., and K. Vestues, "Knowledge Sharing and Process Improvement in Large-Scale Agile Development," *Proceedings Scientific Workshop of XP2016 (XP '16 Workshops).* ACM, New York, NY, 2016, Article 7.

[Bla10] Blair, S., et al., "Responsibility-Driven Architecture," *IEEE Software,* vol. 27, no. 3, March–April 2010, pp. 26–32.

[Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition," *CrossTalk,* May 2001, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.642.8250&rep=rep1&type=pdf.

[Boe01b] Boehm, B., and V. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer,* vol. 34, no. 1, January 2001, pp. 135–137.

[Boe08] Boehm, B., "Making a Difference in the Software Century," *IEEE Computer,* vol. 41, no. 3, March 2008, pp. 32–38.

[Boe81] Boehm, B., *Software Engineering Economics,* Prentice Hall, 1981.

[Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer,* vol. 21, no. 5, May 1988, pp. 61–72.

[Boe89] Boehm, B. W., *Software Risk Management,* IEEE Computer Society Press, 1989.

[Boe96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer,* vol. 31, no. 7, July 1998, pp. 33–44.

[Boh00] Bohl, M., and M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic,* 5th ed., Prentice Hall, 2000.

[Boh66] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *CACM,* vol. 9, no. 5, May 1966, pp. 366–371.

[Boi04] Boiko, B., *Content Management Bible,* 2nd ed., Wiley, 2004.

[Boo05] Booch, G., J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*, 2nd ed., Addison-Wesley, 2005.

[Boo11a] Booch, G., "Dominant Design," *IEEE Software,* vol. 28, no. 2, January–February 2011, pp. 8–9.

[Boo11b] Booch, G., "Draw Me a Picture," *IEEE Software,* vol. 28, no. 1, January–February 2011, pp. 6–7.

[Bos00] Bosch, J., *Design & Use of Software Architectures,* Addison-Wesley, 2000.

[Bos11] Bose, B., et al., "Morphing Smartphones into Automotive Application Platforms," *IEEE Computer,* vol. 44, no. 5, May 2011, pp. 28–29.

[Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software," informit.com, February 1, 2002, available at http://www.informit.com/articles/article.aspx?p=25141&seqNum=3.

[Bri09] Briand, L., "Model-Driven Development and Search-Based Software Engineering: An Opportunity for Research Synergy," *2009 1st International Symposium on Search Based Software Engineering,* Windsor, 2009.

[Bri13] Briers, B., "The Gamification of Project Management." PMI® Global Congress 2013—North America, New Orleans, LA. Newtown Square, PA: Project Management Institute, 2013, available at https://www.pmi.org/learning/library/gamification-project-management-5949.

[Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems," *IEEE Computer,* vol. 39, no. 10, October 2006, pp. 72–80.

[Bro10a] Brown, N., R. Nord, and I. Ozkaya, "Enabling Agility through Architecture," *Crosstalk,* November–December 2010, available at https://apps.dtic.mil/dtic/tr/fulltext/u2/a552111.pdf.

[Bro10b] Broy, M., and R. Reussner, "Software Architecture Review: The State of Practice," *IEEE Computer,* vol. 43, no. 10, June 2010, pp. 88–91.

[Bro12] Brown, A., *The Architecture of Open Source Applications,* lulu.com, 2012.

[Bro98] Brown, W. J., et al., *AntiPatterns—Refactoring Software, Architectures, and Projects in Crisis,* Wiley, 1998.

[Bud96] Budd, T., *An Introduction to Object-Oriented Programming,* 2nd ed., Addison-Wesley, 1996.

[Bur16] Burns, A., B. Umbaugh, and C. Dunn, "Introduction to the Mobile Software Development Lifecycle," 2016, available at https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/introduction-to-mobile-sdlc.

[Bus07] Buschmann, F., et al., *Pattern-Oriented Software Architecture, A System of Patterns,* Wiley, 2007.

[Bus10b] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 1," *IEEE Software,* vol. 27, no. 3, May–June 2010, pp. 63–65.

[Bus10c] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 2," *IEEE Software,* vol. 27, no. 4, July–August 2010, pp. 12–14.

[Cac02] Cachero, C., et al., "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification," *Proceedings of the 2nd International Workshop on Web-Oriented Technology,* June 2002, available at http://gplsi.dlsi.ua.es/iwad/ooh_project/papers/iwwost02.pdf.

[Car08] Carrasco, M., "7 Key Attributes of High Performance Software Development Teams," June 30, 2008, available at http://www.realsoftwaredevelopment.com/7-key-attributes-of-high-performance-software-development-teams/.

[Car90] Card, D., and R. Glass, *Measuring Software Design Quality,* Prentice Hall, 1990.

[Cav78] Cavano, J., and J. McCall, "A Framework for the Measurement of Software Quality," *Proceedings ACM Software Quality Assurance Workshop,* November 1978, pp. 133–139.

[CMM18a] *Capability Maturity Model Integration (CMMI),* Software Engineering Institute, 2018, available at https://cmmiinstitute.com/cmmi.

[CMM18b] *People Capability Maturity Model Integration (People CMM),* Software Engineering Institute, 2018, available at https://cmmiinstitute.com/cmmi/pm.

[Cha89] Charette, R., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[Cha92] Charette, R., "Building Bridges over Intelligent Rivers," *American Programmer,* vol. 5, no. 7, September 1992, pp. 2–9.

[Cha93] de Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.

[Chi94] Chidamber, S., and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering,* vol. SE-20, no. 6, June 1994, pp. 476–493.

[Cho16] Choma, J., et al., "Interaction Patterns for User Interface Design of Large Web Applications," *Proceedings 11th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP '16),* 2016, The Hillside Group, Article 8, 11 pages.

[Chr17] Christensen, E., "How to Create a Customer Journey Map," 2017, available at https://www.lucidchart.com/blog/how-to-build-customer-journey-maps.

[Chu09] Chung, L., and J. Leite, "On Non-Functional Requirements in Software Engineering," in A. T. Borgida et al. (eds.), *Conceptual Modeling: Foundations and Applications,* Springer-Verlag, 2009.

[Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings," 2007.

[Cla05] Clark, S., and E. Baniasaad, *Aspect-Oriented Analysis and Design,* Addison-Wesley, 2005.

[Cle03] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies,* Addison-Wesley, 2003.

[Cle10] Clements, P., and L. Bass, "The Business Goals Viewpoint," *IEEE Software,* vol. 27, no. 6, November–December 2010, pp. 38–45.

[CMM07] *Capability Maturity Model Integration (CMMI),* Software Engineering Institute, 2007, available at www.sei.cmu.edu/cmmi/.

[Coa91] Coad, P., and E. Yourdon, *Object-Oriented Analysis,* 2nd ed., Prentice Hall, 1991.

[Coc01a] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer,* vol. 34, no. 11, November 2001, pp. 131–133.

[Coc01b] Cockburn, A., *Writing Effective Use-Cases,* Addison-Wesley, 2001.

[Coc02] Cockburn, A., *Agile Software Development,* Addison-Wesley, 2002.

[Coh05] Cohn, M., "Estimating with Use Case Points," *Methods & Tools,* Fall 2005, available at http://www.mountaingoatsoftware.com/articles/estimating-with-use-case-points.

[Con02] Conradi, R., and A. Fuggetta, "Improving Software Process Improvement," *IEEE Software,* July–August 2002, pp. 2–9, available at http://citeseer.ist.psu.edu/conradi02improving.html.

[Con10] Conway, D., *The Data Science Venn Diagram,* 2010, available at http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram.

[Con95] Constantine, L., "What DO Users Want? Engineering Usability in Software," *Windows Tech Journal,* December 1995, available at http://www.wytsg.org:88/reslib/400/180/110/020/030/050/060/L000000000240585.pdf.

[Cro79] Crosby, P., *Quality Is Free,* McGraw-Hill, 1979.

[Cro07] Cross, M., *Developer's Guide to Web Application Security,* Syngress, 2007.

[Cur09] Curtis, B., and W. Heflley, *The People CMM: A Framework for Human Capital Management*, 2nd ed., Addison-Wesley, 2009.

[Cur90] Curtis, B., and D. Walz, "The Psychology of Programming in the Large: Team and Organizational Behavior," *Psychology of Programming,* Academic Press, 1990.

[DAC03] "An Overview of Model-Based Testing for Software," Data and Analysis Center for Software, CR/TA 12, June 2003.

[Dah72] Dahl, O., E. Dijkstra, and C. Hoare, *Structured Programming,* Academic Press, 1972.

[Dak14] Daka, E., and G. Fraser, "A Survey on Unit Testing Practices and Problems," *2014 IEEE 25th International Symposium on Software Reliability Engineering,* Naples, 2014, pp. 201–211.

[Dam17] Dam, R., and T. Siang, "Test Your Prototype: How to Gather Feedback and Maximize Learning," October 2017, available at https://www.interaction-design.org/literature/article/test-your-prototypes-how-to-gather-feedback-and-maximise-learning.

[Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems,* Software Engineering Institute, 2001, available at www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.

[Dar91] Dart, S., "Concepts in Configuration Management Systems," *Proceedings Third International Workshop on Software Configuration Management,* ACM SIGSOFT, 1991, available at https://dl.acm.org/citation.cfm?id=111063.

[Das15] Dasanayake, S., et al., "Software Architecture Decision-Making Practices and Challenges: An Industrial Case Study," *Proceedings of the 24th Australasian Software Engineering Conference,* SA, Australia, 2015. pp. 88–97.

[Dav93] Davis, A., et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proceedings of the. First International Software Metrics Symposium,* IEEE, Baltimore, MD, May 1993, pp. 141–152.

[Dav95a] Davis, M., "Process and Product: Dichotomy or Duality," *Software Engineering Notes,* ACM Press, vol. 20, no. 2, April 1995, pp. 17–18.

[Dav95b] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.

[Day99] Dayani-Fard, H., et al., "Legacy Software Systems: Issues, Progress, and Challenges," IBM Technical Report: TR-74.165-k, April 1999.

[Dem86] Deming, W., *Out of the Crisis,* MIT Press, 1986.

[DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.

[DeM98] DeMarco, T., and T. Lister, *Peopleware,* 2nd ed., Dorset House, 1998.

[Den73] Dennis, J., "Modularity," in *Advanced Course on Software Engineering,* F. L. Bauer (ed.), Springer-Verlag, 1973, pp. 128–182.

[Des08] de Sáa, M., and L. Carriocço, "Lessons from Early Stages Design of Mobile Applications," *Proceedings of 10th International Conference on Human Computer Human with Mobile Services and Devices,* 2008, pp. 127–136.

[Des09] de Souza, C., H. Sharp, G. Venolia, and L. Cheng, "Guest Editors' Introduction: Cooperative and Human Aspects of Software Engineering," *IEEE Software,* vol. 26, no. 6, 2009, pp. 17–19.

[Det11] Deterding, S., et al., "From Game Design Elements to Gamefulness: Defining Gamification," *Proceedings of the 2011 Annual Conference Extended Abstracts on Human Factors in Computing Systems—CHI EA '11,* 2011, p. 2425.

[Dia14] Díaz-Bossini, J., and L. Moreno, "Accessibility to Mobile Interfaces for Older People," *Procedia Computer Science,* vol. 27, 2014, pp. 57–66.

[Dij65] Dijkstra, E., "Programming Considered as a Human Activity," *Proceedings 1965 IFIP Congress,* North-Holland Publishing Co., 1965.

[Dij72] Dijkstra, E., "The Humble Programmer," 1972 ACM Turing Award Lecture, *CACM,* vol. 15, no. 10, October 1972, pp. 859–866.

[Dij76a] Dijkstra, E., "Structured Programming," in *Software Engineering, Concepts and Techniques*, J. Buxton et al. (eds.), Van Nostrand-Reinhold, 1976.

[Dij76b] Dijkstra, E., *A Discipline of Programming,* Prentice Hall, 1976.

[Dij82] Dijksta, E., "On the Role of Scientific Thought," in *Selected Writings on Computing: A Personal Perspective,* Springer-Verlag, 1982.

[Din16] Dingsøyr, T., and C. Lassenius, "Emerging Themes in Agile Software Development: Introduction to the Special Section on Continuous Value Delivery," *Information and Software Technology,* vol. 77, 2016, pp. 56–60.

[Dix99] Dix, A., "Design of User Interfaces for the Web," *Proceedings User Interfaces to Data Systems Conference,* September 1999, available at www.comp.lancs.ac.uk/computing/users/dixa/topics/webarch/.

[Don99] Donahue, G., S. Weinschenck, and J. Nowicki, "Usability Is Good Business," Compuware Corp., July 1999, available at www.compuware.com.

[Duc01] Ducatel, K., et al., Scenarios for Ambient Intelligence in 2010, ISTAG-European Commission, 2001, available at https://www.researchgate.net/publication/262007900_Scenarios_for_ambient_intelligence_in_2010.

[Dun82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982.

[Dut15] Dutra, E., and G. Santos, "Software Process Improvement Implementation Risks: A Qualitative Study Based on Software Development Maturity Models Implementations in Brazil," in *Product-Focused Software Process Improvement,* PROFES 2015 Lecture Notes in Computer Science, P. Abrahamsson et al. (eds.), vol. 9459, Springer, 2015.

[DXL18] DX Lab Design Sprint, "Make Your UX Design Process Agile Using Google's Methodology," available at https://www.interaction-design.org/literature/article/make-your-ux-design-process-agile-using-google-s-methodology, 2018.

[Dye15] Dyer, R., et al., "Boa: Ultra-Large-Scale Software Repository and Source-Code Mining," *ACM Transactions on Software Engineering Methodology,* vol. 25, no. 1, Article 7, December 2015.

[Ebe14] Ebert, C. "Software Product Management," *IEEE Software,* vol. 31, no. 3, May–June 2014, pp. 21–24.

[Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends,* vol. 2, no. 5, May 1995, pp. 37–42.

[Eis01] Eisenstein, J., et al., "Applying Model-Based Techniques to the Development of UIs for Mobile Computers," *Proceedings of Intelligent User Interfaces,* January 2001.

[Eji91] Ejiogu, L., *Software Engineering with Formal Metrics,* QED Publishing, 1991.

[Elb16] Elbanna, A., and S. Sarker, "The Risks of Agile Development: Learning from Adopters," *IEEE Software,* vol. 33, no. 5, September–October 2016, pp. 72–79.

[Era09] Erasmus, H., "The Seven Traits of Superprofessionals," *IEEE Software,* vol. 26, no. 4, July–August 2009, pp. 4–6.

[Erd10] Erdogmus, H., "Déjà vu: The Life of Software Engineering Ideas," *IEEE Software,* vol. 27, no. 1, January–February 2010, pp. 2–3.

[Eri15] Ericson, C., *Hazard Analysis Techniques for System Safety,* 2nd ed., Wiley, 2015.

[Eve09] Everett, G., and B. Meyer, "Point/Counterpoint," *IEEE Software,* vol. 26, no. 4, July–August 2009, pp. 62–65.

[Fag86] Fagan, M., "Advances in Software Inspections," *IEEE Transactions on Software Engineering,* vol. 12, no. 6, July 1986.

[Fai17] Fairley, R., and M. J. Willshire, "Better Now Than Later: Managing Technical Debt in Systems Development," *Computer,* vol. 50, no. 5, May 2017, pp. 80–87.

[Fal10] Falessi, D., et al., "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture," *IEEE Software,* vol. 27, no. 3, March–April 2010, pp. 23–25.

[Fel07] Feller, J., et al. (eds.), *Perspectives on Free and Open Source Software,* The MIT Press, 2007.

[Fel18] Feldt, R., et al., "Ways of Applying Artificial Intelligence in Software Engineering," In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE '18).* ACM, New York, NY, 2018, pp. 35–41.

[Fel89] Felican, L., and G. Zalateu, "Validating Halstead's Theory for Pascal Programs," *IEEE Transactions on Software Engineering,* vol. SE-15, no. 2, December 1989, pp. 1630–1632.

[Fen91] Fenton, N., *Software Metrics,* Chapman and Hall, 1991.

[Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering,* vol. SE-20, no. 3, March 1994, pp. 199–206.

[Fer14] Ferrucci F., M. Harman, and F. Sarro, "Search-Based Software Project Management," in *Software Project Management in a Changing World,* G. Ruhe and C. Wohlin (eds.), Springer, 2014.

[Fir13] Firesmith, D., *Security and Safety Requirements for Software-Intensive Systems,* Auerbach, 2013.

[Fow00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000.

[Fow01] Fowler, M., and J. Highsmith, "The Agile Manifesto," *Software Development Magazine,* August 2001.

[Fow02] Fowler, M., "The New Methodology," available at www.martinfowler.com/articles/newMethodology.html#N8B, June 2002.

[Fow03] Fowler, M., et al., *Patterns of Enterprise Application Architecture,* Addison-Wesley, 2003.

[Fow04] Fowler, M., *UML Distilled,* 3rd ed., Addison-Wesley, 2004.

[Fow16] Fowler, M., and Sutherland, J., *The Scrum Guide,* 2016, available at, http://www.scrumguides.org/.

[Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models,* Addison-Wesley, 1997.

[Fra93] Frankl, P., and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Transactions on Software Engineering,* vol. 19, no. 8, August 1993, pp. 770–787.

[Fre90] Freedman, D., and G. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews,* 3rd ed., Dorset House, 1990.

[Fri10] Fricker, S., "Handshaking with Implementation Proposals: Negotiating Requirements Understanding," *IEEE Software,* vol. 27, no. 2, March–April 2010, pp. 72–80.

[Fug14] Fuggetta, A., and E. Di Nitto, "Software Process," in *Proceedings of the on Future of Software Engineering (FOSE 2014),* ACM, New York, NY, 2014, pp. 1–12.

[Gag04] Gage, D., and J. McCormick, "We Did Nothing Wrong," *Baseline Magazine,* March 4, 2004, available at http://www.baselinemag.com/c/a/Projects-Processes/We-Did-Nothing-Wrong.

[Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences," Technical Report, University of Calgary, September 1995, downloadable at https://www.lri.fr/~mbl/ENS/FundHCI/2017/papers/Gaines-BRETAM99.pdf.

[Gal16] Galster, M., et al., "Variability and Complexity in Software Design," ACM SIGSOFT Software Engineering Notes, vol. 41, no. 6, November 2016, pp. 27–30.

[Gal17] Galster, M., et al., "Variability and Complexity in Software Design: Towards a Research Agenda," SIGSOFT Software Engineering Notes, vol. 41 no. 6, January 2017, pp. 27–30.

[Gam95] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[Gar08] GartnerGroup, "Understanding Hype Cycles," 2008, available at https://www.gartner.com/en/documents/3887767.

[Gar09a] Garlan, D., et al., "Architectural Mismatch: Way Reuse Is Still So Hard," *IEEE Software,* vol. 26, no. 4, July–August 2009, pp. 66–69.

[Gar10a] Garcia-Crespo, A., et al., "A Qualitative Study of Hard Decision Making in Managing Global Software Development Teams," *Journal of Management Information Systems,* vol. 27, no. 3, June 2010, pp. 247–252.

[Gar10b] Garrett, J. J., *The Elements of User Experience: User-Centered Design for the Web and Beyond,* 2nd ed., New Riders Publishing, 2010.

[Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review,* Fall 1984, pp. 25–45.

[Gar95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* vol. I, V. Ambriola and G. Tortora (eds.), World Scientific Publishing Company, 1995.

[Gas17] Gasparic, M., et al., "GUI Design for IDE Command Recommendations," *Proceedings of the 22nd International Conference on Intelligent User Interfaces (IUI '17),* 2017, ACM, New York, NY, pp. 595–599.

[Gau89] Gause, D., and G. Weinberg, *Exploring Requirements: Quality Before Design,* Dorset House, 1989.

[Ger17] Géron, A., *Hands-On Machine Learning with Scikit-Learn and TensorFlow,* O'Reilly Media, 2017.

[Gey01] Geyer-Schulz, A., and M. Hahsler, "Software Engineering with Analysis Patterns," Technical Report 01/2001, Institut für Informationsverarbeitung undwirtschaft, Wirschaftsuniversität Wien, November 2001, available at https://www.researchgate.net/publication/250241449_Software_Engineering_with_Analysis_Patterns.

[Gha14] Ghazi, P., A. M. Moreno, and L. Peters, "Looking for the Holy Grail of Software Development," *IEEE Software,* vol. 31, no. 1, January–February 2014, pp. 96.

[Gil06] Gillis, D., "Pattern-Based Design," tehan + lax blog, September 14, 2006.

[Gil88] Gilb, T., *Principles of Software Project Management,* Addison-Wesley, 1988.

[Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, January 1995.

[Gla02] Gladwell, M., *The Tipping Point,* Back Bay Books, 2002.

[Gla98] Glass, R., "Defining Quality Intuitively," *IEEE Software,* May–June 1998, pp. 103–104, 107.

[Gli07] Glinz, M., and R. Wieringa, "Stakeholders in Requirements Engineering," *IEEE Software,* vol. 24, no. 2, March–April 2007, pp. 18–20.

[Glu94] Gluch, D., "A Construct for Describing Software Development Risks," CMU/SEI-94-TR-14, Software Engineering Institute, 1994.

[Gna99] Gnaho, C., and F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering," *Proceedings of the First ICSE Workshop on Web Engineering,* ACM, Los Angeles, May 1999.

[Goa14] Gao, J., et al., "Mobile Application Testing: A Tutorial," *Computer,* vol. 47, no. 2, 2014, pp. 46–55.

[Gon04] Gonzales, R., "Requirements Engineering," Sandia National Laboratories, 2004, a slide presentation.

[Gon17] Gonzalez, D., et al., "A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension," *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17).* IEEE Press, Piscataway, NJ, 2017, pp. 391–401.

[Goo18] Google, Design Sprint Kit, 2018, available at https://designsprintkit.withgoogle.com/.

[Gor06] Gorton, I., *Essential Software Architecture,* Springer, 2006.

[Got11] Gotel, O., and S. Morris, "Requirements Tracery," *IEEE Software,* vol. 28, no. 5, September–October 2011, pp. 92–94.

[Got18] Gotterbarn, D., et al., "Thinking Professionally: The Continual Evolution of Interest in Computing Ethics," *ACM Inroads,* vol. 9, no. 2, April 2018, pp. 10–12.

[Gra18] Gray, C., et al., "The Dark (Patterns) Side of UX Design," *Proceedings 2018 CHI Conference on Human Factors in Computing Systems (CHI '18).* ACM, New York, NY, Paper 534, 2018.

[Gra92] Grady, R. G., *Practical Software Metrics for Project Management and Process Improvement,* Prentice Hall, 1992.

[Gru00] Gruia-Catalin, R., et al., "Software Engineering for Mobility: A Roadmap," *Proceedings of the 22nd International Conference on the Future of Software Engineering,* 2000.

[Gup15] Gupta, S., and V. Suma, "Data Mining: A Tool for Knowledge Discovery in Human Aspect of Software Engineering," *2015 2nd International Conference on Electronics and Communication Systems (ICECS),* Coimbatore, 2015, pp. 1289–1293.

[Gut15] Gutiérrez, J., M. Escalona, and M. Mejías, "A Model-Driven Approach for Functional Test Case Generation," *Journal of Systems and Software,* vol. 109, 2015, pp. 214–228.

[Hac98] Hackos, J., and J. Redish, *User and Task Analysis for Interface Design,* Wiley, 1998.

[Hal77] Halstead, M., *Elements of Software Science,* North-Holland, 1977.

[Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development,* Addison-Wesley, 1998.

[Har11] Harris, N., and P. Avgeriou, "Pattern-Based Architecture Reviews," *IEEE Software,* vol. 28, no. 6, November–December 2011, pp. 66–71.

[Har12a] Hardy, T., *Software and System Safety,* Authorhouse, 2012.

[Har12b] Harman, M., "The Role of Artificial Intelligence in Software Engineering," *Proceedings First International Workshop on Realizing AI Synergies in Software Engineering (RAISE '12).* IEEE Press, Piscataway, NJ, 2012, pp. 1–6.

[Har14] Harman, M., et al., "Search Based Software Engineering for Product Line Engineering: A Survey and Directions for Future Work," *Proceedings 18th International Software Product Line Conference,* SPL14, Florence, Italy, vol. 1, September 2014, pp. 5–18.

[Har98] Harrison, R., S. Counsell, and R. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering,* vol. SE-24, no. 6, June 1998, pp. 491–496.

[Hee15] Heeager, L., and J. Rose, "Optimising Agile Development Practices for the Maintenance Operation: Nine Heuristics," *Journal of Empirical Software Engineering,* vol. 20, no. 6, 2015, pp. 1762–1784.

[Hei02] Heitmeyer, C., "Software Cost Reduction," in *Encyclopedia of Software Engineering,* J. J. Marciniak (ed.), 2 vols., John Wiley & Sons, 2002, pp. 1374–1380.

[Hel18] Helfrich, J., *Security for Software Engineers,* Chapman and Hall/CRC, 2018.

[Her06] Hernan, S., et al., "Uncover Security Design Flaws Using the STRIDE Approach," *MSDN Magazine,* November 2006.

[Het84] Hetzel, W., *The Complete Guide to Software Testing,* QED Information Sciences, 1984.

[Hig02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2," *Cutter IT Journal,* vol. 15, no. 1, January 2002.

[Hig95] Higuera, R., "Team Risk Management," *CrossTalk,* U.S. Dept. of Defense, January 1995, pp. 2–4.

[Hil17] Hill, C., et al., "Gender-Inclusiveness Personas vs. Stereotyping: Can We Have It Both Ways?" *Proceedings 2017 CHI Conference on Human Factors in Computing Systems (CHI '17).* ACM, New York, NY, 2017, pp. 6658–6671.

[Hne11] Hneif, M., and S. Lee, "Using Guidelines to Improve Quality in Software Nonfunctional Attributes," *IEEE Software,* vol. 28, no. 5, November–December 2011, pp. 72–73.

[Hoe16] Hoegl, M., and M. Muethel, "Enabling Shared Leadership in Virtual Project Teams: A Practitioners' Guide," *Project Management Journal,* vol. 47, no. 1, February/March 2016, pp. 7–12.

[Hog04] Hoglund, G., and G. McGraw, *Exploiting Software: How to Break Code,* Addison-Wesley Professional, 2004.

[Hol06] Holzner, S., *Design Patterns for Dummies,* For Dummies Publishers, 2006.

[Hoo12] Hoober, S., and E. Berkman, *Designing Mobile Interfaces,* O'Reilly Media, 2012.

[Hoo96] Hooker, D., "Seven Principles of Software Development," September 1996, a summary of these principles is available at https://lingualeo.com/pt/jungle/seven-principles-of-software-development-by-david-hooker-48432#/page/1.

[Hop90] Hopper, M., "Rattling SABRE, New Ways to Compete on Information," *Harvard Business Review,* May–June 1990.

[Hor03] Horch, J., *Practical Guide to Software Quality Management,* 2nd ed., Artech House, 2003.

[Hua17] Huang, J., et al., "Cross-Validation Based K Nearest Neighbor Imputation for Software Quality Datasets: An Empirical Study," *Journal of Systems and Software,* 2017, pp. 226–252.

[Hub99] Hubbard, R., "Design, Implementation, and Evaluation of a Process to Structure the Collection of Software Project Requirements," PhD dissertation, Colorado Technical University, 1999.

[Hus15] Hussain, A., et al., "Usability Evaluation of Mobile Game Applications: A Systematic Review," *International Journal of Computer and Information Technology,* vol. 4, no. 3, May 2015, pp. 547–551.

[Hya96] Hyatt, L., and L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," NASA SATC, 1996, available at http://articles.adsabs.harvard.edu/cgi-bin/nph-iarticle_query?1996ESASP.377..209H&data_type=PDF_HIGH&whole_paper=YES&type=PRINTER&filetype=.pdf.

[IBM13] IBM, Web Services Globalization Model, 2013, downloadable from ftp://public.dhe.ibm.com/software/globalization/.../webservicesglobalizationmodel.pdf.

[IBM81] "Implementing Software Inspections," course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.

[IEE05] IEEE Std. 982.1-2005, *IEEE Standard Dictionary of Measures of the Software Aspects of Dependability,* 2005.

[IEE11] IEEE-Std-42010:2011(E), *Systems and Software Engineering-Architectural Description,* 2011, available at https://ieeexplore.ieee.org/document/6129467.

[IEE17] ISO/IEC/IEEE 24765:2017(E), *ISO/IEC/IEEE International Standard: Systems and Software Engineering—Vocabulary,* available at https://standards.ieee.org/findstds/standard/24765-2017.html.

[ISO08] ISO SPICE 2008, an earlier description is available at https://www.cs.helsinki.fi/u/paakki/Pyhajarvi.pdf.

[ISO14] ISO/IEC 90003:2014, Second Edition: *Software Engineering—Guidelines for the Application of ISO 9001:2008 to Computer Software,* International Organization for Standardization, 2014.

[ISO18] ISO/IEC/IEEE 90003:2018, *Software Engineering—Guidelines for the Application of ISO 9001:2015 to Computer Software,* International Organization for Standardization, 2018.

[ISO15] *Plain English Summary of ISO 9001: 2015,* 2015, available at http://praxiom.com/iso-9001.htm.

[ISO11] ISO/IEC 25010:2011, *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*, 2011, available at https://www.iso.org/standard/35733.html.

[Ive04] Iversen, J., L. Mathiassen, and P. Nielsen, "Managing Risk in Software Process Improvement: An Action Research Approach," *MIS Quarterly,* vol. 28, no. 3, September 2004, pp. 395–433.

[Ivo01] Ivory, M., R. Sinha, and M. Hearst, "Empirically Validated Web Page Design Metrics," ACM SIGCHI'01, March 31–April 4, 2001, available at http://webtango.berkeley.edu/papers/chi2001/.

[Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal,* vol. 15, no. 1, January 2002, pp. 18–24.

[Jac02b] Jacyntho, D., D. Schwabe, and G. Rossi, "An Architecture for Structuring Complex Web Applications," 2002, available at https://www.semanticscholar.org/paper/A-Software-Architecture-for-Structuring-Complex-Web-Jacyntho-Schwabe/2809668ede5034ed8d65e7669c6b0b463e9ee464.

[Jac04] Jacobson, I., and P. Ng, *Aspect-Oriented Software Development,* Addison-Wesley, 2004.

[Jac92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software,* July 1998, pp. 43–45.

[Jac99] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.

[Jai18] Jain, N., et al., "Digital Consumers, Emerging Markets, and the $4 Trillion Future," September 18, 2018, available at https://www.bcg.com/en-us/publications/2018/digital-consumers-emerging-markets-4-trillion-dollar-future.aspx.

[Jal04] Jalote, P., et al., "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software,* vol. 70, no. 2, 2004, pp. 117–127.

[Jam13] James, G., et al., *An Introduction to Statistical Learning with Applications in R,* Springer, 2013.

[Jan16] Jan, S., et al., "An Innovative Approach to Investigate Various Software Testing Techniques and Strategies," *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET),* vol. 2, no. 2, March–April 2016, pp. 682–689.

[Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success," *CrossTalk,* October 2004, available at http://www.pauldee.org/se-must-have/jones-failure-success.pdf.

[Jon86] Jones, C., *Programming Productivity,* McGraw-Hill, 1986.

[Jon91] Jones, C., *Systematic Software Development Using VDM,* 2nd ed., Prentice Hall, 1991.

[Jov15] Jovanovic M., A. Mesquida, and A. Mas, "Process Improvement with Retrospective Gaming in Agile Software Development," *Systems, Software and Services Process Improvement,* EuroSPI 2015 Communications in Computer and Information Science, R. O'Connor, et al. (eds.), vol. 543, Springer, 2015.

[Joy00] Joy, B., "The Future Doesn't Need Us," *Wired,* vol. 8, no. 4, April 2000.

[Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, available at http://www.exampler.com/testing-com/test-patterns/patterns/pattern-scenario-testing-kaner.html.

[Kan93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software,* 2nd ed., Van Nostrand-Reinhold, 1993.

[Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs," 1995, available at www.badsoftware.com/plaintif.htm.

[Kap15] Kapyaho, M., and M. Kauppinen, "Agile Requirements Engineering with Prototyping: A Case Study," *Proceedings IEEE 23rd International Requirements Engineering Conference,* August 2015, ON, Canada, pp. 334–343.

[Kar12] Kar S., S. Das, A. Kumar Rath, and S. K. Kar, "Self-assessment Model and Review Technique for SPICE: SMART SPICE," in *Software Process Improvement and Capability Determination,* A. Mas et al. (eds.), SPICE 2012, Communications in Computer and Information Science, vol. 290, 2012, Springer, Berlin, Heidelberg.

[Kar94] Karten, N., *Managing Expectations,* Dorset House, 1994.

[Kau11] Kaur, A., and S. Goel, "COTS Components Usage Risks in Component Based Software Development." *International Journal of Information Technology and Knowledge Management,* vol. 4, no. 2, July–December 2011, pp. 573–575.

[Kaz98] Kazman, R., et al., *The Architectural Tradeoff Analysis Method,* Software Engineering Institute, CMU/SEI-98-TR-008, July 1998, summarized at http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm.

[Kea07] Keane, "Testing Mobile Business Applications," a white paper, 2007. A 40-point checklist that complements this white paper is available at https://softcrylic.com/blogs/40-point-checklist-testing-mobile-applications/.

[Kei18] Keith, J., "10 Great Sites for UI Design Patterns," 2018, available at https://www.interaction-design.org/literature/article/10-great-sites-for-ui-design-patterns.

[Kei98] Keil, M., et al., "A Framework for Identifying Software Project Risks," *CACM,* vol. 41, no. 11, November 1998, pp. 76–83.

[Ker17] Kerzner, H., *Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 12th ed., Wiley, 2017.

[Kho12] Khode, A., "Getting Started with Mobile Apps Testing," 2012, available at http://www.mobileappstesting.com/getting-started-with-mobile-apps-testing/.

[Kim16a] Kim, G., et al., *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations,* Revolution Press, 2016.

[Kim16b] Kim, M., et al., "The Emerging Role of Data Scientists on Software Development Teams," *Proceedings 38th International Conference on Software Engineering (ICSE '16).* ACM, New York, NY, 2016, pp. 96–107.

[Kir94] Kirani, S., and W. Tsai, "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.

[Kiz05] Kizza, J., *Computer Network Security,* Springer, 2005.

[Kna16] Knapp, J., J. Zeratsky, and B. Kowitz, *Sprint: How to Solve Big Problems and Test New Ideas in Just Five Days,* Simon and Schuster, 2016.

[Koe12] Koester, J., "The Seven Deadly Sins of MobileApp Design," *Venture Beat/Mobile,* May 31, 2012, available at http://venturebeat.com/2012/05/31/the-7-deadly-sins-of-mobile-app-design/.

[Kor03] Korpipaa, P., et al., "Managing Context Information in Mobile Devices," *IEEE Pervasive Computing,* vol. 2, no. 3, July–September 2003, pp. 42–51.

[Kou14] Kouzes, J., *Five Practices of Exemplary Leadership—Technology,* Wiley, 2014.

[Kra88] Krasner, G., and S. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming,* vol. 1, no. 3, August–September 1988, pp. 26–49.

[Kra95] Kraul, R., and L. Streeter, "Coordination in Software Development," *CACM,* vol. 38, no. 3, March 1995, pp. 69–81.

[Kru05] Krutchen, P., "Software Design in a Postmodern Era," *IEEE Software,* vol. 22, no. 2, March–April 2005, pp. 16–18.

[Kru06] Kruchten, P., H. Obbink, and J. Stafford (eds.), "Software Architectural" (special issue), *IEEE Software,* vol. 23, no. 2, March–April 2006.

[Kru09] Kruchten, P., et al., "The Decision View's Role in Software Architecture Practice," *IEEE Software,* vol. 26, no. 2, March–April 2009, pp. 70–72.

[Kub17] Kubat, M., *An Introduction to Machine Leaning,* 2nd ed., Springer, 2017.

[Kul13] Kulkarni, V., "Model Driven Software Development," *Modelling Foundations and Applications,* ECMFA 2013. Lecture Notes in Computer Science, Van Gorp, et al. (eds.), vol. 7949, Springer, 2013.

[Kur05] Kurzweil, R., *The Singularity Is Near,* Penguin Books, 2005.

[Kur13] Kurzweil, R., *How to Create a Mind,* Viking, 2013.

[Kyb84] Kyburg, H., *Theory and Measurement,* Cambridge University Press, 1984.

[Laa00] Laakso, S., et al., "Improved Scroll Bars," In *CHI '00 Extended Abstracts on Human Factors in Computing Systems (CHI EA '00).* ACM, New York, NY, 2000, pp. 97–98.

[Lag10] Lago, P., et al., "Software Architecture: Framing Stakeholders' Concerns," *IEEE Software,* vol. 27, no. 6, November–December 2010, pp. 20–24.

[Lai02] Laitenberger, A., "A Survey of Software Inspection Technologies," in *Handbook on Software Engineering and Knowledge Engineering,* World Scientific Publishing Company, 2002.

[Lam09] Lamsweerde, A., "Goal-Oriented Requirements Engineering: A Guided Tour," *Proceedings of 5th IEEE International Symposium on Requirements Engineering,* Toronto, August 2009, pp. 249–263.

[Lan01] Lange, M., "It's Testing Time! Patterns for Testing Software," June 2001, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.5064&rep=rep1&type=pdf.

[Lar16] Larrucea, X., et al., "Software Process Improvement in Very Small Organizations," *IEEE Software,* vol. 33, no. 2, March–April 2016, pp. 85–89.

[Laz11] Lazzaroni, M., et al., *Reliability Engineering,* Springer, 2011.

[Leh97a] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change,* Academic Press, 1997.

[Leh97b] Lehman, M., et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings 4th International Software Metrics Symposium (METRICS '97),* IEEE, 1997, available at www.ece.utexas.edu/~perry/work/papers/feast1.pdf.

[Let04] Lethbridge, T., and R. Lagraniere, *Object-Oriented Software Engineering; Practical Software Development Using UML and Java,* 2nd ed., McGraw-Hill, 2004.

[Lev01] Levinson, M., "Let's Stop Wasting $78 billion a Year," *CIO Magazine,* October 15, 2001, available at https://www.cio.com/article/2441228/software-development---let-s-stop-wasting--78-billion-a-year.html.

[Lev12] Leveson, N., *Engineering a Safer World: Systems Thinking Applied to Safety* (Engineering Systems), MIT Press, 2012.

[Li16] Li, W., Z. Huang, and Q. Li, "Three-Way Decisions Based Software Defect Prediction," *Knowledge-Based Systems,* vol. 91, 2016, pp. 263–274.

[Lin16] Lin, Y., et al., "Interactive and Guided Architectural Refactoring with Search-Based Recommendation," *Proceedings 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* November 2016, pp. 535–546.

[Lin79] Linger, R., H. Mills, and B. Witt, *Structured Programming,* Addison-Wesley, 1979.

[Lis88] Liskov, B., "Data Abstraction and Hierarchy," *SIGPLAN Notices,* vol. 23, no. 5, May 1988.

[Maa07] Maassen, O., and S. Stelting, "Creational Patterns: Creating Objects in an OO System," 2007, available at www.informit.com/articles/article.asp?p=26452&rl=1.

[Mac10] Maciel, C., et al., "An Integration Testing Approach Based on Test Patterns and MDA Techniques," *Proceedings 8th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP '10),* ACM, New York, NY, Article 14, 2010.

[Mal16] Malhotra, R., "An Empirical Framework for Defect Prediction Using Machine Learning Techniques with Android Software," *Applied Software Computing,* vol. 49, December 2016, pp. 1034–1050.

[Mal17] Malhotra, R., M. Khanna, and R. Raje, "On the Application of Search-Based Techniques for Software Engineering Predictive Modeling: A Systematic Review and Future Directions," *Swarm and Evolutionary Computation,* vol. 32, February 2017, pp. 85–109.

[Man17] Mansoor, U., et al., "Multi-view Refactoring of Class and Activity Diagrams Using a Multi-Objective Evolutionary Algorithm," *Software Quality Journal,* vol. 25, no. 2, 2017, pp. 529–552.

[Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM,* vol. 24, no. 3, March 1981, pp. 106–113.

[Man97] Mandel, T., *The Elements of User Interface Design,* Wiley, 1997.

[Mao17] Mao, K., M. Harman, and Y. Jia, "Crowd Intelligence Enhances Automated Mobile Testing," *Proceedings 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017),* IEEE Press, Piscataway, NJ, 2017, pp. 16–26.

[Mar00] Martin, R., "Design Principles and Design Patterns," 2000, available at https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf.

[Mar02] Marick, B., "Software Testing Patterns," 2002.

[Mar94] Marick, B., *The Craft of Software Testing,* Prentice Hall, 1994.

[Mat94] Matson, J., et al., "Software Cost Estimation Using Function Points," *IEEE Transactions on Software Engineering,* vol. 20, no. 4, April 1994, pp. 275–287.

[Max16] Maxim, B. R., and M. Kessentini, "An Introduction to Modern Software Quality Assurance," in *Software Quality Assurance,* I. Mistrik et al. (eds.), Morgan Kaufman, 2016, pp. 19–46.

[McC09] McCaffrey, J., "Analyzing Risk Exposure and Risk Using PERIL," *MSDN Magazine,* January 2009, available at http://msdn.microsoft.com/en-us/magazine/dd315417.aspx.

[McC76] McCabe, T., "A Software Complexity Measure," *IEEE Transactions on Software Engineering,* vol. SE-2, December 1976, pp. 308–320.

[McC77] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977.

[McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 143–144.

[McG06] McGraw, G., *Software Security: Building Security In,* Addison-Wesley Professional, 2006.

[McG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes,* vol. 16, no. 4, October 1991, pp. 53–54.

[McG94] McGregor, J., and T. Korson, "Integrated Object-Oriented Testing and Development Processes," *CACM,* vol. 37, no. 9, September, 1994, pp. 59–77.

[McK17] McKinney, W., *Python for Data Analysis Data Wrangling with Pandas, NumPy, and IPython,* O'Reilly Media, 2017.

[McT16] McTear, M., Z. Callejas, and D. Griol, *The Conversational Interface: Talking to Smart Devices,* Springer, 2016.

[Mea05] Mead, N., E. Hough, and T. Stehney, "Security Quality Requirements Engineering (SQUARE) Methodology" (CMU/SEI-2005-TR-009, ADA452453), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005, available at http://www.sei.cmu.edu/publications/documents/05.reports/05tr009.html.

[Mea16] Mead, N., and C. Woody, *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance,* Addison-Wesley, 2016.

[Mea18] Mead, N., et al., "A Hybrid Threat Modeling Method," Software Engineering Institute CMU/SEI Report Number: CMU/SEI-2018-TN-002, March 2018, downloadable at https://resources.sei.cmu.edu/asset_files/TechnicalNote/2018_004_001_516627.pdf.

[Mei09] Meier, J., et al., *Microsoft Application Architecture Guide,* 2nd ed., Microsoft Press, 2009, available at http://msdn.microsoft.com/en-us/library/ff650706.

[Mei12] Meier, J., et al., "Chapter 19: Mobile Applications," *Application Architecture Guide, 2.0,* 2012, available at https://guidanceshare.com/wiki/Application_Architecture_Guide_-_Chapter_19_-_Mobile_Applications.

[Mei18] Meinke, K., and A. Bennaceur, "Machine Learning for Software Engineering: Models, Methods, and Applications," *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeding (ICSE '18).* ACM, 2018.

[Mel06] Mellado, D., et al., *Applying a Security Requirements Engineering Process,* Springer, 2006, available at https://pdfs.semanticscholar.org/379f/941eaf2878341948fbc30f6da246d90702ab.pdf.

[Men01] Mendes, E., N. Mosley, and S. Counsell, "Estimating Design and Authoring Effort," *IEEE Multimedia,* vol. 8, no. 1, January–March 2001, pp. 50–57.

[Men13] Menzies, T., and T. Zimmermann, "Software Analytics: So What?" *IEEE Software,* vol. 30, no. 4, July 2013, pp. 31–37.

[Mer93] Merlo, E., et al., "Reengineering User Interfaces," *IEEE Software,* January 1993, pp. 64–73.

[Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns," *MSDN,* May 2004, available at http://msdn2.microsoft.com/en-us/library/ms978700.aspx.

[Mic09] Microsoft Patterns & Practices Team, *Microsoft Application Architecture Guide,* 2nd ed., Microsoft Press, 2009.

[Mic10] Microsoft Security Development Lifecycle Version 5.0, 2010, available at http://download.microsoft.com/download/F/2/0/F205C451-C59C-4DC7-8377-9535D0A208EC/Microsoft%20SDL_Version%205.0.docx.

[Mic13a] Microsoft Accessibility Technology for Everyone, 2013, available at www.microsoft.com/enable/.

[Mic13b] Microsoft, "Patterns and Practices," *MSDN,* 2013, available at http://msdn.microsoft.com/en-us/library/ff647589.aspx.

[Mic17] Microsoft Corporation, "SDL Threat Modeling Tool," *Security Development Lifecycle,* November 10, 2017, available at https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx.

[Mic18] *The Microsoft Security Development Lifecycle (SDL)*, available at https://www.microsoft.com/en-us/securityengineering/sdl/, Microsoft, 2018.

[Mil00] Mili, A., and R. Cowan, "Software Engineering Technology Watch," April 6, 2000, available at https://www.researchgate.net/publication/222828018_Software_engineering_technology_watch.

[Mil04] Miler, J., and J. Gorski, "Risk Identification Patterns for Software Projects," *Foundations of Computing and Decision Sciences,* vol. 29, no. 1, 2004, pp. 115–131, available at http://iag.pg.gda.pl/RiskGuide/papers/Miler-Gorski_Risk_Identification_Patterns.pdf.

[Mil72] Mills, H., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.

[Mit14] Mitre Corp., "Common Weakness Ennumeration: A Community-Developed Dictionary of Software Weakness Types," 2014, available at http://cwe.mitre.org.

[Mob12] "Mobile UI Patterns," 2012, available at http://mobile-patterns.com/.

[Mof04] Moffett, J., et al., "Core Security Requirements Artefacts," Technical Report 2004/23. Milton Keynes, UK: Department of Computing, The Open University, June 2004, available at http://computing.open.ac.uk.

[Mol12] Molitor, M., "Software Configuration Management and Continuous Integration," 2012, available at https://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/6-scm.pdf.

[Mor05] Morales, A., "The Dream Team," Dr. Dobbs Portal, March 3, 2005, available at www.ddj.com/dept/global/184415303.

[Mor81] Moran, T., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal of Man-Machine Studies,* vol. 15, 1981, pp. 3–50.

[Mun17] Munaiah, N., et al., "Do Bugs Foreshadow Vulnerabilities? An In-depth Study of the Chromium Project," *Empirical Software Engineering,* vol. 22, 2017, pp. 1305–1347.

[Mus87] Musa, J., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures,* McGraw-Hill, 1987.

[Mye78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.

[Mye79] Myers, G., *The Art of Software Testing,* Wiley, 1979.

[Nan14] Ramanathan, N., A. Lal, and R. Parmar, "State of the Art in Software Quality Assurance," *ACM SIGSOFT Software Engineering Notes,* vol. 39, 2014, pp. 1–6.

[NAS07] NASA, "Software Risk Checklist, Form LeR-F0510.051," March 2007, available at https://www.scribd.com/document/250436/Software-Risk-Checklist-Department-of-Defense-NASA-USA.

[Nat15] National Instruments, "7 Steps in Creating a Functional Prototype," November 2015, available at http://www.ni.com/white-paper/10590/en/.

[Nei14] Neil, T., *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps,* 2nd ed., O'Reilly Media, 2014.

[Nei93] Nielsen, J., *Usability Engineering.* Morgan Kaufmann Publishers Inc., 1993.

[Net18] Neto, F., et al., "Improving Continuous Integration with Similarity-Based Test Case Selection," *Proceedings 13th International Workshop on Automation of Software Test (AST '18),* ACM, New York, NY, 2018, pp. 39–45.

[Nie00] Nielsen, J., *Designing Web Usability,* New Riders Publishing, 2000.

[Nie10] Nielsen, D., "Successfully Building a Software Prototype," July 2010, http://www.nuwavetech.com/it-project-blog/bid/43839/successfully-building-a-software-prototype (downloaded January 16, 2018).

[Nie13] Nielsen, L., "Personas," in *The Encyclopedia of Human-Computer Interaction*, 2nd ed., M. Soegaard (ed.), The Interaction Design Foundation, 2013.

[Nie94] Nielsen, J., and J. Levy, "Measuring Usability: Preference vs. Performance," *CACM,* vol. 37, no. 4, April 1994, pp. 65–75.

[Nie96] Nielsen, J., and A. Wagner, "User Interface Design for the WWW," *Proceedings CHI '96 Conf. on Human Factors in Computing Systems,* ACM Press, 1996, pp. 330–331.

[Nor13] Norman, D. A., *The Design of Everyday Things,* Revised Expanded Edition, Basic Books, Inc., 2013.

[Nor70] Norden, P., "Useful Tools for Project Management," in *Management of Production,* M. K. Starr (ed.), Penguin Books, 1970.

[Nor88] Norman, D., *The Design of Everyday Things,* Doubleday, 1988.

[Nov05] Novotny, O., "Next Generation Tools for Object-Oriented Development," *The Architecture Journal,* January 2005, available at http://msdn2.microsoft.com/en-us/library/aa480062.aspx.

[Num18] NumFOCUS, "Python Data Analysis Library," 2018. Retrieved from pandas: http://pandas.pydata.org/.

[Nun11] Nunes, N., L. Constantine, and R. Kazman, "iUCP: Estimating Interactive Software Project Size with Enhanced Use Case Points," *IEEE Software,* vol. 28, no. 4, July–August 2011, pp. 64–73.

[Nun17] Nunez-Iglesias, J., S. Walt, and H. Dashnow, *Elegant SciPy,* O'Reilly Media, 2017.

[Nyg11] Nygard, M., "Documenting Architecture Decisions," 2011, available at http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions.

[Off02] Offutt, J., "Quality Attributes of Web Software Applications," *IEEE Software,* March–April 2002, pp. 25–32.

[Ols99] Olsina, L., et al., "Specifying Quality Characteristics and Attributes for Web Sites," *Proceedings 1st ICSE Workshop on Web Engineering,* ACM, Los Angeles, May 1999.

[OMG03] Object Management Group, *OMG Unified Modeling Language Specification,* version 1.5, March 2003, available at www.rational.com/uml/resources/documentation/.

[Oth17] Othmane, B. L., et al., "Time for Addressing Software Security Issues: Prediction Models and Impacting Factors," *Data Science Engineering,* vol. 2, no. 2, 2017, pp. 107–124.

[Osb90] Osborne, W. M., and E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle," *IEEE Software,* January 1990, pp. 10–11.

[OSO12] OpenSource.org, 2012, available at www.opensource.org/.

[Oun17] Ouni, A., M. Kessentini, and M. Cinneide, "MORE: A Multi-Objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells," *Journal of Software: Evolution and Process,* 2017, available at https://doi.org/10.1002/smr.1843.

[OWA16] "Open Web Application Security Project—Buffer Overflow," 2016, available at https://www.owasp.org/index.php/Buffer_Overflow.

[OWA18] "Open Web Application Security Project—Attack Surface Cheat Sheet," 2018, available at https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.md.

[Pad18] Padhy, N., et al., "Software Reusability Metrics Estimation: Algorithms, Models and Optimization Techniques," *Computers and Electrical Engineering,* vol. 69, July 2018, pp. 653–668.

[Pag85] Page-Jones, M., *Practical Project Management,* Dorset House, 1985.

[Par11] Pardo, C., et al., "Harmonizing Quality Assurance Processes and Product Characteristics," *IEEE Computer,* June 2011, pp. 94–96.

[Par15] Parunak, H., and S. Brueckner, "Software Engineering for Self-Organizing Systems," *The Knowledge Engineering Review,* vol. 30, no. 4, September 2015, pp. 419–434.

[Par72] Parnas, D., "On Criteria to Be Used in Decomposing Systems into Modules," *CACM,* vol. 14, no. 1, April 1972, pp. 221–227.

[Par96b] Park, R. E., W. B. Goethert, and W. A. Florac, *Goal Driven Software Measurement—A Guidebook,* CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.

[Pas10] Passos, L., et al., "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software,* vol. 27, no. 5, September–October 2010, pp. 82–89.

[Pau94] Paulish, D., and A. Carleton, "Case Studies of Software Process Improvement Measurement," *Computer,* vol. 27, no. 9, September 1994, pp. 50–57.

[Ped15] Pedreira, O., et al., "Gamification in Software Engineering—A Systematic Mapping," *Information and Software Technology,* vol. 57, 2015, pp. 157–168.

[Per74] Persig, R., *Zen and the Art of Motorcycle Maintenance,* Bantam Books, 1974.

[Pet18] Petke, J., et al., "Genetic Improvement of Software: A Comprehensive Survey," in *IEEE Transactions on Evolutionary Computation,* vol. 22, no. 3, June 2018, pp. 415–432.

[Pew18] PEW Research Center, "Mobile Fact Sheet," 2018, available at http://www.pewinternet.org/fact-sheet/mobile/.

[Phi02] Phillips, M., "CMMI V1.1 Tutorial," April 2002, available at www.sei.cmu.edu/cmmi/.

[Pit18] Pitoura, E., et al., "On Measuring Bias in Online Information," *SIGMOD,* vol. 46, no. 4, February 2018, pp. 16–21.

[Pol45] Polya, G., *How to Solve It,* Princeton University Press, 1945.

[Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve,* 2008, available at www.faithpopcorn.com/.

[Por18] Port, D., and B. Taber, "Actionable Analytics for Strategic Maintenance of Critical Software: An Industry Experience Report," *IEEE Software,* vol. 35, no. 1, January–February 2018, pp. 58–63.

[Pre05] Pressman, R., "Adaptable Process Model, Version 2.0," R. S. Pressman & Associates, 2005, available at www.rspa.com/apm/index.html.

[Pre08] Pressman, R., and D. Lowe, *Web Engineering: A Practitioner's Approach,* McGraw-Hill, 2008.

[Pre88] Pressman, R., *Making Software Engineering Happen,* Prentice Hall, 1988.

[Pre94] Premerlani, W., and M. Blaha, "An Approach for Reverse Engineering of Relational Databases," *CACM,* vol. 37, no. 5, May 1994, pp. 42–49.

[Pri10] Prince, B., "10 Most Dangerous Web App Security Flaws," *eWeek.com,* April 19, 2010, available at https://www.eweek.com/security/10-most-dangerous-web-app-security-risks.

[Pun17] Punchoojit, L., and N. Hongwarittorrn, "Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review," *Advances in Human-Computer Interaction,* available at https://www.hindawi.com/journals/ahci/2017/6787504/

[Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Transactions on Software Engineering,* vol. SE-4, no. 4, July 1978, pp. 345–361.

[Put92] Putnam, L., and W. Myers, *Measures for Excellence,* Yourdon Press, 1992.

[Pyz14] Pyzdek, T., and P. Keller, *The Six Sigma Handbook,* 4th ed., McGraw-Hill, 2014.

[Rad02] Radice, R., *High-Quality Low Cost Software Inspections,* Paradoxicon Publishing, 2002.

[Raj14] Rajagoplan, S., "Review of the Myths on Original Development Model," *International Journal of Software Engineering and Applications,* vol. 5, no. 6, November 2014, pp. 103–111.

[Ray12] Raymond P., L. Buse, and T. Zimmermann. "Information Needs for Software Development Analytics," *Proceedings 34th International Conference on Software Engineering (ICSE '12),* IEEE Press, Piscataway, NJ, 2012, pp. 987–996.

[Ree99] Reel, J., "Critical Success Factors in Software Projects," *IEEE Software,* May 1999, pp. 18–23.

[Rem14] Rempel, P., et al., "Mind the Gap: Assessing the Conformance of Software Traceability to Relevant Guidelines," *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* ACM, New York, NY, 2014, pp. 943–954.

[Reu12] Reuveni, D., "Crowdsourcing Provides Answer to App Testing Dilemma," 2012, available at https://www.ecnmag.com/article/2010/02/crowdsourcing-provides-answer-app-testing-dilemma.

[Ric04] Rico, D., *ROI of Software Process Improvement,* J. Ross Publishing, 2004, available at http://davidfrico.com/rico03a.pdf.

[Rob10] Robinson, W., "A Roadmap for Comprehensive Requirements Monitoring," *IEEE Computer,* vol. 43, no. 5, May 2010, pp. 64–72.

[Rod16] Rodríguez, A., F. Ortega, and R. Concepción, "A Method for the Evaluation of Risk in IT Projects," *Expert Systems with Applications,* vol. 45, 2016, pp. 273–285.

[Rod17] Rodríguez, P., et al., "Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study," *Journal of Systems and Software,* vol. 123, 2017, pp. 263–291.

[Rod98] Rodden, T., et al., "Exploiting Context in HCI Design for Mobile Systems," *Proceedings of Workshop on Human Computer Interaction with Mobile Devices,* 1998.

[Roo09] Rooksby, J., et al., "Testing in the Wild: The Social and Organizational Dimensions of Real World Practice," *Journal of Computer Supported Work,* vol. 18, no. 5–6, December 2009, pp. 559–580.

[Ros17] Rosa, W., and C. Wallshein, "Software Effort Estimation Models for Contract Cost Proposal Evaluation," *Proceedings 2017 ICEAA Professional Development & Training Workshop,* June 2017, pp. 1–8.

[Ros75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer,* vol. 8, no. 5, May 1975.

[Rot02] Roth, J., "Seven Challenges for Developers of Mobile Groupware," *Proceedings of Computer Human Interaction Workshop on Mobile Ad Hoc Collaboration,* 2002.

[Rou02] Rout, T. (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide,* 2002, downloadable at http://www.noginfo.com.br/arquivos/SPICE.pdf.

[Roy70] Royce, W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings WESCON,* August 1970.

[Roz11] Rozanski, N., and E. Woods, *Software Systems Architecture,* 2nd ed., Addison-Wesley, 2011.

[Rya11] Ryan, T., *Statistical Methods for Quality Improvement,* Wiley, 2011.

[San17] Sancetta, G., et al., "Risk Patterns, Structural Characteristics, and Organizational Configurations," *Strategic Change,* 2017, available at https://doi.org/10.1002/jsc.2138.

[Sce02] Sceppa, D., *Microsoft ADO.NET,* Microsoft Press, 2002.

[Sca15] Scandariato, R., K. Wuyts, and W. Joosen, "A Descriptive Study of Microsoft's Threat Modeling Technique," *Requirements Engineering,* vol. 20, no. 2, June 2015, pp. 163–180.

[Sch01a] Schneider, G., and J. Winters, *Applying Use Cases,* 2nd ed., Que, 2001.
[Sch01b] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM,* Prentice Hall, 2001.
[Sch06] Schmidt, D., "Model-Driven Engineering," *IEEE Computer,* vol. 39, no. 2, February 2006, pp. 25–31.
[Sch09] Schumacher, R. (ed.), *Handbook of Global User Research,* Morgan-Kaufmann, 2009.
[Sch11] Schilit, B., "Mobile Computing: Looking to the Future," *IEEE Computer,* vol. 44, no. 5, May 2011, pp. 28–29.
[Sch15] Schell, M., and J. O'Brien, *Communicating the UX Vision: 13 Anti-Patterns That Block Ideas,* Morgan Kaufman, 2015.
[Sch98] Schneider, G., and J. Winters, *Applying Use Cases,* Addison-Wesley, 1998.
[Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics," *IEEE Transactions,* SE, vol. 25, no. 6, November–December 1999, pp. 768–781.
[Sci18] SciPy Developers, *SciPy Library,* 2018, available at https://scipy.org/scipylib/index.html.
[SEI02] SEI, "Maintainability Index Technique for Measuring Program Maintainability," 2002.
[SEI08] Software Engineering Institute, "The Ideal Model," 2008, available at https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=20208.
[Ser15] Serrador, P., and J. Pinto, "Does Agile Work?—A Quantitative Analysis of Agile Project Success," *International Journal of Project Management,* vol. 33, no. 5, 2015, pp. 1040–1051.
[Sha05] Shalloway, A., and J. Trott, *Design Patterns Explained,* 2nd ed., Addison-Wesley, 2005.
[Sha09] Shaw, M., "Continuing Prospects for an Engineering Discipline of Software," *IEEE Software,* vol. 26, no. 8, November–December 2009, pp. 64–67.
[Sha15] Shaw, M., and D. Garla, *Software Architecture: Perspectives on an Emerging Discipline,* Pearson, 2015.
[Sha17] Sharma, S., and B. Coyne, *DevOps for Dummies,* 3rd ed., Wiley, 2017.
[Shn09] Shneiderman, B., et al., *Designing the User Interface,* 5th ed., Addison-Wesley, 2009.
[Shn16] Shneiderman, B., et al., *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* 6th ed., Pearson, 2016.
[Sho14] Shostack, A., *Threat Modeling: Designing for Security.* John Wiley & Sons, 2014.
[Shu12] Shull, F., "Designing a World at Your Fingertips: A Look at Mobile User Interfaces," *IEEE Software,* vol. 29, no. 4, July–August 2012, pp. 4–7.
[Shu13] Shunn, A., et al., *Strengths in Security Solutions, Software Engineering Institute,* Carnegie Mellon University, 2013, available at http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=77878.
[Shu16] Shull, F., *Evaluation of Threat Modeling Methodologies,* Software Engineering Institute, Carnegie Mellon University, 2016, available at http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=474197.
[Sin00] Sindre, G., and A. Opdahl, "Eliciting Security Requirements by Misuse Cases," *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37'00),* New York, NY: IEEE Press, 2000, pp. 120–131.
[Sin01] Sindre, G., and A. Opdahl, "Templates for Misuse Case Description," *Seventh International Workshop on Requirements Engineering: Foundation for Software Quality,* 2001, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.8190.
[Sla12] Slattery, K., "Study Shows the Importance of Localization Testing," 2012. A related article can be found at https://www.welocalize.com/6-reasons-localization-qa-testing-important/.
[Smo08] Smolander, K., et al., "Software Architectures: Blueprint, Literature, Language, or Decision?" *European Journal of Information Systems,* vol. 17, 2008, pp. 575–588.
[Sne18] Snee, R., and R. Hoerl, *Leading Six Sigma,* 2nd ed., Pearson, 2018.
[Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems," *IEEE Software,* January 1995, pp. 24–25.
[Soa10] Soares, G., et al., "Making Program Refactoring Safer," *IEEE Software,* vol. 37, no. 4, July–August 2010, pp. 52–57.
[Soa11] SOASTA, White Paper: "Five Strategies for Performance Testing Mobile Applications," 2011, available at http://hosteddocs.ittoolbox.com/whitepapersoastamobile.pdf.
[Som97] Somerville, I., and P. Sawyer, *Requirements Engineering,* Wiley, 1997.
[Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial," *IEEE Software,* vol. 22, no. 1, January–February 2005, pp. 16–23.

[Sou08] de Sousa, C., and D. Redmiles, "An Empirical Study of Software Developer's Management of Dependencies and Changes," *ICSE Proceedings,* May 2008, available at www.ics.uci.edu/~redmiles/publications/C078-deSR08.pdf.

[Spa11] Spagnolli, B., et al., "Eco-Feedback on the Go: Motivating Energy Awareness," *IEEE Computer,* vol. 44, no. 5, May 2011, pp. 38–45.

[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction," Version 1.0, ISO/IEC JTC1, 1999.

[SSO08] Software-Supportability.org, 2008, available at www.software-supportability.org/.

[Ste10] Stephens, M., and D. Rosenberg, *Design Driven Testing,* Apress, 2010.

[Ste16] Steed, S., et al., "An 'In the Wild' Experiment on Presence and Embodiment Using Consumer Virtual Reality Equipment," *IEEE Transactions on Visualization and Computer Graphics,* vol. 22, no. 4, April 2016, pp. 1406–1414.

[Ste18] Steffens, A., H. Lichter, and J. Döring, "Designing a Next-Generation Continuous Software Delivery System: Concepts and Architecture," *Proceedings 4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18),* ACM, New York, NY, 2018, pp. 1–7.

[Ste74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, no. 2, 1974, pp. 115–139.

[Sto05] Stone, D., et al., *User Interface Design and Evaluation,* Morgan Kaufman, 2005.

[Sul11] Sullivam, B., and V. Liu, *Web Application Security, A Beginner's Guide,* McGraw-Hill, 2011.

[Sun15] Sun X., et al., "What Information in Software Historical Repositories Do We Need to Support Software Maintenance Tasks? An Approach Based on Topic Model," in *Computer and Information Science,* Studies in Computational Intelligence, R. Lee (ed.), vol. 566, Springer, 2015.

[SWE14] *Software Engineering Body of Knowledge,* version 3, 2014, available at https://www.computer.org/web/swebok (accessed December 9, 2018).

[Tai12] Taivalsaari, A., and K. Systa, "Mobile Content as a Service: A Blueprint for a Vendor-Neutral Cloud of Mobile Devices," *IEEE Software,* vol. 29, no. 4, July–August 2012, pp. 28–33.

[Tai89] Tai, K., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes,* vol. 14, no. 2, April 1989, pp. 58–61.

[Tan01] Tandler, P., "Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing," *Proceedings of UbiComp 2001: Ubiquitous Computing,* 2001.

[Tao17] Tao, H., and J. Gao, "On Building a Cloud-Based Mobile Testing Infrastructure Service System," *Journal of Systems and Software,* vol. 124, 2017, pp. 39–55.

[Tay09] Taylor, R., N. Medvidovic, and E. Dashofy, *Software Architecture,* Wiley, 2009.

[Tho04] Thomas, J., et al., *Java Testing Patterns,* Wiley, 2004.

[Tho92] Thomsett, R., "The Indiana Jones School of Risk Management," *American Programmer,* vol. 5, no. 7, September 1992, pp. 10–18.

[Tic18] *TickIT plus,* 2018, available at http://www.tickitplus.org/.

[Tid11] Tidwell, J., *Designing Interfaces: Patterns for Effective Interaction Design,* 2nd ed., O'Reilly, 2011.

[Til00] Tillman, H., "Evaluating Quality on the Net," Babson College, May 30, 2000, available at http://www.dronet.org/lineeguida/ligu_pdf/evelqual.pdf.

[Toc18] Tonchia, S., "Project Time Management," in *Industrial Project Management. Management for Professionals,* Springer, 2018.

[Tog01] Tognozzi, B., "First Principles," askTOG, 2001, available at www.asktog.com/basics/firstPrinciples.html.

[Tos17] Tosun, A., A. Bener, and S. Akbarinasaji, "A Systematic Literature Review on the Applications of Bayesian Networks to Predict Software Quality," *Software Quality Journal,* vol. 25, no. 1, March 2017, pp. 273–305.

[Tri03] Trivedi, R., *Professional Web Services Security,* Wrox Press, 2003.

[Tyr05] Tyree, J., and A. Akerman, "Architectural Decisions: Demystifying Architecture," *IEEE Software,* vol. 22, no. 2, March–April 2005.

[Uni03] Unicode, Inc., *The Unicode Home Page,* 2003, available at www.unicode.org/.

[USA87] U.S. Air Force, "Management Quality Insight," AFCSP 800-14, January 20, 1987.

[Ute12] UTest, E-book: *Essential Guide to Mobile App Testing,* 2012, available at http://go.applause.com/rs/539-CKP-074/images/The-Essential-Guide-to-Mobile-App-Testing.pdf.

[Vac06] Vacca, J., *Practical Internet Security,* Springer, 2006.

[Vak18] Vakkuri, V., and P. Abrahamsson, "The Key Concepts of Ethics of Artificial Intelligence," *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC),* Stuttgart, 2018, pp. 1–6, doi:10.1109/ICE.2018.8436265.

[Van16] VanderPlas, J., *Python Data Science Handbook Essential Tools for Working with Data,* O'Reilly Media, 2016.

[Vel16] Veloso, A., and L. Costa, "Heuristics for Designing Digital Games in Assistive Environments: Applying the Guidelines to an Ageing Society," *2016 1st International Conference on Technology and Innovation in Sports, Health and Wellbeing (TISHW),* Vila Real, 2016, pp. 1–8.

[Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer," *Artima Developer,* December 8, 2003, available at www.artima.com/intv/contracts.html.

[Vit03] Vitharana, P., "Risks and Challenges of Component-Based Software Development," *CACM,* vol. 46, no. 8, August 2003, pp. 67–72.

[Vit17] Vitharana, P., "Defect Propagation at the Project-Level: Results and a Post-Hoc Analysis on Inspection Efficiency," *Empirical Software Engineering,* vol. 22, no. 1, February 2017, pp. 57–79.

[Voa12] Voas, J., et al., "Mobile Software App Takeover," *IEEE Software,* vol. 29, no. 4, July–August 2012, pp. 25–27.

[Voe14] Voehl, F., H. Harrington, C. Mignosa, and R. Charron, *The Lean Six Sigma Black Belt Handbook,* Productivity Press, 2014, https://doi.org/10.1201/b15163.

[W3C18] *W3C Web Content Accessibility Guidelines (WCAG 2.1),* 2018, available at https://www.w3.org/TR/WCAG21/.

[Wal12] Walker, J., "Computer Programmers Learn Tough Lesson in Sharing," *The Wall Street Journal,* vol. 260, no. 48, August 27, 2012, p. 1.

[War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices," TMNNet, 2007, available at www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.

[Was10] Wasserman, A., "Software Engineering Issues for Mobile Application Development," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research,* 2010.

[Web05] Weber, S., *The Success of Open Source,* Harvard University Press, 2005.

[Web13] Web Application Security Consortium, 2013, available at http://www.webappsec.org/.

[Wee11] Weevers, I., "Seven Guidelines for Designing High Performance Mobile User Experiences," *Smashing Magazine,* July 18, 2011, available at http://uxdesign.smashingmagazine.com/2011/07/18/seven-guidelines-for-designing-high-performance-mobile-user-experiences/.

[Wel01] vanWelie, M., "Interaction Design Patterns," 2001. A related article can be found at https://www.interaction-design.org/literature/article/10-great-sites-for-ui-design-patterns.

[Whi08] White, J., "Start Your Engines: Mobile Application Development," April 22, 2008, available at http://www.devx.com/SpecialReports/Article/37693.

[Whi12] Whittaker, J., et al., *How Google Tests Software,* Addison-Wesley, 2012.

[Whi15] Whigham, P. A., C. A. Owen, and S. G. MacDonell, "A Baseline Model for Software Effort Estimation," *ACM Transactions on Software Engineering and Methodology,* vol. 24, no. 3, 2015, pp. 1–11.

[Whi97] Whitmire, S., *Object-Oriented Design Measurement,* Wiley, 1997.

[Wie02] Wiegers, K., *Peer Reviews in Software,* Addison-Wesley, 2002.

[Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software," *Computerworld,* March 21, 2005, available at https://www.computerworld.com/article/2563708/q-a--quality-software-means-more-secure-software.html.

[Wil97] Williams, R., J. Walker, and A. Dorofee, "Putting Risk Management into Practice," *IEEE Software,* May 1997, pp. 75–81.

[Wir71] Wirth, N., "Program Development by Stepwise Refinement," *CACM,* vol. 14, no. 4, 1971, pp. 221–227.

[Wir90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software,* Prentice Hall, 1990.

[WMT02] *Web Mapping Testbed Tutorial,* 2002. A related presentation can be found at http://proceedings.esri.com/library/userconf/devsummit17/papers/dev_int_114.pdf.

[Woo04] Woody, C., *Eliciting and Analyzing Quality Requirements: Management Influences on Software Quality Requirements* (CMU/SEI-2005-TN-010, ADA441310). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004, available at http://www.sei.cmu.edu/publications/documents/05.reports/05tn010.html.

[Woo14] Woody, C., R. Ellison, and W. Nichols, *Predicting Software Assurance Using Quality and Reliability Measures,* CMU/SEI-2014-TN-026, Software Engineering Institute, Carnegie Mellon University, 2014, http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=428589.

[Woo89] Wood, J., and D. Silver, *Joint Application Design: How to Design Quality Systems in 40% Less Time,* John Wiley & Sons, 1989.

[Wri11] Wright, A., "Lessons Learned: Architects Are Facilitators, Too!" *IEEE Software,* vol. 28, no. 2, January–February 2011, pp. 70–72.

[Xia16] Xiao, L., et al., "Identifying and Quantifying Architectural Debt," *Proceedings of 38th ACM International Conference on Software Engineering,* May 2016, pp. 488–498.

[Xie18] Xie, T., "Intelligent Software Engineering: Synergy between AI and Software Engineering," *Proceedings of the 11th Innovations in Software Engineering Conference (ISEC '18).* ACM, New York, NY, Article 1, 2018.

[Yad17] Yadav, H., and D. Yadav, "Early Software Reliability Analysis Using Reliability Relevant Software Metrics," *International Journal of System Assurance Engineering and Management*, vol. 8, suppl., December 2017, pp. 2097–2108.

[Yau11] Yau, S., and H. An, "Software Engineering Meets Services and Cloud Computing," *IEEE Computer,* vol. 44, no. 10, October 2011, pp. 47–53.

[Yoo13] Yoo, S., and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Journal of Software: Testing, Verification, and Reliability,* vol. 22, no. 2, 2013, pp. 67–120.

[You01] Young, R., *Effective Requirements Practices,* Addison-Wesley, 2001.

[You18] Young, S., T. Abdou, and A. Bener, "A Replication Study: Just-in-Time Defect Prediction with Ensemble Learning," *Proceedings of the ACM/IEEE Sixth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering,* ACM, 2018, pp. 42–47.

[You75] Yourdon, E., *Techniques of Program Structure and Design,* Prentice Hall, 1975.

[Zah90] Zahniser, R., "Building Software in Groups," *American Programmer,* vol. 3, no. 7–8, July–August 1990.

[Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development,* March 1994, pp. 35–38.

[Zan15] Zanoni, M., Fontana, F., and Stella, F. "On Applying Machine Learning Techniques for Pattern Detection," *Journal of Systems and Software,* vol. 103, May 2015, pp. 102–117.

[Zan18] Zancan, B. A. G., et al., "Accessibility Guidelines for Virtual Environments," in M. Antona and C. Stephanidis (eds.), *Universal Access in Human-Computer Interaction. Virtual, Augmented, and Intelligent Environments,* UAHCI 2018. Lecture Notes in Computer Science, vol. 10908, 2018.

[Zha13] Zhang, D., et al., "Software Analytics in Practice," *IEEE Software,* September–October 2013, vol. 30, no. 5, pp. 30–37.

[Zim11] Zimmermann, O., "Architectural Decisions as Reusable Design Assets," *IEEE Software,* vol. 28, no. 1, January–February 2011, pp. 64–69.

[Zus90] Zuse, H., *Software Complexity: Measures and Methods,* DeGruyter, 1990.

[Zus97] Zuse, H., *A Framework of Software Measurement,* DeGruyter, 1997.

INDEX

Abstraction, 87, 163, 168
Abstraction dimension, 171–172
Abuse cases, 363–364
Acceptance testing, 48, 68–69, 95, 430
Access control, 450
Accessibility, 233, 237, 259–261, 432–433
Action, defined, 10
Active state, 150
Activity, defined, 9–10
Activity diagrams, 146–148, 151–154, 223, 622–625
Activity networks, 525–526
Activity state, 627
Actors, 114–115, 131
Adapters, 230
Adaptive maintenance, 70, 553
Aesthetic design, 237, 277
Aesthetic metrics, 472
Aggregation, 614–615
Agile Alliance, 40–41
Agile manifesto, 37, 38, 42
Agile spirit, 41
Agile teams, 40, 41, 78, 495
Agility
  application of, 38–39
  in change management, 453–458
  characteristics of, 38, 40
  comparison of techniques for, 52
  cost of change and, 39–40
  in design, 60, 174, 185–186
  development of, 38, 41
  DevOps approach to, 50–51
  estimation and, 519
  Kanban method for, 48–50, 56
  principles of, 40–41
  process models for, 42–51, 56, 57
  in requirements engineering, 58, 104
  in reviews, 336
  Scrum framework for, 42–45, 56
  testing and, 95
  XP framework for, 46–48
Algorithms, 62–63, 352, 606
Alpha tests, 430
Ambient intelligence, 590
Analysis
  attack surface, 367
  boundary value, 389–390
  data, 633
  errors/defects, 342
  gap, 573
  hazards, 545
  impact, 443
  inventory, 563–564
  source code, 561
  static, 368
  task, 243, 247–248
  threats, 363
  user interface, 243–249
  work environment, 248–249
Analysis classes
  attributes for, 140
  defined, 104–105
  identifying, 137–140
  manifestations of, 138
  selection characteristics for, 139–140
  state diagrams for, 150–151
Analysis models and modeling. *see also* Requirements models
  building, 118–122
  domains of, 93
  elements of, 119–121
  patterns in, 122
  principles of, 129–130
  purpose of, 118
  rules of thumb for, 128–129
  terminology for, 126
Analysis patterns, 122
Analytics, 462–463, 509–511, 558–559
Anchor point milestones, 29
Antibugging, 382
Anti-patterns, 302–304
Anti-requirements, 383
Application domains, 7
Application objects, 251
Application software, 7
Appraisal costs, 317
Archetypes, 196–198
Architectural context diagram (ACD), 196
Architectural decision record (ADR), 184
Architectural decisions, 184–185, 194–196
Architectural description (AD), 183–185
Architectural description language (ADL), 164
Architectural design
  agility and, 60, 185–186
  alternatives in, 201–204
  archetypes in, 196–198
  economy of, 193
  elements of, 175
  emergence in, 194
  functions of, 158, 182
  instantiations of system in, 200
  metrics for, 466–467
  organization and refinement of, 193
  patterns, 187, 192–193, 203–204, 291, 299–300
  preliminary, 59–60
  properties of, 164
  refining into components, 198–200
  representing system in context, 196–197
  spacing in, 194
  styles of, 186–193
  symmetry in, 194
  visibility in, 194
  of WebApps, 278–280
Architectural mismatch, 229–230
Architectural patterns, 187, 192–193, 203–204, 291, 299–300
Architectural reviews, 202–204
Architecture
  call-and-return, 189
  conformance checking, 204
  content, 279
  data-centered, 187–188
  data-flow, 188
  defined, 163–164, 173, 182
  erosion of, 204
  functional, 226
  importance of, 183
  information, 235, 279
  layered, 189–192
  mobile, 273–274
  MVC, 189, 191–192, 279–280
  object-oriented, 189
  refactoring, 561–562
  responsibility-driven, 186
  testing, 375, 376
Architecture trade-off analysis method (ATAM), 201–202
Artificial intelligence (AI)
  applications involving, 7
  for decision making, 229
  integration-level testing and, 403
  machine learning and, 294–295, 322
  to model reliability, 351–352
  software engineering and, 589, 606
  software support and, 558
  testing, 428–429
Assessment. *see also* Evaluation; Risk assessment
  architectural, 202
  of quality, 314–315
  in software improvement process, 572–573
  of software process, 24–25
Assessment effort, 328
Associations, 144, 154, 613–615

Mc Graw Hill
501-1111

Atomic modules, 399
Attack patterns, 363–364
Attack surface, 366–367
Attributes
  defining, 140–141
  of metrics, 462
  in software quality assurance, 346
Audits, 342, 452, 458, 579
Audit trails, 445
Authoring, 456
Automated testing tools, 50, 413, 415, 422, 430
Automation phase of technology innovation, 585
Availability, 351

Backlog, 43–45
Backward impact management, 451
Baselines, 441, 442, 450, 481
Basis path testing, 384–386
Bayesian inference, 351–352
Behavioral elements, 120–121
Behavioral modeling
  activity diagrams in, 151–154
  characteristics of, 127
  identifying events with use case in, 149–150
  state diagrams in, 150–151
  steps for creation of, 149
Behavioral patterns, 292–293
Behavioral testing, 388–390, 392–393
Beta tests, 430
Big bang approach to integration testing, 398
Binary classification problems, 634
Black box metrics, 466
Black-box testing, 388–390, 397
Blueprint metaphor, 183
Bottom-up integration, 399–400
Boundary testing, 381–382
Boundary value analysis (BVA), 389–390
Breadth-first integration, 398
Breakthrough phase of technology innovation, 585
Buffer overflow, 367
Bugs, 326. *see also* Defects
Building blocks, 591–592
Building Security in Maturity Model (BSIMM), 370
Business activity monitoring, 123
Business goals, 485
Business risks, 534

Call-and-return architectures, 189
Capability Maturity Model Integration (CMMI), 370, 576–579
Capture/playback tools, 403
Casual meetings, 331
Casual reviews, 326, 331
Categorical variables, 633
Certification testing, 414
Change, sources of, 439
Change control, 441, 446, 448–450, 453–455
Change control authority (CCA), 448, 450
Change costs, 39–40
Change descriptions, 455
Change management
  agile, 453–458
  process for, 447–452
  in SCM, 445
  in SQA, 342
Change reports, 448
Change requests, 448
Change sets, 446
Changes to legacy systems, 8
Checklists
  mobile product quality, 285
  for reviews, 331
  risk item, 536
  validation requirements, 106
Chunking, 227–228
CK metrics suite, 469–471
Class-based design metrics, 469–470
Class-based elements, 120
Class-based modeling
  characteristics of, 127
  defining attributes and operations in, 140–141
  identifying analysis classes in, 137–140
  UML, 141–144
Class coupling, 218
Class diagrams, 120, 141, 143, 612–615
Classes. *see also* Analysis classes
  attributes, 140
  collaborating, 207
  defined, 120
  dependent, 404
  design, 169–171
  equivalence, 389
  hierarchies, 469, 470
  independent, 404
  operations, 141
  server, 404
Classification problems, 634
Class-responsibility-collaborator (CRC) model, 47, 144–146
Class testing, 390–391
Cloud-based testing, 428
Cloud computing, 7, 273–274
Clustering technique, 637
Cluster testing, 404
CMMI-DEV, 24
Code quality, 345
Code refactoring, 561, 564
Coding activity, 48, 95, 96, 367–368
Cognitive walkthroughs, 246, 253
Cohesion, 167, 170, 216–218, 468
Collaboration
  benefits of, 89
  in development, 595–596
  in requirements gathering, 110–113
  among stakeholders, 108
  teams and, 587
Collaboration diagrams, 220, 406, 621
Collaborators, in CRC modeling, 144
Collection subsystem, 455–456
Command labeling, 260
Common closure principle (CCP), 215
Common reuse principle (CRP), 215
Communication
  effectiveness of modes of, 89
  by interface, 257
  principles of, 88–90
  in process framework, 10
  in prototyping paradigm, 27
  task set for, 23, 24, 500
  in teams, 496, 604
Communicational cohesion, 216
Communication diagrams, 189, 190, 621–622
Compartmentalization, 521, 604
Compatibility testing, 414
Completeness, 468
Complexity, 468, 588–589
Component-based software engineering (CBSE), 228–230, 509
Component diagrams, 177
Component elements of SCM, 440
Component-level design
  cohesion in, 216–218
  coupling in, 218–219
  elements of, 176–177
  example of, 210–211
  functions of, 158, 206
  guidelines for, 215–216
  for MobileApps, 226–227
  patterns in, 300–301
  principles of, 173, 212–215
  specialized, 225–230
  steps for, 219–225
  for traditional components, 227–228
  for WebApps, 226, 282
Component-level testing
  black-box, 388–390
  elements of, 372
  object-oriented, 390–393
  planning and recordkeeping in, 378–380
  strategic approach to, 373–378
  test case design in, 381–383
  white-box, 383–387
Components
  class-based, 212–219
  defined, 207
  elaboration of, 207–209
  naming, 215–216
  object-oriented view of, 207–209
  process-related view of, 211–212
  refactoring, 230–231
  reuse of, 214–215, 228, 229
  traditional view of, 209–211
Composition, 615
Computational intelligence, 638

Computer-aided software engineering, 9
Concept development projects, 524–526
Concerns, 165. *see also* Separation of concerns
Condition testing, 386
Condition-transition-consequence (CTC) format, 542
Configuration audits, 452
Configuration management, 11, 437, 445. *see also* Software configuration management
Configuration objects, 441–443, 445–446, 450, 452, 457–458
Configuration reviews, 408
Configuration status reporting, 452
Connectionism, 636–637
Connectivity and connectivity testing, 414, 587
Consistency, 87, 240–241, 257
Construction
  of interface, 243
  as phase of Unified Process, 33
  principles of, 95–98
  in process framework, 10
Construction elements of SCM, 440
Constructive cost model (COCOMO), 511
Content architecture, 279
Content coupling, 218
Content design, 277–278
Content management, 455, 456
Content metrics, 473
Content objects, 277–278
Content repository, 444
Content testing, 420–421
Context-aware apps, 274–275
Context-free questions, 108
Context variables, 427
Contingency planning, 86, 92, 544
Continuous integration (CI), 400–402, 446–447
Continuous process improvement, 341
Continuous variables, 633
Contracted software, 342
Control coupling, 218
Control structure testing, 386–387
Convergence, 157
Coordination of teams, 496
Corrective maintenance, 69–70, 553
Cost of change, 39–40
Cost of quality, 317–318
Coupling, 167, 170, 174, 218–219, 468
Coupling between object classes (CBO), 469–470
Creational patterns, 292
Critical path, 520, 526
Critical practices for project management, 502
Crowdsourcing, 423
Customer journey maps, 244–245
Customers, defined, 88
Customer satisfaction, 40–41
Customer testing. *see* Acceptance testing
Cyclomatic complexity, 385–386

Data abstraction, 163
Data analysis, 633
Data analytics, 509–511
Data-centered architectures, 187–188
Data cleaning, 633
Data collection, 633
Data complexity, 467
Data design, 173, 174
Data-flow architectures, 188
Data flow diagrams (DFDs), 365, 366
Data flow testing, 381, 386
Data models, 127
Data munging, 630, 633
Data name rationalization, 561
Data patterns, 291
Data processing, 605
Data record standardization, 561
Data redesign, 561
Data refactoring, 561, 564–565
Data science
  analysis of data in, 633
  characteristics of, 461, 629
  cleaning data in, 633
  clustering technique in, 637
  collecting data in, 633
  computational intelligence and, 638
  decision trees in, 634–635
  dimensional reduction in, 637
  language selection for, 629–631
  libraries and tools for, 631
  machine learning and, 631–637
  nearest neighbor technique in, 635–637
  neural networks in, 636–637
  search-based software engineering and, 638
  training set fabrication in, 633
  transformation of data in, 633
Data transformation, 633
Debugging, 123
Decision metaphor, 184
Decision trees, 634–635
Decision view, 195
Decomposition, 497–500, 511–519
Defect removal efficiency (DRE), 482–484, 486
Defects
  amplification of, 327
  collection and analysis of, 342
  cost impact of, 326–327
  defined, 478
  prediction of, 322
  propagation of, 327
  tracking, 446
Deficiency lists, 408
Degree of rigor, 524
Degree of structural uncertainty, 506
Dependencies, 154, 216, 442–443, 445, 614
Dependency inversion principle (DIP), 214
Dependent classes, 404
Deployment
  in DevOps approach, 50
  in mobile development life cycle, 269
  principles of, 98–100
  in process framework, 11
  prototyping and, 63
Deployment diagrams, 177, 178, 225, 615–616
Deployment-level design, 177–178
Depth-first integration, 398
Depth of the inheritance tree (DIT), 469, 475
Descriptor form of deployment diagrams, 178
Design
  agility in, 60, 174, 185–186
  architectural (*see* Architectural design)
  component-level (*see* Component-level design)
  concepts of (*see* Design concepts)
  content, 277–278
  in context of software engineering, 157–159
  convergence in, 157
  of data, 173, 174
  defined, 181
  deployment-level, 177–178
  diversification in, 157
  dominant, 184
  evaluation of, 160, 161, 253–256
  evolution of, 161–162
  importance of, 3, 159
  interface (*see* Interface design)
  level, 456
  metrics for, 466–473, 475
  mobile (*see* MobileApps)
  in mobile development life cycle, 268
  models (*see* Design models)
  navigation, 280–282
  object-oriented approach to, 161, 173
  pattern-based (*see* Pattern-based design)
  practice, 156
  principles, 156
  process for, 159–162
  quality of, 159–161, 282–285, 311, 345
  refactoring, 48, 168, 225
  refinement of, 167–168
  task set for, 162
  test case, 381–383, 405–407
  user interaction, 236
  visual, 237, 277
  in XP framework, 47–48
Design classes, 169–171

Design concepts
abstraction, 163, 168
architecture, 163–164
classes, 169–171
functional independence, 167
importance of, 156
information hiding, 166
modularity, 165–166
patterns, 164–165
refactoring, 48, 168, 225
separation of concerns, 165
stepwise refinement, 167–168
Design models
architectural, 175
characteristics represented by, 93
component-level, 176–177
for data, 174
deployment-level, 177–178
dimensions of, 171–172
for interfaces, 175–176
principles for, 173–174
translation from requirements models, 158
Design patterns. *see* Pattern-based design
Design recovery, 564
Design structure quality index (DSQI), 468
Desk checks, 331–332
Determinate software, 7
Developer notes, 195
Development teams, 43–45, 67
Device compatibility testing, 414
DevOps approach, 50–51
Diagrams
activity, 146–148, 151–154, 223, 622–625
architectural context, 196
class, 120, 141, 143, 612–615
collaboration, 220, 406, 621
communication, 189, 190, 621–622
component, 177
data flow, 365, 366
deployment, 177, 178, 225, 615–616
sequence, 148–149, 618–621
state, 120–121, 150–151, 392, 625–628
swimlane, 151, 153–154
use case, 118, 119, 136, 137, 616–618
Dimensional reduction, 637
Direct measures, 479–480
Distributed debugging, 123
Diversification, 157
Do-activity, 627
Documentation testing, 434
Document restructuring, 564
Domain-specific modeling languages (DSMLs), 597
Dominant design, 184
Drivers, 379, 399
Dynamic models, 164
Dynamic testing, 429

Education and training, 342, 573
Efficiency, 257
Effort validation, 521
Egoless programming, 64–65
Elaboration
in communication activity, 23
of components, 207–209
as phase of Unified Process, 33
problem, 497–498
refinement as process of, 167–168, 248
in requirements engineering, 104–105
Elicitation. *see also* Requirements gathering
agile, 104
in communication activity, 23, 24
in requirements engineering, 104, 109
of security needs, 362
work products produced during, 114
Embedded software, 7
Empirical estimation models, 510–511
Empiricism phase of technology innovation, 585
Encapsulation, 475
End users, defined, 88
Engineering. *see also* Software engineering
algorithms, 62–63
feature, 634
forward, 565
reengineering, 554, 562–565
reverse, 553–557, 564
security, 357, 360–363
Engineering change order (ECO), 448, 450, 452
Engineering software, 7
Ensemble learning environment, 635
Environmental resources, 509
Equivalence classes, 389
Equivalence partitioning, 389
Errors
collection and analysis of, 342
cost of, 317–318
defined, 326, 478
density of, 328, 329
handling systems, 260
Estimation. *see also* Project planning
for agile development, 519
complexity of projects and, 505–506
data analytics and, 509–511
decomposition techniques, 511–519
empirical models for, 510–511
FP-based, 514–515
LOC-based, 512–513
objectives of, 92
problem-based, 512
process-based, 515–516
quality and, 321
reconciling estimates, 518–519
of resources, 60–61, 505
of risk, 538–541
size of projects and, 506
software sizing, 511
use case, 517–518
Ethics, 607–609
Ethnographic observation, 272
Evaluation. *see also* Assessment
of design, 160, 161, 253–256
of interface design, 253–256
post-mortem, 336
in prototype development, 64–65, 68–69, 253–254
in software process improvement, 575
Evolutionary process flow, 22, 23
Evolutionary process models, 29–31, 56–57, 67
Exceptions, 134–135
Exhaustive testing, 380
Exposure to risk, 540–541
Extensible Markup Language (XML), 273
External coupling, 218
External failure costs, 317
Extra-functional properties, 164
Extreme Programming (XP), 46–48, 67

Failure costs, 317
Failure curves, 5–6
Failures in time (FIT), 351
Families of related systems, 164
Fan-in (FIN), 475
Fault-based testing, 405–406
Faults, 326. *see also* Defects
Feature engineering, 634
Feature vectors, 633
Filters, 188
Financial leverage, 571
Fire-fighting mode, 533
First law of system engineering, 438–439
Flexibility, 258
Flow graphs, 384
Flow-oriented models, 127
Formal change control, 450
Formal technical reviews (FTRs), 327, 332–335
Formal use cases, 127, 135
Forward engineering, 565
Forward impact management, 451
FP-based estimation, 514–515
Frameworks
models, 164
pattern-based design, 293
risk management, 364–365
Scrum, 42–45, 56, 66–67
software process, 10–11, 21, 23
software process improvement, 569–570
Functional architecture, 226
Functional cohesion, 216
Functional independence, 167, 173
Functional modeling, 146–149, 164
Functional testing, 97, 388–390, 397
Function-oriented metrics, 481
Function point (FP), 481, 514–515
Fuzzy logic, 539

Gamification, 544–545
Gantt charts, 527
Gap analysis, 573
Generalizations, 613
Generative patterns, 291
Generic process models, 21–23
Generic risks, 535
Genetic algorithms, 352, 606
Gesture testing, 415–416
Glass-box testing, 383–387, 397
Globalization, 587–588
Global software development (GSD) teams, 80–81
Go, no-go decisions, 65–67
Goal in context, 135
Goals
  business, 485
  defined, 104
  of interface design, 243
  of requirements gathering, 24
  security, 362
  of software quality assurance, 345–346
"Good enough" software, 316
Grammatical parse, 137, 138, 141
Granularity, 92
Graphic design, 237, 277
Graphic design metrics, 472
Graphic icons, 276
Graphic images, 276

Handshaking, 122
Hardware, failure curve for, 5–6
Hazards, 352–353, 545
Help facilities, 260, 434
Heuristic evaluations, 253
Heuristics, 7, 157, 162, 554
High-granularity plans, 92
High-order testing, 377
Historical data, 481
Hooks, 293
Human elements of SCM, 440
Human-focused path, 599
Human interface objects, 258
Human resources, 508
Hype cycle for emerging technologies, 586–587

Identification
  of analysis classes, 137–140
  of events with use cases, 149–150
  of risk, 535–537
  of stakeholders, 107
Idioms, 292
Impact analysis, 443
Impact management, 443, 451
Impact of risk, 540–541
Implementation model, 242
Inception
  in communication activity, 23
  in mobile development life cycle, 268
  as phase of Unified Process, 32
  in requirements engineering, 104
Incremental development strategies, 40
Incremental process models, 55–56
Increments. *see* Software increments
Independent classes, 404
Independent paths, 384–385
Independent test groups (ITGs), 375
Indeterminate software, 7
Indicators, defined, 462
Informal change control, 450
Information architecture, 235, 279
Information delivery, 4
Information domain, 129
Information hiding, 166
Information objectives, 497
Information technology, 605
Inheritance, 216, 475
Inspections, 326, 332
Installation, 574–575
Instance form of deployment diagrams, 178
Integrability, 187
Integration
  bottom-up, 399–400
  breadth-first, 398
  continuous, 400–402, 446–447
  depth-first, 398
  in DevOps approach, 50
  top-down, 398–399
Integration-level testing
  of architecture, 375, 376
  artificial intelligence and, 403
  black-box, 397
  bottom-up, 399–400
  challenges of, 395
  in construction activity, 95
  fundamentals of, 396–397
  objectives of, 398
  object-oriented, 404–407
  patterns, 409
  regression, 403–404
  top-down, 398–399
  validation, 407–408
  white-box, 397
  work products, 402
Integrity, 483
Intelligent software engineering, 606
Interaction design, 236
Interaction frames, 620
Interaction mechanisms, 234
Interdependency, 521
Interface analysis, 243
Interface construction, 243
Interface design. *see also* User experience design
  elements of, 175–176
  evaluation of, 253–256
  functions of, 158
  goals of, 243
  golden rules of, 234, 238–241
  guidelines for, 216, 259
  metrics for, 471–473
  for MobileApps, 261, 270–271
  models for, 241–242, 270–271
  patterns in, 252–253, 291, 304–305
  for placing user in control, 238–239
  principles of, 173
  process for, 242–243
  for reducing user's memory load, 239–240
  steps for, 251–252
  for WebApps, 275–276
Interface segregation principle (ISP), 214
Interface testing, 388, 421
Interface validation, 243
Internal failure costs, 317
Internationalization, 260–261, 423
Inventory analysis, 563–564
ISO 9001:2015, 353–354
Issues lists, 112
Issues tracking, 69, 446
Iterative planning, 91, 92
Iterative process flow, 22, 23

Jelled teams, 76–77, 494
Justification activity, 573–574

Kanban method, 48–50, 56, 67
Key performance indicators (KPIs), 462
Knowledge discovery, 605–606
Known risks, 535

Lack of cohesion in methods (LCOM), 470, 475
Language metaphor, 183
Languages
  architectural description, 164
  for data science, 629–631
  domain-specific modeling, 597
  pattern, 297
  UML (*see* Unified modeling language)
Latency reduction, 258
Law of Demeter, 170
Layer cohesion, 216, 217
Layered architectures, 189–192
Layered behavioral model, 75–76
Layout, 277, 431
Leadership, 493–494
Learnability, 258
Legacy software, 8
Level design, 456
Liability, 320
Libraries, 631
Life-cycle security models, 357–359
Linear process flow, 22, 23
Linear sequential model. *see* Waterfall model
Liskov substitution principle (LSP), 214
Listening, 46, 89
Literature metaphor, 184
Load testing, 425–426
Localization, 260, 423
LOC-based estimation, 512–513
LOC-based metrics, 481

Loop testing, 387
Low-granularity plans, 92

Machine learning, 294–295, 322, 558, 606, 631–637
Main program/subprogram architectures, 189
Maintainability, 483, 553
Maintenance
  adaptive, 70, 553
  challenges of, 552
  corrective, 69–70, 553
  defined, 69
  importance of, 3, 71
  metrics for, 476
  perfective, 70, 553
  practice principles and, 88
  preventive, 70, 553
  tasks related to, 554–555
Make facilities, 446
Management. *see also* Project management; Risk management; Software configuration management
  change, 342, 445, 447–452
  of complexity, 588–589
  configuration, 11, 437, 445
  content, 455, 456
  impact, 443, 451
  quality, 9, 340
  release, 550–551
  in requirements engineering, 106
  reusability, 11
  security, 342
  vendor, 342
  version, 446
Management subsystem, 456–457
Man-in-the-middle attacks, 267
Manufacturer view of quality, 311
Maturity levels, 577, 578
Maturity models, 570–571
Maturity phase of technology innovation, 585
Mean time between failure (MTBF), 350–351
Mean time to change (MTTC), 483
Measurement. *see also* Metrics
  of availability, 351
  defined, 461–462
  direct, 479–480
  as management tool, 461
  of productivity, 480, 482
  of reliability, 350–351
  of security, 368–369
  as umbrella activity, 11
  usefulness of, 460, 461
Measures, defined, 461
Meetings
  casual, 331
  Kanban, 50
  question-and-answer format for, 109
  requirements gathering, 112
  review, 332–333
  Scrum, 44–45, 56, 66–67
  sprint, 44, 45
Melding product and process, 498, 499
Mental model, 241–242
Menu labeling, 260
Meta-model, 443–444
Metaphors, 258, 276
Meta-questions, 109
Methods, 9
Metrics
  aesthetic, 472
  attributes of, 462
  black-box, 466
  class-based, 469–470
  content, 473
  for conventional software, 465–468
  defined, 462
  design, 466–473, 475
  function-oriented, 481
  for interface design, 471–473
  key performance indicators, 462
  LOC-based, 481
  for maintenance, 476
  for MobileApps, 465–466
  morphology, 467, 468
  navigation, 473
  for object-oriented software, 468–470
  private, 480, 482
  process, 476–478
  product, 461, 463–464, 469, 480, 482
  productivity, 481
  program establishment, 485–487
  project, 476–478
  public, 480
  quality, 482–484
  for requirements model, 464–466
  for reviews, 327–330
  size-oriented, 480–481
  for small organizations, 486
  for software quality assurance, 346
  for source code, 473–474
  for testing, 474–475
  usefulness of, 461
  for WebApps, 472, 473
Middleware, 266
Migration, 574–575
Milestones, 521
Mini-specifications, 112
Misuse cases, 363–364
MobileApps (mobile applications)
  agile change management for, 453–458
  alerts and extraordinary conditions, 417
  architecture for, 273–274
  best practices for design, 285–287
  challenges related to, 265–268
  component-level design for, 226–227
  context-aware, 274–275
  development considerations, 265
  gestures and, 415–416
  guidelines for design, 271–272
  interface design for, 261, 270–271
  internationalization of, 423
  metrics for, 465–466
  mistakes in design of, 272
  patterns for, 292, 305–306
  performance of, 424–426
  prevalence of, 264
  quality of, 282–285
  real-time applications, 426–428
  security of, 424
  stages of development, 268–272
  technical considerations, 266–268
  testing, 413–417, 424–428
  tools for, 265, 267
  types of, 7
  usability of, 257, 258, 415–417
  virtual keyboard input and, 416
  voice input and recognition, 416–417
Model-based testing (MBT), 429–430
Model-driven software development, 596–597
Models and modeling. *see also* Analysis models and modeling; Requirements models and modeling
  behavioral, 127, 149–154
  class-based, 127, 137–146
  constructive cost, 511
  CRC, 47, 144–146
  data, 127
  design, 93, 158, 171–178
  dynamic, 164
  empirical estimation, 510–511
  flow-oriented, 127
  framework, 164
  functional, 146–149, 164
  for interface design, 241–242, 270–271
  maturity, 570–571
  principles of, 92–94
  process (*see* Process models)
  in process framework, 10
  product quality, 313–314
  quality in use, 313
  scenario-based, 127, 128, 130–137
  security, 357–359
  structural, 164
  system approach, 446
  threats, 357, 365–366
  user, 241, 245–246
Model-View-Controller (MVC) architecture, 189, 191–192, 279–280
Modularity, 87, 90, 165–166
Modules, 165, 209. *see also* Components
Morphology metrics, 467, 468
Multiclass classification problems, 634
Multilabel classification problems, 634
Multiplicity, 614

Navigation
  design of, 280–282
  menus for, 276
  syntax of, 280
  visible, 258

Navigational nodes (NN), 280
Navigation metrics, 473
Navigation semantic links (NSLs), 281
Navigation semantic units (NSUs), 280–282
Navigation testing, 421–423
Nearest neighbor technique, 635–637
Negative test cases, 383
Negligence, 320
Negotiation, 23, 90, 105, 122–123
Nested loops, 387
Neural networks, 636–637
90–90 rule, 500
Nomadic networks, 266
Nonfunctional requirements (NFRs), 109
Nongenerative patterns, 291
Notes, 615
Number of children (NOC), 469, 475
Number of root classes (NOR), 475

Object-oriented architectures, 189
Object-oriented design metrics, 468–470
Object-oriented testing, 390–393, 404–407
OO design metrics, 469–470, 475
Open-closed principle (OCP), 212–213
Open source movement, 592
Open-world software, 589–590
Operations, 141
Ordered Categorical variables, 633

Pair programming, 48, 332
Paper prototypes, 62, 272
Parallel process flow, 22, 23
Pareto principle, 97–98
Partitioning, 130, 389, 497–498
Passive state, 150
Pattern-based architecture review (PBAR), 203–204
Pattern-based design
  common mistakes in, 298–299
  in context, 290, 295–296
  defined, 164, 290
  describing, 293–294
  framework for, 293
  machine learning in, 294–295
  objectives of, 165
  organizing tables in, 298, 299
  repository for, 295
  tasks in, 297–298
  template for, 293–294
  thinking in, 296–297
  types of, 291–293
  use of, 87
  for user interfaces, 252–253
Pattern languages, 297
Pattern-organizing tables, 298, 299
Patterns. *see also* Pattern-based design
  analysis of, 122
  anti-patterns, 302–304
  architectural, 187, 192–193, 203–204, 291, 299–300
  attack, 363–364
  behavioral, 292–293
  component-level, 300–301
  creational, 292
  data, 291
  of defects, 98
  generative, 291
  interface design, 252–253, 291, 304–305
  mobile, 292
  for MobileApps, 292, 305–306
  nongenerative, 291
  pipe-and-filter, 188
  process, 24
  risk, 536
  structural, 292
  testing, 409
  for WebApps, 291–292
Peer reviews, 326
People Capability Maturity Model, 577
People Capability Maturity Model (People-CMM), 491
Percent public and protected (PAP), 475
Perfective maintenance, 70, 553
Performance testing, 414, 424–426
Pipe-and-filter pattern, 188
Planning. *see also* Project planning
  contingency, 86, 92, 544
  iterative, 91, 92
  principles of, 91–92
  in process framework, 10
  for software quality assurance, 343–344, 354
  for testing, 378–380
  in XP framework, 46–47
Platform model, 270
Playability testing, 433–434
Plug points, 293
Postconditions, 214
Postmortem evaluations (PMEs), 336
Practice (software engineering), 12–15, 85–88
Preconditions, 135, 214
Predictable risks, 535
Predictive technologies, 416
Preparation effort, 328
Prescriptive process models, 25–33
Presentation model, 270
Prevention costs, 317
Preventive maintenance, 70, 553
Primary actors, 115
Primary scenarios, 134
Primitiveness, 468
Priority points, 108
Private metrics, 480, 482
Proactive risk strategies, 534
Proactive software support, 557–560
Probabilistic reasoning, 558
Problem-based estimation, 512
Problem decomposition, 497–498
Problem elaboration, 497–498
Problem solving, 12–13
Procedural abstraction, 163
Process. *see* Software process
Process adaptation, 11–12
Process-based estimation, 515–516
Process dimension, 171–172
Process elements of SCM, 440
Process flow, 22–23, 25, 31
Process framework, 10–11, 21, 23
Process improvement. *see* Software process improvement
Processing narratives, 138, 141
Process maturity, 570
Process metrics, 476–478
Process migration, 574
Process models
  agile, 42–51, 56, 57
  comparison of, 33–34
  criticisms of, 38
  in design, 164
  evolutionary, 29–31, 56–57, 67
  generic, 21–23
  guidelines for use of, 55
  incremental, 55–56
  prescriptive, 25–33
  prototyping, 26–29
  reengineering, 563
  selection of, 28, 31
  spiral, 29–30, 56–57
  unified, 31–33
  waterfall, 25–26, 55
Process patterns, 24
Producers, 161, 333
Product, relationship to process, 34–35
Product backlog, 43–45
Production phase of Unified Process, 33
Productivity measures, 480, 482
Productivity metrics, 481
Product-line software, 7, 164, 590
Product metrics, 461, 463–464, 469, 480, 482
Product quality model, 313–314
Product requests, 110–111
Product scope, 491–492, 497
Product-specific risks, 535–536
Product view of quality, 311
Progressive sign-off, 186
Project complexity, 505–506
Project databases, 441, 442, 445–446
Project feasibility, 507
Projection, risk, 538–541
Project level change control, 450
Project management
  characteristics for success, 500–501
  critical practices for, 502
  people in, 491, 493–496
  process and, 492, 498–500
  product issues and, 491–492, 497–498
  for quality, 322
  spectrum of, 491–492
  W^5HH principle for, 501

Project metrics, 476–478
Project planning. *see also* Estimation
  data analytics in, 509–511
  objectives of, 506
  overview, 504
  in process framework, 10
  resources in, 507–509
  scheduling in, 520–523, 526–529
  scope and feasibility in, 507
  task networks in, 525–526
  task sets in, 507, 523–525
Project risks, 534, 536–537
Project size, 506
Project tables, 527–528
Project velocity, 47
Prototype development
  in agile process models, 56
  architectural design for, 59–60
  evaluation of, 64–65, 68–69, 253–254
  evolutionary model of, 67–68
  first prototype construction, 61–63
  go, no-go decisions in, 65–67
  incremental model for, 55–56
  for mobile devices, 272
  release stage of, 68–69
  scope determination in, 67
  spiral model for, 56–57
  stakeholders in, 58–59, 61–62
  in user experience design, 250
Prototype user interface, 62
Prototyping paradigm, 26–29
Public access to data members (PAD), 475
Public metrics, 480
Publishing subsystem, 457
Putnam-Norden-Rayleigh (PNR) curve, 522–523

Quality
  checklist for, 285
  of codes, 345
  of conformance, 311
  cost of, 317–318
  definitions of, 311, 312
  of design, 159–161, 282–285, 311, 345
  dilemmas related to, 315–321
  Garvin's dimensions of, 311
  of "good enough" software, 316
  guidelines for achievement of, 321–323
  importance of, 3
  of legacy software, 8
  management actions and impact on, 321
  McCall's factors for, 313
  metrics for, 482–484
  of MobileApps, 282–285
  planning for, 92
  qualitative assessment of, 314–315
  quantitative assessment of, 315
  of requirements model, 345
  reviews for, 341–342
  risk and, 319–320
  security and, 320
  standards for, 313–314, 341, 353–354
  of WebApps, 282–285
Quality assurance, 11, 323, 353. *see also* Software quality assurance
Quality control, 322–323, 346
Quality dilemma, 315–321
Quality in use model, 313
Quality management, 9, 340, 349. *see also* Software quality assurance
Quality requirements tree, 283
Question-and-answer sessions, 109

Reactive risk strategies, 533
Readability, 258, 431
Real-time testing, 426–428
Reconciliation of estimates, 518–519
Recorders, 161, 333
Recordkeeping, 333–334, 378–380
Recovery testing, 417
Reengineering, 554, 562–565
Refactoring
  architecture, 561–562
  code, 561, 564
  components, 230–231
  data, 561, 564–565
  defined, 48, 554, 560
  design, 48, 168, 225
  interfaces, 221
  tools for, 230, 561
Refinement
  of risk, 542–543
  stepwise, 167–168, 248
  of task sets, 524–525
Regression problems, 634
Regression testing, 68, 403–404
Release management, 550–551
Release reuse equivalency principle (REP), 214–215
Reliability, 350–353
Remote procedure call architectures, 189
Replicator phase of technology innovation, 585
Reporting, 333–334, 402, 448, 452, 458
Repositories, 295, 441, 443–446
Representation State Transfer (REST), 273
Requirements
  analysis of, 127–130
  architectural design and, 59–60
  collaboration on, 108
  emergent, 590–591
  monitoring, 123
  multiple viewpoints on, 107–108
  negotiating, 122–123
  nonfunctional, 109
  test cases for, 382–383
  tracing, 445
  understanding, 102–103
  validation of, 123–124

Requirements engineering
  agile, 58, 104
  defined, 57, 103
  first questions in, 108–109
  goal-oriented, 104
  for security, 360–363
  specifications in, 105, 407
  tasks in, 104–106
  for technology, 596
  traceability in, 109–110
Requirements gathering, 24, 110–114. *see also* Elicitation
Requirements models and modeling. *see also* Analysis models and modeling
  behavioral, 127, 149–154
  class-based, 127, 137–146
  domains of, 93
  functional, 146–149
  metrics for, 464–466
  objectives and philosophy of, 128
  principles of, 129–130
  quality of, 345
  rules of thumb for, 128–129
  scenario-based, 127, 128, 130–137
  terminology for, 126
  translation into design model, 158
  types of, 127
Requirements quality, 345
Requirements tasks
  elaboration, 104–105
  elicitation, 104, 109
  inception, 104
  management, 106
  negotiation, 105, 122–123
  specification, 105
  validation, 105–106, 123–124
Resources
  environmental, 509
  estimation of, 60–61, 505
  human, 508
  in project planning, 507–509
  reusable, 509
Response for a class (RFC), 470
Response time, 259
Responsibilities, in CRC modeling, 144
Responsibility-driven architecture (RDA), 186
Restructuring, 564
Return on investment (ROI), 580
Reusability management, 11
Reusable software resources, 509
Reuse, 214–215, 228, 229
Reverse engineering, 553–557, 564
Review effort, 328
Review issues list, 333–334
Review leaders, 161, 333
Review meetings, 332–333
Reviews. *see also* Technical reviews
  agile, 336
  architectural, 202–204
  casual, 326, 331

checklists for, 331
desk checks, 331–332
formal, 327, 332–335
guidelines for, 334–335
informal, 331–332
inspections, 326, 332
metrics for, 327–330
peer, 326
purpose of, 325
for quality, 341–342
reporting and recordkeeping for, 333–334
walkthroughs, 326, 332
Rework effort, 328
Risk. *see also* Risk assessment; Risk management
business, 534
categorization of, 534–535
characteristics of, 534
in component-based software engineering, 229
components of, 537
drivers of, 537
estimation of, 538–541
exposure to, 540–541
generic, 535
identification of, 535–537
impact of, 540–541
known, 535
low-quality software and, 319–320
patterns of, 536
predictable, 535
product-specific, 535–536
project, 534, 536–537
refinement of, 542–543
technical, 534
unpredictable, 535
Risk assessment
collective, 539
contingency plans and, 86
in prototype development, 66
security, 362, 364–365
in spiral model, 30, 56, 57
Risk information sheet (RIS), 546, 547
Risk item checklists, 536
Risk management
framework for, 364–365
gamification and, 544–545
intentional, 533
principles of, 535
proactive strategies, 534
reactive strategies, 533
RMMM plan for, 546
for software process improvement, 575–576
in software quality assurance, 342
as umbrella activity, 11
Risk Mitigation, Monitoring, and Management (RMMM), 539, 543–547
Risk-oriented decisions, 321
Risk projection, 538–541
Risk tables, 538–539
Run-time validation, 123
Run-time verification, 123

SafeHome
activity diagram, 147, 152
applying CK metrics, 471
applying patterns, 301–302
archetypes, 197–198
architectural assessment, 202
architectural context diagram, 197
behavioral modeling, 121
choosing an architectural style, 192
class models, 142
class testing, 391
cohesion in action, 217–218
communication mistakes, 90
conclusion, 604
considering agile software development, 43
considering first prototype, 63
control panel, 115–117
coupling in action, 219
CRC models, 145–146
debating product metrics, 464
design classes, 171
design concepts, 168–169
designing unique tests, 382
design patterns, 292–293
design vs. coding, 159
establishing a metrics approach, 479
estimating, 513–514
evaluating architectural decisions, 194–195
evaluating first prototype, 65
formulating mobile device requirements, 269–270
gamification and risk management, 545–546
grammatical parse, 138
graphic design, 237
instantiations of, 200
interface design review, 254–255
metrics-based quality approach, 484
mini-specifications, 112
negotiation, 122–123
open-closed principle in action, 213
overall architectural structure, 199–200
preparing for testing, 377
preparing for validation, 408
product request, 111
quality assurance, 345
quality issues, 319, 335
regression testing, 403–404
requirements gathering meeting, 112
risk analysis, 541–542
SCM issues, 451
selecting process model, 28, 31
sequence diagram, 148
starting projects, 16
state diagram, 151
swimlane diagram, 153
team structure, 79, 496
tracking the schedule, 529
use case diagram, 118, 119, 137
use cases for user interface design, 247
use case template, 135–136
user scenario, 113–114, 132–133
using cyclomatic complexity, 386
violating interface golden rules, 240
WebApp testing, 419–420
Safety considerations, 342, 352–353, 545
Scaffolding, 64, 379
Scenario-based elements, 119
Scenario-based modeling
actors and user profiles in, 131
characteristics of, 127
overview, 130
popularity of, 128
use cases in, 131–137
Scenario-based testing, 407
Scheduling, 321, 520–523, 526–529
Scientific software, 7
Scope
of products, 491–492, 497
of projects, 91, 114
in prototype development, 67
of software, 497, 507
of use cases, 135
of user experience design, 234
Scrum framework, 42–45, 56, 66–67
Search-based software engineering (SBSE), 161, 558, 597, 638
Secondary actors, 115
Secondary scenarios, 134
Secure coding, 367
Security
attack patterns and, 363–364
attack surface and, 366–367
coding and, 367–368
importance of, 356, 357
in integrity measurement, 483
life-cycle models for, 357–359
management of, 342
measurement of, 368–369
misuse and abuse cases, 363–364
of MobileApps, 424
process improvement and maturity models, 370
quality and, 320
requirements engineering for, 360–363
risk assessment, 362, 364–365
testing, 414, 423–424
threat modeling methods, 365–366
tools for, 358, 365
touchpoints for, 359
of WebApps, 424
Security Development Lifecycle (SDL), 357–359, 365
Selection and justification activity, 573–574

Selective testing, 381
Self-organizing teams, 41, 45, 78, 86, 495
Semantic testing, 420–421
Sensitivity points, 201
Separation of concerns, 87, 130, 165, 194
Sequence diagrams, 148–149, 618–621
Server classes, 404
Service computing, 273
Service-oriented architecture decision (SOAD), 195–196
Shrink-wrapped packages, 342
Simple Object Access Protocol (SOAP), 273
Six Sigma, 9, 349
Size-oriented metrics, 480–481
Slots, 293
Smoke testing, 400–401
Social media, 79–80, 559, 604
Social networking tools, 79
Software
  application domains, 7
  building blocks for, 591–592
  contracted, 342
  defined, 5
  failure curves for, 6
  "good enough," 316
  importance of, 2–3, 603
  key questions regarding, 4
  legacy, 8
  model-driven development of, 596–597
  nature of, 4–8, 74
  open-world, 589–590
  product-line software, 7, 164, 590
  realities related to, 3
  reliability of, 350–353
  safety of, 342, 352–353, 545
Software analytics, 462–463, 558–559
Software architecture. *see* Architecture
Software Assurance Maturity Model (SAMM), 370
Software capital, 20
Software components. *see* Components
Software configuration, defined, 438
Software configuration audits, 452
Software configuration items (SCIs), 438, 441–442
Software configuration management (SCM)
  baselines in, 441, 442, 450
  continuous integration and, 446–447
  of dependencies and changes, 442–443
  elements of, 440
  repository for, 441, 443–446
  scenario for, 439–440
  tasks involved in, 447–448
  version control systems, 445–446
  work flow for, 438
Software defects. *see* Defects
Software design. *see* Design
Software engineering. *see also* Trends in software engineering
  component-based, 228–230, 509
  computer-aided, 9
  defined, 8
  as discipline, 585–586
  ethics and, 607–609
  goals of, 438
  grand challenge for, 594–595
  intelligent, 606
  layers of, 9
  long view of, 606–607
  practice, 12–15, 85–88
  principles of, 14–15, 85–100
  psychology of, 75–76
  search-based, 161, 558, 597, 638
  social media and, 79–80
  start of projects, 15–16
  video games, 1–2
Software engineering environment (SEE), 509, 599–600
Software engineers, 75, 84
Software equation, 523
Software evolution, 550, 562–565
Software increments, 11, 32, 40, 58
Software maintenance. *see* Maintenance
Software maturity index (SMI), 476
Software measurement. *see* Measurement
Software platform solutions, 592
Software process
  adaptation of, 11–12
  agility and (*see* Agility)
  assessment and improvement of, 24–25
  components of, 9–10
  decomposition of, 498–500
  defined, 9, 20, 21
  flow of, 22–23, 25, 31
  framework for, 10–11, 21, 23
  models for (*see* Process models)
  principles guiding, 85–86
  project management and, 492, 498–500
  recommended steps, 71
  relationship to product, 34–35
  schematic representation of, 21
  trends related to, 593–594
Software process improvement (SPI)
  assessment and gap analysis in, 572–573
  continuous, 341
  education and training in, 573
  evaluation in, 575
  frameworks for, 569–570, 576–579
  installation/migration in, 574–575
  maturity models for, 570–571
  return on investment, 580
  risk management for, 575–576
  security and, 370
  selection and justification in, 573–574
  for small organizations, 571
  steps for, 571–576
  trends in, 580–581
Software process redesign (SPR), 574
Software product lines, 7, 164
Software project plans. *see* Project planning
Software quality assurance (SQA)
  attributes in, 346
  background issues related to, 341
  elements of, 340–343
  formal approaches to, 347
  goals of, 345–346
  metrics for, 346
  plans for, 343–344, 354
  processes and product characteristics, 343
  reliability, 350–353
  safety, 342, 352–353
  Six Sigma for, 9, 349
  statistical, 347–349, 378
  tasks of, 343–344
Software reengineering, 554, 562–565
Software reviews. *see* Reviews
Software scope, 497, 507
Software security. *see* Security
Software sizing, 511
Software support. *see also* Maintenance
  analytics in, 558–559
  cost of, 559–560
  elements of, 549, 550
  iterative model of, 551
  proactive, 557–560
  social media and, 559
Software teams. *see* Teams
Source code analysis, 561
Source code metrics, 473–474
Source objects, 251
Spacing, 194
Specifications
  in communication activity, 23
  mini-specifications, 112
  in requirements engineering, 105, 407
  template for, 105
  test, 402
SPICE model, 579
Spiral model, 29–30, 56–57
Sprints, 42–45, 61, 186
SQA groups, 341, 343–344
SQUARE process, 360–363
Stakeholders
  collaboration among, 108
  communication with, 23, 24
  defined, 10, 107
  feedback from, 27, 45, 55
  identifying, 107
  negotiation with, 122
  in planning, 91
  in project management, 493
  in prototype development, 58–59, 61–62

Standards for quality, 313–314, 341, 353–354
State, defined, 120
Statecharts, 224
State diagrams, 120–121, 150–151, 392, 625–628
Static analysis tools, 368
Static architecture-conformance analysis (SACA), 204
Static testing, 429
Statistical quality assurance, 347–349, 378
Statistical software process improvement (SSPI), 478
Status reporting, 452
Stepwise refinement, 167–168, 248
Stereotypes, 613
Storyboarding, 186
Stress testing, 425–426
STRIDE model, 365, 366
Structural complexity, 467
Structural models, 164
Structural patterns, 292
Structural properties, 164
Structural testing, 97, 383–387, 397
Structural uncertainty, 506
Structure charts, 210
Structured analysis, 128
Stubs, 379
Sufficiency, 468
Supervised learning, 634
Supportability, 552. *see also* Software support
Swimlane diagrams, 151, 153–154
Symmetry, 194
Synchronization control, 450
System complexity, 467
System coupling, 129
System modeling, 446
System of forces, 290
System software, 7
System testing, 376, 377, 396

Tailored shell, 342
Talent mix, 591
Target environment, 509
Target objects, 251
Task, defined, 10
Task analysis, 243, 247–248
Task model, 270
Task networks, 525–526
Task sets
  communication, 23, 24, 500
  concept development projects, 524
  defined, 21
  design, 162
  project planning, 507, 523–525
  refinement of, 524–525
Teams
  agile, 40, 41, 78, 495
  collaboration and, 587
  coordination and communication, 496, 604
  development, 43–45, 67
  global, 80–81
  jelled, 76–77, 494
  Kanban, 48–50
  leaders of, 493–494
  in project management, 494–496
  Scrum, 43, 44
  self-organizing, 41, 45, 78, 86, 495
  structure of, 78–79, 496
  talent mix and, 591
  toxicity of, 77, 495
Technical debt, 157, 230, 327, 533
Technical reviews (TRs)
  cost effectiveness of, 329
  criteria for, 330–331
  of design quality, 160, 161
  formal, 327, 332–335
  informal, 331–332
  purpose of, 326
  for quality, 341–342
  reference model for, 330
  for requirements validation, 105–106
  as umbrella activity, 11
Technical risks, 534
Technology
  collaborative development of, 595–596
  innovations in, 584–585
  predictive, 416
  process trends, 593–594
  requirements engineering for, 596
Templates
  for pattern-based design, 293–294
  for publication, 457
  for software testing, 373
  for specifications, 105
  for system modeling, 446
  use case, 135–136
Test case design, 381–383, 405–407
Test-driven development (TDD), 598–599
Test frames, 62
Testing. *see also* Component-level testing; Integration-level testing
  acceptance, 48, 68–69, 95, 430
  accessibility, 432–433
  in agile processes, 95
  alerts and extraordinary conditions, 417
  alpha, 430
  architecture, 375, 376
  artificial intelligence systems, 428–429
  basis path, 384–386
  behavioral, 388–390, 392–393
  beta, 430
  black-box, 388–390, 397
  boundary, 381–382
  certification, 414
  class, 390–391
  cloud-based, 428
  cluster, 404
  condition, 386
  connectivity, 414
  content, 420–421
  control structure, 386–387
  cost-effective, 379–380
  criteria for "done," 377–378
  data flow, 381, 386
  documentation, 434
  dynamic, 429
  exhaustive, 380
  fault-based, 405–406
  functional, 97, 388–390, 397
  gesture, 415–416
  glass-box, 383–387, 397
  help facilities, 434
  high-order, 377
  interface, 388, 421
  loop, 387
  metrics for, 474–475
  MobileApps, 413–417, 424–428
  model-based, 429–430
  navigation, 421–423
  object-oriented, 390–393, 404–407
  organizing for, 374–375
  patterns, 409
  performance, 414, 424–426
  playability, 433–434
  preparing for, 377
  principles of, 96–98
  prototypes, 63–65, 68–69
  for quality, 342
  real-time, 426–428
  recovery, 417
  regression, 68, 403–404
  scenario-based, 407
  security, 414, 423–424
  selective, 381
  semantic, 420–421
  smoke, 400–401
  spiral illustration of, 375–376
  static, 429
  steps for, 376–377
  structural, 97, 383–387, 397
  system, 376, 377, 396
  thread-based, 404
  tools for, 50, 413, 415, 422, 430
  unit, 48, 95, 376, 378–381
  usability, 415–417, 430–432
  use-based, 404
  user, 255–256
  user experience, 414–417
  validation, 95, 373–377, 407–408
  verification in, 373–374
  virtual environments, 430–434
  virtual keyboard input, 416
  voice input and recognition, 416–417
  WebApps, 418–426
  white-box, 383–387, 397
Testing in the wild, 414, 426–427
Test reports, 402
Test sets, 633
Test specifications, 402

Theory phase of technology innovation, 585
Thread-based testing, 404
Threats
  analysis of, 363
  in integrity measurement, 483
  mitigation of, 357, 358
  modeling, 357, 365–366
TickIT auditing method, 579
Time allocation, 521
Time-boxes, 42, 44, 45, 528–529
Time-line charts, 527
Time window, 508
Tools
  capture/playback, 403
  data science, 631
  function of, 9
  interface design, 261
  for MobileApps, 265, 267
  reengineering, 562
  refactoring, 230, 561
  restructuring, 564
  reverse engineering, 555, 564
  scheduling, 521
  for security, 358, 365
  social networking, 79
  static analysis, 368
  testing, 50, 413, 415, 422, 430
  trends in, 599–600
  version control, 445
Top-down integration, 398–399
Total quality management (TQM), 9, 349
Touchpoints for security, 359
Toxic team environment, 77, 495
Traceability, 109–110, 383, 442
Traceability matrix, 109–110, 442
Tracking
  dependencies, 445
  issues, 69, 446
  progress, 92
  schedules, 528–529
  state of user, 258
  as umbrella activity, 11
Training and education, 342, 573
Training set, 632, 633
Transcendental view of quality, 311
Transition phase of Unified Process, 33
Trends in software engineering
  changing perceptions of value, 592
  connectivity and collaboration, 587
  emergent requirements, 590–591
  globalization, 587–588
  managing complexity, 588–589
  observation of, 586–587
  open source movement, 592
  open-world software, 589–590
  soft trends, 587–592
  software building blocks, 591–592
  talent mix, 591
  technological, 584–585, 593–599
  tools-related, 599–600
Triggers, 135, 150
Trustworthiness, 431

Umbrella activities, 10, 11, 21–22
UML notation
  activity diagrams, 146–148, 151–154, 223, 622–625
  archetypes, 198
  class diagrams, 120, 612–615
  communication diagrams, 189, 190, 621–622
  component diagrams, 177
  deployment diagrams, 177, 178, 615–616
  sequence diagrams, 148–149, 618–621
  state diagrams, 121, 150–151, 625–628
  swimlane diagrams, 151, 153–154
  use case diagrams, 119, 616–618
Unadjusted actor weight (UAW), 517, 518
Unadjusted use case weight (UUCW), 517, 518
Uncertainty, 534
Unified modeling language (UML)
  actors in, 131
  associations in, 144, 154
  class models, 141–144
  dependencies in, 154
  development of, 32, 611
  for multiple viewpoints, 88
  notation (*see* UML notation)
  profiles in, 131
Unified Process (UP), 31–33
Unified theory for software evolution, 550
Unit testing, 48, 95, 376, 378–381
Unordered Categorical variables, 633
Unpredictable risks, 535
Unsupervised learning, 634
Usability
  definitions of, 233, 256
  engineering, 236–237
  guidelines for, 257–259
  software quality metric, 483
  testing, 415–417, 430–432
Usage scenarios, 113–114, 271
Use-based testing, 404
Use case diagrams, 118, 119, 136, 137, 616–618
Use case points (UCPs), 517
Use cases
  creating, 131–135
  defined, 31, 616
  development of, 114–118
  documenting, 135–137
  estimation and, 517–518
  exceptions, 134–135
  formal, 127, 135
  identifying events with, 149–150
  questions to be answered by, 115
  refinement of, 135
  in requirements gathering, 113
  scope of, 135
  for testing requirements, 382–383
  in Unified Process, 31–33
  for user interface design, 247
User experience testing, 414–417
User experience (UX) design. *see also* Interface design
  defined, 234
  elements of, 175, 234–237
  information architecture in, 235
  interaction design in, 236
  scope of, 234
  steps for, 249–250
  usability engineering in, 236–237
  visual design in, 237
User interaction design, 236
User interface. *see also* Interface design
  accessibility of, 233, 237, 259–261
  analysis of, 243–249
  consistency of, 240–241, 257
  construction of, 243
  interaction mechanisms in, 234
  prototype of, 62
  usability of, 233, 236–237, 256–259
  validation of, 243
User modeling, 241, 245–246
User-observable functionality, 146
User personas, 245–246
User profiles, 131
User research, 244–245
User stories, 46, 48, 56, 58, 61–63, 67–68
User testing, 255–256
User view of quality, 311

Validation
  in communication activity, 23
  of effort, 521
  of interface, 243
  in requirements engineering, 105–106, 123–124
  run-time, 123
  in user experience design, 250
Validation testing, 95, 373–377, 407–408
Value, changing perceptions of, 592
Value-based view of quality, 311
Variability-intensive systems, 162, 590
Vendor management, 342
Verification, 123, 373–374
Version control, 445–446, 457–458
Versioning, 445
Version management, 446
Viewpoints, multiple, 87–88, 107–108
Virtual environments, 430–434
Virtual keyboard input, 416
Visual design, 237, 277
Voice input, 416–417
Voice recognition, 416–417
Volatility, 118, 468

Walking skeleton, 203
Walkthroughs, 326, 332
Waterfall model, 25–26, 55
Ways of navigating (WoN), 280–282
WebApps (Web applications)
  accessibility of, 259
  aesthetic design of, 277
  agile change management for, 453–458
  architectural design of, 192, 278–280
  component-level design for, 226, 282
  content design of, 277–278
  design pyramid for, 275–282
  graphic design of, 237
  interface design for, 275–276
  metrics for, 472, 473
  navigation design of, 280–282
  patterns for, 291–292
  performance of, 424–426
  quality of, 282–285
  security of, 424
  testing, 418–426
  types of, 7
  usability of, 257–259
Web Services Description Language (WSDL), 273
Weighted device platform matrix (WDPM), 427
Weighted methods per class (WMC), 469
W^{5}HH principle, 501
White-box testing, 383–387, 397
Work breakdown structure (WBS), 526
Work environment analysis, 248–249
Work flow. *see* Process flow
Work in progress (WIP), 49
Work products, 11, 86, 114, 402
Work product size (WPS), 328
Work tasks. *see* Task sets