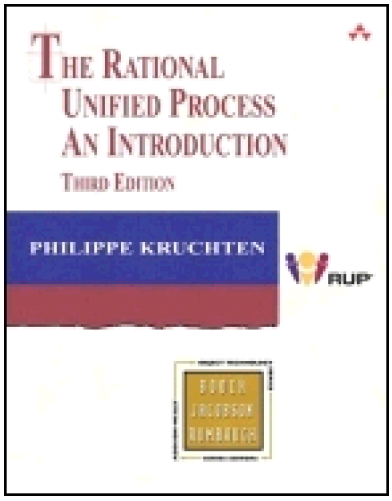


- Table of Contents

Rational Unified Process, The: An Introduction, Third Edition

By Philippe Kruchten

Publisher: Addison Wesley
Pub Date: December 19, 2003
ISBN: 0-321-19770-4
Pages: 336

The Rational Unified Process, Third Edition, is a concise introduction to IBM's Web-enabled software engineering process. Rational Unified Process(R), or RUP(R), Lead Architect Philippe Kruchten quickly and clearly describes the concepts, structure, content, and motivation that are central to the RUP. Readers will learn how this approach to software development can be used to produce high-quality software, on schedule and on budget, using the Unified Modeling Language (UML), software automation, and other industry best practices.

The RUP unifies the entire software development team and optimizes the productivity of each team member by bringing them the experience of industry leaders and lessons learned from thousands of projects. It provides detailed and practical guidance through all phases of the software development life cycle, but it is not inflexible. The RUP can be tailored to suit a wide variety of projects and organizations.

This new edition has been updated to reflect all the changes integrated into the latest version of the Rational Unified Process-RUP 2003. It includes a four-color poster that lists key RUP elements: disciplines, workflows, artifacts, phases, and milestones.

RUP 2003 also offers

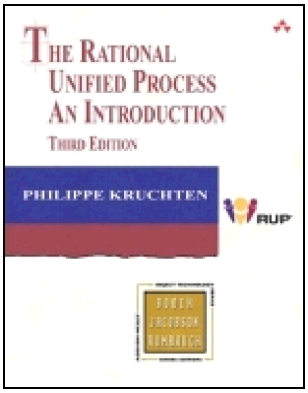
- The RUP Builder with enhanced configurability, process views, and three base configurations.
- MyRUP, enabling personalized views of the RUP configuration.
- Additions to the Rational Process Workbench, used to define process components, perform modifications to the RUP, and create process plugins.

- The introduction of a separate Process Engineering Process offering guidance on adapting the RUP, creating plugins, and deploying the RUP.

The Rational Unified Process, Third Edition, is a reliable introduction to the Rational Unified Process that will serve project managers and software professionals alike.

[Team LiB]

NEXT



Rational Unified Process, The: An Introduction, Third Edition

By Philippe Kruchten

Start Reading

Publisher: Addison Wesley
Pub Date: December 19, 2003
ISBN: 0-321-19770-4
Pages: 336

Copyright

Library of Congress Cataloging-in-Publication Data

Kruchten, Philippe.
The rational unified process : an introduction / Philippe Kruchten—3rd ed.
p. cm.
ISBN 0-321-19770-4 (alk. paper)
1. Computer software—Development. 2. Software engineering. I. Title.

QA76.76.D47K79 2003
005.1—dc22
2003020385

Boston, MA 02116
Fax: (617) 848-7047

Text printed on recycled paper

12345678910—CRS—0706050403

First printing, December 2003

Dedication

To Sylvie, Alice, Zoé, and Nicolas

PREVIOUS NEXT

The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors

For more information, check out the series web site at www.awprofessional.com/otseries.

Ahmed/Umrysh, *Developing Enterprise Java Applications with J2EE™and UML*

Arlow/Neustadt, *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*

Arlow/Neustadt, *UML and the Unified Process: Practical Object-Oriented Analysis and Design*

Armour/Miller, *Advanced Use Case Modeling: Software Systems*

Bellin/Simone, *The CRC Card Book*

Bergström/Råberg, *Adopting the Rational Unified Process: Success with the RUP*

Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*

Bittner/Spence, *Use Case Modeling*

Booch, *Object Solutions: Managing the Object-Oriented Project*

Booch, *Object-Oriented Analysis and Design with Applications*, *2E*

Booch/Bryan, *Software Engineering with ADA*, *3E*

Booch/Rumbaugh/Jacobson, *The Unified Modeling Language User Guide*

Box/Brown/Ewald/Sells, *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications*

Carlson, *Modeling XML Applications with UML: Practical e-Business Applications*

Collins, *Designing Object-Oriented User Interfaces*

Conallen, *Building Web Applications with UML*, *2E*

D'Souza/Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*

Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*

Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*

Douglass, *Real-Time UML, 2E: Developing Efficient Objects for Embedded Systems*

Eeles/Houston/Kozaczynski, *Building J2EE™ Applications with the Rational Unified Process*

Fontoura/Pree/Rumpe, *The UML Profile for Framework Architectures*

Fowler, *Analysis Patterns: Reusable Object Models*

Fowler et al., *Refactoring: Improving the Design of Existing Code*

Fowler, *UML Distilled, 3E: A Brief Guide to the Standard Object Modeling Language*

Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*

Graham, *Object-Oriented Methods, 3E: Principles and Practice*

Heinckiens, *Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*

Hofmeister/Nord/Dilip, *Applied Software Architecture*

Jacobson/Booch/Rumbaugh, *The Unified Software Development Process*

Jordan, *C++ Object Databases: Programming with the ODMG Standard*

Kleppe/Warmer/Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*

Kroll/Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*

Kruchten, *The Rational Unified Process 3E, An Introduction*

Lau, *The Art of Objects: Object-Oriented Design and Architecture*

Leffingwell/Widrig, *Managing Software Requirements, 2E: A Use Case Approach*

Manassis, *Practical Software Engineering: Analysis and Design for the .NET Platform*

Marshall, *Enterprise Modeling with UML: Designing Successful Software through Business Analysis*

McGregor/Sykes, *A Practical Guide to Testing Object-Oriented Software*

Mellor/Balcer, *Executable UML: A Foundation for Model-Driven Architecture*

Naiburg/Maksimchuk, *UML for Database Design*

Oestereich, *Developing Software with UML, 2E: Object-Oriented Analysis and Design in Practice*

Page-Jones, *Fundamentals of Object-Oriented Design in UML*

Pohl, *Object-Oriented Programming Using C++, 2E*

Pollice et al. *Software Development for Small Teams: A RUP-Centric Approach*

Quatrani, *Visual Modeling with Rational Rose 2002 and UML*

Rector/Sells, *ATL Internals*

Reed, *Developing Applications with Visual Basic and UML*

Rosenberg/Scott, *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*

Rosenberg/Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*

Royce, *Software Project Management: A Unified Framework*

Rumbaugh/Jacobson/Booch, *The Unified Modeling Language Reference Manual*

Schneider/Winters, *Applying Use Cases, 2E: A Practical Guide*

Shan/Earle, *Enterprise Computing with Objects: From Client/Server Environments to the Internet*

Smith/Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*

Stevens/Pooley, *Using UML, Updated Edition: Software Engineering with Objects and Components*

Unhelkar, *Process Quality Assurance for UML-Based Projects*

van Harmelen, *Object Modeling and User Interface Design: Designing Interactive Systems*

Wake, *Refactoring Workbook*

Warmer/Kleppe, *The Object Constraint Language, 2E: Getting Your Models Ready for MDA*

White, *Software Configuration Management Strategies and Rational ClearCase®: A Practical Introduction*

The Component Software Series

Clemens Szyperski, Series Editor

For more information, check out the series web site at www.awprofessional.com/csseries.

Allen, *Realizing eBusiness with Components*

Apperly et al., *Service- and Component-based Development: Using the Select Perspective™ and UML*

Atkinson et al., *Component-Based Product Line Engineering with UML*

Cheesman/Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*

Szyperski, *Component Software, 2E: Beyond Object-Oriented Programming*

Whitehead, *Component-Based Development: Principles and Planning for Business Systems*

Preface

The Rational Unified Process is a software engineering process developed and marketed originally by Rational Software, and now IBM. It is a disciplined approach to assigning and managing tasks and responsibilities in a development organization. The goal of this process is to produce, within a predictable schedule and budget, high-quality software that meets the needs of its end users.

The Rational Unified Process captures many of the best practices in modern software development and presents them in a tailorable form that is suitable for a wide range of projects and organizations. The Rational Unified Process delivers these best practices to the project team online in a detailed, practical form.

This book provides an introduction to the concepts, structure, contents, and motivation of the Rational Unified Process.

Goals of This Book

In this book, you will

- learn what the Rational Unified Process is and what it is not;
- master the vocabulary of the Rational Unified Process and understand its structure;
- develop an appreciation for the best practices that we have synthesized in this process; and
- understand how the Rational Unified Process can give you the guidance you need for your specific responsibility in a project.

This book is not the complete Rational Unified Process. Rather, it is a small subset to introduce the RUP. In the full Rational Unified Process you will find the detailed guidance needed to carry out your work. The full Rational Unified Process—the online *knowledge base*—is a product that can be obtained from IBM.

This book makes numerous references to the Unified Modeling Language (UML), but it is not an introduction to the UML. That is the focus of two other books: *The Unified Modeling Language User Guide* (Addison-Wesley, 1999) and *The Unified Modeling Language Reference Manual* (Addison-Wesley, 1999).

This introductory book speaks about modeling and object-oriented techniques, but it is not a design method, and it does not teach you how to model. Detailed steps and guidance on the various techniques that are embedded in the Rational Unified Process can be found only in the process product.

Several chapters of this book discuss project management issues. They describe aspects of planning an iterative development, managing risks, and so on. This book, however, is by no means a complete manual on project management and software economics. For more information, we refer you to the book *Software Project Management: A Unified Framework* (Addison-Wesley, 1998).

The Rational Unified Process is a specific and detailed instance of a more generic process described in the textbook *The Unified Software Development Process* (Addison-Wesley, 1998).

Who Should Read This Book?

The Rational Unified Process, An Introduction, Third Edition, is written for a wide range of people involved in software development: project managers, developers, quality engineers, process engineers, method specialists, system engineers, and analysts.

This book is relevant especially to members of an organization that has adopted the Rational Unified Process or is about to adopt it. It is likely that an organization will tailor the Rational Unified Process to suit its needs, but the core process described in this book should remain the common denominator across all instances of the Rational Unified Process.

This book will be a useful companion to students taking one of the many professional education courses delivered by IBM Rational Software and its partners in industry and academia. It provides a general context for the specific topics covered by the course.

We assume that you have a basic understanding of software development. It is not necessary that you have specific knowledge of a programming language, of an object-oriented method, or of the Unified Modeling Language.

How to Use This Book

Software professionals who are working in an organization that has adopted the Rational Unified Process, in whole or part, should read the book linearly. The chapters have been organized in a natural progression.

Project managers can limit their reading to Chapters 1, 2, 4, and 7, which provide an introduction to the implications of an iterative, risk-driven development process.

Process engineers or methodologists may have to tailor and install the Rational Unified Process in their organizations. They should carefully study Chapters 3, 14, and 17, which describe the process structure and the overall approach to implementing the Rational Unified Process.

Organization and Special Features

The book has two parts.

Part I describes the process, its context, its history, its structure, and its software development lifecycle. It describes some of the key features that differentiate the Rational Unified Process from other software development processes:

- Chapter 1: Software Development Best Practices
- Chapter 2: The Rational Unified Process
- Chapter 3: Static Structure: Process Description
- Chapter 4: Dynamic Structure: Iterative Development
- Chapter 5: An Architecture-Centric Process
- Chapter 6: A Use-Case-Driven Process

Part II gives an overview of the various components of process, or disciplines. There is one chapter for each discipline.

- Chapter 7: The Project Management Discipline
- Chapter 8: The Business Modeling Discipline
- Chapter 9: The Requirements Discipline
- Chapter 10: The Analysis and Design Discipline
- Chapter 11: The Implementation Discipline
- Chapter 12: The Test Discipline
- Chapter 13: The Configuration and Change Management Discipline
- Chapter 14: The Environment Discipline
- Chapter 15: The Deployment Discipline
- Chapter 16: Typical Iteration Plans
- Chapter 17: Implementing the Rational Unified Process

Most discipline chapters are organized into six sections:

- Purpose of the discipline
- Definitions and key concepts
- Roles and artifacts
- A typical workflow: an overview of the activities

- Tool support
- Summary

Two appendixes summarize all the roles of the process and artifacts (the work products of the process) that are introduced in Chapters 7 through 15. A list of acronyms and glossary of common terms are provided, as is an annotated bibliography.

PREVIOUS NEXT

For More Information

Information about the Rational Unified Process, such as a data sheet and a downloadable demo version, can be obtained from IBM, Rational Software via the Internet at www.rational.com/rup/.

If you are already using the Rational Unified Process, additional information is available from the Rational Developers Network, which has extra goodies, updates, and links to partners. The hyperlink to the RDN is in the online version of RUP.

Academic institutions can contact IBM for information on a special program for including the Rational Unified Process in the curriculum.

Second Edition

The second edition of *The Rational Unified Process: An Introduction* made the book current with the Rational Unified Process 2000.

Third Edition

This third edition of *The Rational Unified Process: An Introduction* makes the book current with the Rational Unified Process 2003. This edition implements several changes of terminology, some redesign of parts of the process (in particular, the test and environment disciplines), as well as the further development of RUP as a process framework. New elements are introduced—the concept of process components and process plug-ins, a new tool set comprising RUP Modeler, RUP Organizer, RUP Builder, and MyRUP, and a separate Process Engineering Process. It incorporates the feedback of many readers and acknowledges the acquisition of Rational Software by IBM.

Acknowledgments

The Rational Unified Process reflects the wisdom of a great many software professionals from Rational Software and elsewhere. The history of the process can be found in Chapter 2. But putting together a book, even as small and modest as this one, required the dedicated effort of a slate of people, whom I would like to recognize here.

Early members of the Process Development Group assembled the Rational Unified Process and contributed to this introduction. Stefan Bylund, then Kurt Bittner, and finally Bruce Macisaac contributed to the analysis and design chapter. Maria Ericsson developed the business engineering and requirements management aspect with Leslee Probasco. The test part was started by Bruce Katz and then completely redesigned by Paul Szymkowiak. John Smith expanded the project management aspects for RUP 2000. Jas Madhur contributed the ideas on configuration management, change management, and deployment. Håkan Dyrhage contributed many ideas to the organization and structure of the process and to its implementation and configuration. Margaret Chan was responsible for the product integration and for the assembly of most of the artwork in this book with Susan Buie. Sigurd Hopen developed the PEP with the help of Björn Gustafsson.

Per Kroll is the manager of the development group, Mike Barnard the product manager. Alfredo Bencomo, Chinh Vo, Philip Denno, Lars Jenzer, Glenys Macisaac, and Fionna Chong developed the product infrastructure. And Debbie Gray is the devoted administrative assistant of a team spread across nine time zones.

The Rational Unified Process and this book benefited over the past 8 years from the reviews and ideas of Stefan Ahlqvist, Dave Bernstein, Grady Booch, Murray Cantor, Geoff Clemm, Catherine Connor, Mike Devlin, Christian Ehrenborg, Ian Gavin, Christina Gisselberg, Sam Guckenheimer, Björn Gustafsson, Matt Herdon, Ivar Jacobson, Paer Jansson, Ron Krubek, Dean Leffingwell, Andrew Lyons, Bruce Malasky, Roger Oberg, Gary Pollice, Terri Quatrani, Walker Royce, Jim Rumbaugh, Ian Spence, John Smith, and Brian White.

We are very grateful to Grady Booch for writingChapter 1.

Special thanks go to the Rational field, members of our Chat_RUP list, and our users, members of the RUP_Forum, for their feedback, and contributions.

And finally many thanks to our editor Mary O'Brien, our former editor J. Carter Shanklin, as well as Brenda Mulligan, Amy Fleischer and her team, and all the staff at Addison-Wesley for getting this book out as quickly and smoothly as they did.

Philippe Kruchten
Vancouver, B.C., Canada

Part I: The Process

Chapter 1. Software Development Best Practices

Grady Booch

This chapter surveys best practices for software development and establishes a context for the Rational Unified Process.

The Value of Software

Software is the fuel on which modern businesses are run, governments rule, and societies become better connected. Software has helped us create, access, and visualize information in previously inconceivable ways and forms. Globally, the breathtaking pace of progress in software has helped drive the growth of the world's economy. On a more human scale, software-intensive products have helped cure the sick and have given voice to the speechless, mobility to the impaired, and opportunity to the less able. From all these perspectives, software is an indispensable part of our modern world.[1]

[1] Grady Booch, "Leaving Kansas," *IEEE Software* 15(1), Jan.–Feb. 1998, pp. 32–35.

The good news for software professionals is that worldwide economies depend increasingly on software. The kinds of software-intensive systems that technology makes possible and society demands are expanding in size, complexity, distribution, and importance.

The bad news is that the expansion of these systems in size, complexity, distribution, and importance pushes the limits of what we in the software industry know how to develop. Trying to advance legacy systems to more modern technology brings its own set of technical and organizational problems. Compounding the problem is that businesses continue to demand increased productivity and improved quality with faster development and deployment. Additionally, the supply of qualified development personnel is not keeping pace with the demand.

The net result is that building and maintaining software is hard and getting harder; building quality software in a repeatable and predictable fashion is harder still.

Symptoms and Root Causes of Software Development Problems

Different software development projects fail in different ways—and, unfortunately, too many of them fail—but it is possible to identify a number of common symptoms that characterize these kinds of projects:[2] [3]

[2] Capers Jones, *Patterns of Software Systems Failure and Success*. London: International Thompson Computer Press, 1996.

[3] Edward Yourdon, *Death March: Managing "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice-Hall, 1997.

- Inaccurate understanding of end-user needs
- Inability to deal with changing requirements
- Modules that don't fit together
- Software that's hard to maintain or extend
- Late discovery of serious project flaws
- Poor software quality
- Unacceptable software performance
- Team members in each other's way, making it impossible to reconstruct who changed what, when, where, and why
- An untrustworthy build-and-release process

Unfortunately, treating these symptoms does not treat the disease. For example, the late discovery of serious project flaws is only a symptom of larger problems, namely, subjective project status assessment and undetected inconsistencies in the project's requirements, designs, and implementations.

Although different projects fail in different ways, it appears that most of them fail because of a combination of the following root causes:

- Ad hoc requirements management
- Ambiguous and imprecise communication
- Brittle architectures
- Overwhelming complexity
- Undetected inconsistencies in requirements, designs, and implementations
- Insufficient testing
- Subjective assessment of project status
- Failure to attack risk

- Uncontrolled change propagation
- Insufficient automation

Software Best Practices

If you treat these root causes, not only will you eliminate the symptoms, but you'll also be in a much better position to develop and maintain quality software in a repeatable and predictable fashion.

That's what software best practices are all about: commercially proven approaches to software development that, when used in combination, strike at the root causes of software development problems.[4] They are "best practices" not so much because you can precisely quantify their value but rather because they are commonly used in industry by successful organizations. In this chapter we will describe six important best practices:

[4] See the Software Program Manager's Network best practices work at *http://www.spmn.com*.

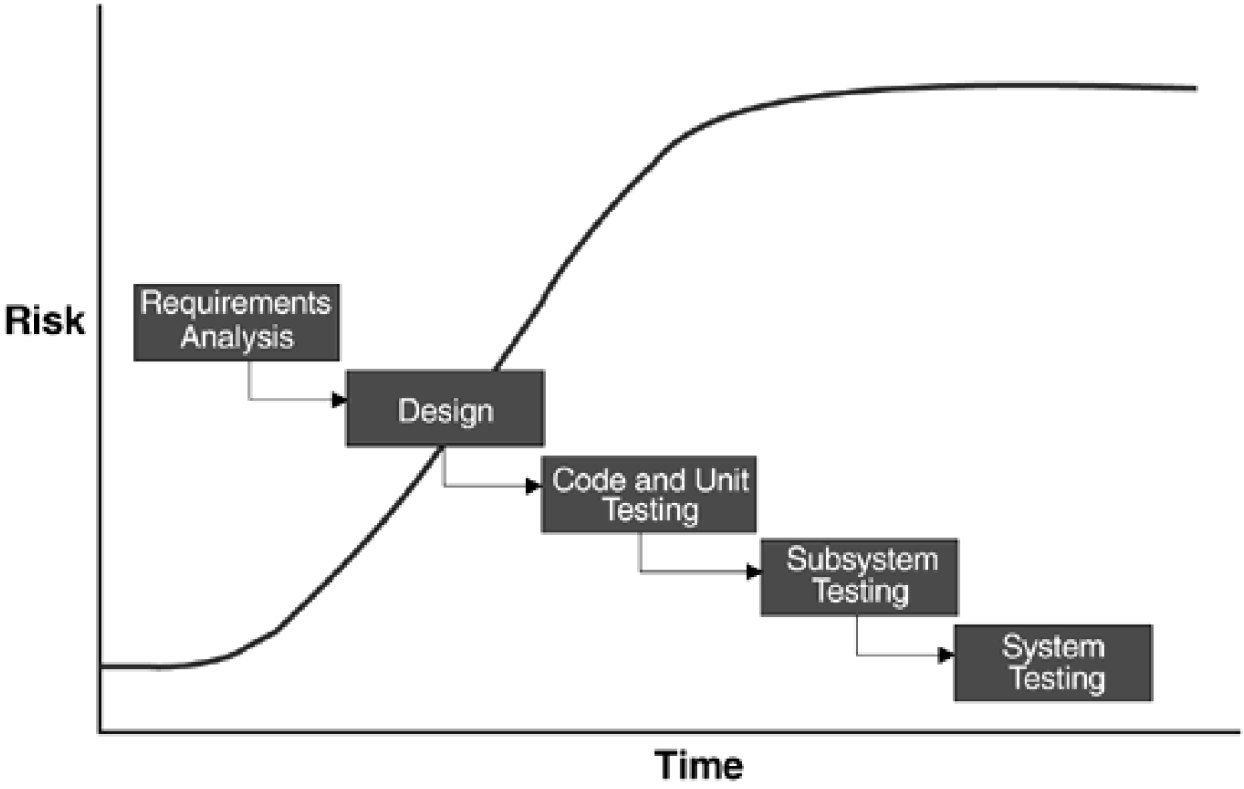
1. Develop software iteratively.
2. Manage requirements.
3. Use component-based architectures.
4. Visually model software.
5. Continuously verify software quality.
6. Control changes to software.

But there are many other software development best practices that will be covered in subsequent chapters.

Develop Software Iteratively

Classic software development processes follow the waterfall lifecycle, as illustrated in Figure 1-1. In this approach, development proceeds linearly from requirements analysis through design, code and unit testing, subsystem testing, and system testing.

Figure 1-1. The waterfall lifecycle



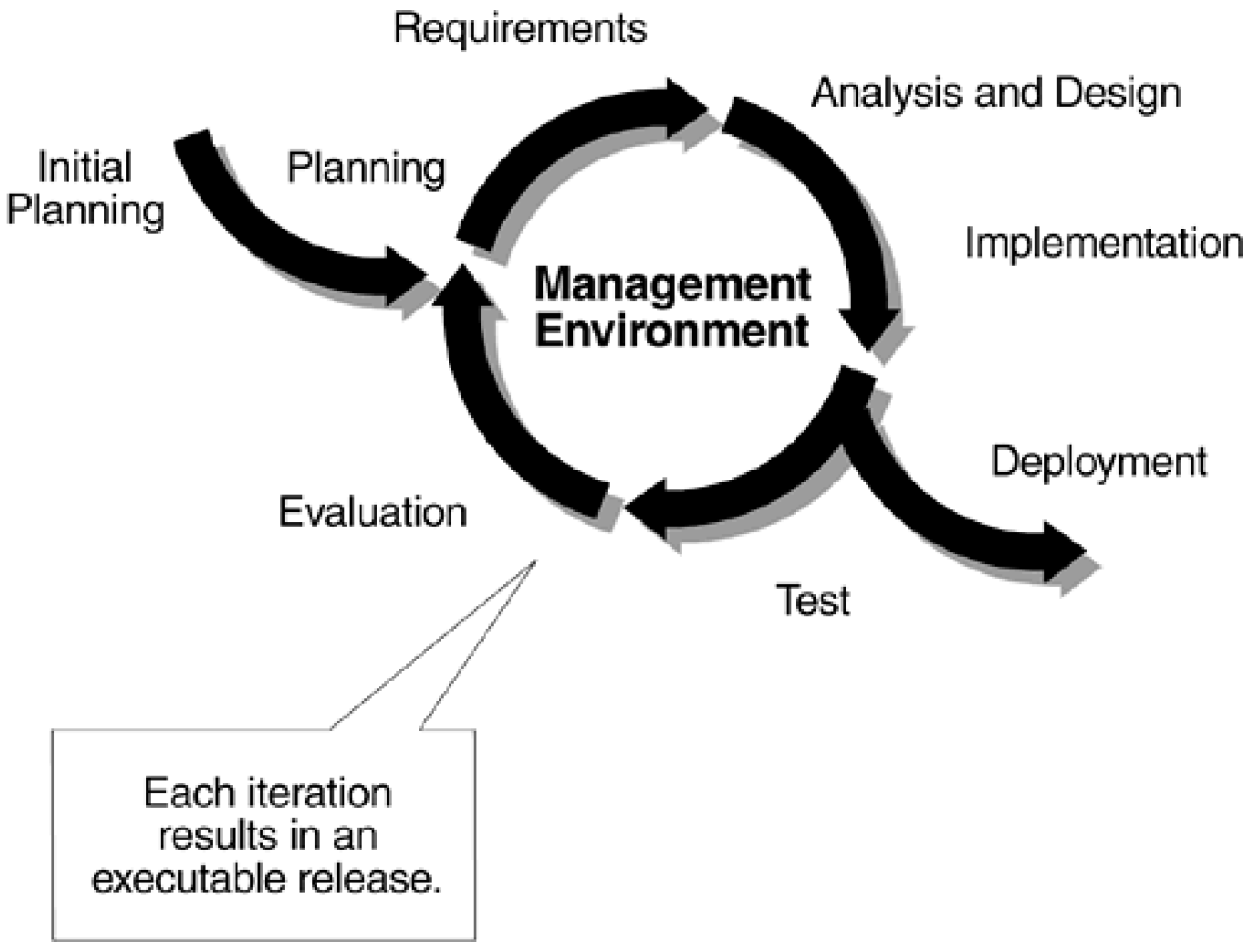
The fundamental problem of this approach is that it pushes risk forward in time so that it's costly to undo mistakes from earlier phases. An initial design will likely be flawed with respect to its key requirements, and, furthermore, the late discovery of design defects tends to result in costly overruns or project cancellation. As Tom Gilb aptly said, "If you do not actively attack the risks in your project, they will actively attack you."[5] The waterfall approach tends to mask the real risks to a project until it is too late to do anything meaningful about them.

[5] Tom Gilb, *Principles of Software Engineering Management*. Harlow, UK: Addison-Wesley, 1988, p. 73.

An alternative to the waterfall approach is the iterative and incremental process, as shown in Figure 1-2. In this approach, building on the work of Barry Boehm's spiral model,[6] the identification of risks to a project is forced early in the lifecycle, when it's possible to attack and react to them in a timely and efficient manner. This approach is one of continuous discovery, invention, and implementation, with each iteration forcing the development team to drive the project's artifacts to closure in a predictable and repeatable way.

[6] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988, pp. 61–72.

Figure 1-2. An iterative and incremental process



Developing software iteratively offers a number of solutions to the root causes of software development problems:

1. Serious misunderstandings are made evident early in the lifecycle when it's possible to react to them.
2. This approach enables and encourages user feedback so as to elicit the system's real requirements.
3. The development team is forced to focus on those issues that are most critical to the project and are shielded from those issues that distract them from the project's real risks.
4. Continuous, iterative testing enables an objective assessment of the project's status.
5. Inconsistencies among requirements, designs, and implementations are detected early.
6. The workload of the team, especially the testing team, is spread more evenly throughout the project's lifecycle.
7. The team can leverage lessons learned and therefore can continuously improve the process.
8. Stakeholders in the project can be given concrete evidence of the project's status throughout its lifecycle.

Manage Requirements

The challenge of managing the requirements of a software-intensive system is that they are dynamic: You must expect them to change during the life of a software project. Furthermore, identifying a system's true requirements—those that weigh most heavily on the system's economic and technical goals—is a continuous process. Except for the most trivial system, it is impossible to completely and exhaustively state a system's requirements before the start of development. Indeed, as a new or evolving system changes, a user's understanding of the system's requirements also changes.

A *requirement* is a condition or capability a system must have. The active management of requirements encompasses three activities: eliciting, organizing, and documenting the system's required functionality and constraints; evaluating changes to these requirements and assessing their impact; and tracking and documenting trade-offs and decisions.

Managing the requirements of your project offers a number of solutions to the root causes of software development problems:

1. A disciplined approach is built into requirements management.
2. Communications are based on defined requirements.
3. Requirements can be prioritized, filtered, and traced.
4. An objective assessment of functionality and performance is possible.
5. Inconsistencies are detected more easily.
6. With suitable tool support, it is possible to provide a repository for a system's requirements, attributes, and traces, with automatic links to external documents.

Use Component-Based Architectures

Visualizing, specifying, constructing, and documenting a software-intensive system demand that the system be viewed from a number of perspectives. Each of the stakeholders—end users, analysts, developers, system integrators, testers, technical writers, and project managers—brings a different agenda to a project, and each of them looks at that system in a different way at different times over the project's life. A system's architecture is perhaps the most important deliverable that can be used to manage these various viewpoints and thereby control the iterative and incremental development of a system throughout its lifecycle.

A system's architecture encompasses the set of significant decisions about the following elements:

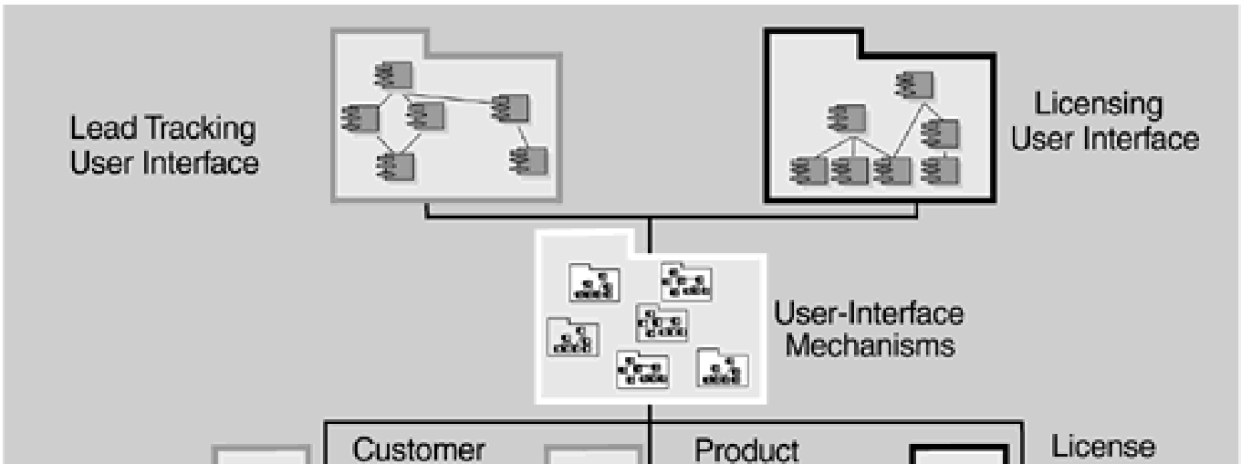
- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified by the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
- The architectural style that guides this organization: these elements and their interfaces, their collaborations, and their composition

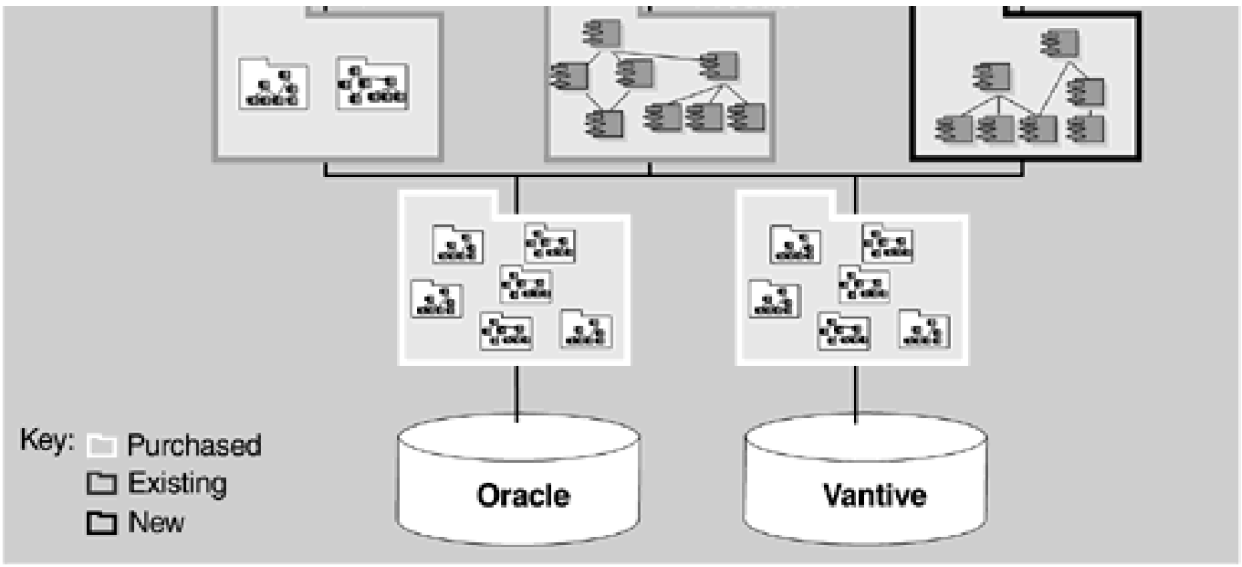
Software architecture is concerned not only with structure and behavior but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technologic constraints and trade-offs, and aesthetic concerns.

Building resilient architectures is important because they enable economically significant degrees of reuse, offer a clear division of work among teams of developers, isolate hardware and software dependencies that may be subject to change, and improve maintainability.

Component-based development (CBD) is an important approach to software architecture because it enables the reuse or customization of components from thousands of commercially available sources. Microsoft's component object model (COM), the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), and Sun Microsystems' Enterprise JavaBeans (EJB) offer pervasive and widely supported platforms on which component-based architectures are made possible. As Figure 1-3 indicates, components make reuse possible on a larger scale, enabling systems to be composed from existing parts, off-the-shelf third-party parts, and a few new parts that address the specific domain and glue the other parts together.

Figure 1-3. A component-based software architecture





Coupled with the practice of developing software iteratively, using component-based architectures involves the continuous evolution of a system's architecture. Each iteration produces an executable architecture that can be measured, tested, and evaluated against the system's requirements. This approach permits the team to attack continuously the most important risks to the project.

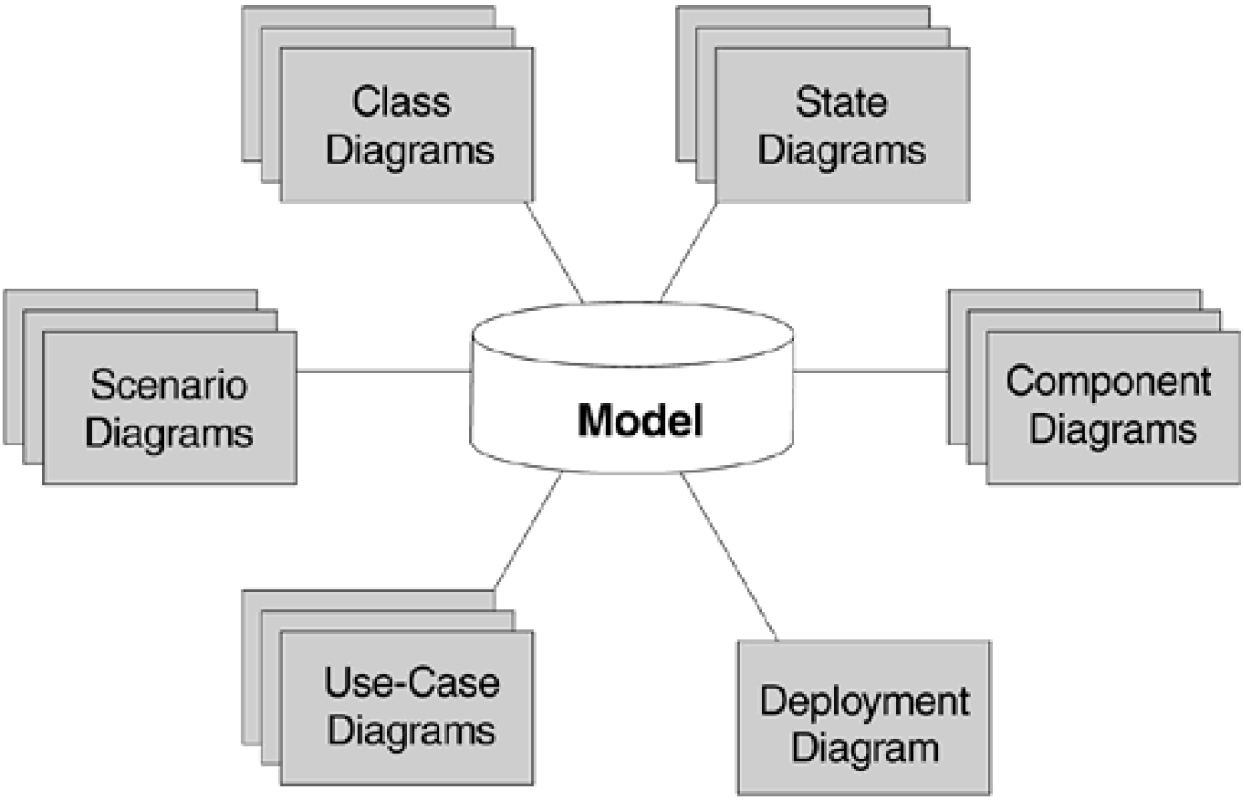
Using component-based architectures offers a number of solutions to the root causes of software development problems:

1. Components facilitate resilient architectures.
2. Modularity enables a clear separation of concerns among elements of a system that are subject to change.
3. Reuse is facilitated by leveraging standardized frameworks (such as COM+, CORBA, and EJB) and commercially available components.
4. Components provide a natural basis for configuration management.
5. Visual modeling tools provide automation for component-based development.

Visually Model Software

A model is a simplification of reality that completely describes a system from a particular perspective, as shown in Figure 1-4. We build models so that we can understand better the system we are modeling; we build models of complex systems because we cannot comprehend such systems in their entirety.

Figure 1-4. Modeling a system from different perspectives



Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system's architecture. Using a standard modeling language such as the Unified Modeling Language (UML), members of the development team can unambiguously communicate their decisions to one another.

Visual modeling tools facilitate the management of these models, letting you hide or expose details as necessary. Visual modeling also helps to maintain consistency among a system's artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity.

Coupled with the practice of developing software iteratively, visual modeling helps you expose and assess architectural changes and communicate those changes to the entire development team. With the right kind of tools, you can then synchronize your models and source code during each iteration.

Modeling your software visually offers a number of solutions to the root causes of software development problems:

1. Use cases and scenarios unambiguously specify behavior.
2. Models unambiguously capture software design.

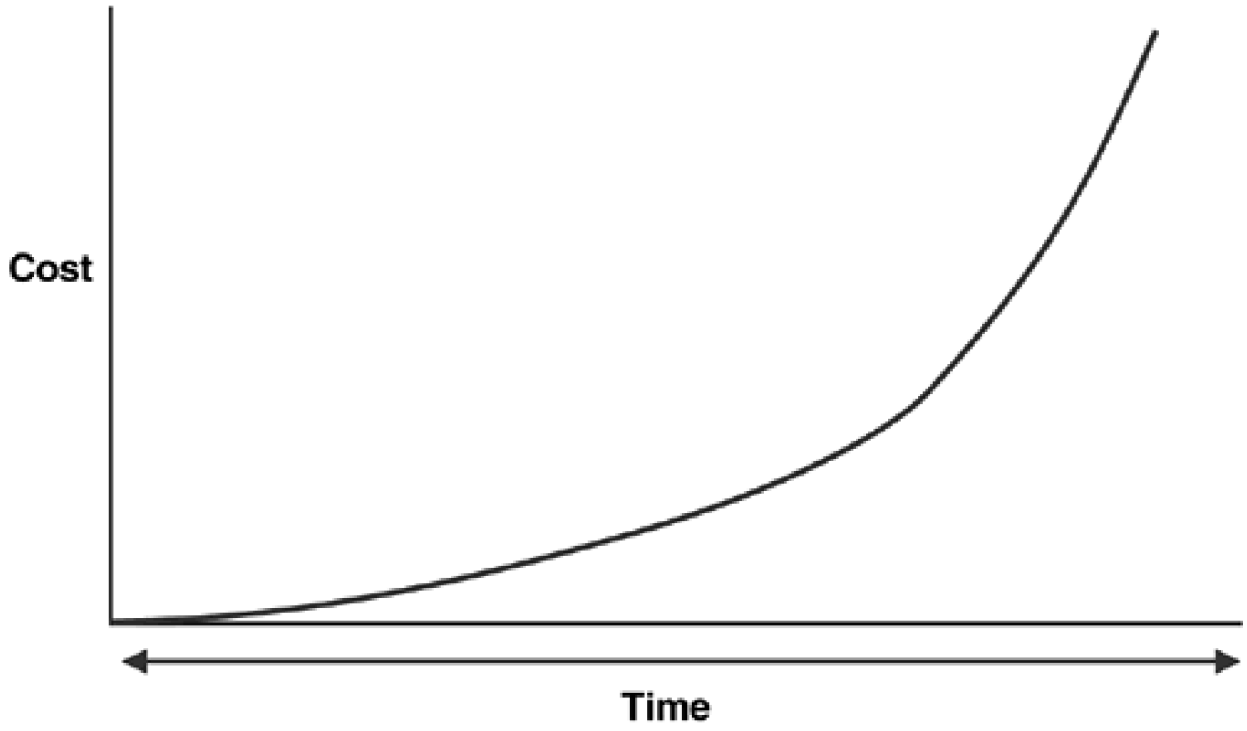
3. Nonmodular and inflexible architectures are exposed.
4. Details can be hidden when necessary.
5. Unambiguous designs reveal their inconsistencies more readily.
6. Application quality starts with good design.
7. Visual modeling tools provide support for UML modeling.

PREVIOUS NEXT

Continuously Verify Software Quality

As Figure 1-5 illustrates, software problems are 100 to 1,000 times more expensive to find and repair after deployment than beforehand. For this reason, it's important to assess continuously the quality of a system with respect to its functionality, reliability, application performance, and system performance.

Figure 1-5. The cost of fixing problems



Verifying a system's functionality—the bulk of the testing activity—involves creating tests for each key scenario, each of which represents some aspect of the system's desired behavior. You can assess a system's functionality by asking which scenarios failed and where, as well as which scenarios and corresponding code have not yet been exercised. As you are developing your software iteratively, you test at every iteration, a process of continuous, quantitative assessment.

Verifying software quality offers a number of solutions to the root causes of software development problems:

1. Project status assessment is made objective, not subjective, because test results, not paper documents, are evaluated.
2. This objective assessment exposes inconsistencies in requirements, designs, and implementations.
3. Testing and verification are focused on areas of highest risk, thereby increasing the quality and effectiveness of those areas.
4. Defects are identified early, radically reducing the cost of fixing them.
5. Automated testing tools provide testing for functionality, reliability, and performance.

Control Changes to Software

A key challenge when you're developing software-intensive systems is that you must cope with multiple developers organized into teams, possibly at different sites, working together on multiple iterations, releases, products, and platforms. In the absence of disciplined control, the development process rapidly degenerates into chaos.

Coordinating the activities and the artifacts of developers and teams involves establishing repeatable workflows for managing changes to software and other development artifacts. This coordination allows a better allocation of resources based on the project's priorities and risks, and it actively manages the work on those changes across iterations. Coupled with developing your software iteratively, this practice lets you continuously monitor changes so that you can actively discover and then respond to problems.

Coordinating iterations and releases involves establishing and releasing a tested baseline at the completion of each iteration. Maintaining traceability among the elements of each release and among elements across multiple, parallel releases is essential for assessing and actively managing the impact of change.

Controlling changes to software offers a number of solutions to the root causes of software development problems:

1. The workflow of requirements changes is defined and repeatable.
2. Change requests facilitate clear communications.
3. Isolated workspaces reduce interference among team members working in parallel.
4. Change rate statistics provide good measurements for objectively assessing project status.
5. Workspaces contain all artifacts, which facilitates consistency.
6. Change propagation is assessable and controlled.
7. Changes can be maintained in a robust, customizable system.

The Rational Unified Process

A software development process has four roles:[7]

[7] Grady Booch, *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison-Wesley, 1995.

1. Provide guidance as to the order of a team's activities.
2. Specify which artifacts should be developed and when they should be developed.
3. Direct the tasks of individual developers and the team as a whole.
4. Offer criteria for monitoring and measuring the project's products and activities.

Without a well-defined process, your development team will develop in an ad hoc manner, with success relying on the heroic efforts of a few dedicated individual contributors. This is not a sustainable condition.

By contrast, mature organizations that employ a well-defined process can develop complex systems in a repeatable and predictable way. Not only is that a sustainable business, but it's also one that can improve with each new project, thereby increasing the efficiency and productivity of the organization as a whole.

Such a well-defined process enables and encourages all of the best practices described earlier. When you codify these practices into a process, your development team can build on the collective experience of thousands of successful projects.

The Rational Unified Process, as described in the remainder of this book, builds on these six best practices plus several others to deliver a well-defined process. This is the context for the Rational Unified Process, a software development process focused on ensuring the production of quality systems in a repeatable and predictable fashion.

Summary

- Building quality software in a repeatable and predictable fashion is hard.
- There are a number of symptoms of common software development problems, and these symptoms are the observable results of the root causes.
- To strike at the root causes of these software development problems, use software development best practices:
    - Develop software iteratively.
    - Manage requirements.
    - Use component-based architectures.
    - Visually model software.
    - Continuously verify software quality.
    - Control changes to software.
- The Rational Unified Process brings these best practices, and many others, together in a form that is suitable for a wide range of projects and organizations.

Chapter 2. The Rational Unified Process

This chapter gives an overview of the Rational Unified Process, introduces the process structure, describes the process product, and outlines its main features.

What Is the Rational Unified Process?

The Rational Unified Process is a *software engineering process.* It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget.

The Rational Unified Process is a *process product*. It is developed and maintained by Rational Software and integrated with its suite of software development tools. It is available from IBM on CD-ROM or through the Internet. This book is an integral part of the Rational Unified Process but represents only a small fraction of the Rational Unified Process knowledge base. Later in this chapter we describe the physical structure of the process product.

The Rational Unified Process is also a *process framework* that can be adapted and extended to suit the needs of an adopting organization. Chapter 3 describes in more detail how the process framework is organized and introduces the *process model,* the elements that compose the process framework.

The Rational Unified Process captures many of the *best practices* in modern software development in a form that is suitable for a wide range of projects and organizations. Along with many others, it covers the practices introduced in Chapter 1:

1. Develop software iteratively.
2. Manage requirements.
3. Use component-based architectures.
4. Visually model software.
5. Continuously verify software quality.
6. Control changes to software.

The Rational Unified Process as a Product

Many organizations have slowly become aware of the importance of a well-defined and well-documented software development process to the success of their software projects. Over the years, they have collected their knowledge and shared it with their developers. This collective know-how often grows out of methods, published textbooks, training programs, and small how-to notes amassed over several projects. Unfortunately, these practices often end up gathering dust in nice binders on a developer's shelf—rarely updated, rapidly becoming obsolete, and almost never followed.

"Software processes are software, too," wrote Lee Osterweil.[1] In contrast with the dusty-binder approach, the Rational Unified Process is designed, developed, delivered, and maintained like any software tool. The Rational Unified Process shares many characteristics with software products:

[1] Lee Osterweil, "Software Processes Are Software Too," *Proceedings of the Ninth International Conference on Software Engineering*, pp. 2–13, Mar. 30–Apr. 2, 1987, Monterey, CA.

- IBM releases regular upgrades.
- It is delivered online using Web technology, so it is literally at the fingertips of the developers.
- It can be tailored and configured to suit the specific needs of a development organization.
- It is integrated with many of the software development tools in the IBM Rational Suites so that developers can access process guidance from within the tool they are using.

This approach of treating the process as a software product provides the following benefits:

- The process is never obsolete; at regular intervals companies get new releases with improvements and up-to-date techniques.
- All project members can access the latest version of their process configuration on an intranet.
- Java applets, such as a process browser and a built-in search engine, allow developers to reach instantaneously process guidance or policies, including the latest document templates they should use.
- Hyperlinks provide navigation from one part of the process to another, eventually branching out to a software development tool or to an external reference or guideline document.
- Local, project- or company-specific process improvements or special procedures are included easily.
- Each project or department can manage its own version or variant of the process.

This online Rational Unified Process product gives you benefits that are difficult to achieve with a process that is available only in the form of a book or binder.

Organization of the Process Product

The RUP product is delivered on a CD-ROM or via the Internet and consists of the following:

1. An online version of the Rational Unified Process, which is the electronic guide for the RUP: a fully hyperlinked Web site description of the process in HTML

2. A tool called RUP Builder that allows you to compose and publish your own tailored configuration of the Rational Unified Process (see Chapter 14)

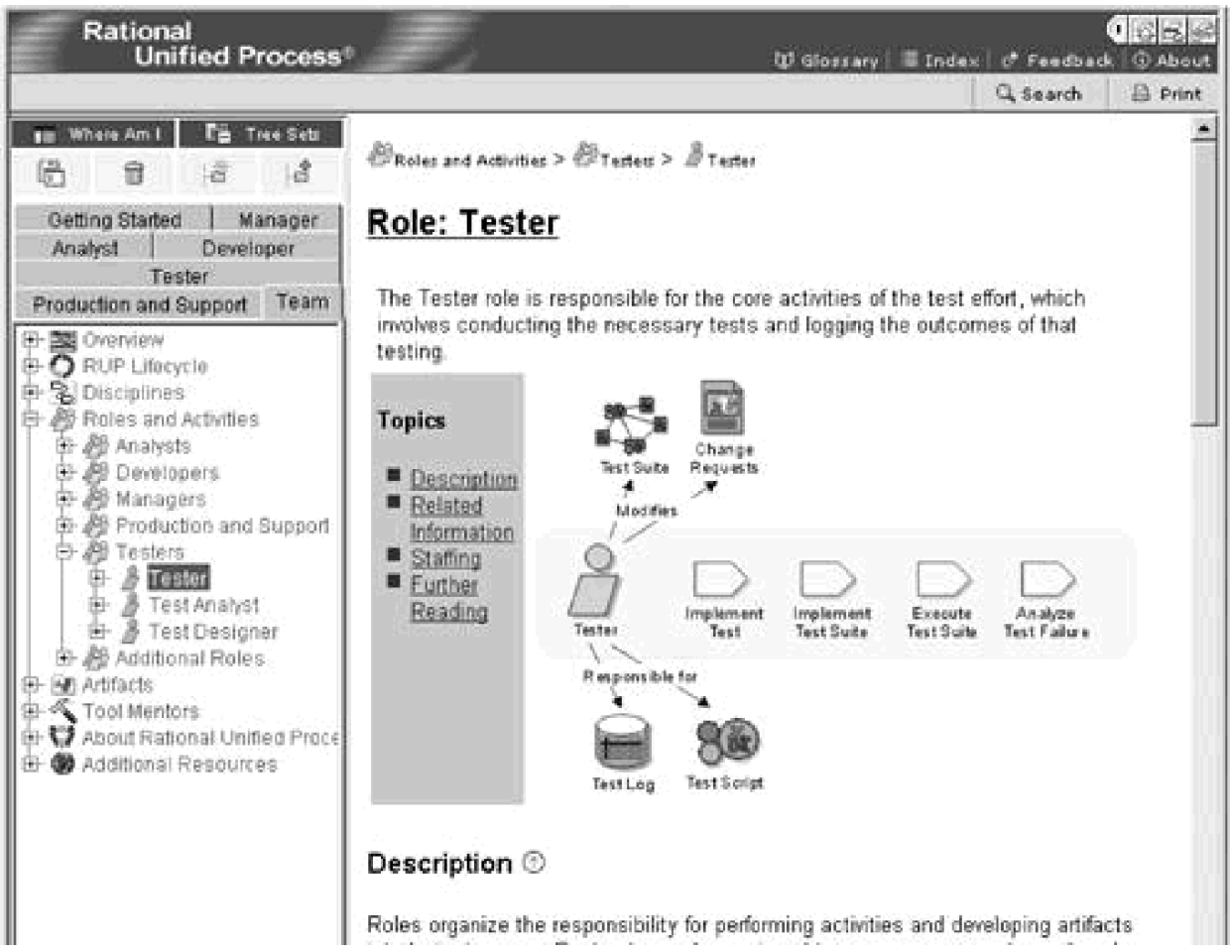
3. Additions to the Rational Unified Process, called Process Plug-ins, which cover specialized topics and which can be added to your configuration with RUP Builder. Many such plug-ins are available for download from the Rational Developers Network or can be obtained from IBM partners

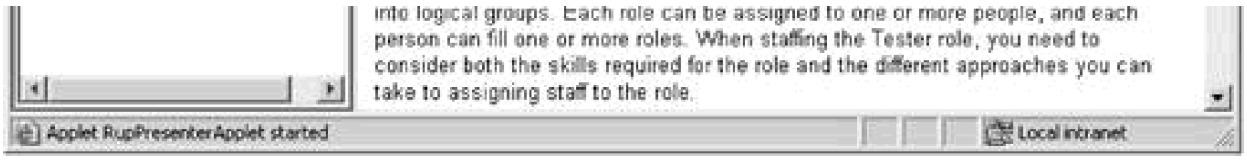
A RUP configuration can be used with any of the popular Web browsers, such as Netscape Navigator™ and Microsoft Internet Explorer™. Information is easy to find, thanks to these elements:

- Extensive hyperlinking
- Graphical navigation (Most graphical elements are hyperlinked to the process elements they depict.)
- A hierarchical tree browser
- An exhaustive index
- A built-in search engine

A tool called MyRUP enables navigation through a RUP configuration and allows a user to define a personalized view of a RUP configuration. You can locate these facilities as shown in Figure 2-1, which is a snapshot of a RUP page.

Figure 2-1. The Rational Unified Process online





In a process configuration, you will find not only a complete description of the process itself but also the following:

- Tool mentors, which provide additional guidance when you're working with any of the software development tools offered by IBM Rational Software, such as Rational XDE (eXtended Develoment Environment) for visual modeling and Rational ClearCase for configuration management
- Templates for all major process artifacts, for example:
    - Microsoft Word templates and Adobe FrameMaker templates for most plain documents and reports
    - Rational SoDA templates, to automate the assembly of documents from multiple sources
    - RequisitePro templates to help manage requirements
    - Microsoft Project templates, to help plan an iterative project based on the RUP
    - HTML templates for extending the online process itself
- Examples of artifacts for simple projects

For the Process Engineers

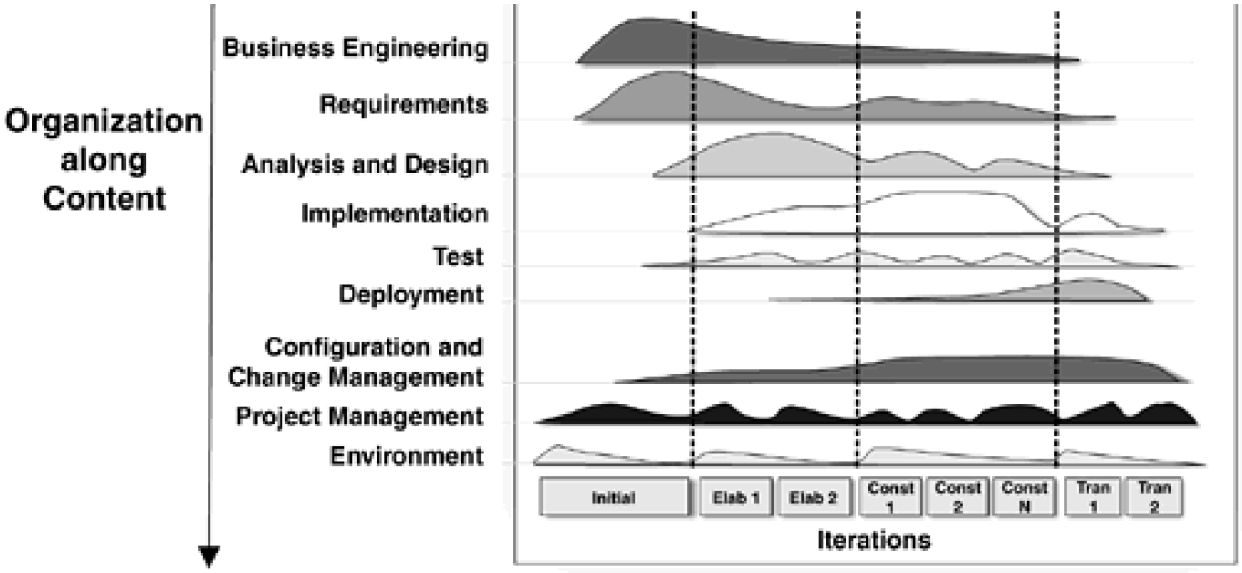
For process engineers additional tools and guidance are available. The Rational Process Workbench (RPW) is the tool used to define process components, perform extensive modification to RUP, and create process plug-ins. It is composed of the RUP Modeler, an add-in to Rational XDE, and RUP organizer. A separate process configuration, called the Process Engineering Process, provides guidance on the tailoring, the customization of RUP, the creation of plug-ins, and the deployment of the Rational Unified Process in an organization. We will examine these topics in Chapters 14 and 17.

Process Structure: Two Dimensions

Figure 2-2 shows the overall *architecture* of the Rational Unified Process. The process has two structures or, if you prefer, two dimensions:

1. The horizontal axis represents time and shows the lifecycle aspects of the process as it unfolds.
2. The vertical axis represents core process disciplines, which group activities logically by nature.

Figure 2-2. Process structure—two dimensions



The first dimension represents the dynamic aspect of the process as it is enacted, and it is expressed in terms of cycles, phases, iterations, and milestones. This is described further in Chapter 4, Dynamic Structure: Iterative Development, and inChapter 7, The Project Management Discipline.

The second dimension represents the static aspect of the process: its description in terms of process components, activities, disciplines, artifacts, and roles. This is described further in Chapter 3, Static Structure: Process Description.

Software Best Practices in the Rational Unified Process

This section revisits the best practices introduced by Grady Booch in Chapter 1 and maps them to major components of the Rational Unified Process. The RUP, however, contains many other best practices in all aspects of software development.

Iterative Development

The iterative approach recommended by the Rational Unified Process is generally superior to a linear, or waterfall, approach for a number of reasons:

- It lets you take into account changing requirements. The truth is that requirements usually change. Changing requirements and requirements "creep" have always been primary sources of project trouble, which lead to late delivery, missed schedules, unsatisfied customers, and frustrated developers.
- In the Rational Unified Process, integration is not one "big bang" at the end; instead, elements are integrated progressively. This iterative approach is almost a process of continuous integration. What used to be a lengthy time of uncertainty and pain—taking up to 40% of the effort at the end of a project—is now broken into six to nine smaller integrations that begin with far fewer elements to integrate.
- The iterative approach lets you mitigate risks earlier because integration is generally the only time that risks are discovered or addressed. As you unroll the early iterations, you go through all process components, exercising many aspects of the project, such as tools, off-the-shelf software, people skills, and so on. Perceived risks will prove not to be risks, and new, unsuspected risks will be revealed.
- It provides management with a means of making tactical changes to the product for whatever reason, for example, to compete with existing products. You can decide to release a product early with reduced functionality to counter a move by a competitor, or you can adopt another vendor for a given technology.
- It facilitates reuse because it is easy to identify common parts as they are partially designed or implemented instead of identifying all commonality in the beginning before anything has been designed or implemented. Identifying and developing reusable parts is difficult. Design reviews in early iterations allow architects to identify unsuspected potential reuse and then develop and mature common code for it in subsequent iterations.
- It results in a very robust architecture because you correct errors over several iterations. Flaws are detected even in the early iterations as the product moves beyond inception into elaboration rather than in one massive testing phase at the end. Performance bottlenecks are discovered at a time when they can still be addressed instead of creating panic on the eve of delivery.
- Developers can learn along the way, and their various abilities and specialties are employed more fully during the entire lifecycle. Testers start testing early, technical writers write early, and so on. In a noniterative development, the same people would be waiting around to begin their work, making plan after plan but not making concrete progress. What can a tester test when the product consists only of three feet of design documentation on a shelf? Training needs or the need for additional people is spotted early, during assessment reviews.
- The development process itself can be improved and refined along the way. The assessment at the end of an iteration not only looks at the status of the project from a product/schedule perspective but also analyzes what should be changed in the organization and in the process to make it perform better in the next iteration.

Project managers often resist the iterative approach, seeing it as a kind of endless and uncontrolled hacking. In the Rational Unified Process, the iterative approach is very controlled; iterations are planned in number, duration, and objectives. The tasks and responsibilities of the participants are defined. Objective measures of progress are captured. Some reworking takes place

from one iteration to the next, but this, too, is controlled carefully.

Chapter 4 describes this iterative approach in more detail, and Chapter 7 describes how to manage an iterative process and, in particular, how to plan it.

Requirements Management

Requirements management is a systematic approach to eliciting, organizing, communicating, and managing the changing requirements of a software-intensive system or application.

The benefits of effective requirements management include the following:

- *Better control of complex projects*

  Lack of understanding of the intended system behavior and requirements "creep" are common factors in out-of-control projects.

- *Improved software quality and customer satisfaction*

  The fundamental measure of quality is whether a system does what it is supposed to do. This can be assessed only when all *stakeholders*[2] have a common understanding of what must be built and tested.

  > [2] A stakeholder is any person or representative of an organization who has a stake—a vested interest—in the outcome of a project. A stakeholder can be an end user, a purchaser, a contractor, a developer, a project manager, or anyone else who cares enough or whose needs must be met by the project.

- *Reduced project costs and delays*

  Fixing errors in requirements is very expensive; therefore, decreasing these errors early in the development cycle cuts project costs and prevents delays.

- *Improved team communication*

  Requirements management facilitates the involvement of users early in the process, helping to ensure that the application meets their needs. Well-managed requirements build a common understanding of the project needs and commitments among the stakeholders: users, customers, management, designers, and testers.

In Chapter 9, The Requirements Discipline, we revisit and expand on this important feature of the Rational Unified Process. Chapter 13, The Configuration and Change Management Discipline, discusses the aspects related to tracking changes.

Architecture and Use of Components

Use cases drive the Rational Unified Process throughout the entire lifecycle, but the design activities are centered on the notion of *architecture*—either system architecture or, for software-intensive systems, software architecture. The main focus of the early iterations of the process is to produce and validate a software architecture that, in the initial development cycle, takes the form of an executable architectural prototype that gradually evolves to become the final system in later iterations.

The Rational Unified Process provides a methodical, systematic way to design, develop, and validate an architecture. It offers templates for describing an architecture based on the concept of multiple architectural views. It provides for the capture of

architectural style, design rules, and constraints. The design process component contains specific activities aimed at identifying architectural constraints and architecturally significant elements, as well as guidelines on how to make architectural choices. The management process shows how planning the early iterations takes into account the design of an architecture and the resolution of the major technical risks.

A software *component* can be defined as a nontrivial piece of software, a module, a package, or a subsystem that fulfills a clear function, has a clear boundary, and can be integrated into a well-defined architecture. It is the physical realization of an abstraction in your design. Component-based development can take various flavors:

- In defining a modular architecture, you identify, isolate, design, develop, and test well-formed components. These components can be tested individually and integrated gradually to form the whole system.
- Furthermore, some of these components can be developed to be reusable, especially the components that provide common solutions to a wide range of common problems. These reusable components are typically larger than mere collections of utilities or class libraries. They form the basis of reuse within an organization, thereby increasing overall software productivity and quality.
- More recently, the advent of commercially successful infrastructures that support the concept of software components has launched a whole industry of off-the-shelf components for various domains, allowing developers to buy and integrate components rather than develop them in-house.

The first point exploits the old concepts of modularity and encapsulation, bringing the concepts underlying object-oriented technology a step further. The final two points shift software development from programming software (one line at a time) to composing software (by assembling components).

The Rational Unified Process supports component-based development in several ways:

- The iterative approach allows developers to identify components progressively and decide which ones to develop, which ones to reuse, and which ones to buy.
- The focus on software architecture allows you to articulate the structure. The architecture enumerates the components and the ways they integrate as well as the fundamental mechanisms and patterns by which they interact.
- Concepts such as packages, subsystems, and layers are used during analysis and design to organize components and specify interfaces.
- Testing is organized around single components first and then is gradually expanded to include larger sets of integrated components.

Chapter 5 defines and expands the concept of architecture and its central role in the Rational Unified Process.

Modeling and the UML

A large part of the Rational Unified Process is about developing and maintaining *models* of the system under development. Models help us to understand and shape both the problem and its solution. A model is a simplification of the reality that helps us master a large, complex system that cannot be comprehended in its entirety. We introduce several models in this book: a use-case model (Chapter 6), business models (Chapter 8), and design models and analysis models (Chapter 10).

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard means of writing the system's blueprints, covering conceptual items such as business processes and system functions as well as concrete items such as classes written in a specific programming language, database schemas, and reusable software components.[3]

[3] Grady Booch et al., *UML Users Guide.* Reading, MA: Addison-Wesley, 1998.

The UML is a common language to express the various models, but it does not tell you how to develop software. It provides the vocabulary, but it doesn't tell you how to write the book. That is why Rational has developed the Rational Unified Process hand-in-hand with the UML to complement our work with the UML. The Rational Unified Process is a guide to the effective use of the UML for modeling. It describes which models you need, why you need them, and how to construct them. RUP 2000 and later versions use UML version 1.4.

Configuration and Change Management

Particularly in an iterative development, many work products are modified and modified often. By allowing flexibility in the planning and execution of the development and by allowing the requirements to evolve, iterative development emphasizes the vital issues of tracking changes and ensuring that everything and everyone is in sync. Focused closely on the needs of the development organization, change management is a systematic approach to managing changes in requirements, design, and implementation. It also covers the important activities of keeping track of defects, misunderstandings, and project commitments as well as associating these activities with specific artifacts and releases. Change management is tied to configuration management and to measurements.

Chapter 13, The Configuration and Change Management Discipline, expands on these important aspects of software management and their interrelationships.

Other Key Features of the Rational Unified Process

In addition to these best practices, three important features of the Rational Unified Process are worth mentioning now:

- The role of use cases in driving many aspects of the development
- Its use as a process framework that can be tailored and extended by an adopting organization
- The need for software development tools to support the process

Use-Case-Driven Development

It is often difficult to look at a traditional object-oriented system model and tell how the system does what it is supposed to do. We believe that this difficulty stems from the lack of a consistent, visible thread through the system when it performs certain tasks. In the Rational Unified Process, use cases provide that thread by defining the behavior performed by a system.

Use cases are not required in object orientation, but they provide important links between system requirements and other development artifacts such as design and tests. Other object-oriented methods provide use-case-like representation but use different names for it, such as *scenarios* and *threads.*

The Rational Unified Process is a *use-case-driven* approach, which means that the use cases defined for the system are the foundation for the rest of development process. Use cases play a major role in several of the process disciplines, especially requirements, design, test, and management. Use cases are also key to business modeling.

If you are unfamiliar with the concept of use case, Chapter 6, A Use-Case-Driven Process, introduces it in more detail and shows how use cases are used.

Process Configuration

You could envisage to use one of the predefined configurations of the RUP "as is." But in doing so you run into the risk of doing too much, of getting lost in the details, or of developing artifacts that bring no value to your project in your specific context.

A process should not be followed blindly, generating useless work and producing artifacts that are of little added value. Instead, the process must be made *as lean as possible* while fulfilling its mission to rapidly produce predictably high-quality software. The adopting organization should complement the process with its own best practices and with its specific rules and procedures.

The Rational Unified Process is a process framework that the adopting organization can modify, adjust, and expand to accommodate the specific needs, characteristics, constraints, and history of its organization, culture, and domain. RUP 2003 offers two process *bases*—one for small projects and one for larger and more formal projects—and it brings an array of process components to choose from, to cover various additional disciplines (real-time system design, system engineering, user interface design), various technologies (IBM Websphere, Microsoft .NET, J2EE), programming languages, tools, and techniques.

Chapter 17, Implementing the Rational Unified Process, explains how the process is implemented in an adopting organization.

Tools Support

To be effective, a process must be supported by adequate tools. The Rational Unified Process is supported by a vast palette of tools that automate steps in many activities. These tools are used to create and maintain the various artifacts—models in particular—of the software engineering process, namely, visual modeling, programming, and testing. The tools are invaluable in supporting the bookkeeping associated with the change management and the configuration management that accompany each iteration.

Chapter 3 introduces the concept of *tool mentors*, which provide tool-related guidance. As we go through the various process disciplines in Chapters 7 through 15, we introduce the tools of choice to support the activities of each discipline.

Who Is Using the Rational Unified Process?

Probably about half a million users worldwide working in more than three thousand companies were using the Rational Unified Process in 2003. They were using it in various domains of applications and for large and small projects. This shows the versatility and wide applicability of the Rational Unified Process. Here are some examples of various uses in companies that have offices around the world:

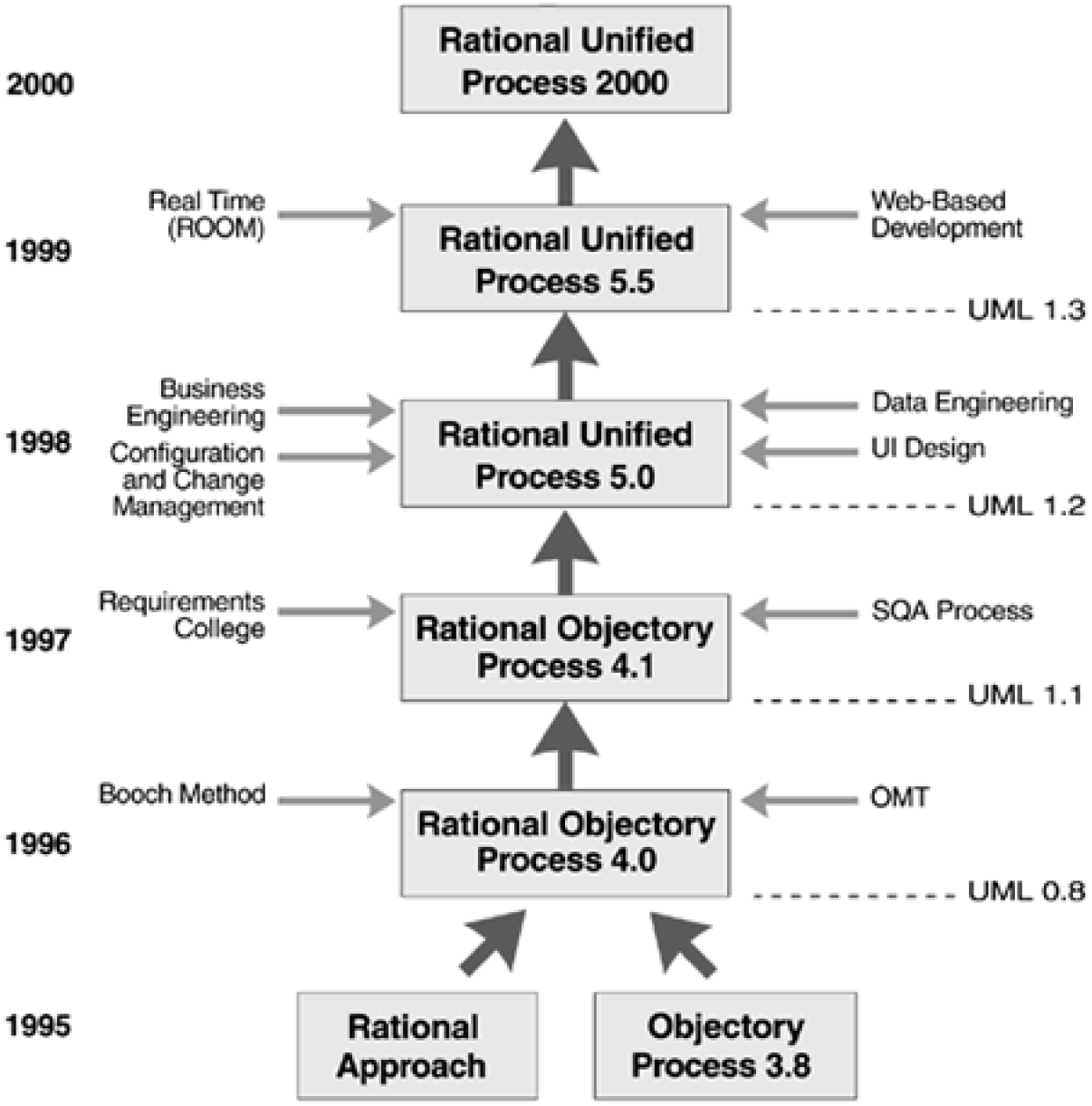
- Telecommunications: Ericsson, Alcatel
- Transportation, aerospace, defense: Lockheed-Martin, British Aerospace
- Manufacturing: Xerox, Volvo, Intel
- Finances: Visa, Merrill Lynch, Schwab
- System integrators: CAP Gemini Ernst & Young, Oracle, Deloitte & Touche

The way these organizations use the Rational Unified Process varies greatly. Some use it very formally; they have evolved their own company process from the Rational Unified Process, which they follow with great care. Others use it more informally, a sort of electronic coach on software engineering, taking the Rational Unified Process as a repository of advice, templates, and guidance, which they use as they go along.

A Brief History of the Rational Unified Process

The Rational Unified Process has matured over the years and reflects the collective experience of the many people and companies that today make up the rich heritage of IBM's Rational Software division. Let us take a quick look at the rich ancestry of RUP 2003, as illustrated in Figure 2-3.

Figure 2-3. Genealogy of the Rational Unified Process



Going backward in time, the Rational Unified Process was brought into the IBM offering by the acquisition of the 20-year-old Rational Software Corporation by IBM Software Group in February 2003. The Rational Unified Process incorporates material in the areas of data engineering, business modeling, project management, and configuration management, the latter as a result of a

merger with Pure-Atria in 1997. It includes elements of the Real-Time Object-Oriented Method, developed by the founders of ObjecTime, acquired by Rational in 2000. It also brings a tighter integration to the Rational Software suite of tools.

The Rational Unified Process is the direct successor to the Rational Objectory Process (ROP), version 4, which was the result of the integration of the Rational Approach and the Objectory Process (version 3.8) after the merger of Rational Software Corporation and Objectory AB in 1995. From its Objectory ancestry, the process has inherited its process model (described in Chapter 3) and the central concept of use case (described in Chapter 6). From its Rational background, it gained the current formulation of iterative development (described in Chapters 4 and 11) and architecture (described in Chapter 5). This ROP version 4 also incorporated material on requirements management from Requisite, Inc., and a detailed test process inherited from SQA, Inc., companies that also were acquired by Rational Software. Finally, this version of the process was the first to use the newly created Unified Modeling Language (UML 0.8).

The Objectory Process was created in Sweden in 1987 by Ivar Jacobson.[4]

[4] Ivar Jacobson et al., *Object-Oriented Software Engineering: A Use-Case-Driven Approach*. Reading, MA: Addison-Wesley, 1992.

Summary

- The Rational Unified Process is a software development process that covers the entire software development lifecycle.
- It is a process product that brings a wealth of knowledge, accumulated over 20 years from a vast range of sources, right to the practitioner's workstation in the form of an online electronic guide.
- It embeds guidance on many modern techniques and approaches: object technology and component-based development, modeling and UML, architecture, iterative development, and so on.
- It is an open process framework that software development organizations can configure and extend to suit their own needs.
- It is not a frozen product; rather, it is alive, constantly maintained, and continuously evolving, with new process components made available online.
- It is based on a solid process architecture, and that allows a development organization to configure and tailor it to suit its needs.
- It supports key best practices for software development.
- It is supported by an extensive palette of tools developed by IBM Rational.

Chapter 3. Static Structure: Process Description

This chapter describes how the Rational Unified Process is represented. We introduce the key concepts of roles, activities, artifacts, workflows, and disciplines as well as other elements used in the process's description.

A Model of the Rational Unified Process

A process describes *who* is doing *what, how*, and *when*. The Rational Unified Process is represented using five primary elements:

- Roles: the *who*
- Activities: the *how*
- Artifacts: the *what*
- Workflows: the *when*
- Disciplines: the "container" for the preceding four kinds of element

The first three elements are shown in Figure 3-1, and Figure 3-6 shows a workflow.

Figure 3-1. Roles, activities, and artifact

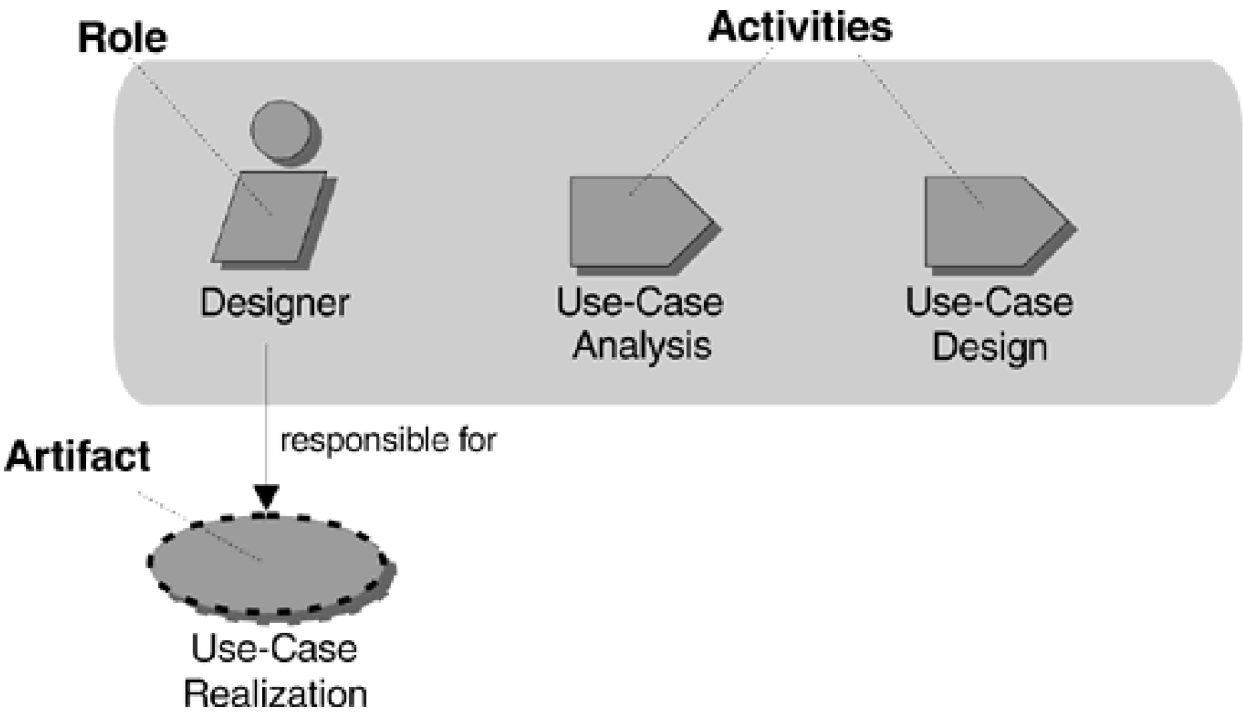
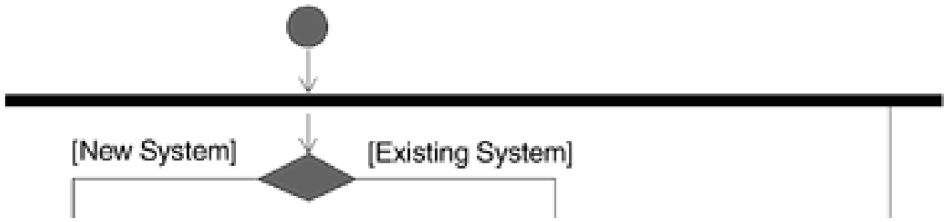
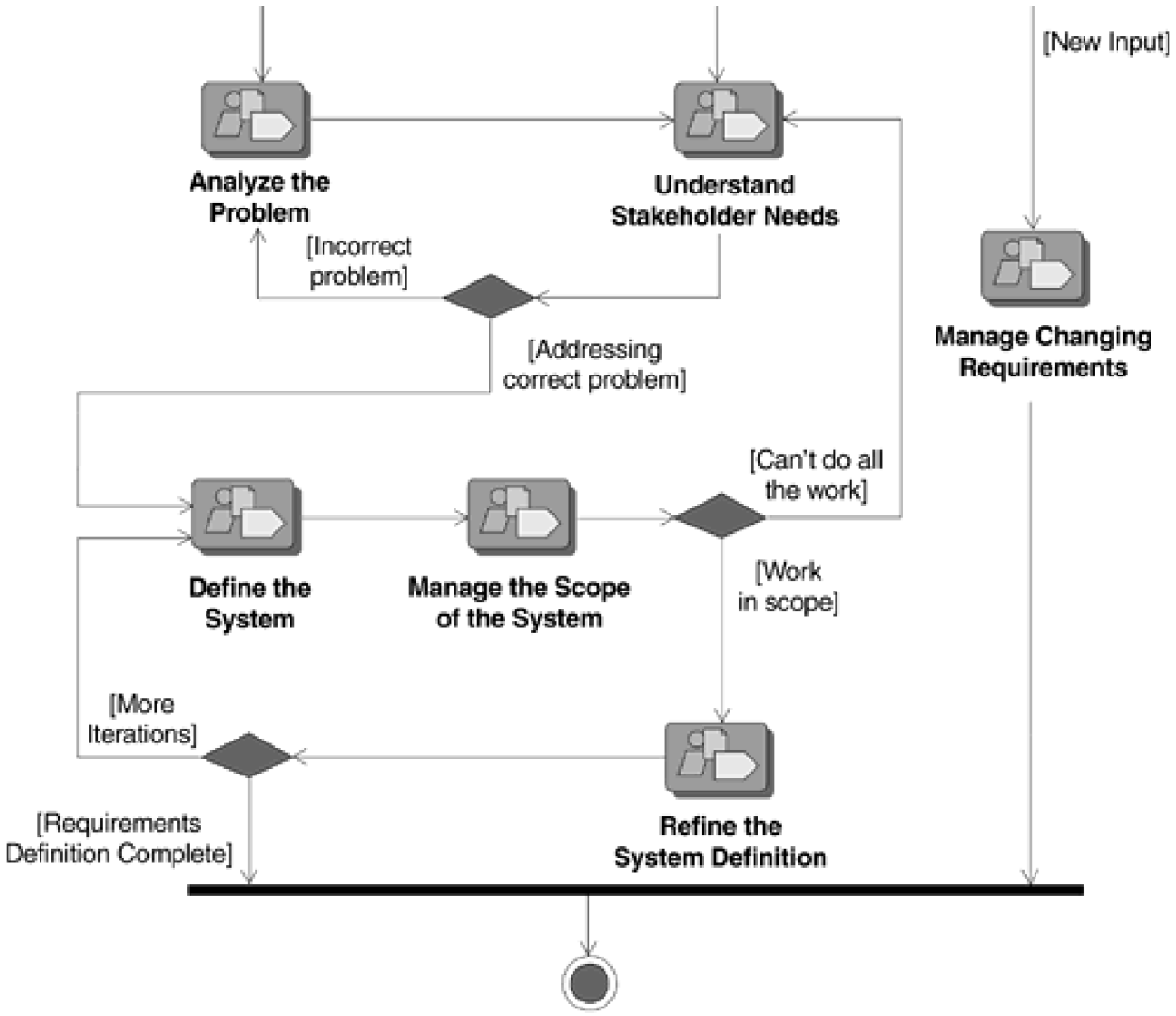


Figure 3-6. Example of a workflow




[New Input]
Analyze the Problem
Understand Stakeholder Needs
[Incorrect problem]
[Addressing correct problem]
Manage Changing Requirements
[Can't do all the work]
Define the System
Manage the Scope of the System
[Work in scope]
[More Iterations]
[Requirements Definition Complete]
Refine the System Definition

Roles

The central concept in the process is that of a role. A *role* defines the behavior and responsibilities of an individual or of a group of individuals working together as a team. The behavior is expressed in terms of *activities* the role performs, and each role is associated with a set of cohesive activities. "Cohesive" in this sense means activities that are best performed by one person. The responsibilities of each role are usually expressed in relation to certain *artifacts* that the role creates, modifies, or controls.

Roles are not individuals, nor job titles. You can play the Project Manager role for an hour, then play the architect role for the rest of the morning, and switch from coder role to tester role during the afternoon. Your colleagues Joe, Stefan, and Mary might all play simultaneously the Design Reviewer role in your review later in the afternoon.

The following are examples of roles:

- *System Analyst*

  An individual acting as a system analyst leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality and delimiting the system.

- *Designer*

  An individual acting as a designer defines the responsibilities, operations, attributes, and relationships of one or more classes and determines how they should be adjusted to the implementation environment.

- *Test Designer*

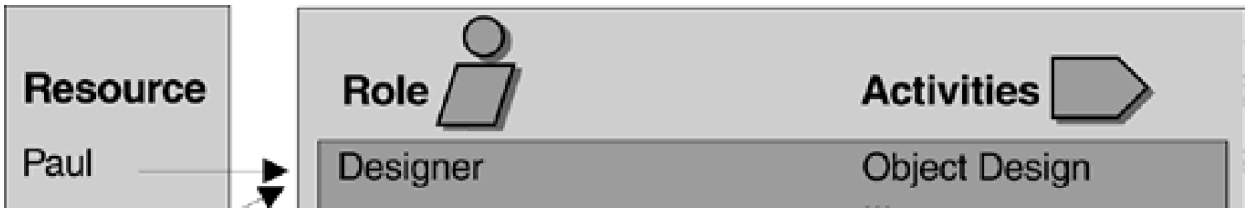
  An individual acting as a test designer is responsible for the planning, design, implementation, and evaluation of tests, including generating the test plan and test model, implementing the test procedures, and evaluating test coverage, results, and effectiveness.

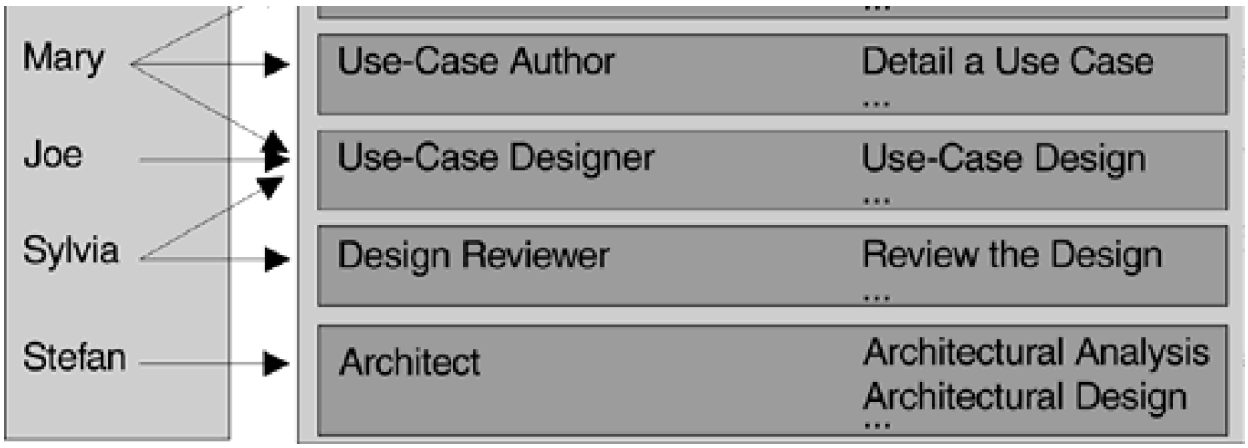
Note that roles are not individuals; instead, they describe how individuals should behave in the business and the responsibilities of each individual. Individual members of the software development organization wear different hats or play different parts or roles.[1] The mapping from individual to role is performed by the project manager when planning and staffing the project, and it is based on the skills and competencies associated with the role and the actual skills and competencies of the members of the team. This mapping allows an individual to act several roles and for a role to be played by several individuals.

> [1] However, we often write, "The designer of class X does this" when, strictly speaking, we should write, "The person playing the designer role for class X does this."

In the example in Figure 3-2, one individual, Sylvia, can be a Database Designer in the morning and act as a Design Reviewer in the afternoon. Paul and Mary are both Designers, although they are likely responsible for different classes or different design packages.

Figure 3-2. People and roles





Each role is associated with a set of skills required to perform the activities of the role. Sylvia must understand how to design a use case and how to review a part of the design.

Roles are organized in five main categories:

1. Analyst roles
2. Developer roles
3. Tester roles
4. Manager roles
5. Production and support roles

There are a few generic roles that are used as placeholders for activities that may be done by a variety of people. See Appendix A for a complete "cast of characters."

Roles are usually denoted in the RUP prefixed with the word *Role,* as in Role: Integration Tester. Appendix A lists all roles defined in the Rational Unified Process.

Activities

Roles have activities, which define the work they perform. An *activity* is a unit of work that an individual in that role may be asked to perform and that produces a meaningful result in the context of the project. The activity has a clear purpose, usually expressed in terms of creating or updating artifacts, such as a model, a class, or a plan. Every activity is assigned to one specific role.

The granularity of an activity is generally a few hours to a few days. It usually involves one person in the associated role and affects one or only a small number of artifacts. An activity should be usable as an element of planning and progress; if it is too small, it will be neglected, and if it is too large, progress will have to be expressed in terms of the activity's parts.

Activities may be repeated several times on the same artifact, especially from one iteration to another as the system is refined and expanded. Repeated activities are performed by the same role but not necessarily the same individual.

In object-oriented terms, the role is an active object, and the activities that the role performs are operations performed by that object. The following are examples of activities:

- *Plan an iteration:* performed by the Role: Project Manager
- *Find use cases and actors:* performed by the Role: System Analyst
- *Review the design:* performed by the Role: Design Reviewer
- *Execute a performance test:* performed by the Role: Performance Tester

Activities are often prefixed with the word *Activity,* as in*Activity: Find use case and actors.* Chapters 7 through 15 give an overview of all activities in the Rational Unified Process.

Activity Steps

Activities are broken into steps. Steps fall into three main categories:

- *Thinking steps*

  The person playing the role understands the nature of the task, gathers and examines the input artifacts, and formulates the outcome.

- *Performing steps*

  The person playing the role creates or updates some artifacts.

- *Reviewing steps*

  The person playing the role inspects the results against some criteria.

Not all steps are necessarily performed each time an activity is invoked, so they can be expressed in the form of alternative flows.

For example, the Activity: Find Use Cases and Actors decomposes into these seven steps:

1. Find actors.
2. Find use cases.
3. Describe how actors and use cases interact.
4. Package use cases and actors.
5. Present the use-case model in use-case diagrams.
6. Develop a survey of the use-case model.
7. Evaluate your results.

The finding part (steps 1 through 3) requires some thinking; the performing part (steps 4 through 6) involves capturing the result in the use-case model; the reviewing part (step 7) requires the role to evaluate the result to assess completeness, robustness, intelligibility, and other qualities.

PREVIOUS NEXT

Artifacts

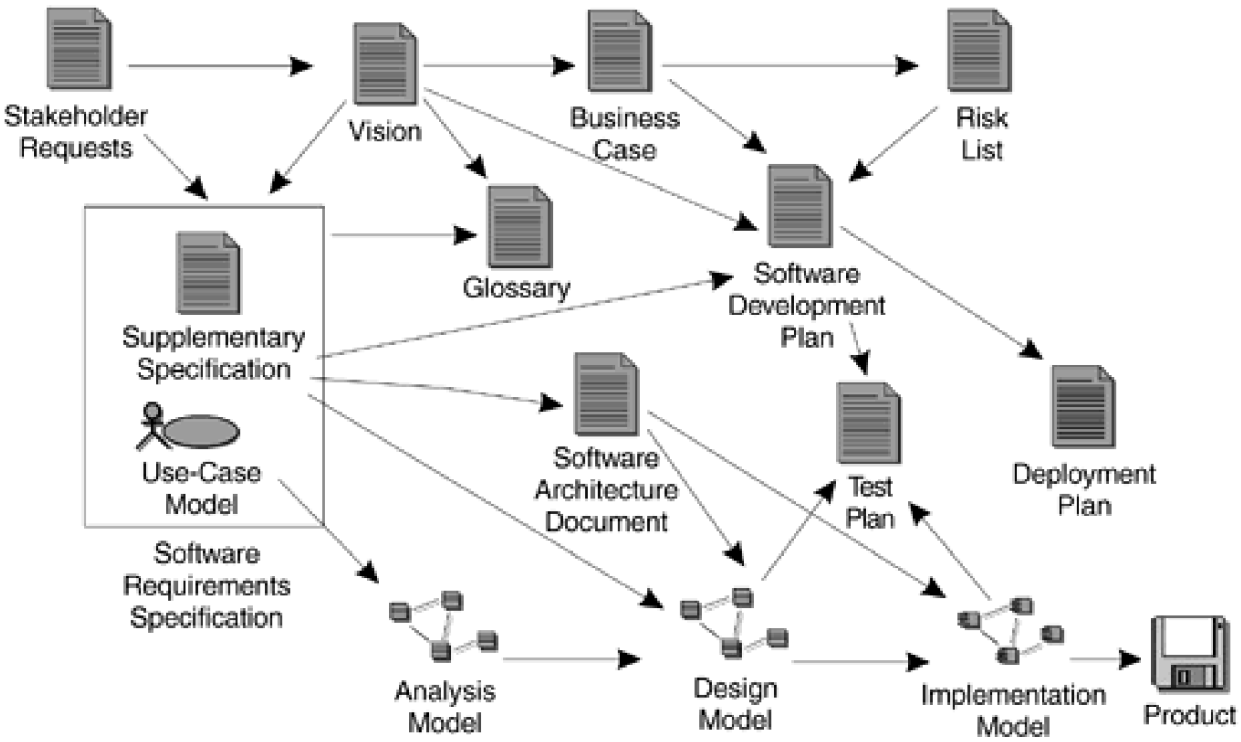
Activities have input and output artifacts. An *artifact* is a piece of information that is produced, modified, or used by a process. Artifacts are the tangible products of the project: the things the project produces or uses while working toward the final product. Artifacts are used as input by roles to perform an activity and are the result or output of such activities. In object-oriented design terms, just as activities are operations on an active object (the role), artifacts are the parameters of these activities.

Artifacts take various shapes or forms:

- A model, such as the use-case model or the design model
- A model element—an element within a model—such as a class, a use case, or a subsystem
- A document, such as a business case or software architecture document
- Source code
- Executables

Note that *artifact* is the term used in the Rational Unified Process. Other processes use terms such as *work product, work units*, *deliverables*, and so on, to denote the same thing. Deliverables in RUP are only the subset of all artifacts that end up in the hands of the customers and end users. Figure 3-3 shows some of the major artifacts of the Rational Unified Process.

Figure 3-3. Major artifacts of the Rational Unified Process



Artifacts can also be composed of other artifacts. For example, the design model contains many classes; the software development plan contains several other plans: a staffing plan, a phase plan, a measurement plan, iteration plans, and so on.

Artifacts are very likely to be subject to version control and configuration management. Sometimes this can be achieved only by versioning the container artifact when it is not possible to do it for the elementary, contained artifacts. For example, you may control the versions of a whole design model or design package and not the individual classes they contain.

Typically, artifacts are *not* documents. Many processes place an excessive focus on documents and in particular on paper documents. The Rational Unified Process discourages the systematic production of paper documents. The most efficient and pragmatic approach to managing project artifacts is to maintain the artifacts *within* the appropriate tool used to create and manage them. When necessary, you can generate documents (snapshots) from these tools on a just-in-time basis.

You should also consider delivering artifacts to the interested parties inside and together with the tool rather than on paper. This approach ensures that the information is always up-to-date and is based on actual project work, and producing it shouldn't require additional effort.

The following are examples of artifacts:

- A design model in UML stored in Rational XDE
- A project plan stored in Microsoft Project
- A defect stored in Rational ClearQuest
- A project requirements database in Rational Requisite Pro

However, certain artifacts must be plaintext documents, as in the case of external input to the project, and sometimes plaintext is simply the best means of presenting descriptive information.

To promote the idea that every piece of information should be the responsibility of a specific person, artifacts are the responsibility of a single role. Even though one person may "own" the artifact, many people may use this artifact, perhaps even updating it if they have been given permission.

Artifacts are denoted in the process prefixed with the word *artifact,* as in Artifact: Use-Case Storyboard.

Reports

Models and model elements can have associated reports. A *report* extracts information about models and model elements from a tool. For example, a report presents an artifact or a set of artifacts for a review. Unlike regular artifacts, reports are not subject to version control. You can reproduce them at any time by going back to the artifacts that generated them.

Sets of Artifacts

The artifacts of the Rational Unified Process have been organized into *information sets* that are aligned with the core disciplines.

The *management set* groups all artifacts related to the software business and to the management of the project:

- Planning artifacts, such as the software development plan (SDP), the business case, the actual process instance used by the project (the development case), and so on
- Operational artifacts, such as a release description, status assessments, deployment documents, and defect data

The *requirements set* groups all artifacts related to the definition of the software system to be developed:

- The vision document
- Requirements in the form of stakeholders' needs, use-case model, and supplementary specification

The *design set* contains a description of the system to be built (or as built) in these forms:

- The design model
- The architecture description

The *implementation set* includes these elements:

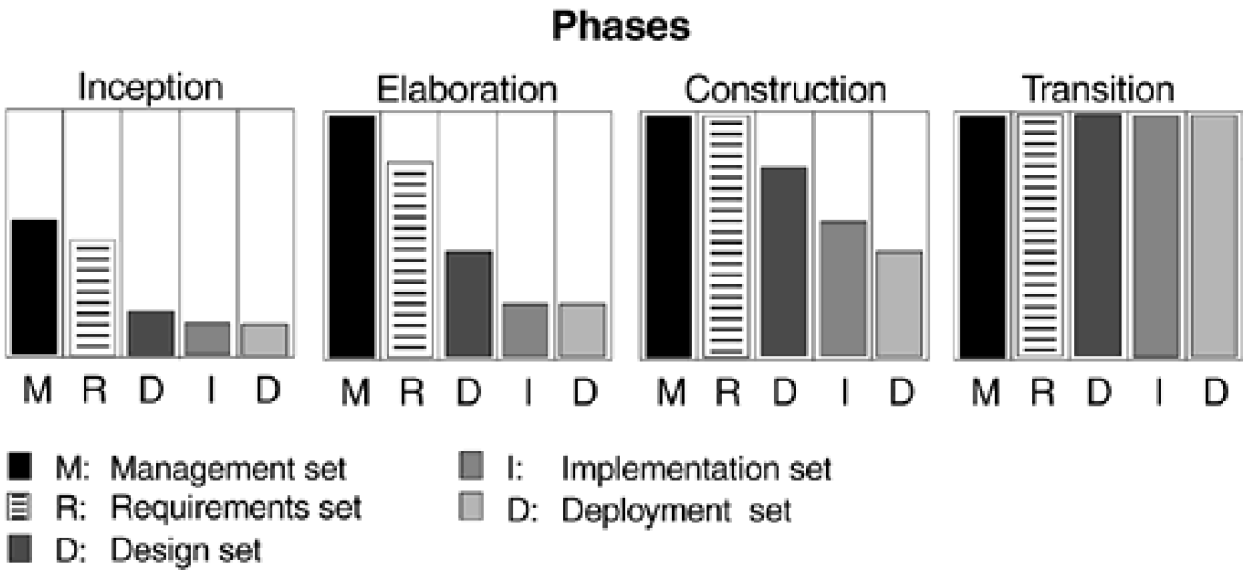
- The source code and executables
- The associated data files or the files needed to produce them

The *deployment set* contains all the information delivered, including the following:

- Installation material
- User documentation
- Training material

In an iterative development process, the various artifacts are not produced, completed, or even frozen in one phase before you move to the next phase. On the contrary, the information sets evolve throughout the development cycle. They are *grown,* as depicted in Figure 3-4. Appendix B lists all artifacts defined in the Rational Unified Process, organized by discipline.

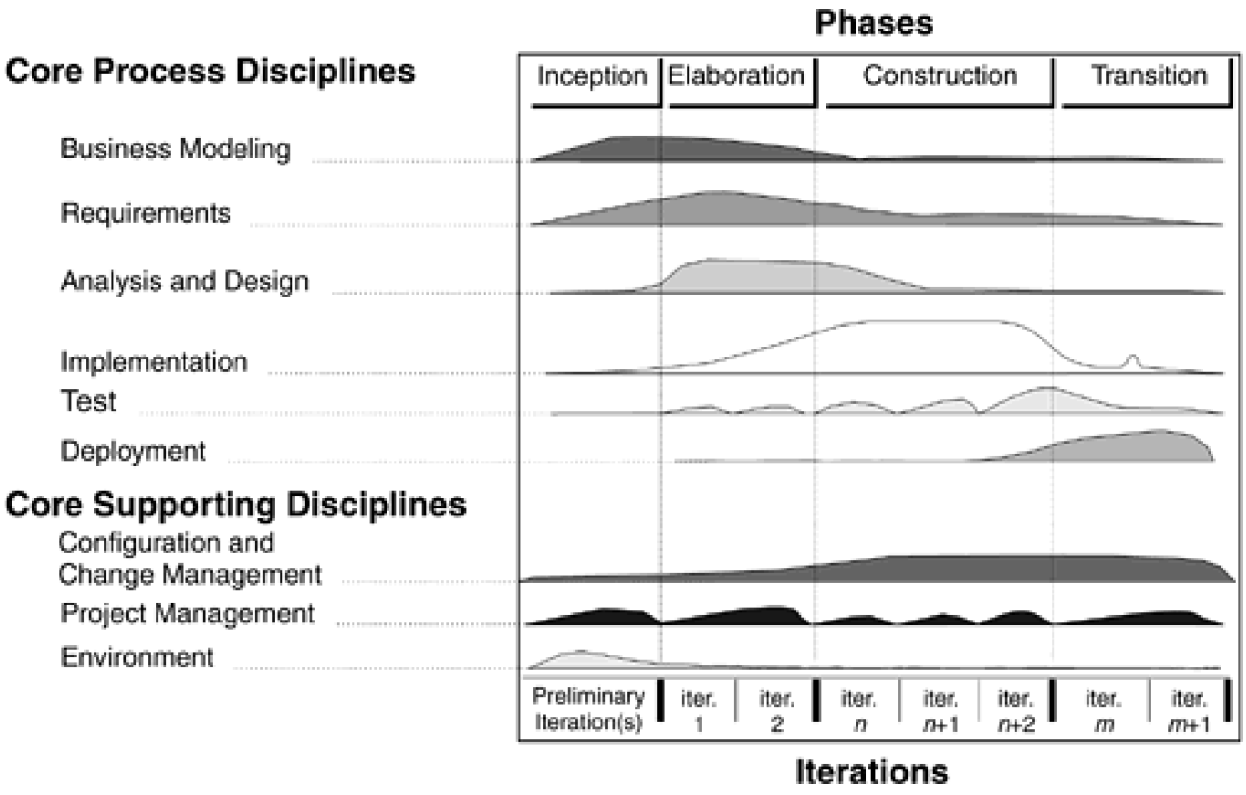
Figure 3-4. Growing the information sets



Disciplines

Disciplines are "containers" used to organize activities of the process. There are nine main disciplines in the Rational Unified Process, and they represent a partitioning of all roles and activities into logical groupings by areas of concern or specialty (see Figure 3-5). The nine core disciplines are divided into six technical disciplines and three supporting disciplines.

Figure 3-5. Nine core disciplines



The technical disciplines are as follows:

- Business modeling discipline
- Requirements discipline
- Analysis and design discipline
- Implementation discipline
- Test discipline
- Deployment discipline

The supporting disciplines are as follows:

- Project management discipline

- Configuration and change management discipline
- Environment discipline

Although the names of the six technical disciplines may evoke the sequential phases in a traditional waterfall process, you will see in Chapter 4 that the *phases* of an iterative process are quite different and that these disciplines and the activities and workflows they contain are revisited again and again throughout the lifecycle. The nine main disciplines are discussed in detail in Chapters 7 through 15.

PREVIOUS NEXT

Workflows

A mere enumeration of all roles, activities, and artifacts does not quite constitute a process. We need a way to describe meaningful sequences of activities that produce some valuable result and to show interactions between and among roles. A *workflow* is a sequence of activities that produces a result of observable value. In UML terms, a workflow can be expressed as a sequence diagram, a collaboration diagram, or an activity diagram. We use a form of activity diagrams in this book. Figure 3-6 is an example of a workflow.

Note that it is rarely possible or practical to represent all the dependencies between activities. Often, two activities are more tightly interwoven than shown, especially when they involve the same role or the same individual. People are not machines, and the workflow cannot be interpreted literally as a program that people are to follow exactly and mechanically.

The Rational Unified Process uses three types of workflow:

- Core workflows, associated to each discipline
- Workflow details, to refine the core workflow
- Iteration plans

Core Workflows

In each discipline there is a *core workflow* that gives the overall flow of activities.

Workflow Details

Each core workflow covers a lot of ground. To break them down, we use *workflow details* to express a specific group of closely related activities. For example, the activities are performed together or in a cyclical fashion; they are performed by a group of people working together in a workshop; or they produce an interesting intermediate result. Workflow details also show information flows—the artifacts that are input to and output from the activities—to show how activities interact through the various artifacts.

Iteration Plans

Iteration plans are another means of presenting the process, describing it more from the perspective of what happens in a typical iteration. They are actually the closest to what a workflow engine would handle. You can look at them as instantiations of the process for one given iteration, selecting the activities that will be effectively run during the iteration and replicating them as necessary. There are an infinite number of ways you can instantiate the process. The Rational Unified Process contains descriptions of a few typical iteration plans. They are given primarily for pedagogical purposes, as you can see with the few

examples given in Chapter 16.

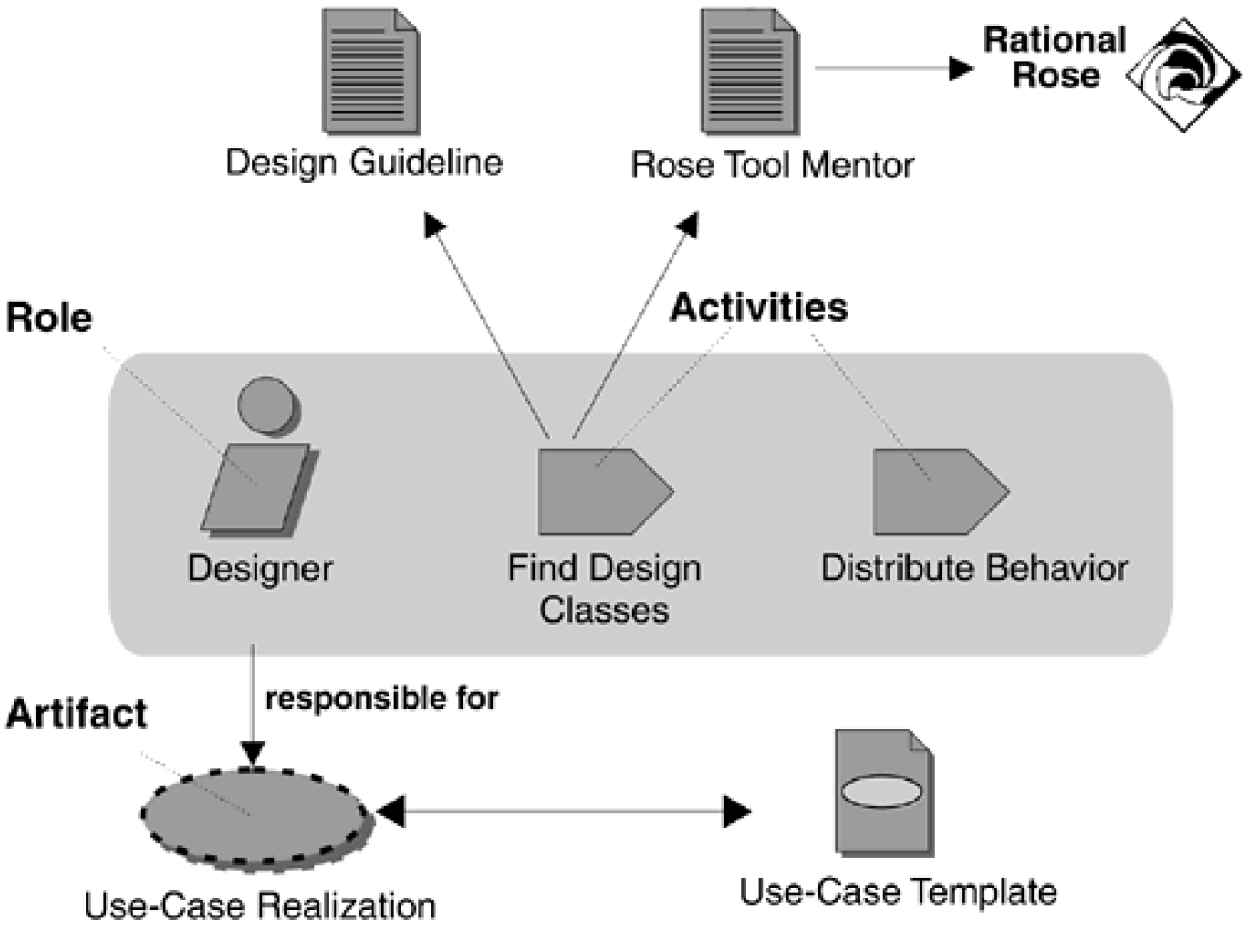
Additional Process Elements

Roles, activities, workflows, and artifacts, organized in disciplines, represent the backbone of the Rational Unified Process static structure. But other elements are added to activities or artifacts to make the process easier to understand and use and to provide more comprehensive guidance to the practitioners. These additional process elements are:

- Guidelines
- Templates
- Tool mentors
- Concepts

These elements enhance the primary elements, as shown in Figure 3-7.

Figure 3-7. Adding templates, tool mentors, and guidelines



Guidelines

Activities and steps are kept brief and to the point because they are intended to serve as references for what needs to be done. Therefore, they must be useful for neophytes looking for guidance as well as for experienced practitioners needing a reminder.

Attached to activities, steps, or artifacts are *guidelines.* Guidelines are rules, recommendations, or heuristics that support activities and steps. They describe well-formed artifacts, focusing on their specific qualities, for example, what constitutes a good use case or a good design class. Guidelines also describe specific *techniques* to create certain artifacts or the transformations from one artifact to another or the use of the Unified Modeling Language (UML). Guidelines are used also to assess the quality of artifacts—in the form of checklists associated with artifacts—or to review activities. The following are types of guidelines:

- Work guidelines, which give practical advice on how to undertake an activity, especially for group activities; for example:
    - Work guidelines for reviews
    - Work guidelines for a use-case workshop
    - Work guidelines for programming, such as programming in pairs
- Artifact guidelines are associated with a particular artifact and describe how to develop, evaluate, and use the artifact; for example:
    - Modeling guidelines, which describe well-formed modeling elements such as use cases, classes, and test cases
    - Programming guidelines, for languages such as Java, C++, or Ada, describing well-formed programs
    - User-interface guidelines
    - Guidelines on how to create a specific artifact, such as a risk list or an iteration plan
    - Checkpoints to be used as part of a review to verify that an activity is complete

Some guidelines need to be refined or specialized for a given organization or project to accommodate project specifics, such as the use of a particular technique or tool. The following are examples of this latter type of guideline:

- User-interface guidelines, such as a description of the windowing style specific to a project: color palette, fonts, gallery of icons, and so on
- Programming guidelines, such as a description of naming conventions specific to the project

Templates

Templates are models, or prototypes, of artifacts. Associated with the artifact description are one or more templates that can be used to create the corresponding artifacts. Templates are linked to the tool that is to be used. For example:

- Microsoft Word templates for documents and some reports
- SoDA templates for Microsoft Word or FrameMaker that extract information from tools such as Rational Rose, RequisitePro, or TeamTest
- HTML templates for the various elements of the process
- Microsoft Project template for the project plan and iteration plan

As with guidelines, organizations may want to customize the templates before using them by adding the company logo, some project identification, or information specific to the type of project.

Tool Mentors

Activities, steps, and associated guidelines provide general guidance to the practitioner. To go one step further, *tool mentors* are an additional means of providing guidance by showing you *how* to perform the steps using a specific software tool. Tool mentors are provided in the Rational Unified Process, linking its activities with tools such as Rational Rose, Rational XDE, RequisitePro, ClearCase, ClearQuest, and TestStudio. The tool mentors almost completely encapsulate the dependencies of the process on the tool set, keeping the activities free from tool details. A development organization can extend the concept of tool mentor to provide guidance for other tools.

Concepts

Some of the key concepts, such as iteration, phase, artifact, risk, performance testing, and so on, are introduced in separate sections of the process, usually attached to the most appropriate discipline. Many of these concepts are also introduced in this book.

A Process Framework

With this structure, the Rational Unified Process constitutes a process *framework*. Roles, artifacts, activities, guidelines, concepts, and mentors are the elements that you can add or replace to evolve or adapt the process to the organization's needs. They are the building blocks of the knowledge that you can use to capture and package process know-how in the form of process components and "compile" those in process plug-ins. The way to do this is developed in Chapter 14, The Environment Discipline.

The organization of the Rational Unified Process that we described in this chapter is based on an industry standard called the *Software Process Engineering Metamodel (SPEM),*[2] developed around 2000 by a group of companies, including Rational Software and IBM. SPEM is an object-oriented metamodel in UML, as well as a UML profile.

[2] Object Management Group. *Software Process Engineering Metamodel (SPEM)*. OMG, doc ad/01-03-08, April 2, 2001.

Summary

- The Rational Unified Process model is built on three fundamental entities: roles, activities, and artifacts.
- Disciplines are natural groupings of process activities, roles, and artifacts.
- Workflows relate activities, artifacts, and roles in sequences that produce valuable results.
- Guidelines, templates, and tool mentors complement the description of the process by providing detailed guidance to the practitioner.
- The Rational Unified Process is an open process framework based on an industry standard called SPEM.

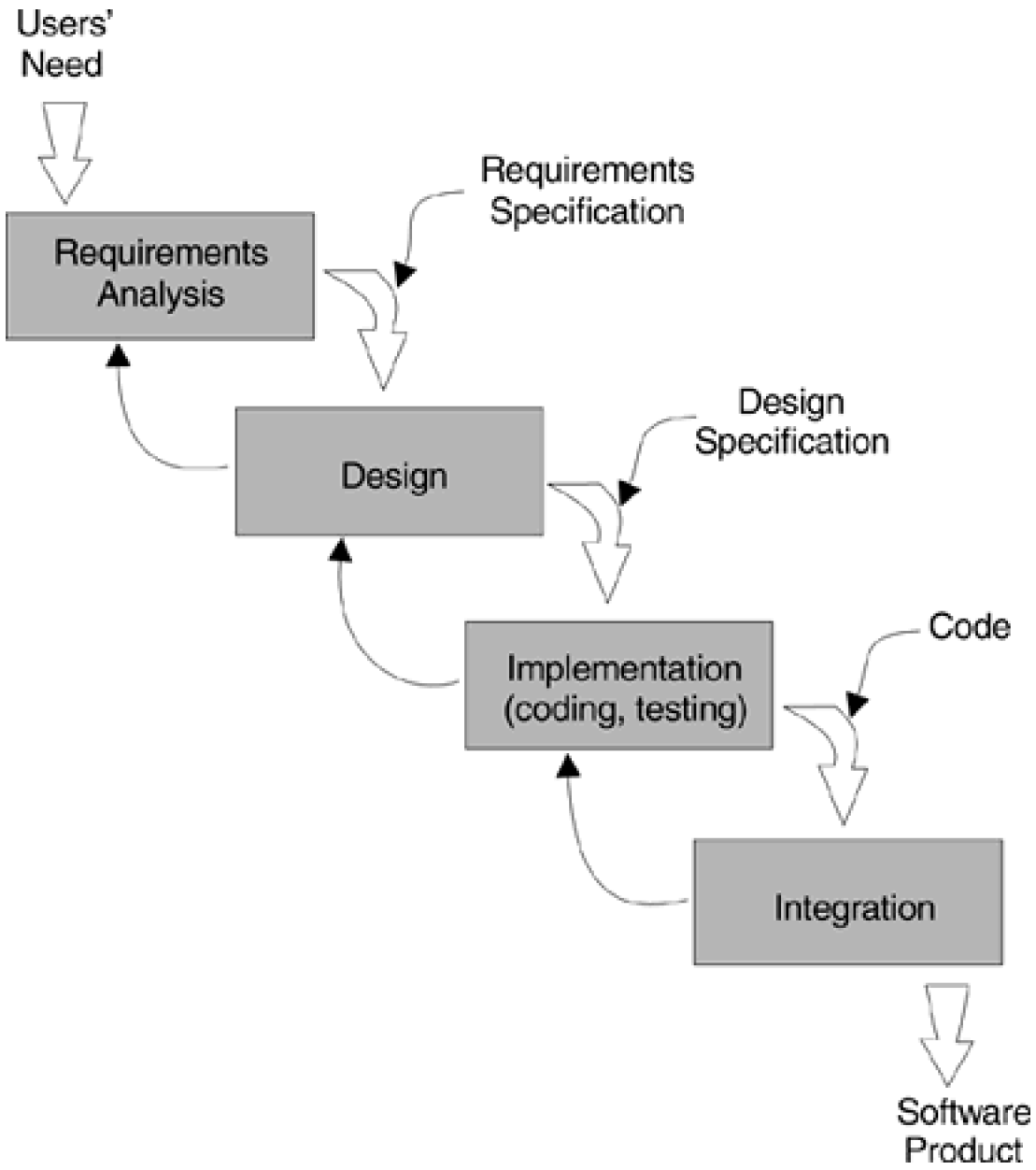
Chapter 4. Dynamic Structure: Iterative Development

This chapter describes the lifecycle structure of the Rational Unified Process—that is, how the process rolls out over time. We introduce the concept of iterative development, with its phases, milestones, and iterations, as well as the factors that drive the process: risk mitigation and incremental evolution.

The Sequential Process

It has become fashionable to blame many problems and failures in software development on the sequential, or waterfall, process depicted in Figure 4-1. This is rather surprising because at first this method seems like a reasonable approach to system development.

Figure 4-1. The sequential process



A Reasonable Approach

Many engineering problems are solved using a sequential process, which typically goes through the following five steps:

1. Completely understand the problem to be solved, its requirements, and its constraints. Capture them in writing and get all interested parties to agree that this is what they need to achieve.
2. Design a solution that satisfies all requirements and constraints. Examine this design carefully and make sure that all interested parties agree that it is the right solution.
3. Implement the solution using your best engineering techniques.
4. Verify that the implementation satisfies the stated requirements.
5. Deliver. Problem solved!

That is how skyscrapers and bridges are built. It's a rational way to proceed but only because the problem domain is relatively well known; engineers can draw on hundreds of years of experimentation in design and construction.

By contrast, software engineers have had only a few decades to explore their field. Software developers worked very hard, particularly in the seventies and eighties, to accumulate experimental results in software design and construction. In 1980 I would have sworn that the sequential process was the one and only reasonable approach.

If the sequential process is ideal, however, why aren't the projects that use it more successful? There are many reasons:

- We made the wrong assumptions.
- The context of software development is somewhat different from that of other engineering disciplines.
- We have failed to incorporate some human factors.
- We have tried to stretch an approach that works in certain well-defined circumstances beyond what it can bear.
- We are still only in the exploratory phase of software engineering. We do not have the experience of hundreds of years of trial and error that makes building a bridge appear to be a mechanical process. This is the primary reason.

Let us review two fundamentally wrong assumptions that often hinder the success of software projects.

Wrong Assumption 1: Requirements Will Be Frozen

Notice that in the description of the sequential process we assume in step 1 that we can capture the entire problem at the beginning. We assume we can nail down all the requirements in writing in an unambiguous fashion and begin the project with a stable foundation. Despite all our efforts, though, this almost always proves to be impossible. Requirements will change. We must accept this fact. Unless we are solving a trivial problem, new or different requirements will appear. Requirements change for many reasons. Let's look at a few:

- *The users change.*

  The users' needs cannot be frozen in time. This is especially true when the development time is measured not in weeks or months but in years. Users see other systems and other products, and they want some of the features they see. Their own work environment evolves, and they become better educated.

- *The problem changes.*

  After the system is implemented or while it is being implemented, the system itself affects the perspective of users.

Trying out features or seeing them demonstrated is quite different from reading about them. As soon as the end users see how their intentions have been translated into a system, the requirements change. In fact, the one point when users know exactly what they want is not two years before the system is ready but rather a few weeks or months *after* delivery of the system when they are beyond the initial learning phase. This is known as the IKIWISI effect: "I'll Know It When I See It."[1]

[1] Origin uncertain, sometimes attributed to U.S. Judge Stewart Potter; Barry Boehm used this acronym at a workshop on software architecture at the University of Southern California in 1997.

Users don't really know what they want, but they know what they do not want when they see it. Therefore, efforts to detail, capture, and freeze the requirements may ultimately lead to the delivery of a perfect system with respect to the requirements but the wrong system with respect to the real problem at the time of delivery.

- *The underlying technology changes.*

  New software or hardware techniques and products emerge, and you will want to exploit them. On a multiyear project, the hardware platform bid at the beginning of the project may no longer be manufactured at delivery time.

- *The market changes*.

  The competition might introduce better products to the market. What is the point of developing the perfect product relative to the original spec if you end up with the wrong product relative to what the marketplace expects when you are finally finished?

- *We cannot capture requirements with sufficient detail and precision.*

  Formal methods have held the promise of a solution, but at the beginning of the third millennium, they have not gained significant acceptance in the industry except in small, specialized domains. They are hard to apply and very user-unfriendly. Try teaching temporal logic or colored Petri nets to an audience of bank tellers and branch managers so that they can read and approve the specification of their new system.

Wrong Assumption 2: We Can Get the Design Right on Paper before Proceeding

The second step of the sequential process assumes that we can confirm that our design is the right solution to the problem. By "right" we imply all the obvious qualities: correctness, efficiency, feasibility, and so on. With complete requirements tracing, formal derivation methods, automated proof, generator techniques, and design simulation, some of these qualities can be achieved. Few of these techniques are readily available to practitioners, however, and many of them require that you begin with a formal definition of the problem. You can accumulate pages and pages of design documentation and hundreds of blueprints and spend weeks in reviews, only to discover late in the process that the design has major flaws that cause serious breakdowns.

Software engineering has not reached the level of other engineering disciplines (and perhaps it never will) because the underlying "theories" are weak and poorly understood, and the heuristics are crude. Software engineering may be misnamed. At various times it more closely resembles a branch of psychology, sociology, philosophy, or art than engineering. Relatively straightforward laws of physics underlie the design of a bridge, but there is no strict equivalent in software design. Software is "soft" in this respect.

Bringing Risks into the Picture

The sequential, or waterfall, process does work. It has worked fine for me on small projects lasting from a few weeks to a few months, on projects in which we could clearly anticipate what would happen, and on projects in which all hard aspects were well understood. For projects having little or no novelty, you can develop a plan and execute it with little or no surprise. If the current project is somewhat like the one you completed last year—and the one the year before—and if you use the same people, the same tools, and the same design, the sequential approach will work well.

The sequential process breaks down when you tackle projects that have a significant level of novelty, unknowns, and risks. You cannot anticipate the difficulties you may encounter, let alone how you will address them. The only thing you can do is to build some slack into the schedule and cross your fingers.

The absence of fundamental "laws of software" to match the fundamental laws of physics that support other engineering disciplines, and the pace at which software evolves make it a risky domain. Techniques for reinforcing concrete have not changed dramatically since my grandfather used them in the early twenties in an engineering bureau. Software tools, techniques, and products, on the other hand, have a lifetime of a few years at best. So every time we try to build a system that is a bit more complicated, somewhat larger, or a little more challenging, we are in dangerous and risky territory, and we must take this into account.

That's why we bring risk analysis into the picture.

Stretching the Time Scale

If you stretch what works for a three-month project to fit a three-year project, you expose the project not only to the changing contexts we have discussed but also to other subtle effects related to the people involved. Software developers who know that they will see tangible results within the next two to three months can remain well focused on the real outcome. Very quickly, they will get feedback on the quality of their work. If small mistakes are discovered along the way, the developers won't have to go very far back in time to correct them.

But imagine the mindset of developers in the middle of the design phase of a three-year project. The target is to finish the design within four months. In a sequential process, the developers may not even be around to see the final product up and running. Progress is measured in pages or diagrams and not in operational features. There is nothing tangible, nothing to get the adrenaline flowing.

There is little feedback on the quality of the current activity because defects will be found later, during integration or test, perhaps 18 months from now. The developers have few opportunities to improve the way they work. Moreover, strange things discovered in the requirements text mean that developers must revisit discussions and decisions made months ago. Is it any wonder that they have a hard time staying motivated? The original protagonists are no longer in the project, and the contract with the customer is as hard and inflexible as a rock.

The developers have only one shot at each kind of activity, with little opportunity to learn from their mistakes. You have one shot at design, and it had better be good. You say you've never designed a system like this? Too bad! You have one shot at coding, and it had better be good. You say this is a new programming language? Well, you can work longer hours to learn its new features. There's only one shot at testing, and it had better be a no-fault run. You say this is a new system and no one really knows how it's supposed to work? Well, you'll figure it out. If the project introduces new techniques or new tools or new people, the sequential process gives you no latitude for learning and improvement.

Pushing Paperwork on the Shelves

In the sequential process, the goal of each step except the last one is to produce and complete an intermediate artifact (often a paper document) that is reviewed, approved, frozen, and then used as the starting point for the next step. In practice, sequential processes place an excessive emphasis on the production and freezing of documents. Some limited amount of feedback to the preceding step is tolerated, but feedback on the results of earlier steps is seen as disruptive. This is related to the reluctance to

change requirements and to the loss of focus on the final product that is often seen during long projects.

Volume-Based versus Time-Based Scheduling

Often, timeliness is the most important factor in the success of a software project. In many industries, delivery of a product on time and with a short turnaround for new features is far more important than delivery of a complete, full-featured, perfect system. To achieve timeliness, you must be able to adjust the contents dynamically by dropping or postponing some features to deliver incremental value on time. With a linear approach, you do not gain much on the overall schedule if you decide in the middle of the implementation to drop feature X. You have already expended the time and effort to specify, design, and code the feature. That's why this model isn't suitable when a company wants to work with schedules that are time-based (for example, in three months we can do the first three items on your list, and three months later we'll have the next two, and so on) and not volume-based (it will take us nine months to do everything that you want).

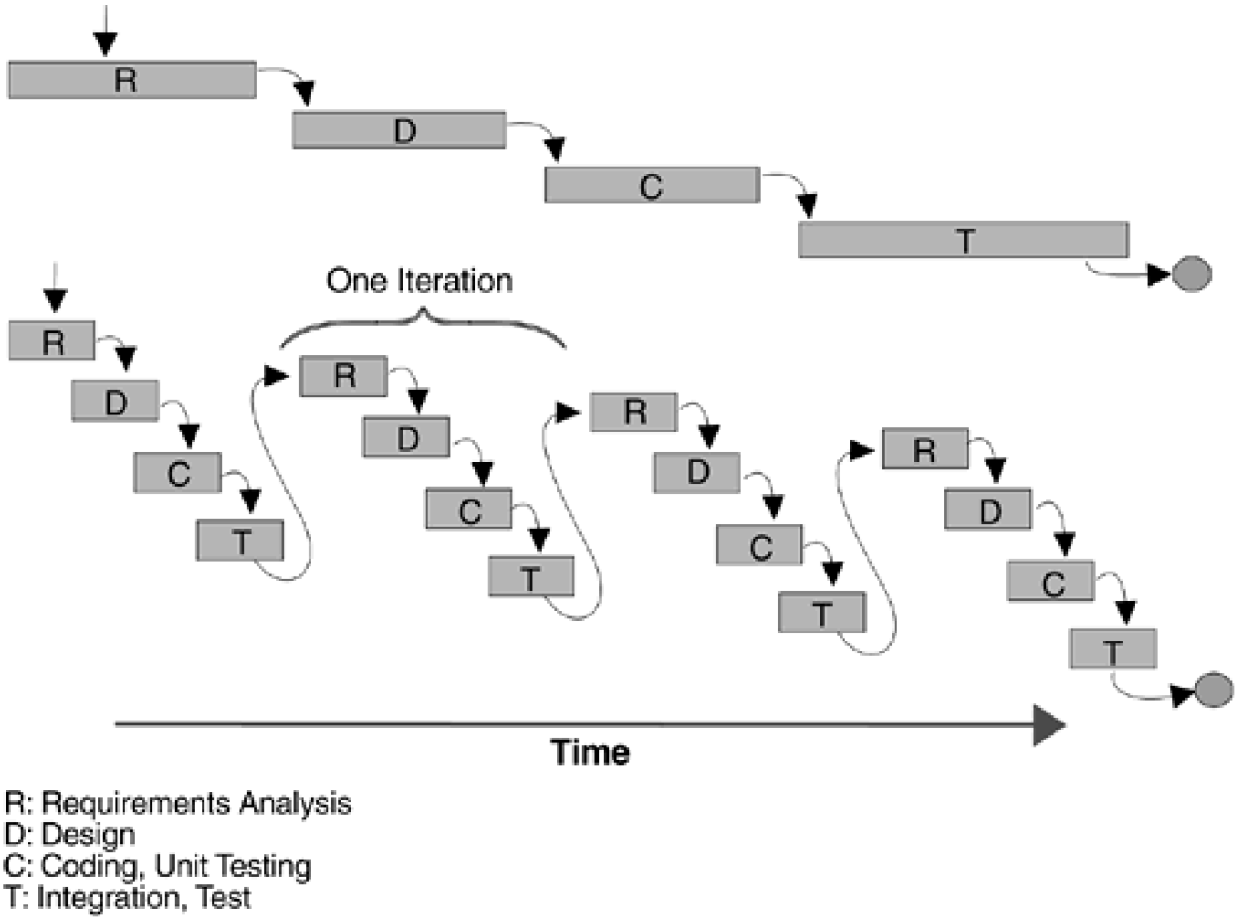
For these reasons and a few others that we will cover later, software organizations have tried another approach.

Overcoming Difficulties: Iterate!

How do you eat an elephant? One bite at a time! If the sequential, or waterfall, approach is reasonable and even successful for short projects or those with a small amount of novelty or risk, why not break down the lifecycle of a large project into a succession of small waterfall projects? In this way, you can address some requirements and some risks, design a little, implement a little, validate it, and then take on more requirements, design some more, build some more, validate, and so on, until you are finished. This is the *iterative* approach.

Figure 4-2 compares the iterative approach and the sequential approach. It shows four phases of the sequential process, then it shows what things look like in an iterative process for one development cycle—from the initial idea to the point when a complete, stable, quality product is delivered to the end users.

Figure 4-2. From a sequential to an iterative lifecycle



Do not take this too literally, however; an iteration is not quite like a single waterfall project, and activities are sequenced in a more opportunistic, tactical, adaptive fashion.

The iterative technique is easy to illustrate but not very easy to achieve. It raises at first more questions than it answers:

- How does this work converge to become a product? How do you avoid having each iteration start over from scratch?

- How do you select what to do in each iteration? Which requirements do you consider, and which risks do you address?
- How does this approach solve the major issues we identified earlier?

The Rational Unified Process answers these questions. We'll discuss the answers in the rest of this chapter and further in Chapter 7, The Project Management Discipline.

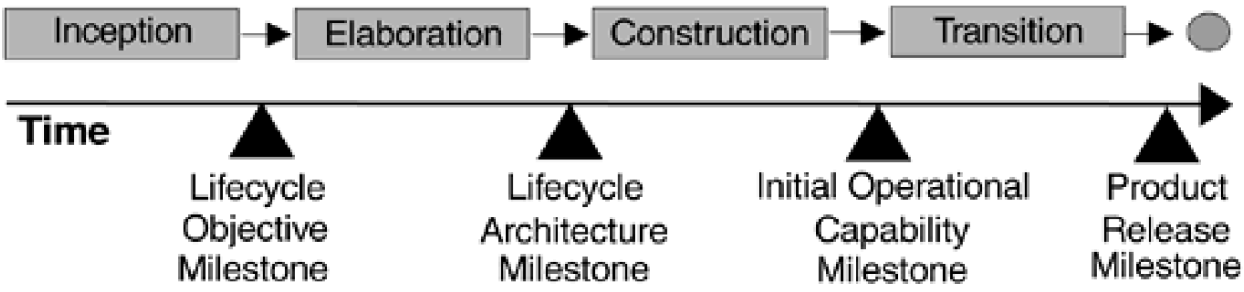
Gaining Control: Phases and Milestones

From a project management perspective, we need a way to assess progress to ensure that we are not wandering aimlessly from iteration to iteration but are actually converging on a product. From a management perspective, we must also define points in time to operate as gating functions based on clear criteria. These *milestones* provide points at which we can decide to proceed, abort, or change course. Finally, we must partition and organize the sequence of iterations according to specific short-term objectives. Progress will be measured in the number of use cases or features completed, as well as test cases passed, performance requirements satisfied, and above all, risks eliminated.

The iterative process is organized in phases, as shown in Figure 4-3. But unlike the steps in the waterfall approach, the phases here are not the traditional sequence of requirements analysis, design, coding, integration, and test. They are completely orthogonal to the traditional phases. Each phase is concluded by a major milestone.[2]

[2] The milestone definition and names are identical to the ones proposed by Barry Boehm in the article, "Anchoring the Software Process," *IEEE Software*, July 1996, pp. 73–82, and not by accident.

Figure 4-3. The four phases and milestones of the iterative process



Let's look at the four phases in more detail.

- *Inception*

  The good idea—specifying the end-product vision and its business case and defining the scope of the project.[3] The inception phase is concluded by the lifecycle objective (LCO) milestone.

  [3] *American Heritage Dictionary* defines *inception* as "the beginning of something, such as an undertaking, a commencement."

- *Elaboration*

  Planning the necessary activities and required resources; specifying the features and designing the architecture.[4] The elaboration phase is concluded by the lifecycle architecture (LCA) milestone.

  [4] *American Heritage Dictionary* defines *elaboration* as the process "to develop thoroughly, to express at greater length or greater detail."

- *Construction*

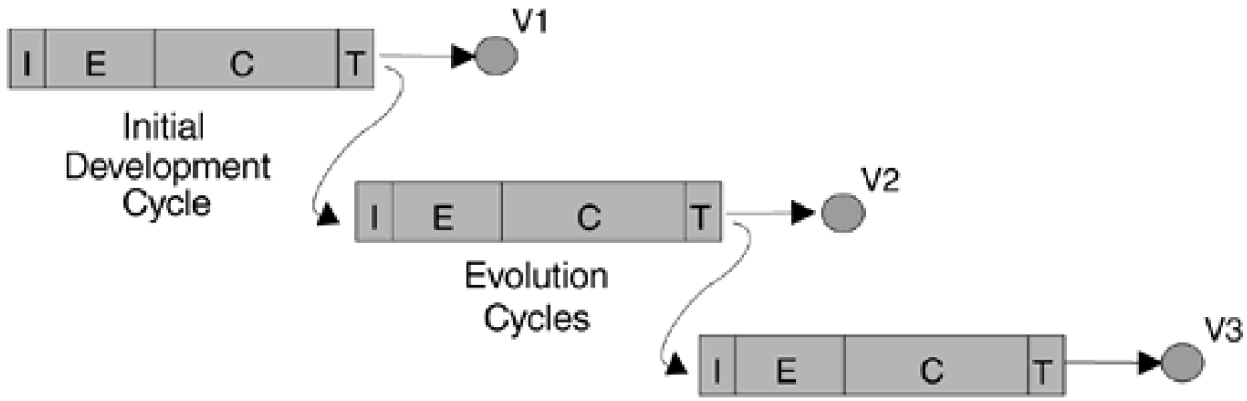
Building the product and evolving the vision, the architecture, and the plans until the product—the completed vision—is ready for delivery to its user community. The construction phase is concluded by the initial operational capability (IOC) milestone.

- *Transition*

  Transitioning the product to its users, which includes manufacturing, delivering, training, supporting, and maintaining the product until users are satisfied. It is concluded by the product release (PR) milestone, which also concludes the cycle.

The four phases (I, E, C, and T) constitute a development *cycle* and produce a software*generation.* A software product is created in an *initial development cycle.* Unless the life of the product stops at this point, a product will evolve into its next generation by a repetition of the sequence of inception, elaboration, construction, and transition phases, but with different emphases on the various phases. We call these periods *evolution cycles* (see Figure 4-4).

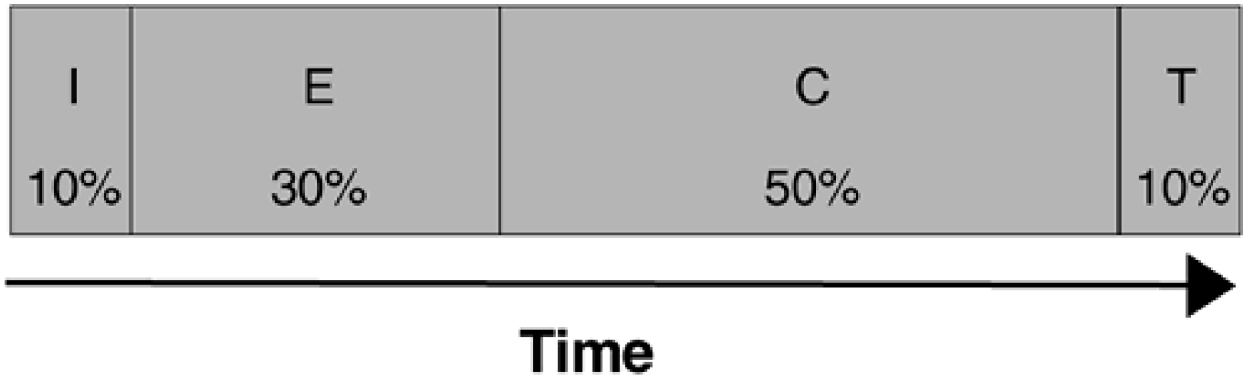
Figure 4-4. Initial and evolution cycles



As the product goes through several evolution cycles, new generations of the product are produced. Evolution cycles can be triggered by user-suggested enhancements, changes in the user's context, changes in the underlying technology, or reaction to the competition. In practice, cycles may overlap slightly: the inception and elaboration phase may begin during the final part of the transition phase of the previous cycle. We revisit this issue in Chapter 7.

Do not interpret the various figures to mean that the phases are of equal duration; their length varies greatly depending on the specific circumstances of the project. What is important is the goal of each phase and the milestone that concludes it. Figure 4-5 shows the time line of a typical project.

Figure 4-5. Typical time line for initial development cycles



For example, a two-year project would have the following:

- A 2½-month inception phase

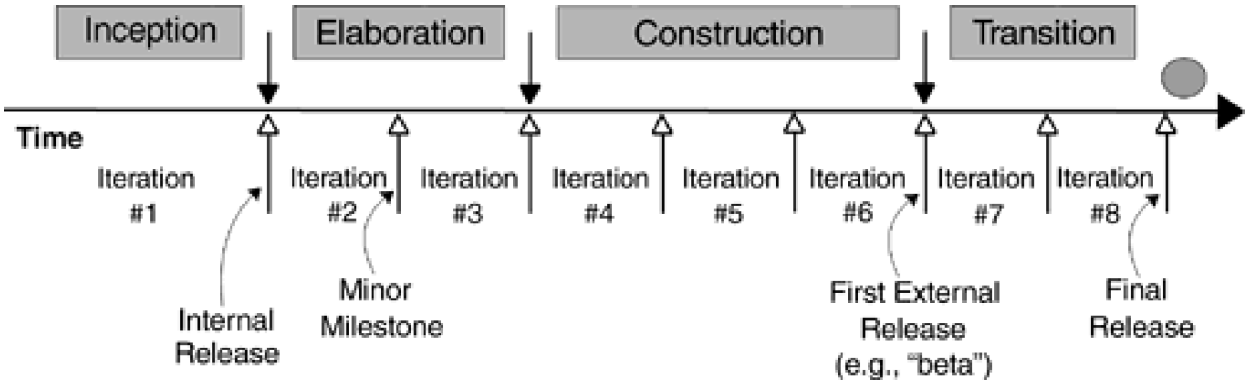
- A 7-month elaboration phase
- A 12-month construction phase
- A 2½-month transition phase

How does this time line relate to iterations? In each phase you progress iteratively, and each phase consists of one or several iterations.[5] The lifecycle is shown in Figure 4-6.[6]

[5] In rare cases, a phase may contain no real iteration.

[6] The number of iterations per phase in this diagram is for illustration only. We discuss the number of iterations in Chapter 7.

Figure 4-6. Introducing iterations, internal and external releases, and minor milestones



A Shifting Focus across the Cycle

Remember that the workflow of *each iteration* contains the activities of requirements elicitation and analysis, of design and implementation, and of integration and test, not necessarily in this order. But from one iteration to the next and from one phase to the next, the emphasis on the various activities changes.

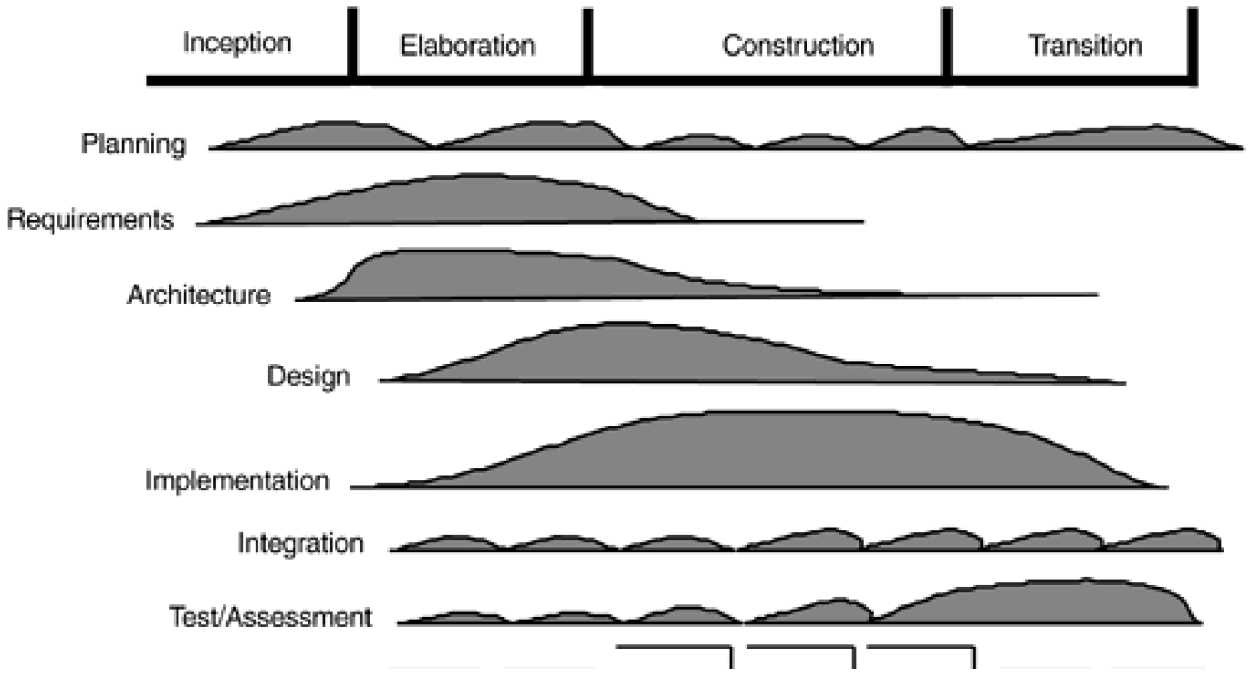
Figure 4-7 shows the relative emphasis of the various types of activities over time. If you look at a cross-section of the activities in the middle of each phase, you will notice a number of things:

- In the inception phase, the focus is on understanding the overall requirements and determining the scope of the development effort.
- In the elaboration phase, the focus is on requirements, but some software design and implementation is aimed at prototyping the architecture, mitigating certain technical risks by trying solutions, and learning how to use certain tools and techniques. You finally produce an executable architectural prototype that will serve as the baseline[7] for the next phase.

  [7] A baseline is a reviewed and approved release of artifacts that constitutes an agreed-on basis for further evolution or development and that can be changed only through a formal procedure, such as change and configuration control.

- In the construction phase, the focus is on design and implementation. Here you evolve and flesh out the initial prototype into the first operational product.
- In the transition phase, the focus is on ensuring that the system has the right level of quality to meet your objectives; you fix bugs, train users, adjust features, and add missing elements. You produce and deliver the final product.

Figure 4-7. Importance of activities across one development cycle




Preliminary Iteration
Iteration #1
Iteration #2 ...
Iteration #n+1
Iteration #...
Iteration #m
Iteration #m+1
Iteration #m+2 ...

Phases Revisited

This section examines in more detail the purpose of each phase and the evaluation criteria used at each major milestone.[8]

[8] The phases were developed in cooperation with Walker Royce, and this section is adapted from Chapter 6 of his book *Software Project Management: A Unified Framework*. Reading, MA: Addison-Wesley, 1998.

The Inception Phase

The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The primary objectives of the inception phase include the following:

- Establish the project's software scope and boundary conditions, including an operational concept, acceptance criteria, and descriptions of what is and is not intended to be part of the product.
- Discriminate the critical use cases of the system, that is, the primary scenarios of behavior that will drive the system's functionality and will shape the major design trade-offs.
- Exhibit, and perhaps demonstrate, at least one candidate architecture against some of the primary scenarios.
- Estimate the overall cost and schedule for the entire project and provide detailed estimates for the elaboration phase that will immediately follow.
- Estimate risks (the sources of unpredictability).

The essential activities of the inception phase are as follows:

- Formulate the scope of the project, that is, capture the context and the most important requirements and constraints so that you can derive acceptance criteria for the end product.
- Plan and prepare a business case and evaluate alternatives for risk management, staffing, project plan, and trade-offs among cost, schedule, and profitability.
- Synthesize a candidate architecture, evaluate trade-offs in design, and assess make/buy/reuse decisions so that cost, schedule, and resources can be estimated.

The outcome of the inception phase is creation of these artifacts:

- A vision document, that is, a general vision of the core project's requirements, key features, and main constraints
- The use-case model survey, which lists all use cases and actors that can be identified at this early stage
- An initial project glossary
- An initial business case, which includes the following:

- Business context
- Success criteria (revenue projection, market recognition, and so on)
- Financial forecast

- An initial risk assessment
- A project plan, which shows the phases and iterations

For a commercial software product, the business case should include a set of assumptions about the project and the order of magnitude of the return on investment (ROI) if these assumptions are true. For example, the ROI will be a magnitude of five if the project is completed in one year, two if it is completed in two years, and a negative number after that. These assumptions are checked again at the end of the elaboration phase when the scope and plan are defined with more accuracy.

The resource estimate might encompass either the entire project through to delivery or only the resources needed for the elaboration phase. Estimates of the resources required for the entire project should be viewed as very rough, a "guesstimate" at this point. This estimate is updated during each phase and each iteration and becomes more accurate with every iteration.

The inception phase may also produce the following artifacts:

- An initial use-case model (10% to 20% complete) when dealing with an initial development cycle
- A domain model, which is more sophisticated than a glossary (see Chapter 8)
- A business model if necessary (see Chapter 8)
- A preliminary development case description to specify the process used (see Chapter 17)
- One or several prototypes (see Chapter 11)

Milestone: Lifecycle Objective

At the end of the inception phase is the first major project milestone: the lifecycle objective milestone. The evaluation criteria for the inception phase are as follows:

- Stakeholder concurrence on scope definition and cost and schedule estimates
- Requirements understanding as evidenced by the fidelity of the primary use cases
- Credibility of the cost and schedule estimates, priorities, risks, and development process
- Depth and breadth of any architectural prototype that was developed
- Actual expenditures versus planned expenditures

If the project fails to pass this milestone, it may be canceled or considerably rethought.

The Elaboration Phase

The purpose of the elaboration phase is to analyze the problem domain, establish a sound architectural foundation, develop the

project plan, and eliminate the project's highest-risk elements. To accomplish these objectives, you must have a "mile wide and inch deep" view of the system. Architectural decisions must be made with an understanding of the whole system: its scope, major functionality, and nonfunctional requirements such as performance requirements.

It is easy to argue that the elaboration phase is the most critical of the four phases. At the end of this phase, the hard "engineering" is considered complete, and the project undergoes its most important day of reckoning: the decision of whether to commit to the construction and transition phases.

For most projects, this phase corresponds to the transition from a mobile, nimble, low-risk operation to a high-cost, high-risk operation that has substantial inertia. Although the process must always accommodate changes, the elaboration-phase activities ensure that the architecture, requirements, and plans are stable enough, and the risks are sufficiently mitigated, that you can predictably determine the cost and schedule for the completion of the development. Conceptually, this level of fidelity corresponds to the level necessary for an organization to commit to a fixed-price construction phase.

In the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, risk, and novelty of the project. At minimum, this effort should address the critical use cases identified in the inception phase, which typically expose the project's major technical risks. Although an evolutionary prototype of a production-quality component is always the goal, this does not exclude the development of one or more throwaway prototypes to mitigate a given risk or explore some trade-offs between design and requirements. Nor does it rule out a feasibility study or demonstrations to investors, customers, and end users.

The primary objectives of the elaboration phase include the following:

- Define, validate, and baseline[9] the architecture as rapidly as practical.

  [9] To *baseline* is to create a baseline, a reference; that is, to put a validated release under configuration control so that it can serve as the starting point and reference for further development.

- Baseline the vision.
- Baseline a high-fidelity plan for the construction phase.
- Demonstrate that the baseline architecture will support this vision for a reasonable cost in a reasonable time.

The essential activities of the elaboration phase are as follows:

- The vision is elaborated, that is, fully articulated, and a solid understanding is established of the most critical use cases that drive the architectural and planning decisions.
- The process, the infrastructure, and the development environment are elaborated, and the process, tools, and automation support are put into place.
- The architecture is elaborated and the components are selected. Potential components are evaluated, and the make/buy/reuse decisions are sufficiently understood to determine the construction-phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios. Lessons learned from these activities may result in a redesign of the architecture, taking into consideration alternative designs or reconsideration of the requirements.

The outcomes of the elaboration phase are as follows:

- A use-case model (at least 80% complete) in which all use cases have been identified in the use-case model survey, all actors have been identified, and most use-case descriptions have been developed
- Supplementary requirements that capture the nonfunctional requirements and any requirements that are not associated with a specific use case
- A software architecture description

- An executable architectural prototype
- A revised risk list and a revised business case
- A development plan for the overall project, including the coarse-grained project plan, which shows iterations and evaluation criteria for each iteration
- An updated development case that specifies the process to be used
- A preliminary user manual (optional)

Milestone: Lifecycle Architecture

At the end of the elaboration phase is the second important project milestone: the lifecycle architecture milestone. At this point, you examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks.

The main evaluation criteria for the elaboration phase involve the answers to the following questions:

- Is the vision of the product stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risks have been addressed and credibly resolved?
- Is the construction phase plan sufficiently detailed and accurate? Is it backed up with a credible basis for the estimates?
- Do all stakeholders agree that the current vision can be achieved if the current plan is executed to develop the complete system, in the context of the current architecture?
- Is the actual resource expenditure versus planned expenditure acceptable?

If the project fails to pass this milestone, it may be aborted or considerably rethought.

The Construction Phase

During the construction phase, all remaining components and application features are developed and integrated into the product, and all features are tested thoroughly. The construction phase is, in one sense, a manufacturing process in which emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mindset undergoes a transition from the development of intellectual property during inception and elaboration to the development of deployable products during construction and transition.

If the project is large enough, parallel construction increments can be spawned. These parallel activities can significantly accelerate the availability of deployable releases; they can also increase the complexity of resource management and workflow synchronization. A robust architecture and an understandable plan are highly correlated. In other words, one of the critical qualities of the architecture is its ease of construction. This is one reason that the balanced development of the architecture and the plan is stressed during the elaboration phase.

Primary construction phase objectives include the following:

- Minimize development costs by optimizing resources and avoiding unnecessary scrap and rework.

- Achieve adequate quality as rapidly as practical.
- Achieve useful versions (alpha, beta, and other test releases) as rapidly as practical.

The essential activities of the construction phase are as follows:

- Resource management, resource control, and process optimization
- Complete component development and testing against the defined evaluation criteria
- Assessment of product releases against acceptance criteria for the vision

The outcome of the construction phase is a product ready to put in the hands of its end users. At minimum, it consists of the following:

- The software product integrated on the adequate platforms
- The user manuals
- A description of the current release

Milestone: Initial Operational Capability

At the end of the construction phase is the third major project milestone: initial operational capability. At this point, you are to decide whether the software, the sites, and the users are ready to become operational without exposing the project to high risks. This release is often called a *beta* release.

The evaluation criteria for the construction phase involve answering the following questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all stakeholders ready for the transition into the user community?
- Are the actual resource expenditures versus planned expenditures still acceptable?

Transition may have to be postponed by one release if the project fails to reach this milestone.

The Transition Phase

The purpose of the transition phase is to move the software product to the user community. After the product has been given to the end user, issues usually arise that require you to develop new releases, correct some problems, or finish features that were postponed.

The transition phase is entered when a baseline is mature enough to be deployed in the end-user domain. Typically, this means that some usable subset of the system has been completed to an acceptable level of quality and that user documentation is available so that the transition to the user will provide positive results for all parties. This phase includes the following:

- Beta testing to validate the new system against users' expectations

- Parallel operation with the legacy system that the project is replacing
- Conversion of operational databases
- Training of users and maintainers
- Rollout of the product to the marketing, distribution, and sales teams

The transition phase concludes when the deployment baseline has achieved the completed vision. For some projects, this lifecycle endpoint coincides with the starting point of the next cycle, leading to the next generation or version of the product. For other projects, it coincides with a delivery of the artifacts to a third party responsible for operation, maintenance, and enhancements of the delivered system.

The transition phase focuses on the activities required to place the software into the hands of the users. Typically, this phase comprises several iterations, including beta releases, general availability releases, and bug-fix and enhancement releases. Considerable effort is expended in developing user-oriented documentation, training users, supporting users in the initial use of the product, and reacting to user feedback. At this point in the lifecycle, however, user feedback should be confined primarily to product tuning, configuring, installation, and usability issues.

The primary objectives of the transition phase include the following:

- Achieve user self-supportability.
- Achieve stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision.
- Achieve final product baseline as rapidly and cost-effectively as practical.

The essential activities of the transition phase are as follows:

- Deployment-specific engineering, that is, cutover, commercial packaging and production, sales rollout, and field personnel training
- Tuning activities, including bug fixing and enhancement for performance and usability
- Assessing the deployment baselines against the vision and the acceptance criteria for the product

In the transition phase, the activities performed during an iteration depend on the goal; for fixing bugs, implementation and testing are usually enough. If new features must be added, the iteration is similar to those of the construction phase.

Depending on the type of product, this phase can range from simple to extremely complex. For example, a new release of an existing desktop product may be simple, whereas replacing a nation's air traffic control system would be complex.

Milestone: Product Release

At the end of the transition phase is the fourth important project milestone: the product release milestone. At this point, you decide whether the objectives were met and whether you should start another development cycle. In some cases, this milestone coincides with the end of the inception phase for the next cycle.

The primary evaluation criteria for the transition phase involve the answers to the following questions:

- Is the user satisfied?
- Are the actual resources expenditures versus planned expenditures still acceptable?

Benefits of an Iterative Approach

Compared with the traditional waterfall process, the iterative process has the following advantages:

- Risks are mitigated earlier.
- Change is more manageable.
- There is a higher level of reuse.
- The project team can learn along the way.
- The product has better overall quality.

Risk Mitigation

An iterative process lets you mitigate risks earlier than a sequential process where the final integration is generally the only time that risks are discovered or addressed. As you roll out the early iterations, you go through all process components, exercising many aspects of the project, including tools, off-the-shelf software, and people skills. Perceived risks will prove not to be risks, and new, unsuspected risks will be discovered.

If a project must fail for some reason, let it fail as soon as possible, before a lot of time, effort, and money are expended. Do not bury your head in the sand too long; instead, confront the risks. Among other risks, such as building the wrong product, there are two categories of risk that an iterative development process helps to mitigate early:

- Integration risks
- Architectural risks

An iterative process results in a more robust architecture because you correct errors over several iterations. Flaws are detected in early iterations as the product moves beyond inception. Performance bottlenecks are discovered when they can still be addressed instead of on the eve of delivery.

Integration is not one "big bang" at the end of the lifecycle; instead, elements are integrated progressively. Actually, the iterative approach that we recommend involves almost continuous integration. What used to be a lengthy time of uncertainty and pain—taking as much as 40% of the effort at the end of a project—is now broken into six to nine smaller integrations that begin with far fewer elements to integrate.

Accommodating Changes

You can envisage several categories of change.

Changes in Requirements

An iterative process lets you take into account changing requirements. The truth is that requirements normally change. Changing requirements and requirements "creep" have always been primary sources of project trouble, leading to late delivery, missed schedules, unsatisfied customers, and frustrated developers. But by exposing users (or representatives of the users) to an early version of the product, you can ensure a better fit of the product to the task.

Tactical Changes

An iterative process provides management with a way to make tactical changes to the product, for example, to compete with existing products. You can decide to release a product early with reduced functionality to counter a move by a competitor, or you can adopt another vendor for a given technology. You can also reorganize the contents of iteration to alleviate an integration problem that needs to be fixed by a supplier.

Technological Changes

To a lesser extent, an iterative approach lets you accommodate technological changes. You can use it during the elaboration phase, but you should avoid this kind of change during construction and transition because it is inherently risky.

Learning as You Go

An advantage of the iterative process is that developers can learn along the way, and the various competencies and specialties are employed during the entire lifecycle. For example, testers start testing early, technical writers write early, and so on, whereas in a noniterative development, the same people would be waiting to begin their work, making plan after plan. Training needs or the need for additional (perhaps external) help are spotted early during assessment reviews.

The process itself can also be improved and refined along the way. The assessment at the end of an iteration looks at the status of the project from a product/schedule perspective and analyzes what should be changed in the organization and in the process so that it can perform better in the next iteration.

Increased Opportunity for Reuse

An iterative process facilitates reuse of project elements because it is easy to identify common parts as they are partially designed or implemented instead of identifying all commonality in the beginning. Identifying and developing reusable parts is difficult. Design reviews in early iterations allow architects to identify unsuspected potential reuse and to develop and mature common code in subsequent iterations. It is during the iterations in the elaboration phase that common solutions for common problems are found, and patterns and architectural mechanisms that apply across the system are identified. For more about this issue, see

Better Overall Quality

The product that results from an iterative process will be of better overall quality than that of a conventional sequential process. The system has been tested several times, improving the quality of testing. The requirements have been refined and are related more closely to the users' real needs. At the time of delivery, the system has been running longer.

Summary

- The sequential, or waterfall, process is fine for small projects that have few risks and use a well-known technology and domain, but it cannot be stretched to fit projects that are long or involve a high degree of novelty or risk.
- An iterative process breaks a development cycle into a succession of iterations. Each iteration looks like a small project and involves the activities of requirement capture, design, implementation, and assessment.
- To control the project and to give the appropriate focus to each iteration, a development cycle is divided into a sequence of four phases that partition the sequence of iterations. The phases are inception, elaboration, construction, and transition.
- The iterative approach accommodates changes in requirements and in implementation strategy. It confronts and mitigates risks as early as possible. It allows the development organization to grow, to learn, and to improve. It focuses on real, tangible objectives.

Chapter 5. An Architecture-Centric Process

This chapter defines architecture and explains why it plays a central role in the Rational Unified Process.

The Importance of Models

A large part of the Rational Unified Process focuses on modeling. Models help us understand and shape both the problem and the solution. A model is a simplification of reality that helps us master a large, complex system that cannot be comprehended easily in its entirety. The choice of models and the choice of techniques used to express them have a profound effect on the way we think about the problem and shape the solution. The model is not the reality ("the map is not the territory"[1]), but the best models are the ones that stick very close to reality.[2]

[1] "The map is not the territory" is fundamental to the book,*Language in Thought and Action* by S. I. Hayakawa. New York: Harcourt-Brace, 1939.

[2] Grady Booch, James Rumbaugh, and Iver Jacobson,*The Unified Modeling Language Users Guide*. Reading, MA: Addison-Wesley, 1999.

No single model is sufficient to cover all aspects of software development. We need multiple models to address different concerns. These models must be coordinated carefully to ensure that they are consistent and not too redundant.

Architecture

Models are complete, consistent representations of the system to be built. The models of complex systems can be very large.

Suppose you are given the task of describing a system so that designers, programmers, users, and managers would be able to do the following:

- Understand what the system does
- Understand how the system works
- Be able to work on one piece of the system
- Extend the system
- Reuse part of the system to build another one

Now assume that you are given a limited amount of space for this task (for example, a maximum of 60 pages). What you would end up with is a description of the *architecture* of the system. As someone once told me, "Architecture is what remains when you cannot take away any more things and still understand the system and explain how it works."

The Importance of Architecture

For many years, software designers have had the strong feeling that software architecture was an important concept. But this concept was not very well exploited, for a number of reasons:

- The purpose of an architecture was not always well articulated.
- The concept remained fuzzy, trapped somewhere between top-level design, system concept, and requirements.
- There was no accepted way to represent an architecture.
- The process by which an architecture came into life was not described and always seemed to be some kind of art form or black magic.

Architecture ended up taking the form of an unmaintained document of a few diagrams made of boxes and arrows reflecting imprecise semantics that could be interpreted only by its authors. But because many software systems weren't complex, the architecture could remain an implicit understanding among software developers.

However, as systems evolve and grow to accommodate new requirements, things break in a strange fashion, and the systems do not scale up. Integrating new technologies requires that we completely rebuild the systems. Systems, in other words, are not very resilient in response to changes. Moreover, the designers lack the intellectual tools to reason about the parts of the system. It is little wonder that poor architectures (together with immature processes) are often listed as reasons for project failures. Not having an architecture, or using a poor architecture, is a major technical risk for software projects.

Architecture Today

Now that all the simple systems have been built, managing the complexity of large systems has become the number one concern of software development organizations. They want their systems to evolve rapidly, and they want to achieve large-scale reuse across families of systems, building systems from ready-made components. Software has become a major asset, and organizations need conceptual tools to manage it.

The word *architecture* is now used everywhere, reflecting a growing concern and attention, but the variety of contexts in which the word is used suggests that it has not necessarily become a well-mastered concept.

For an organization to adopt an architecture focus, three things are required:

- *An understanding of the purpose*

  Why is architecture important? What benefits can we gain from it? How can we exploit it?

- *An architectural representation*

  The best way to make the concept of architecture less fuzzy is to agree on a way to represent it so that it becomes something concrete that can be communicated, reviewed, commented on, and improved systematically.

- *An architectural proces*

  How do you create and validate an architecture that satisfies the needs of a project? Who does it? What are the

artifacts and the quality attributes of this task?

The Rational Unified Process contains some answers to all three points. But let's start by defining more precisely what we mean by *software architecture*, or rather, by the architecture of a software-intensive system.

A Definition of Architecture

Many definitions of architecture have been proposed. The Rational Unified Process defines *architecture* as follows:[3]

> [3] This definition evolved from one given several years ago by Mary Shaw and David Garlan of Carnegie-Mellon University. See their textbook *Software Architecture—Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Architecture encompasses significant decisions about the following:

- The organization of a software system
- The selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements
- The composition of these elements into progressively larger subsystems
- The architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition

Software architecture is concerned with not only structure and behavior but also context: usage, functionality, performance, resilience, reuse, comprehensibility, economic and technological constraints and trade-offs, and aesthetics.

This definition is long, but it attempts to capture the richness, the complexity, and the multiple dimensions of the concept. We can elaborate on some of the points.

Architecture is a part of design; it is about making decisions about how the system will be built. But it is not all of the design. It stops at the major elements—the elements that have a pervasive and long-lasting effect on the qualities of the system, namely, its evolvability and its performance.

Architecture is about structure and about organization, but it is not limited to structure. It also deals with behavior: what happens at the joints, at the seams, and across the interfaces.

Architecture not only looks inward but also looks at the "fit" of the system in two contexts: the operational context (its end users) and the development context (the organization that develops the system). And it encompasses not only a system's technical aspects but also its economic and sociological aspects.

Architecture also addresses "soft" issues such as style and aesthetics. Can an architecture be pleasing? Yes, to the educated eye it can be. Issues of aesthetics have their place in making a design uniform, easy to understand, and easy to evolve, with minimal surprises for the designers.

Architecture Representation

In one form or another, many different parties are interested in architecture:

- The system analyst, who uses it to organize and articulate the requirements and to understand the technological constraints and risks
- End users or customers, who use it to visualize at a high level what they are buying
- The software project manager, who uses it to organize the team and plan the development
- The designers, who use it to understand the underlying principles and locate the boundaries of their own designs
- Other development organizations (if the system is open), which use it to understand how to interface with it
- Subcontractors, who use it to understand the boundaries of their chunk of development
- Architects, who use it to reason about evolution or reuse

To allow the various stakeholders to communicate, discuss, and reason about architecture, we must have an architectural representation that they all understand. The architecture and its representation are not quite the same thing: By choosing a representation, we omit some of the most intangible or soft aspects.

The various stakeholders have different concerns and are interested in different aspects of the architecture. Hence, a complete architecture is a multidimensional thing. *Architecture is not flat.*

Multiple Views

For a building, different types of blueprints are used to represent different aspects of the architecture:

- Floor plans
- Elevations
- Electrical cabling
- Water pipes, central heating, and ventilation
- The look of the building in its environment (in sketches)

Over decades, blueprints have evolved into standard forms so that every person involved in the design and construction of the building understands how to read and use them. Each of these blueprints tries to convey one aspect of the architecture for one category of stakeholders, but the blueprints are not independent of each other. On the contrary, they must be carefully coordinated.

Similarly, in the architecture of a software-intensive system, you can envisage various blueprints for various purposes:

- To address the logical organization of the system
- To organize the functionality of the system

- To address the concurrency aspects
- To describe the physical distribution of the software on the underlying platform

And there are many more purposes. These are what we call *architectural views*.[4]

[4] This multiple-view approach is to conform to the*IEEE Recommended Practice for Architectural Description*. IEEE Standard 1471-2000.

An architectural view is a simplified description (an abstraction) of a system from a particular perspective or vantage point, covering particular concerns and omitting entities that are not relevant to this perspective. For each view, we need to clearly identify the following:

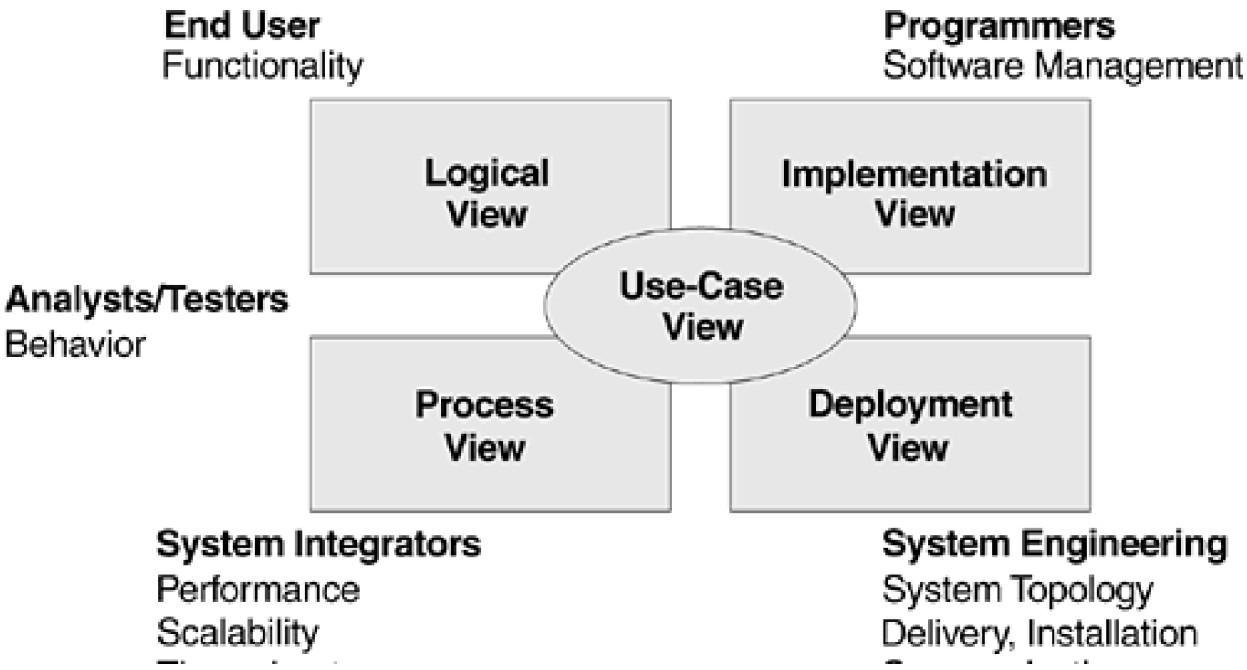
- The point of view— the concerns and the stakeholders to be addressed
- The elements that will be captured and represented in the view, together with their relationships
- The organizational principles used to structure the view
- The way elements of this view are related to those of other views
- The best process to use in creating this view

The 4 + 1 View Model of Architecture

As shown in Figure 5-1, the Rational Unified Process suggests a five-view approach.[5] The following sections summarize the five views.

[5] Philippe Kruchten, "The 4 + 1 View of Architecture,"*IEEE Software*, 12(6) Nov. 1995, pp. 45–50.

Figure 5-1. The 4+1 view model of architecture



The Logical View

This view of the architecture addresses the functional requirements of the system, in other words, what the system should do for its end users. It is an abstraction of the design model and identifies major design packages, subsystems, and classes.

Examples include a flight, a flight plan, an airway, an airport, and an airspace package.

The Implementation View

This view describes the organization of static software modules (source code, data files, components, executables, and other accompanying artifacts) in the development environment in terms of packaging and layering and in terms of configuration management (ownership, release strategy, and so on). It addresses the issues of ease of development, management of software assets, reuse, subcontracting, and off-the-shelf components.

Examples include the source code for a flight class and the library of code for an airspace database.

The Process View

This view addresses the concurrent aspects of the system at runtime—tasks, threads, or processes as well as their interactions. It addresses issues such as concurrency and parallelism, system start-up and shutdown, fault tolerance, and object distribution. It deals with issues such as deadlock, response time, throughput, and isolation of functions and faults. It is concerned with scalability.

Examples are a flight management process, flight plan entry processes, and an airspace management process.

The Deployment View

This view shows how the various executables and other runtime components are mapped to the underlying platforms or computing nodes. It is here that software engineering meets system engineering. It addresses issues such as deployment, installation, and performance.

For example, the flight management process runs on the main flight processor, whereas the flight plan entry processes run on any workstation.

The Use-Case View

This view plays a special role with regard to the architecture. It contains a few key scenarios or use cases. Initially, these are used to drive the discovery and design of the architecture in the inception and elaboration phases, but later they will be used to validate the different views. These few scenarios act to illustrate in the software architecture document how the other views work.

Examples are the entry of a flight plan and the handover of responsibility to another air traffic controller.

Models and Views

The preceding description of the architecture is true also for the models introduced in the first section of this chapter. We create different models to address different needs. But the models are complete representations of the system, whereas an architectural view focuses only on what is architecturally significant. If we do not make this distinction, there is no clear way to distinguish "design" from "architecture." *Not all design is architecture.*

What is "architecturally significant"? An architecturally significant element has a wide impact on the structure of the system and on its performance, robustness, evolvability, and scalability. It is an element that is important for understanding the system.

Architecturally significant elements include the following:

- Major classes, in particular the classes that model major business entities
- Architectural mechanisms that give behavior to these classes, such as persistency mechanisms and communication mechanisms
- Patterns and frameworks
- Layers and subsystems
- Interfaces
- Major processes, or threads of control

Table 5-1. The Relationship between Models and Views

Model	Architectural View
Design model	Logical view
Design model[*]	Process view
Implementation model	Implementation view
Deployment model	Deployment view
Use-case model	Use-case view

[*] or a process model for a complex system

Architectural views are like slices cut through the various models, illuminating only the important, significant elements of the models. Table 5-1 summarizes the relationship between models and views. The views useful for a system are captured in an important artifact: the software architecture description.

Architecture Is More Than a Blueprint

Architecture is more than just a blueprint of the system to be developed. To validate the architecture and to assess its qualities in terms of feasibility, performance, flexibility, and robustness, we must build it.

Building the architecture, validating it, and then baselining it are the primary objectives of the elaboration phase. Therefore, in addition to a software architecture description, the most important artifact associated with the architecture is an architectural prototype that implements the most important design decisions sufficiently to validate them—that is, to test and measure them. This prototype is not a quick-and-dirty throwaway prototype, but it will evolve through the construction phase to become the final system. (For more information, see the section titled Prototypes in Chapter 11.)

PREVIOUS NEXT

An Architecture-Centric Process

After an organization agrees on a representation of the architecture that is suitable for the problem at hand, the next issue is to master an architectural design process.

The Rational Unified Process defines two primary artifacts related to architecture:

- The software architecture description (SAD), which describes the architectural views relevant to the project
- The architectural prototype, which serves to validate the architecture and serves as the baseline for the rest of the development

These two key artifacts are the roots of three others:

- Design guidelines are shaped by some of the architectural choices made and reflect the use of patterns and idioms.
- The product structure in the development environment is based on the implementation view.
- The team structure is based on the structure of the implementation view.

The Rational Unified Process defines a Role: Software Architect, who is responsible for the architecture. Architects, however, are not the only ones concerned with the architecture. Many team members are involved in the definition and the implementation of the architecture, especially during the elaboration phase:

- Designers focus on architecturally significant classes and mechanisms rather than on the details of the classes.
- Integrators integrate major software components, even if their implementation is very rudimentary, to verify the interfaces. Integrators focus mainly on removing integration risks related to major off-the-shelf or reused components.
- Testers test the architectural prototype for performance and robustness.

During the construction phase, the focus shifts to adding the meat and skin to the architectural skeleton. Activities reflect an ongoing concern for the architecture: tuning it, refining it, and making sure that no new design decision is introduced that would weaken or break it.

The bulk of the activities related to architectural design are described in the analysis and design discipline (see Chapter 10), but it spills over to the requirements discipline, the implementation discipline, and the project management discipline.

The Purpose of Architecture

Now that we have embraced architecture, we can revisit its purpose. Why should an organization focus on architecture? Architecture is important for several reasons, which are discussed in the following sections.

Intellectual Control

Architecture lets you gain and retain intellectual control over the project, to manage its complexity, and to maintain system integrity.

A complex system is more than the sum of its parts and more than a succession of independent tactical decisions. It must have a unifying, coherent structure so as to organize the parts systematically. It must also provide precise rules about how to grow the system without having its complexity multiply beyond human understanding.

Maintaining the system's integrity has long been the objective of system architects. Integrity, in its original meaning, is "being one." We want control of the architecture to design and build systems that are almost continuously integrated as "one" without resorting to difficult and painful "integration phases" in which all kinds of mismatched parts must be fitted together by force.

The architecture establishes a means of effective communication and understanding throughout the project by establishing a common set of references and a common vocabulary for discussing design issues. Each designer and developer will understand the context and boundary of the part he or she is working on.

Reuse

Architecture provides an effective basis for large-scale reuse.

By clearly articulating the major components and the critical interfaces between them, an architecture lets you reason about reuse. It assists both internal reuse—the identification of common parts—and external reuse—the incorporation of ready-made, off-the-shelf components.

Architecture also facilitates reuse on a larger scale: the reuse of the architecture itself in the context of a line of products that addresses varying functionality in a common domain. Architects will understand the limits of scalability, identify the generic parts, and define the variation points.

Basis for Development

Architecture provides a basis for project management.

Planning and staffing are organized along the lines of major components: layers and subsystems. Fundamental structural

decisions are made by a small, cohesive architecture team; the decisions are not distributed. Development is partitioned across a set of small teams, each of which is responsible for one or several parts of the system. Different development organizations will understand which interfaces are at their disposal and will understand the limits of their variation.

Architecture and the work done during the elaboration phase provide the basis for further development, including the design guidelines, principles, styles, patterns, and mechanisms to reuse.

PREVIOUS NEXT

Component-Based Development

The Rational Unified Process supports component-based development (CBD), which is the creation and deployment of software-intensive systems that are assembled from components, as well as the development and harvesting of such components.

Component-based development is about building quality systems that satisfy business needs quickly, preferably by using parts rather than handcrafting every individual element. It involves crafting the right set of primitive components from which to build families of systems, including the harvesting of components. Some components are made intentionally; others are discovered and adapted.

A definition of *component* must be broad enough to address conventional components (such as COM/DCOM, CORBA, and JavaBeans components) as well as alternative ones (such as Web pages, database tables, and executables using proprietary communication). At the same time, it shouldn't be so broad as to encompass every possible artifact of a well-structured architecture.

A Definition of Component

A *component* is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Note that component and architecture are two intertwined concepts: the architecture identifies components, their interfaces, and their interactions along several dimensions, and components exist only relative to a given architecture. You cannot mix and match your chosen components if they have not been made to fit.

There are several perspectives on components:

- *Runtime components*

  The things that are delivered, installed, and run, such as executables, processes, and dynamic link libraries (DLLs). They live at runtime on the deployment platform.

- *Development components, as seen from the development organization's point of view*

  Implementation subsystems that have high internal cohesion and low external coupling and are reusable by other developers. There may not always be a one-to-one relationship between runtime components and development components, but they provide a useful "first cut" at the runtime components.

- *Business components*

  Cohesive sets of runtime components (or development components) that fulfill a large chunk of business-level functionality and are units of sale, release, or upgrade.

Other Architectural Concepts

The following sections describe several other topics related to software architecture.

Architectural Style

A software architecture, or an architectural view, may have an attribute called *architectural style,* which reduces the set of possible forms and imposes a certain degree of uniformity on the architecture. The style may be defined by the selection of an architectural framework, by middleware, by a recommended set of patterns, or by an architecture description technique or tool.

Examples of architectural styles are pipe-and-filter, client-server, and event-driven styles.

Architectural Mechanism

An *architectural mechanism* is a class, a group of classes, or a pattern that provides a common solution to a common problem. It is used to give life—that is, behavior—to other classes. Mechanisms are found primarily in the middle and lower layers of an architecture. They are analogous to the standard heating conduits and plumbing fixtures that are available to the architect of a building. They provide a means of rapidly implementing desired solutions.

Examples of architectural mechanisms include a database management system (DBMS), an event broadcasting system, and a transaction server.

Architectural Pattern

A *pattern* addresses and presents a common solution to a recurring design problem that arises in specific design situations. Patterns document existing, well-proven design experience. They identify abstractions that are above the level of classes, instances, and components. They provide a common vocabulary and understanding of design principles and are therefore a means of documenting software architectures. Patterns are larger-scale than mechanisms are; patterns describe broad interactions of abstract design elements that help the architect and designers think about a complex problem in an intuitive shorthand.

Software architectural patterns are analogous to patterns of architecture in buildings. When someone says that a building is a skyscraper, immediately a mental picture of the kind of building and the kind of project leaps to mind. We know that the structure is made of steel and concrete and not of wood. The design includes elevators in addition to stairways.

Patterns and mechanisms provide the architect with a growing toolkit for solving architectural problems. Just as building architects have evolved a representation and symbology of the various problems they encounter, so too have software architects evolved intellectual tools (and software tools) to manage the complexity of systems. Patterns and mechanisms play a growing part in this

toolkit.[6]

[6] See Frank Buschmann, Régine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stahl, *Pattern-Oriented Software Architecture: A System of Patterns.* New York: John Wiley & Sons, 1996.

Examples of architectural patterns are Model-View-Controller (MVC) and Object Request Broker (ORB).

PREVIOUS NEXT

Summary

- System architecture is used in the Rational Unified Process as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.
- Architecture is a complex concept that is best represented by multiple, coordinated architectural views.
- An architectural view is an abstraction of a model that focuses on its structure and its essential elements.

See the bibliography for further readings on software architecture.

Chapter 6. A Use-Case-Driven Process

This chapter introduces the concepts of use case, actor, and scenario. It shows how use cases can be used throughout the development cycle as drivers for many activities, flowing information through several models and encouraging consistency across these models.

Definitions

A large part of the Rational Unified Process focuses on modeling. As we explained in Chapter 5, models help us to understand and shape both the problem we are trying to solve as well as its solution. The choice of models and the choice of techniques used to express them have a significant impact on the way we think about the problem and try to shape the solution.

Many approaches can be taken to model the problem and to express the requirements and constraints on the system to be developed. But keep in mind that once this model is developed, we will then need to formulate a model of the solution. If the model of the problem is too far removed from the model of the solution, a great deal of effort may be expended in attempting to translate the expression of the problem from a form understandable by end users into a form understandable by designers and builders. This presents many opportunities for misinterpretation and often makes it difficult to validate the solution with respect to the stated problem. Moreover, we will be working with several models in our process that must be kept consistent.

In this chapter we focus primarily on one method of understanding and modeling the problem. An effective method, and one recommended by the Rational Unified Process, is the technique of use-case modeling. Use cases provide a means of expressing the problem in a way that is understandable to a wide range of stakeholders: users, developers, and customers.[1]

[1] Use cases were introduced by Ivar Jacobson in *Object-Oriented Software Engineering: A Use-Case-Driven Approach*. Reading, MA: Addison-Wesley, 1992.

Use Case and Actor

To build a use-case model, you must understand two key concepts: use case and actor. The Rational Unified Process defines these terms as follows:

- A *use case* is a sequence of actions a system performs that yields an observable result of value to a particular actor.
- An *actor* is someone or something outside the system that interacts with the system.

Note that the system is the thing under consideration; actors (roles that people or other systems might assume) are things that interact with it, and use cases define these interactions.

In reviewing these definitions, let us consider several key terms and phrases that are used:

- *Actions*

  An action is a computational or algorithmic procedure that is invoked when the actor provides a signal to the system or when the system gets a time event. An action may imply signal transmissions to either the invoking actor or other actors. An action is atomic, which means it is performed either in its entirety or not at all.

- *A sequence of actions*

  The sequence of actions referred to in the definition is a specific flow of events through the system. Many different flows of events are possible, and many of them may be very similar. To make a use-case model understandable, we group similar flows of events into a single use case.

- *The system performs*

  Again, note that we are concerned with what the system does in order to perform the sequence of actions. The use case helps us to define a firm boundary around the system; what is done by the system is described clearly, separate and distinct from the actions in the outside world. In this way, a use case helps us bound the scope of the system.

- *An observable result of value*

  The sequence of actions must yield something that has value to an actor of the system. An actor should not have to perform several use cases in order to achieve something useful. Focusing on useful value provided to an actor ensures that the use case has relevance and is at a level of granularity that the user can understand.

- A *particular actor*

  Focusing on a particular actor forces us to isolate the value provided to specific groups (roles) of users of the system, ensuring that the system does what they need it to do. It helps us avoid building use cases that are too large. It also prevents us from losing focus and building systems that try to satisfy the needs of everyone but in the end satisfy no one.

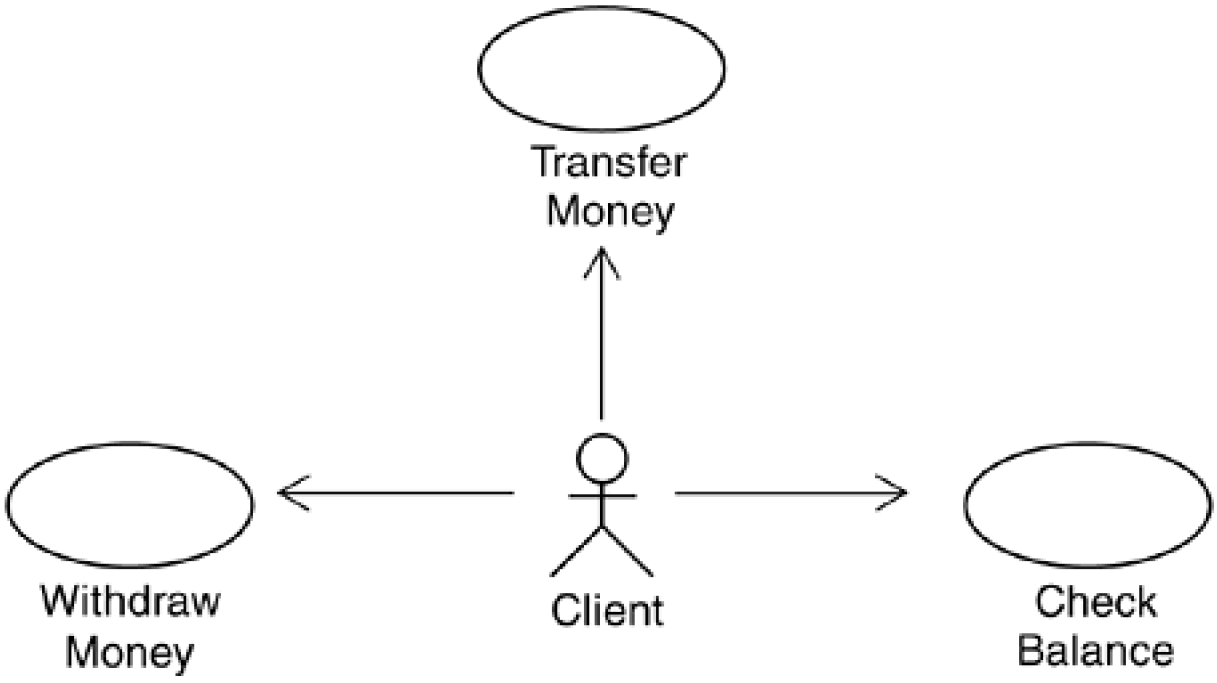
The description of a use case defines what the system does when the use case is performed. A system's functionality is defined by a set of use cases, each of which represents a specific flow of events.

The use-case flow of events expresses the behavior of the system in a "gray box view" of the system, whereas a *use-case realization*, which is used later in design, is the "white box view"; it shows how the use case is actually performed in terms of interacting objects and classes.

Example of Use Cases and Actors

A bank client, for example, can use an automated teller machine (ATM) to withdraw money, transfer money, or check the balance of an account. These capabilities can be represented by a set of use cases, as shown in Figure 6-1.

Figure 6-1. A use-case diagram for an ATM



Each use case represents something that the system does that provides value to the bank's customer, the Client (actor). The collected use cases constitute all the possible ways to use the system. The name of a use case usually conveys the value provided to the actor.

Flow of Events

The most important part of the use case in the requirements discipline is its *flow of events.* The flow of events describes the sequence of actions between the actor and the system. It is written in natural language, in a simple, consistent prose, with a precise use of terms that draws on a common glossary of the problem domain.

Example of a Flow of Events

Let's look at an example of a flow of events for one of the use cases just described. An initial outline of the flow of events of the use case Withdraw Money could be written as follows:

1. The use case begins when the Client inserts a card into the ATM. The system reads and validates information on the card.
2. The system prompts for a personal identification number (PIN). Client enters the PIN. The system validates the PIN.
3. The system asks which operation the Client wishes to perform. Client selects "Withdraw Money."
4. The system requests the amount of withdrawal. Client enters amount.
5. The system requests the account type. Client selects the account type (e.g., checking, savings, credit).
6. The system communicates with the ATM network to validate the account ID, PIN, and availability of the amount requested.
7. The system asks the Client whether a receipt is desired. This step is performed only if there is paper available to print the receipt.
8. The system asks the Client to remove the card. Client removes the card. (The request is a security measure to ensure that Clients do not leave their cards in the machine.)
9. The system dispenses the requested amount of cash.
10. The system prints a receipt, if requested, which ends the use case.

When you try to describe what the system performs by giving complete sequences of actions, you rapidly realize that there are many courses of action—various paths through the use-case flow of events. There are alternative branches and different cases (scenarios) to consider, and different values or effects are produced. Collectively, the use-case flow of events eventually describes all these possible courses.

The path chosen depends on several things such as the following:

- *Input from an actor*

  The actor can choose from several options what to do next. For example, the bank's clients can cancel the transaction

at any time, or they could decide they do not need a receipt.

- *The internal state of the system*

  For example, the ATM may be out of cash or have no paper for receipts, or the cash dispenser or printer may jam.

- *Time-outs and errors*

  For example, if the ATM user does not respond within a specified time interval, the system might automatically cancel the transaction. If the user enters the wrong PIN several times, the transaction might be canceled and the card confiscated.

Scenarios

We do not want to have to express each possible alternate flow in a separate use case; rather, we will group them with other related use-case flows of events. Such a grouping defines a use-case *class*. An instance of a use-case class is one specific flow of events, or a specific path, through the use case. An instance of a use-case class is also called a *scenario*. It is usually sufficient and less cumbersome to refer to a use-case *class* as merely a use case, and a use-case instance as a *scenario*.

Scenarios are used in the process to extract and emphasize a unique sequence of actions or a "thread" through a use case. This is especially useful when defining test cases. Also, when you're trying to find use cases during the early stages of a project, it is easier to start from a specific scenario and then expand it to more flows of events, finally generalizing it to a full-blown use case.

Use-Case Model

The use-case model, then, consists of the set of all use cases for the system, or a portion of the system, together with the set of all actors that interact with these use cases, thus describing the complete functionality of the system. It provides a model of the system's intended functions and its environment, and can serve as a contract between the customer and the developers.

Use cases and actors are defined in the Unified Modeling Language (UML), and the Rational Unified Process uses use-case diagrams and activity diagrams to visualize the use-case model, including possible relationships among use cases.

What about Concurrency?

Sometimes you may be concerned about interactions among use cases at runtime, such as several use cases using the same resources and thereby creating an opportunity for conflict. The purpose of the use-case model is to make sure that all functional requirements are handled by the system.

To remain focused on this, we ignore (for the moment) nonfunctional requirements, including concurrency. You should assume at this stage that all scenarios (use-case instances) can run concurrently without any problem. When the system is designed, you should then ensure that all nonfunctional requirements are handled correctly, including ensuring that concurrency requirements are dealt with properly.

Example of a Use-Case Model

The use-case model for the ATM example discussed earlier could be completed with use cases for the bank staff and use cases for the installation personnel. The use-case model is complemented by nonfunctional specifications that cover the aspects of the system that cannot be described easily in a use case or that apply to many or all use cases. Examples of these are design constraints, performance, reliability, and safety and security requirements (see Chapter 9).

As the defined system grows, it is useful to make sure that certain terms are used consistently throughout all use cases. To address this need, we create as a companion document a project *glossary* or, even better, a simple object model of the *domain* called a *domain model* (see Chapter 8).

PREVIOUS NEXT

Identifying Use Cases

It is often difficult to decide whether a set of user-system interactions, or a particular *dialog* or *scenario,* is one or several use cases. Consider the use of an ATM: You can walk up to the cash dispenser, insert your bank card, punch in your PIN, and then withdraw money, deposit money, check your balances, or transfer funds between accounts. Then you can get a receipt for your transaction and get your card back.

(For the moment, let's forget about multiple transactions.) Is inserting your ATM card, entering your PIN, and having it validated a use case? Does it provide value to you? Suppose you walked up to an ATM, inserted your card, and entered your PIN, and the machine told you that you correctly entered your PIN and then gave you your card back? Would you be happy? Of course not! We use the ATM to get money, transfer funds, and so on. Those are the things of value that the ATM does; those are its use cases: withdraw cash, transfer funds, deposit funds, and inquire into balances. Each use case includes the complete dialog, from inserting the ATM card to selecting the transaction type to getting the receipt and the return of the card.

Use cases emerge when you focus on the results of value that a system provides to an actor and when you group the sequences of actions that a system takes to provide those results of value. Another useful way of thinking about this is to consider that a use case fulfills a particular "goal" that an actor has, to be accomplished by the system. A very useful discussion of this method can be found in the book by Alistair Cockburn.[2]

[2] Alistair Cockburn, *Writing Effective Use Cases*. Boston: Addison-Wesley, 2001.

It is important to keep the actions together so that you can review them at the same time, modify them together, test them together, write manuals for them, deploy them together, and, in general, manage them as a unit. The need for this approach becomes obvious in larger systems. It is different from a "functional decomposition" approach, in which you rapidly lose track of the value and purpose of each little bit of functionality. The reason for grouping pieces of functionality and calling it a use case is that you want to manage these pieces of functionality together throughout the lifecycle.

Evolving Use Cases

You should always begin by developing an outline of a use case (in a step-by-step format) before delving into the details. In early iterations, during the elaboration phase, only a few use cases (those that are considered architecturally significant) are described in detail beyond the brief description.

Your model will often contain use cases that are so simple that they do not need a detailed description of the flow of events, and a step-by-step outline may be sufficient. The criterion for making this decision is that there should be no disagreement among the various stakeholders about what the use case means and that designers and testers are comfortable with the level of detail provided by the step-by-step format. Examples are use cases that describe simple entry or retrieval of data from the system. Toward the end of elaboration, all use cases that will be described in detail should be completed.

Organizing Use Cases

A small system may be expressed as a half-dozen use cases that involve two or three actors. As you tackle larger systems, however, you must define structuring and organizing principles. Otherwise, you can sink under a profusion of use cases, some of which have in common major parts of the flows of events. This profusion may make the requirements hard to understand and can make activities such as planning, prioritizing, and estimating very difficult.

The first structuring concept is that of a *use-case package,* in which you group related use cases into one container. You can also exploit relationships among use cases. To achieve this, you must look closely at the flow of events.

You can view a flow of events as several subflows—one main flow and one or more alternate flows—that, taken together, yield the total flow-of-events description. You can then reuse the description of a subflow in the flow of events of other use cases: Subflows in the description of one use case's flow of events may be the same as those of other use cases. Further along, in design, this can assist you because you should have the same objects perform this same behavior in all the relevant use cases; that is, only one set of objects should perform this behavior no matter which use case is executing.

If the behavior is common to more than two use cases or if it forms an independent part, the model might be clearer if you model the behavior as a use case of its own that can be reused by the original use cases. There are three ways to exploit commonality among use cases:

- Inclusion
- Extension
- Generalization/specialization

Several use cases can *include* a common flow-of-event text organized as a use case. This technique is similar to the factoring you do in programming by means of subprograms. In the ATM example, both Withdraw Money and Check Balance include "User enters PIN." There is no need to express the flow of events associated with the entry of the PIN two or three times in the same model. Both use cases could include a use case "Authenticate the User." But do not overuse inclusion and fall into the trap of doing some extensive functional decomposition.

A use case can *extend* an existing one. For example, in a telephone switch, a basic person-to-person telephone call can be extended by call-forwarding or conference-call functionality. The extension occurs at specific points, called *extension points,* in the extended use case. The extension can be thought of as "injecting" additional descriptive text into the extended use case, at the extension point, under particular conditions. Extension is used primarily to simplify complex flows of events, to represent optional behavior, or to handle exceptions. The result of modeling with extensions is that it should make the basic flow of events of a use case easily understandable and should make the maintenance of the use-case model easier over time. Unlike the earlier "include" example, in which the basic flow of events is incomplete without the inclusion, when you use "extend," the basic flow of events should still be complete; it should not have to refer to the extending flow of events to be understandable.

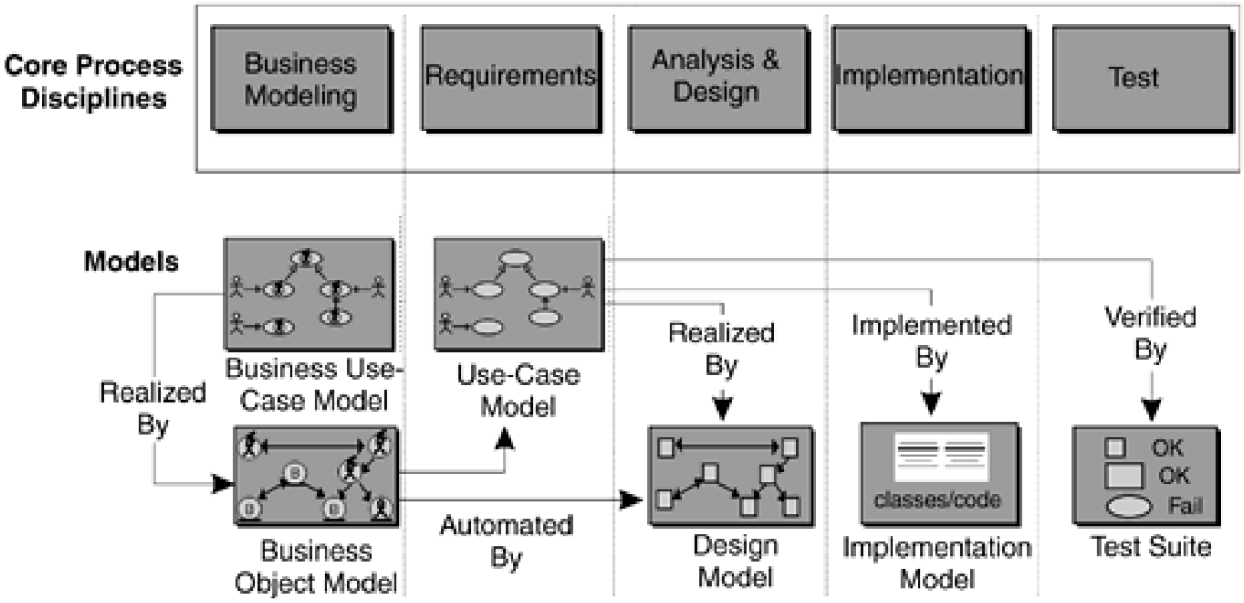
A use case can *specialize* a more general one by adding to or refining the original flow of events. For example, in an air traffic control system, the Entry of an ICAO Flight Plan follows the same basic principles and sequences of actions as the general Entry of a Flight Plan but has some refinements as well as specific steps or attributes. Specialization in the use-case model provides a way to model the behavior of common application frameworks and makes it easy to specify and develop variants of the system.

A common mistake is to overuse these three relationships, a practice that leads to the creation of a use-case model that is difficult to understand and maintain. It also generates a lot of overhead costs.

Use Cases in the Process

The Rational Unified Process is a use-case-driven approach. This means that the use cases defined for a system are the basis for the entire development process (see Figure 6-2).

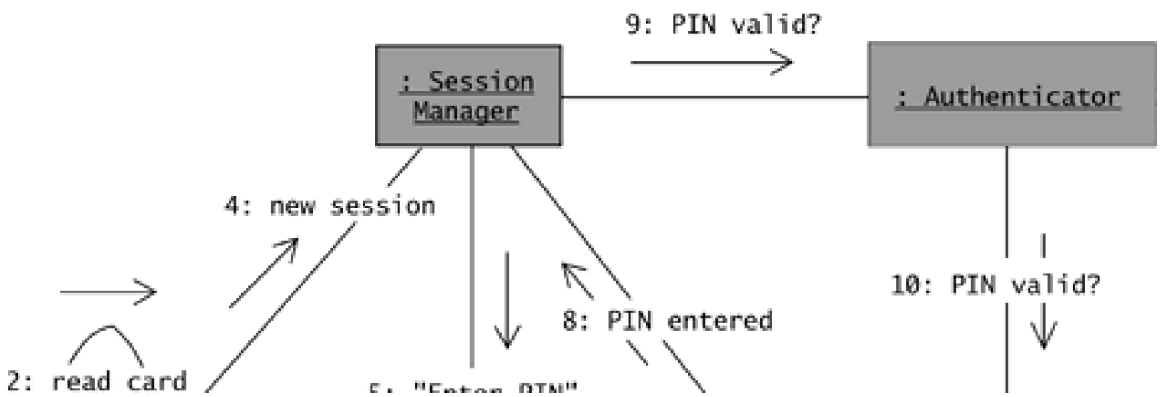
Figure 6-2. Use cases "flow" through the various models

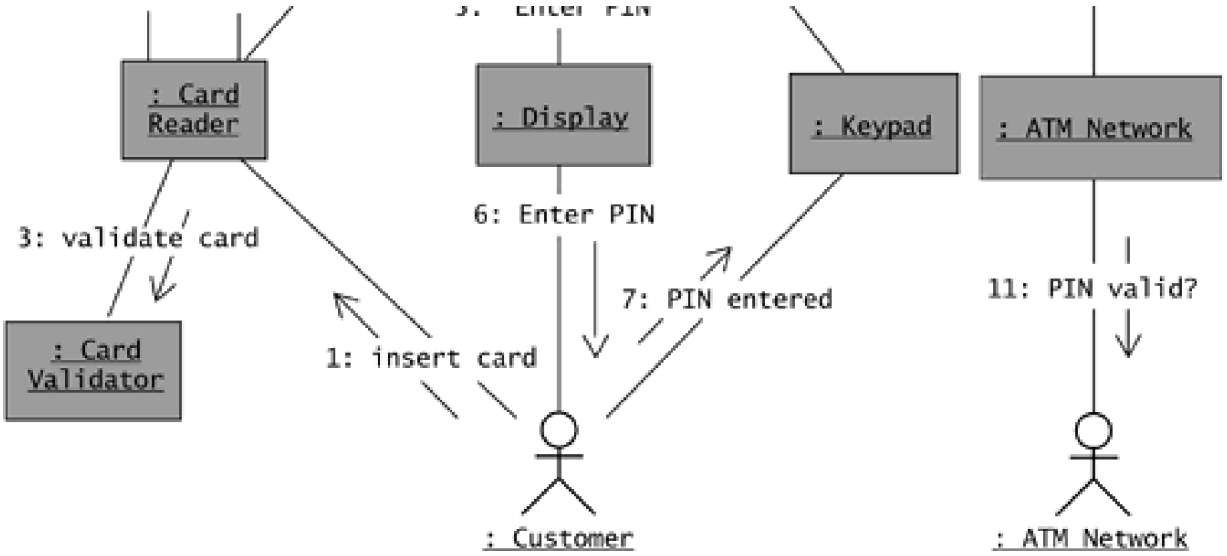


The use-case model is a result of the requirements discipline. In this front-end activity, the use cases are used to capture what the system should do from the user's point of view. Thus, use cases act as the common language for communication between the customer or users and the system developers.

In analysis and design, use cases are the bridge that unites requirements and design activities. They serve as the basis for use-case realizations, which describe how the use case is performed in terms of interacting objects in the design model. The objects and classes are most likely found by walking through the use cases. This technique ensures that all the required behavior is represented in the system design (see Figure 6-3).

Figure 6-3. Example of a collaboration diagram for the use case "Authenticate User" in an ATM





During implementation, the design model is the implementation specification. Because use cases are the basis for the design model, they are implemented in terms of design classes. Use-case realizations in the design model are used to understand the dynamics of the system and determine where to optimize for performance.

During testing, use cases constitute the basis for identifying test cases and test procedures. In other words, each use case is performed to verify the system.

Other activities are also attached to the use cases. For example, because the use cases specify how an actor (a user) interacts with the system, use cases provide much of the structure and content for user manuals. Conversely, when you're rushing to put a use-case model in place for a legacy system, consider using the user manuals as a starting point.

In project management, as you will see in Chapter 7, use cases are used to define the contents of iterations. Estimates of the effort can be derived from the use-case description by techniques such as function point analysis. In deployment, use-case packages can serve to plan a phased deployment or to define system variants. Definition and prototyping of user interfaces can be derived from use cases in the form of Use-Case Storyboards.

Optionally, in business modeling, we use the same concept of use case but at the level of the whole business rather than only of the system under consideration. The business use-case model describes high-level business processes and provides the context and the source of information for expressing the system's use cases. We describe this in Chapter 8, The Business Modeling Discipline.

For a thorough treatment of use case, consult the book by our colleagues Kurt Bittner and Ian Spence, *Use Case Modeling*, Boston: Addison-Wesley, 2003.

Summary

- Use cases are a means of expressing requirements on the functionality of the system.
- Written using concise simple prose, use cases are understandable by a wide range of stakeholders.
- Use cases help synchronize the content of various models.
- A use case is managed as a single unit throughout the development.
- Use cases may also be used to model the business, providing a context for the system development.
- Use cases are organized in a use-case model, which also expresses the relationships among them.
- Scenarios are described instances of use cases.
- Use cases drive numerous activities in the Rational Unified Process:

    - Creation and validation of the design model
    - Definition of the test cases and test procedures in the test model
    - Planning of iterations
    - Creation of user manuals
    - Deployment of the system

Part II: Process Disciplines

Chapter 7. The Project Management Discipline

This chapter introduces the concepts of risk and measures—two key elements in planning and controlling an iterative process. We describe how to plan an iterative process and how to decide on the duration and contents of an iteration.

Purpose

Software project management is the art of balancing competing objectives, managing risk, and overcoming constraints to deliver a product that meets the needs of the customers (the ones who pay the bills) and the end users. The fact that few projects are 100% successful is an indicator of the difficulty of the task. Our goal with the software project management discipline of the Rational Unified Process is to make the task as easy as possible by providing guidance in this area. It is not a recipe for success, but it presents an approach to managing the project that will markedly improve the odds of delivering software successfully.

The project management discipline has the following three purposes:

- To provide a framework for managing software-intensive projects
- To provide practical guidelines for planning, staffing, executing, and monitoring projects
- To provide a framework for managing risk

However, this discipline of the Rational Unified Process does not attempt to cover all aspects of project management.[1] For example, it does not cover issues such as the following:

[1] Project managers are invited to read a companion book by Walker Royce, *Software Project Management: A Unified Framework*. Reading, MA: Addison-Wesley, 1998.

- Managing people: hiring, training, coaching
- Managing budgets: defining, allocating
- Managing contracts with suppliers and customers

This discipline focuses on the specific aspects of an iterative development process:

- Planning an iterative project through the lifecycle and planning a particular iteration
- Risk management
- Monitoring the progress of an iterative project and measurements

Planning an Iterative Project

It is usually not very difficult to convince a project manager of all the benefits of an iterative project (see Chapter 4). But when it comes time to define one, project managers are often perplexed: Few of the traditional planning techniques seem to apply. Some new questions arise:

- How many iterations do I need?
- How long should they be?
- How do I determine the contents and objectives of an iteration?
- How do I track the progress of an iteration?

Among the objectives of project planning are the following:

- To allocate tasks and responsibilities to a team of people over time
- To monitor progress relative to the plan and to detect potential problems as the project is rolled out

Planning also deals with managing inanimate resources such as facilities, equipment, and budgets, but we will not cover this aspect because it is not affected significantly by the iterative approach.

Two Levels of Plan

There have been countless ambitious but doomed attempts to plan large software projects from start to finish in minute detail, as you would plan the construction of a skyscraper. As planners have tried to pin down activities and tasks at the person and day level several months or years in advance, I have seen the resulting Gantt charts and logic networks of activities cover the walls of several rooms and lobbies. For such plans to be realistic, you must have a very good understanding of what will be built, you must have stable requirements and a stable architecture, and you must have built a similar system from which you can derive a detailed work breakdown structure (WBS).

But how can you plan to have Joe code module GGART in week 37 if you do not even know about the existence of module GGART?

The approach works well in industries in which the WBS is more or less standard and stable and the ordering of tasks is fairly deterministic because of, for example, the underlying laws of physics. When constructing a building, you cannot bring up floors 1 and 4 in parallel unless you've done work on floors 2 and 3.

In an iterative process, we recommend that the development be based on two kinds of plans:

- A coarse-grained plan: the phase plan
- A series of fine-grained plans: the iteration plans

The Phase Plan (or Project Plan)

The *phase plan* is a coarse-grained plan, and there is only one per development project. It captures the overall "envelope" of the project for one cycle (and maybe the following cycles, if appropriate). It can be summarized as follows:

- *Dates of the major milestones*
  - Lifecycle objective (end of inception, project well scoped and funded)
  - Lifecycle architecture (end of elaboration, architecture complete)
  - Initial operational capability (end of construction, first beta)
  - Product release (end of transition and of the cycle)
- *Staffing profile*: which resources are required over time
- *Dates of the minor milestones*: end of each iteration and its primary objective, if it is known

The phase plan is produced very early in the inception phase and is updated as often as necessary. It does not require more than one or two pages. It refers to the vision document to define the scope and assumptions of the project (see Chapter 9).

The Iteration Plan

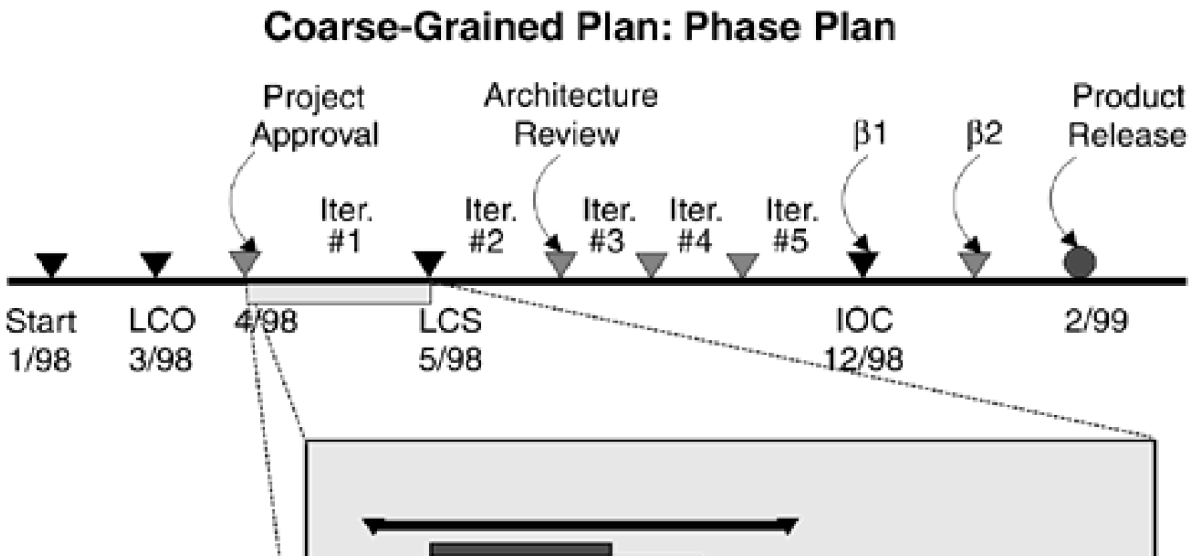
An iteration plan is a fine-grained plan, and there is one per iteration. A project usually has two iteration plans "active" at one time:

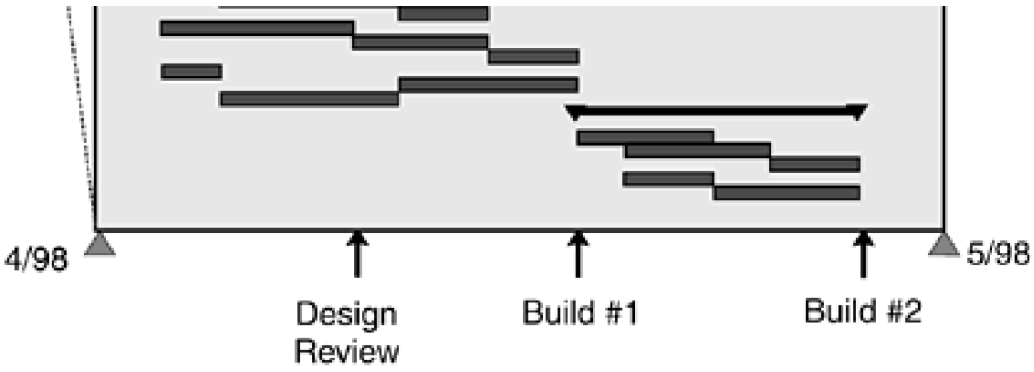
- The current iteration plan (the one for the current iteration), which is used to track progress
- The next iteration plan (the one for the pending iteration), which is built toward the second half of the current iteration and is ready at the end of the current iteration

The iteration plan is built using traditional planning techniques and tools (Gantt charts and so on) to define the tasks and their allocation to individuals and teams. The plan contains important dates, such as major builds, arrival of components from other organizations, and major reviews.

You can picture the iteration plan as a window moving through the phase plan, acting as a magnifier, as shown in Figure 7-1.

Figure 7-1. Project plan and iteration plan





Fine-Grained Plan: Iteration Plan

Because the iterative process is dynamic and is meant to accommodate changes in goals and tactics, no purpose is served by spending an inordinate amount of time producing detailed plans that extend beyond the current planning horizon. Such plans are difficult to maintain, rapidly become obsolete, and typically are ignored by the performing organization. The iteration plan covers about the right span of time and is the right granularity to do a good job of detailed planning.

[Team LiB]

PREVIOUS NEXT

The Concept of Risk

As Tim Lister says, "All the risk-free projects have been done."[2] The software development process primarily takes care of the *known* aspects of software development. You can precisely describe, schedule, assign, or review only what you know must be done. Risk management takes care of the *unknown* aspects. Many organizations work in a "risk denial" mode: estimating and planning proceed as if all variables were known and as if the work were mechanical and the personnel interchangeable.

[2] Tim Lister, "Software Risk Management Is Software Project Management," Seminar at Software Productivity Center, Vancouver, B.C., Canada, May 1996.

What Is a Risk?

Many decisions in an iterative lifecycle are driven by risks. To make effective decisions, you need a good grasp of the risks the project faces and clear strategies for mitigating or dealing with them.

In everyday life, a risk is an exposure to loss or injury or a factor, thing, element, or course that involves uncertain danger. We can define *risk* more specifically in software development as a variable that, within its normal distribution, can take a value that endangers or eliminates success for a project.

In plain terms, a risk is whatever may stand in our way to success and is currently unknown or uncertain. We can define *success* as meeting the entire set of all requirements and constraints held as project expectations by those in power.

We can qualify risks further as direct or indirect:

- *Direct risk:* a risk over which the project has a large degree of control
- *Indirect risk:* a risk over which the project has little or no control

We can also add two important attributes:

- The probability of occurrence (its likelihood)
- The impact on the project (its severity)

These two attributes often can be combined in a single *risk magnitude indicator,* and five discrete values are sufficient: high, significant, moderate, minor, and low.

Strategies: How to Cope with Risks

The key idea in risk management is that you not wait passively until a risk becomes a problem (or kills the project) before you

decide what to do with it. For each perceived risk, you must decide in advance what you are going to do.

Three main routes are possible.[3]

[3] Barry W. Boehm, "Software Risk Management: Principles and Practice," *IEEE Software*, Jan. 1991, pp. 32–41.

- *Risk avoidance:* Reorganize the project so that it cannot be affected by the risk.
- *Risk transfer:* Reorganize the project so that someone or something else bears the risk (the customer, vendor, bank, or another element).
- *Risk acceptance:* Decide to live with the risk as a contingency. Monitor the risk symptoms and determine what to do if the risk materializes.

When accepting a risk, you should do two things:

- *Mitigate the risk:* Take immediate, proactive steps to reduce the probability or the impact of the risk.
- *Define a contingency plan:* Determine the course of action to take if the risk becomes an actual problem; in other words, create a "plan B."

Risks play a major role in planning iterations, as you will see later.

The Concept of Measurement

Why do we measure? We measure primarily to gain control of a project and therefore to manage it. We measure to evaluate how close or far we are from the plan's objectives in terms of completion, quality, and compliance with requirements. We also measure so that we can plan better a new project's effort, cost, and quality based on experience. Finally, we measure to evaluate the effects of changes and assess how we have improved over time the key aspects of the process's performance (see Chapter 17).

Measuring key aspects of a project adds a non-negligible cost, so we do not measure something simply because we can. We must set precise goals for a measurement effort and collect only measurements that allow us to satisfy these goals. There are two kinds of goals:[4]

[4] Kevin Pulford, Annie Kuntzmann-Combelles, and Stephen Shirlaw, *A Quantitative Approach to Software Management—The ami Handbook*. Reading, MA: Addison-Wesley, 1995.

- *Knowledge goals*

  These goals are expressed by the use of verbs such as evaluate, predict, and monitor. They express a desire to understand your development process better. For example, you may want to assess product quality, obtain data to predict testing effort, or monitor test coverage or requirements changes.

- *Change or achievement goals*

  These goals are expressed by the use of verbs such as increase, reduce, improve, and achieve. They express an interest in seeing how things change or improve over time, from one iteration to another, and from one project to another.

The following are examples of goals that might be set in a software development effort:

- Monitor progress relative to the plan.
- Improve customer satisfaction.
- Improve productivity.
- Improve predictability.
- Increase reuse.

These general management goals do not translate readily into measurements. We must translate them into subgoals (or *action* goals), which identify the actions that project members must take to achieve the goal. We also must make sure that the people involved understand the benefits.

For example, the goal "Improve customer satisfaction" would break down into the following action goals:

- Define customer satisfaction.
- Measure customer satisfaction over several releases.
- Verify that satisfaction improves.

The goal "Improve productivity" would include these subgoals:

- Measure effort.
- Measure progress.
- Calculate productivity over several iterations or projects.
- Compare the results.

Some of the subgoals (but not all of them) would require that you collect measurements. For example, "Measure customer satisfaction" can be derived from the following:

- A customer survey (in which the customers would give marks for different support aspects)
- The number and severity of calls to a customer-support hotline

What Is a Measurement?

There are two kinds of measurement:

1. A *measure* is a concrete numeric attribute of an entity (e.g., a number, a percentage, a ratio). For example, project effort expressed in person-months is one measure of a project's size. To calculate this measure you would need to sum all the time-sheet bookings for the project.

2. A *primitive measure* is an item of raw data that is used to calculate a measure. In the preceding example, the time-sheet bookings are the primitive measures. A primitive measure typically exists in a database but is not interpreted in isolation.

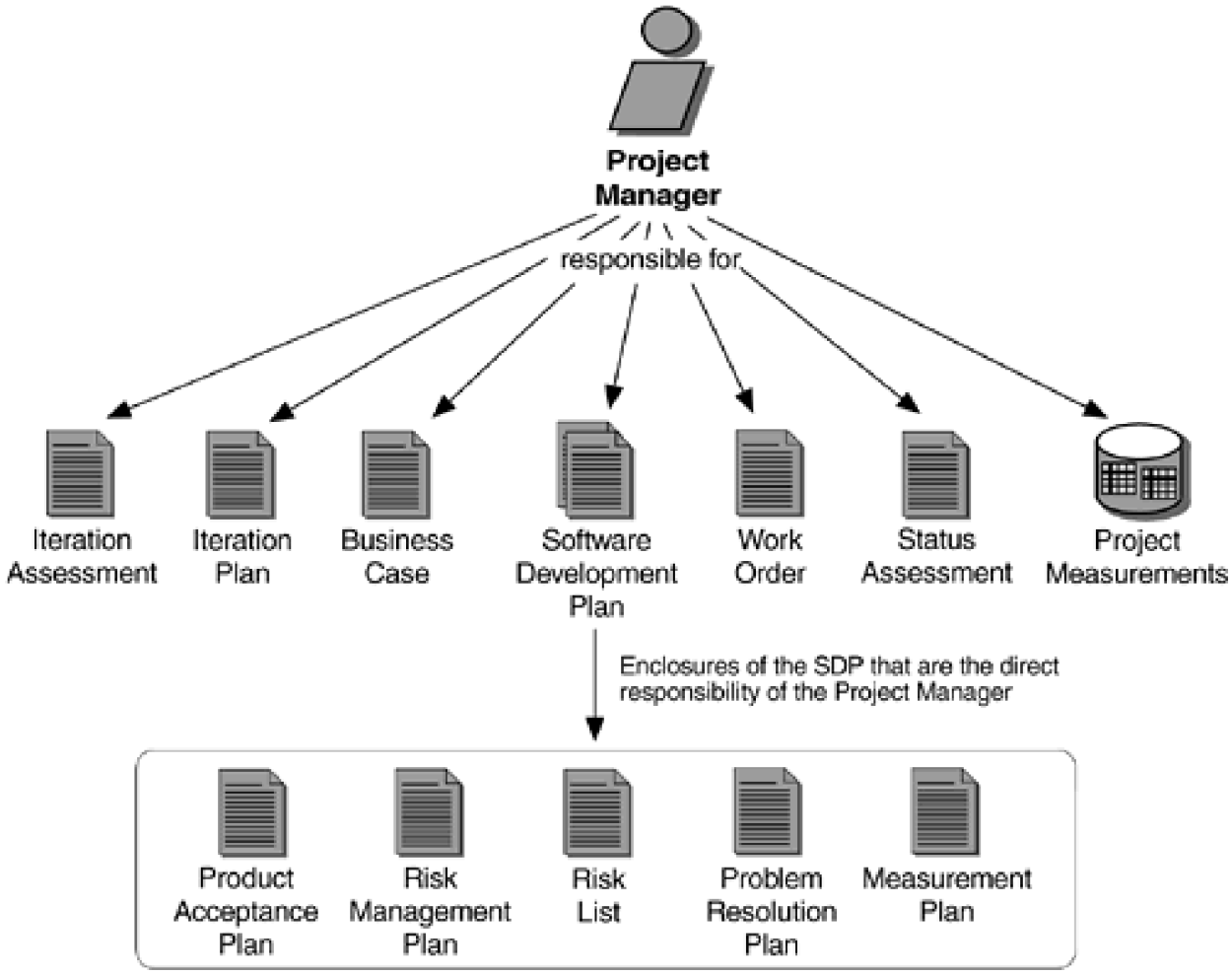
Each measure comprises one or more collected measurements. Consequently, each primitive measure must be clearly identified, and its collection procedure must be defined.

Measures to support change or achievement goals are often *first-derivative* over time (or iterations or project). We are interested in a trend and not in the absolute value of the measure. If our goal is "Improve quality," we must check that the residual level of known defects diminishes over time.

Roles and Artifacts

In the Rational Unified Process, how is all this translated concretely in terms of roles, artifacts, activities, and workflows? Figure 7-2 shows the major role in the project management discipline, the Project Manager, and the artifacts of the discipline. Not shown in the figure is the Project Reviewer, who is responsible for evaluating project planning artifacts and project assessment artifacts at major review points in the project lifecycle.

Figure 7-2. Role and artifacts in the project management discipline



The key artifacts of the project management discipline are as follows:

- The software development plan (SDP), which contains several artifacts:
  - Product acceptance plan
  - Risk management plan (and risk list)
  - Problem resolution plan

- Measurement plan

- The business case
- The iteration plans (one per iteration)
- The iteration assessment
- The (periodic) status assessment
- The work order
- The project measurements database

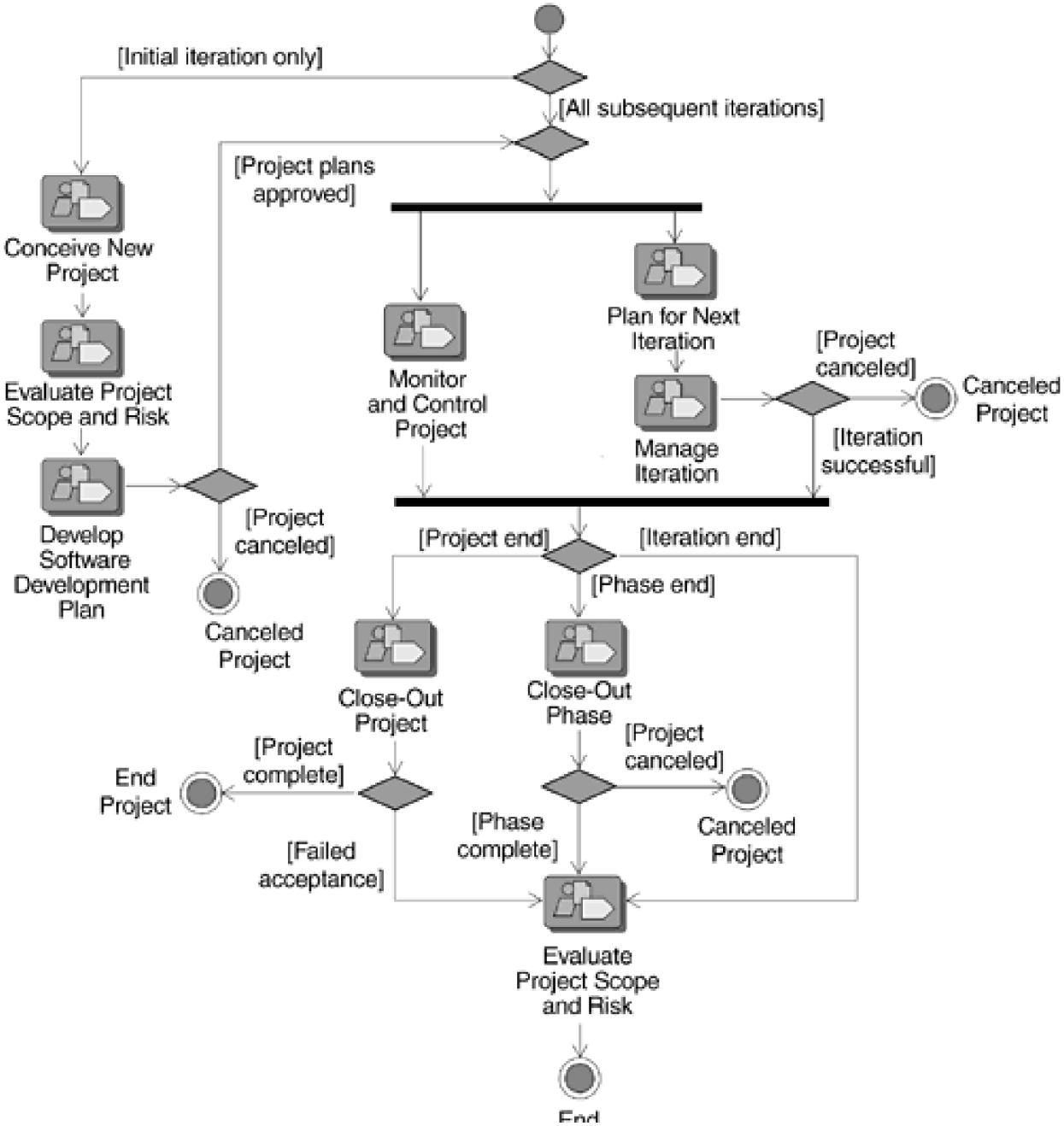
Other plans are part of the SDP, but they are developed by other roles. Here are two examples:

1. Configuration management plan, developed by the Role:Configuration Manager (see Chapter 13)
2. Development case (the process used for the project), developed by the Role:Process Engineer (see Chapters 14 and 17)

Workflow

Figure 7-3 illustrates an iteration in project management in UML activity diagram form. Each activity state represents a workflow detail in the Rational Unified Process, for example, Conceive New Project. Each workflow detail, in turn, is composed of one or more activities. Observe that some workflow details are time-dependent. For example, Conceive New Project is performed only once, at the start of the project, and Close-Out Phase is performed only when terminating the final iteration of each phase.

Figure 7-3. Workflow in project management



End
Iteration

Workflow Details

The workflow is decomposed into a set of workflow details.

Conceive New Project

This workflow detail is composed of the activities Identify and Assess Risks, Develop Business Case, and Initiate Project, performed by the Project Manager, and the Project Approval Review, performed by the Project Reviewer. The purpose of this workflow detail is to bring a project from the germ of an idea to a point at which a reasoned decision can be made to continue or abandon the project.

Evaluate Project Scope and Risk

This workflow detail is composed of the activities Identify and Assess Risks and Develop Business Case, performed by the project manager. The purpose of this workflow detail is to reappraise the project's intended capabilities and characteristics and the risks associated with achieving them. This evaluation is done once after the preferred approach is chosen and the project is initiated to give a solid basis for detailed planning and again at the end of each iteration as more is learned and risks are retired.

Develop Software Development Plan

This workflow detail is composed of the activities:

- Develop measurement plan.
- Develop risk management plan.
- Develop product acceptance plan.
- Develop problem resolution plan.
- Define project organization and staffing.
- Define monitoring and control processes.
- Plan phases and iterations.
- Compile software development plan.

These are performed by the Project Manager, and the Project Planning Review is performed by the Project Reviewer.

The purpose of this workflow detail is to develop the components and enclosures of the software development plan and then have them formally reviewed for feasibility and acceptability to stakeholders and as the basis for the fine-grained plan for the next

iteration (the iteration plan). The major effort in creating these artifacts comes early in the inception phase; thereafter, when this workflow detail is invoked at the beginning of each iteration, it is to revise the software development plan (and its enclosures) on the basis of the previous iteration's experience and the iteration plan for the next iteration. The Project Manager will also collate all other contributions to the software development plan.

Plan for Next Iteration

This workflow detail is composed of the activities Develop Iteration Plan, Develop Business Case, and the workflow detail Develop Software Development Plan, performed by the Project Manager, and the Iteration Plan Review, performed by the Project Reviewer. The purpose of this workflow detail is to create an iteration plan, which is a fine-grained plan, to guide the next iteration. After creating the plan, adjustments may be needed to the software development plan (for example, if planning the next iteration changed the allocation of functionality to subsequent iterations) and the business case (for example, if costs change or the return-on-investment calculation is affected by changes to the availability dates of important features in the software).

Monitor and Control Project

This workflow detail is composed of the activities Schedule and Assign Work, Monitor Project Status, Handle Exceptions and Problems, and Report Status, all performed by the Project Manager, and the Project Review Authority (PRA) Review, performed by the Project Reviewer. This workflow detail captures the daily, continuing work of the Project Manager and covers:

- Dealing with change requests that have been sanctioned by the Change Control Manager, and scheduling these for the current or future iterations
- Continuously monitoring the project in terms of active risks and objective measurements of progress and quality
- Regularly reporting project status, in the status assessment, to the PRA, which is the organizational entity to which the Project Manager is accountable
- Dealing with issues and problems as they are discovered and driving these to closure according to the problem resolution plan

Manage Iteration

This workflow detail is composed of the activities Acquire Staff, Initiate Iteration, and Assess Iteration, all performed by the Project Manager, and the Iteration Evaluation Criteria Review and the Iteration Acceptance Review, both performed by the Project Reviewer. This workflow detail contains the activities that begin, end, and review an iteration. The purpose is to acquire the necessary resources to perform the iteration, allocate the work to be done, and finally, to assess the results of the iteration. An iteration concludes with an iteration acceptance review, which determines, from the iteration assessment artifact, whether the objectives of the iteration were met.

Close-Out Phase

This workflow detail is composed of the activities Prepare for Phase Close-Out, performed by the Project Manager, and the Lifecycle Milestone Review, performed by the Project Reviewer. In this workflow detail, the Project Manager brings a phase to

closure by ensuring the following:

- All major issues from the previous iteration are resolved.
- The state of all artifacts is known (through configuration audit).
- Required artifacts have been distributed to stakeholders.
- Any deployment (e.g., installation, transition, and training) problems are addressed.
- The project's finances are settled if the current contract is ending (with the intent to recontract for the next phase).

A final phase status assessment is prepared for the lifecycle milestone review, at which phase artifacts are reviewed. If project state is satisfactory, sanction is given to proceed to the next phase.

Close-Out Project

This workflow detail is composed of the activities Prepare for Project Close-Out, performed by the Project Manager, and the Project Acceptance Review, performed by the Project Reviewer. In this workflow detail, the Project Manager readies the project for termination. A final status assessment is prepared for the project acceptance review, which, if successful, marks the point at which the customer formally accepts ownership of the software product. The Project Manager then completes the close-out of the project by disposing of the remaining assets and reassigning the remaining staff.

Let us now examine some important aspects of these activities in more detail.

Building a Phase Plan (or Project Plan)

To start building a phase plan, you need at least a rough estimate of the "size of the mountain." You must consider, for example:

- How high is it (total effort)?
- When do I need to arrive at the top (date of final release)?

You then work backward from the end date to "plant" tentative dates for the major milestones.

Staff/Schedule/Scope Trade-off

Many studies (and common sense) have shown again and again that you cannot trade staff for schedule. This is the classic example: "If it takes nine months for a woman to make a baby, why can't we have nine women produce one in a month?" As Fred Brooks wrote, "Adding manpower to a late software project makes it later."[5] You can use a cost model, such as COCOMO (Constructive Cost Model), to validate that you have the right relationship among effort, elapsed time, and staffing level.

> [5] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Reading, MA: Addison-Wesley, 1995.

To reach a reasonable ratio in your product's first generation, you usually must trade off features, or you must be creative in other

ways, such as increasing reuse. Note that you could trade off another parameter—quality—but that is another discussion.

The Rubber Profile

After you have an idea of the limits of the three variables, you must shape your effort profile and refine the dates of milestones, taking into account the specific circumstances of your project. To do this you can start from a typical profile. The profile in Figure 7-4 shows the relative duration and effort per phase. It is suitable for a project that has the following characteristics:

- It is of moderate size and effort.
- It is in an initial development cycle.
- It has no preexisting architecture.
- It has a small number of risks and unknowns.

Figure 7-4. Typical project profile

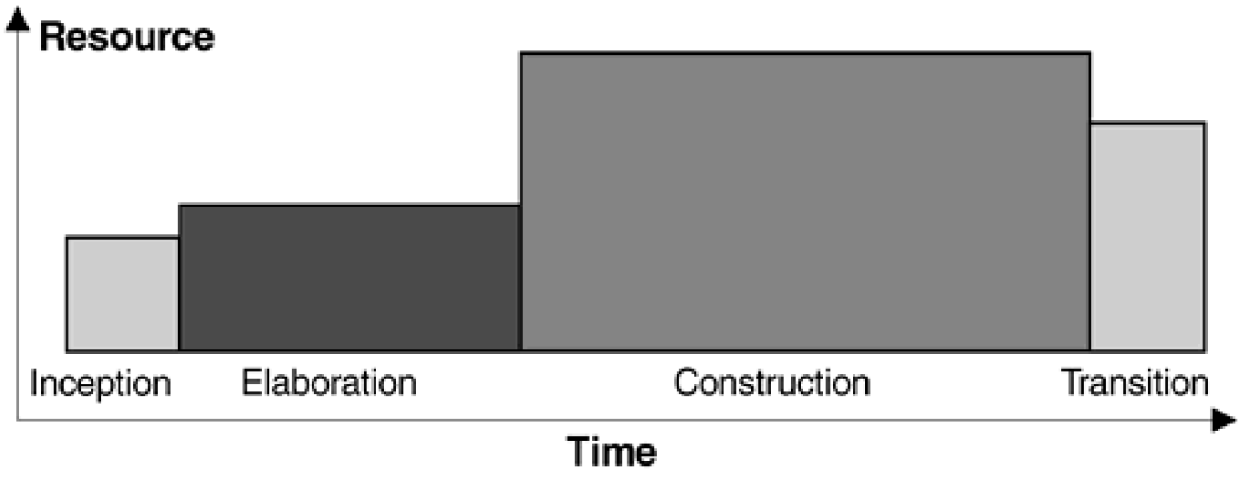


Table 7-1 shows the profile in tabular fashion.

Now let's assume that this profile is made of rubber. Let's stretch it and massage it to fit your circumstances using the following heuristics:

- If you need a long time to scope the project, find the funding, conduct market research, or build an initial proof-of-concept prototype, stretch the inception phase.
- If you have no architecture in place, if you envisage using new and unknown technology, and/or if you have severe performance constraints, a number of technical risks, and a lot of new staff, lengthen the elaboration phase.

Table 7-1. Relative Weight of the Phases of Schedule and Effort for a Typical Project

Phase	Schedule	Effort
Inception	10%	5%
Elaboration	30%	20%
Construction	50%	65%
Transition	10%	10%

- If this is the second generation of a product (an evolution cycle) and if you will make no major changes to the architecture, shrink the inception and elaboration phases.
- If you must hit the market quickly—because you are late or because you are creating the market—and plan to finish the product gradually, you can shorten the construction phase and lengthen the transition phase.
- If you have a complex deployment, such as replacing an old system without interruption of service, or if you have regulatory requirements to satisfy or certification to obtain (as in domains such as medical instrumentation, nuclear industry, avionics, and public telephony), you may have to stretch the transition phase.

Again and again, verify that you are not overstaffing the project.

So, now you have some estimates for the dates of the various major milestones—LCO, LCA, IOC, and PR—and a rough staffing profile across the cycle.

Another element to consider is *how many* iterations you will perform in each phase. Before deciding this, let's first discuss the issue of the duration of an iteration.

Duration of an Iteration

We have defined an iteration as a nearly complete miniproject in which you go through all major workflows, which results, in most cases, in an executable yet incomplete system. Although the cycle (edit, compile, test, debug) sounds like an iteration, it is not what we have called iteration here. The daily or weekly build, in which you incrementally integrate and test increasing numbers of elements of the system, may sound like an iteration, but such builds are not what we call an iteration. An iteration starts with planning and requirements and ends with a release, either internal or external.

Ideally, an iteration should span from two to six weeks, but this varies from project to project. How quickly you can iterate depends primarily on the size of the development organization. Here are some examples:

- Five people can do some planning on a Monday morning, have lunch together every day to monitor progress, reallocate tasks, start doing a build on Thursday, and complete the iteration by Friday evening.
- With 20 people, achieving this scenario would be rather difficult. It would take more time to distribute the work, synchronize the subgroups, and integrate. An iteration might take three or four weeks.
- With 40 people, it takes a week for the nervous influx to go from the brain to the extremities. You have intermediate levels of management, and the common understanding of the objective will require more formal documentation and more ceremony. Three months is a more reasonable iteration length.

Other factors come into play: the degree of the organization's familiarity with the iterative approach; the stability and maturity of the organization; and the level of automation used by the team to manage code (for example, distributed configuration management), distribute information (such as through an internal Web), and perform testing. Also, be aware that an iteration has some fixed overhead for planning, synchronizing, and analyzing the results.

Convinced by the tremendous benefits of the iterative approach, you might be tempted to iterate furiously, but the human limits of your organization will cool your fervor.

Although this is just an empirical approach, Table 7-2 gives some order of magnitude of iteration duration we collected on a few actual iterative projects.[6]

[6] Joe Marasco looked at this sample of projects and noted that the duration *D*, in weeks, was related to the size of the project, *S* (in thousands of lines of code), by the formula $D_{weeks} = \sqrt{S_{ksloc}}$, but you should take this with a grain of salt.

Table 7-2. Iteration Duration for a Range of Iterative Projects

Lines of Code	Number of People	Duration of an Iteration
5,000	4	2 weeks
20,000	10	1 month
100,000	40	3 months
1,000,000	150	6 months

Number of Iterations

After you have an idea of the duration of iteration your organization can tolerate, you can complement the "rubber profile" heuristics by looking at how many iterations you should perform in each phase.

Often, in the inception phase, there will be no real iteration; no software is produced, and there are only planning and marketing activities. In some cases, however, you will have an iteration for the following:

- Building a prototype to convince yourself or your sponsor (perhaps your customer or a venture capitalist) that your idea is a good one
- Building a prototype to mitigate a major technical risk such as trying out a new technology or a new algorithm or verifying that a performance objective is attainable
- Getting your organization up to speed with tools and process

Remember that the goal of inception is not to accumulate code. After all, you want these answers quickly.

So that's zero or one iteration.

In the elaboration phase, you should plan at least one iteration. If you have no architecture to start with and must accommodate a lot of new factors—new people, tools, technology, platform, or programming language—then you should plan two or three iterations. You cannot tackle all the risks at once. You may need to show a prototype to the customer or end users to help them define the requirements better (remember the IKIWISI effect). You will need an iteration to correct your early mistakes on the architecture.

So this gives us one to three iterations.

In the construction phase, you should plan at least one iteration. Two is more reasonable if you want to exploit the benefits of

iterative development and allow a better job of integration and testing. For more ambitious projects, three or more iterations are even better if the organization can support the stress and if there is a sufficient level of automation and process maturity.

So that's one to three iterations.

In the transition phase, plan at least one iteration, for example, final release after beta. Too often, the realities of the market or the (poor) quality of the initial release will force you to do more iterations.

That's one to two iterations.

So over the entire development cycle [I, E, C, T], we have three levels:

Low: three iterations [0, 1, 1, 1]

Typical: six iterations [1, 2, 2, 1]

High: nine iterations [1, 3, 3, 2]

We can summarize by saying that "normal" projects have 6 ± 3 iterations. We call this our "six plus or minus three" rule of thumb.

PREVIOUS NEXT

Building an Iteration Plan

Now let's examine how to build a fine-grained plan for one iteration, an activity that you will repeat once per iteration. We have seen that the nature and focus of iterations vary over time from phase to phase. Although the criteria that we will take into consideration will vary through the lifecycle, the detailed recipe remains the same. So we take an iteration in the elaboration phase as our main example, and later we describe how iterations differ in other phases.

First, consider the iteration, its length, and the resources you have allocated to it to be the bounding box. The idea is to avoid a situation in which you first define ambitious goals and plan to achieve them, only to discover that one iteration is not enough. You must define only enough work to fill the iteration. You are scheduling by time and not by volume.

The iteration plan describes at a fine level of granularity what will happen in the iteration. It allocates activities to roles and allocates individuals to roles. (Remember that we use the term *role* in a special sense; seeChapter 3.) A planning tool, such as Microsoft Project, is useful for handling the details, in particular the dependencies and allocation of tasks to individuals.

To build an iteration plan, follow these four steps:

1. Define objective criteria for the success of the iteration. You will use these criteria in an iteration assessment review at the end of the iteration to decide whether the iteration was a success or a failure.
2. Identify the concrete, measurable artifacts that will need to be developed or updated and the activities that will be required to achieve this.
3. Beginning with a typical iteration work breakdown structure, and massage it to take into account the actual activities that must take place.
4. Use estimates to assign duration and effort to each activity, keeping all numbers within your budget.

Now let's look at how an iteration varies from phase to phase.

Iteration in the Elaboration Phase

There are three main drivers for defining the objectives of an iteration in the elaboration phase:

- Risk
- Coverage
- Criticality

The main driver to determine the iteration objectives is risk. You must mitigate or retire your risks as early as you can. This is especially the case in the elaboration phase, when most of your risks should be mitigated, but it can continue to be a key driver in the construction phase as some risks remain high or new risks are discovered.

But because the goal of the elaboration phase is to baseline an architecture, other considerations must come into play, such as ensuring that the architecture addresses all aspects of the software to be developed (coverage). This process is important

because the architecture will be used for further planning: organization of the teams, estimation of code to be developed, and product structure of the development environment.

Although focusing on risks is important, you should keep in mind the primary missions of the system. It is good to solve all the difficult issues, but it must not be done to the detriment of the core functionality: Make sure that the critical functions or services of the system are covered (criticality) even if no perceived risk is associated with them.

Let's assume that we have a current, up-to-date list of project risks. For the most damaging risks, identify a scenario in one use case that would force the development team to "confront" the risk. The following are two examples:

1. If there is an integration risk, such as "database D working properly with OS Y," be sure to include one scenario that involves some database interaction even if it is only a modest one.

2. If there is a performance risk, such as "excessive time to compute the trajectory of the aircraft," be sure to include one scenario that includes this computation, at least for the most obvious and frequent case.

For criticality, make sure to include the most fundamental function or services provided. Select some scenarios from the use cases that represent the most common or the most frequent form of the service or feature offered by the system.

For example, for a telephone switch, the plain station-to-station call is the obvious "must" for an early iteration. It's far more important to get this right than to perfect the convoluted failure modes in operator configuration of the error-handling subsystem.

For coverage, toward the end of the elaboration phase, include scenarios that touch areas you know will require development even if these scenarios are neither critical nor risky.

It is often economical to create long end-to-end scenarios that address multiple issues at once. For example, suppose that one single scenario is critical and confronts two technical risks. Try to achieve some balance and avoid scenarios that are too "thick" too early—that is, scenarios that try to cover too many aspects, variants, and error cases.

Also, in the elaboration phase, keep in mind that some of the risks may be of a more human nature, such as managing change, fostering a team culture, training team members, and introducing new tools and techniques. Simply reviewing them during an iteration tends to mitigate these risks, but introducing too many changes at the same time becomes destabilizing.

The following are four examples of iteration objectives:

1. *Create one subscriber record on a client workstation; the record is to be stored in the database on the server, including user dialog but not including all fields; assume that no errors are detected*. This scenario combines critical functions with integration risks (database and communication software) and integration issues (dealing with two different platforms). It also forces designers to become familiar with a new GUI design tool. Finally, it leads to production of a prototype that can be demonstrated to end users for feedback.

2. *Make sure that as many as 20,000 subscribers can be created and that access to one subscriber takes no longer than 200 milliseconds*. This scenario addresses key performance issues (data volume and response time) that can dramatically affect the architecture if they are not met.

3. *Undo a change of subscriber address*. This scenario picks a simple feature that forces you to think about a design of all "undo" functions. This may, in turn, trigger the need to communicate with the end users about what can be undone at reasonable cost.

4. *Complete all the use cases relative to supply-chain management*. The goal of the elaboration phase is also to complete the capture of requirements.

Iteration in the Construction Phase

As the project moves into the construction phase, risks remain a key driver, especially as new and unsuspected risks are uncovered. But the completeness of use cases also becomes a driver. You can plan the iterations feature by feature, trying to complete some of the more critical ones early so that they can be tested thoroughly in more than one iteration. Toward the end of

the construction phase, the main goal will be to ensure coverage of full use cases.

The following are examples of iterations in this phase:

1. *Implement all variants of call forwarding, including erroneous ones*. This is a set of related features. One of them may have been implemented during the elaboration phase, and it will serve as a prototype for the rest of the development.
2. *Complete all telephone operator features except night service*. This is another set of features.
3. *Achieve 5,000 transactions per hour on a two-computer setup*. This may increase the required performance relative to what was actually achieved in the previous iteration (only 2,357/hr).
4. *Integrate a new version of geographical information system*. This may be a modest architectural change, perhaps necessitated by a problem discovered earlier.
5. *Fix all level 1 and level 2 defects*. This fixes the defects discovered during testing in the preceding iteration that were not fixed immediately.

Iteration in the Transition Phase

The main goal is to finish this generation of the product. Objectives for an iteration are stated in terms of which bugs are fixed and which improvements in performance or usability are included. If you had to drop (or disable) features to get to the end of the construction phase (the IOC milestone or first beta) on time, you can now complete them or turn them on if they won't jeopardize what has been achieved so far.

The following are five examples of iteration goals in this phase:

1. *Fix all severity 1 problems discovered at beta customer sites*. A goal in terms of quality, this may be related to credibility in the market.
2. *Eliminate all start-up crashes caused by mismatched data*. This is another goal expressed in terms of quality.
3. *Achieve 2,000 transactions per minute*. This is performance tuning and involves some optimization: data structure change, caching, and smarter algorithms.
4. *Reduce the number of different dialog boxes by 30%*. Improve usability by reducing visual clutter.
5. *Produce German and Japanese versions*. The beta was produced only for English-speaking customers because of the lack of time and to reduce rework on the beta release.

Detail the Work in the Iteration

Now that we have objectively defined the pending iteration's goal, we can proceed to the detailed planning stage, as for any other project.

After the scenarios or full-blown use cases to be developed (plus defects to be fixed) have been selected and briefly sketched, you must determine the artifacts that will be affected:

- Which classes are to be revisited?
- Which subsystems are affected or created?
- Which interfaces probably need to be modified?

- Which documents must be updated?

The next step is to identify in the Rational Unified Process the activities that are involved and to place them in your project plan. Some activities are done once per iteration (building an iteration plan), whereas others must be done once per class, per use case, or per subsystem (designing a class). Connect the activities with their obvious dependencies and allocate an estimated effort. Most of the activities described in the process are small enough to be accomplished by one person or a small group of people in a matter of a few hours or a few days.

You will probably discover that there isn't enough time in the iteration to accomplish all this work. Rather than extend the iteration (and thereby extend the final delivery time or reduce the number of iterations), reduce the ambitiousness of the iteration. Depending on which phase you are in, make the scenarios simpler or eliminate or disable features.

Summary

- The project management discipline of the Rational Unified Process is useful for balancing competing objectives, managing risk, and overcoming constraints to deliver a product that meets the needs of the customers (the ones who pay the bills) and the end users.
- In an iterative process, the development should be based on a phase plan and a series of iteration plans.
- Risk is a driver for planning.
- Measurement is a key technique used to control projects.
- In building a phase plan (or overall project plan), you must assess trade-offs among staff, schedule, and project scope.
- The criteria to define the scope of an iteration vary from phase to phase.

Chapter 8. The Business Modeling Discipline

This chapter explains why you might perform business modeling before a system development effort and discusses several ways of doing it. It also describes how to derive software requirements from business models.

Purpose

The goals of business modeling are as follows:

- To understand the structure and the dynamics of the organization in which a system is to be deployed (the target organization)
- To understand current problems in the target organization and identify improvement potentials
- To ensure that customers, end users, and developers have a common understanding of the target organization
- To derive the system requirements to support the target organization

To achieve these goals, the business modeling discipline describes how to develop a vision of the new target organization and, based on this vision, to define the processes, roles, and responsibilities of that organization in a model of the business. This model comprises a business use-case model and a business object model.

Why Business Modeling?

The majority of software applications are no longer "gizmos" built by programming wizards to be used by computer hobbyists who appreciate technically elegant features for their own sake. Instead, most applications are everyday tools used by people in their working environments as well as at home. These applications need to fit intuitively into the organization in which they are used. We must be sure that the applications we build help people in their daily chores and that they are not perceived as gadgets that do not offer added value. To fulfill these needs better, the new standard is to try to understand the business domain before, or in parallel with, a software engineering project.

This situation is becoming even more obvious as organizations are preparing themselves to enter the world of e-business. Then, what is *e-business,* this buzzword that is so frequently used? As with all terms of this kind, its meaning depends somewhat on the messenger. Our definition is that e-business is about building systems (sometimes called business tools) that automate business processes. In a sense, the business tools *are* the business, and they differentiate you from your competitors. For example, an e-commerce business tool automates the sales process.

We see organizations developing e-business software that now consider business modeling the most central part of their projects. They use model-based technologies to develop software fast and in a controlled manner. They involve the CEOs and the marketing directors of the target organizations in developing these models, where before they would typically communicate with some level of "business domain experts" who might have known how business is run but were not empowered to make decisions about changing it.

E-business development is *more* than just automating the existing processes; it forces some reflection on the nature of the business and the way it is run. Business modeling and system definition are of interest not only to people in the IT department; they are of concern to everyone involved in business development. A project to develop a new business tool will involve people from all parts of the organization, from executives with the power to make decisions to "grass roots" and end users who feel the consequences of those decisions.

The business tools built under the umbrella of e-business development can be categorized as follows:

- *Customer to business (C2B):* applications that allow you to order goods over the Internet, such as for electronic book stores
- *Business to business (B2B):* automation of a supply chain across two companies
- *Business to customer (B2C):* provision of information to (otherwise passive) customers, such as by distributing newsletters
- *Customer to customer (C2C):* applications that allow customers to share and exchange information with little input from the service provider, such as for auctions

We do not recommend business modeling for every software engineering effort. It appears that business models add more value when there are more people directly involved in using the system and more information to be handled by the system. For example, if you were simply adding a feature to the software of an existing telecommunication switch, you would not consider business modeling because you would not be radically changing the purpose of the software. On the other hand, if you were building a brand-new order management system to support the sales of telecommunication network solutions (an e-business application), business modeling would be valuable to understand how this new system would affect the way you conduct your business. The sales and order processes in this domain are complex because what you are selling is a custom-made solution, not an off-the-shelf product.

Using Software Engineering Techniques for Business Modeling

One of the major advantages of using a modeling technique for business modeling that is similar to a technique for software engineering is that you are speaking the same language. It facilitates understanding how something described in the business domain might relate to something belonging in the software domain. It also simplifies describing the relationships between and among artifacts in business models and corresponding artifacts in system models.

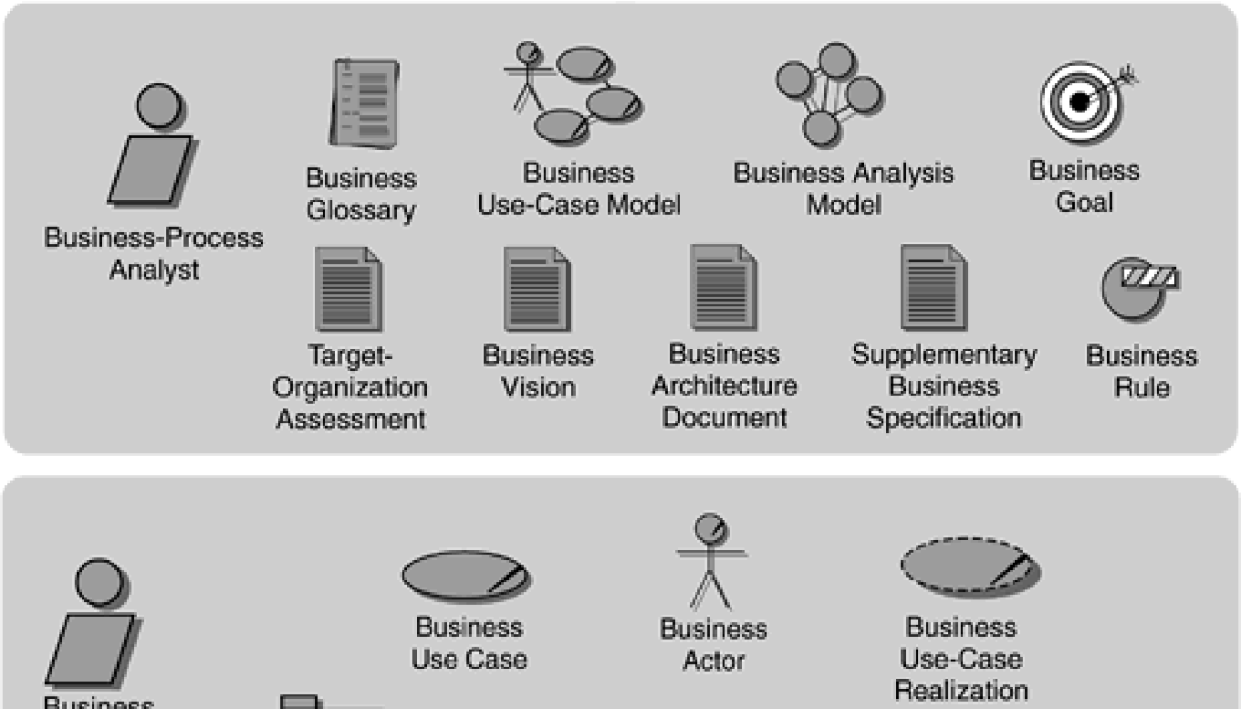
Historically, we have seen that modeling techniques developed and matured in the software domain inspire new ways of visualizing an organization. Because object-oriented visual modeling techniques are common for new software projects, using similar techniques in the business domain comes naturally.[1]

[1] This idea originated in the book by Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Re-engineering with Object Technology*. Reading, MA: Addison-Wesley, 1995.

Our notation for business modeling can be summarized as follows (see Figure 8-1):

- Business users (customers, vendors, or partners) are represented by *business actors.*
- Business processes are represented by *business use cases* and *business use-case realizations.* They refer to *business goals,* and *business rules.*
- The roles people play in an organization are represented by *business workers.*
- The "things" an organization manages or produces are represented by *business entities, business events,* organized in *business systems.*

Figure 8-1. Roles and artifacts in the business modeling discipline




Designer
Business System
Business Entity
Business Worker
Business Event

Business Modeling Scenarios

A business-engineering effort can have different scopes depending on context and need. We list six such scenarios below.

Scenario 1: Organization Chart

You may want to build a simple chart of the organization and its processes so that you get a good understanding of the requirements of the application you are building. In this case, business modeling is a part of the software-engineering project, primarily performed during the inception phase.

Scenario 2: Domain Modeling

If you are building applications with the primary purpose of managing and presenting information (such as an order management system or a banking system), you may choose to build a model of that information at a business level, without considering the workflows of the business. This is referred to as domain modeling. Domain modeling is typically part of the software-engineering project, and performed during the inception and elaboration phases of the project.

Scenario 3: One Business, Many Systems

If you are building a large system, or a family of applications, you may have one business-modeling effort that will serve as input to several software-engineering projects. The business models will help you find functional requirements, as well as serve as input to building the architecture of the application family. In this case, the business-modeling effort is often treated as a project on its own.

Scenario 4: Generic Business Model

If you are building an application that will be used by several organizations (for example, a sales support application or a billing application), it can be useful to go through a business-modeling effort to align the organizations regarding the way they do business to avoid requirements that are too complex for the system. Or, if aligning the organizations is not an option, a business-modeling effort can help you understand and manage differences in the ways the organizations will use the application and make it easier to prioritize application functionality.

Scenario 5: New Business

If an organization has decided to start a completely new line of business, and build information systems to support it, a business-modeling effort needs to be performed. In this case, the purpose of business modeling is not only to find requirements of systems, but also to determine the feasibility of the new line of business. Often the business-modeling effort in this case is treated as a project on its own.

Scenario 6: Revamp

If an organization has decided to completely revamp its way of doing business (business-process reengineering), business modeling is often one of several projects in its own right. Business-process reengineering is typically done in several stages: envision the new business, reverse-engineer the existing business, forward-engineer the new business, and install the new business.

A few years ago, the term *business-process reengineering (BPR)* was very popular and meant a "revolutionary approach to reorganization."[2]

[2] See, for example, the popular book by Michael Hammer and James Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*. New York: HarperBusiness, 1993.

Roles and Artifacts

In the Rational Unified Process, how is this translated concretely in terms of roles, artifacts, activities, and workflow? Figure 8-1 shows the roles and artifacts in business modeling.

The main roles involved in business modeling are as follows:

- The *Business-Process Analyst:* The Business-Process Analyst leads and coordinates business use-case modeling by outlining and delimiting the organization being modeled. For example, the Business-Process Analyst establishes the vision of the new business, captures business goals, and determines which business actors and business use cases exist and how they interact.
- The *Business Designer:* The Business Designer details the specification of a part of the organization by describing one or several business use cases. He or she determines the business workers and business entities needed to realize a business use case, and also how they work together to achieve the realization. The Business Designer defines the responsibilities, operations, attributes, and relationships of one or several business workers and business entities.

Also involved in this discipline are:

- *Stakeholders,* who represent various parts of the organization and provide input and review
- The *Business Reviewer,* who reviews the resulting artifacts

The key artifacts of business modeling are as follows:

- The *Business Vision Document:* a document that defines the objectives and the *Business Goals* of the business-modeling effort
- A *Business Use-Case Model:* a model of the business's intended functions used as an essential input to identify roles and deliverables in the organization
- A *Business Analysis Model:* an object model that describes the realization of business use cases

Other artifacts include the following:

- *Target-Organization Assessment:* description of the current status of the organization in which a system is to be deployed
- *Business Rules:* declarations of policy or conditions that must be satisfied
- *Supplementary Business Specifications:* a document that presents definitions of the business not included in the business use-case model or the business object model
- A *Business Glossary:* definitions of important terms used in the business
- A *Business Architecture Document:* a comprehensive overview of the architecturally significant aspects of the business from a number of perspectives. It should be used only when decisions regarding *changes* to the business need to be made or when the business needs to be described to other parties

A business use-case model consists of business actors and business use cases. The actors represent roles external to the

business (for example, customers), and the business use cases are processes. A business analysis model includes business use-case realizations, which show how the business use cases are "performed" in terms of interacting business roles and business entities.

To reflect groups or departments in an organization, business roles and business entities may be grouped into *business systems*. This parallels the organization of the use-case model and the design model we describe in Chapters 9 and 10. We are using the same modeling technique but at a higher level of abstraction. For example, instead of representing a responsibility in a system, a class at the business level represents a responsibility in an organization.

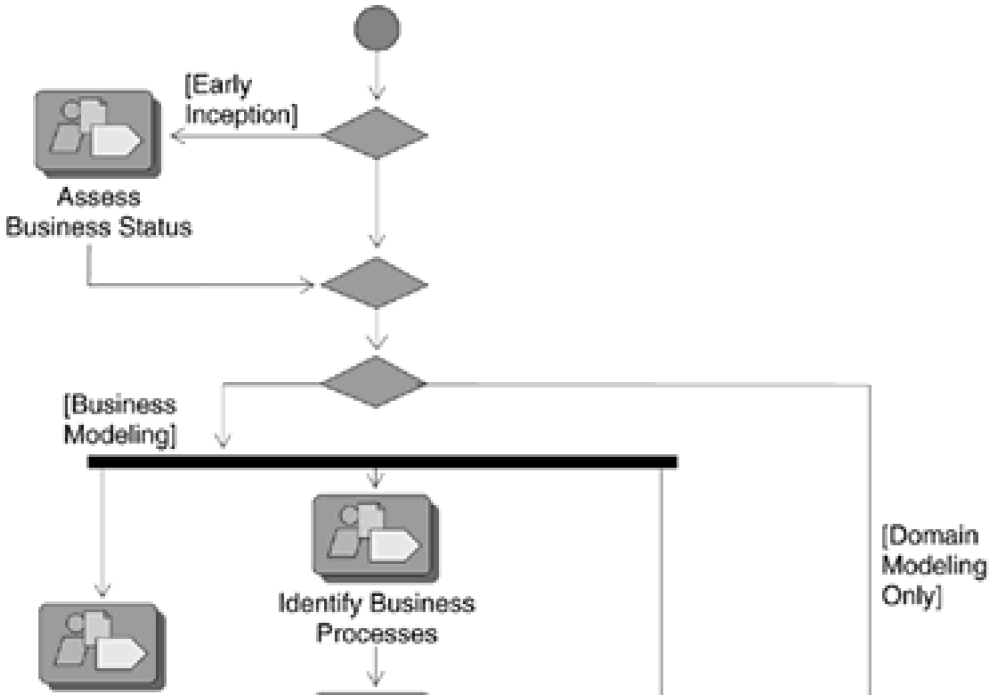
PREVIOUS NEXT

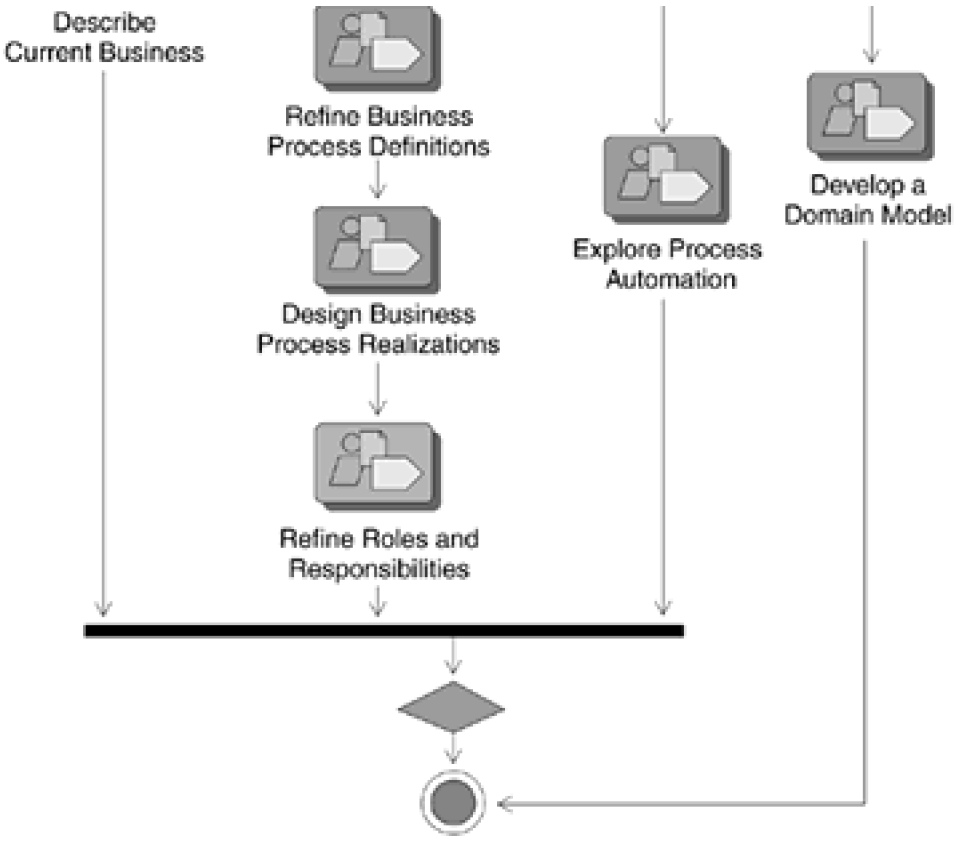
Workflow

Figure 8-2 shows a workflow diagram for business modeling. One may take several paths through this workflow, depending on the purpose of your business modeling, as well as your position in the development lifecycle.

- In the first iteration you will assess the status of the organization in which the eventual system is to be deployed. The primary artifacts produced here are the Target-Organization Assessment and the Business Vision.
- Based on the results of the assessments, you will understand which of the business-modeling scenarios (see preceding) you are following. Then you will be able to make decisions about how to continue in this iteration and also about how to work in subsequent iterations.
- If you determine that you do not need full-scale business models, only a domain model (scenario 2), you will follow the domain-modeling path of this workflow. In the Rational Unified Process, a domain model is a subset of the business analysis model, encompassing the business entities of that model.
- If you determine that no major changes need to occur to the business processes, you need to chart those existing processes and derive system requirements (scenario 1). There is no need to keep a special set of models of the current organization; you can focus directly on describing the target organization. You would follow the business-modeling track, but skip Describe Current Business.
- If you do business modeling with the intention of improving or reengineering a business (scenarios 3, 4, and 6), you would model both the current business and the new business.
- If you do business modeling with the intention of developing a new business more or less from scratch (scenario 5), you would envision the new business and build models of the new business, but skip Describe Current Business.

Figure 8-2. A workflow in business modeling

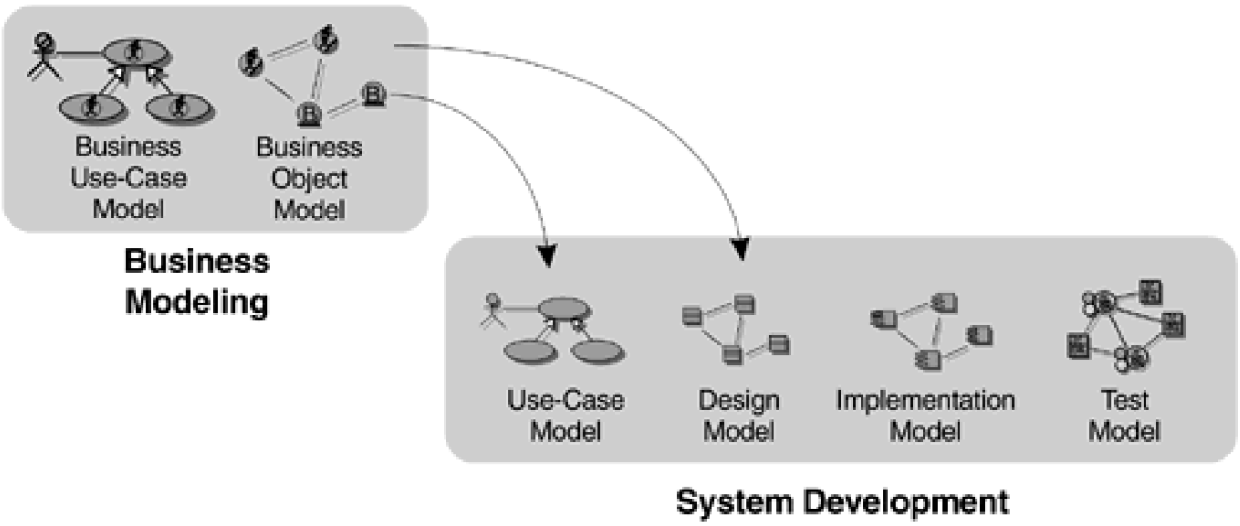



Describe
Current Business
Refine Business
Process Definitions
Design Business
Process Realizations
Refine Roles and
Responsibilities
Explore Process
Automation
Develop a
Domain Model

From the Business Models to the Systems

One advantage of our approach to business modeling is its clear and concise way of showing dependencies between business and system models. Figure 8-3 shows this relationship.

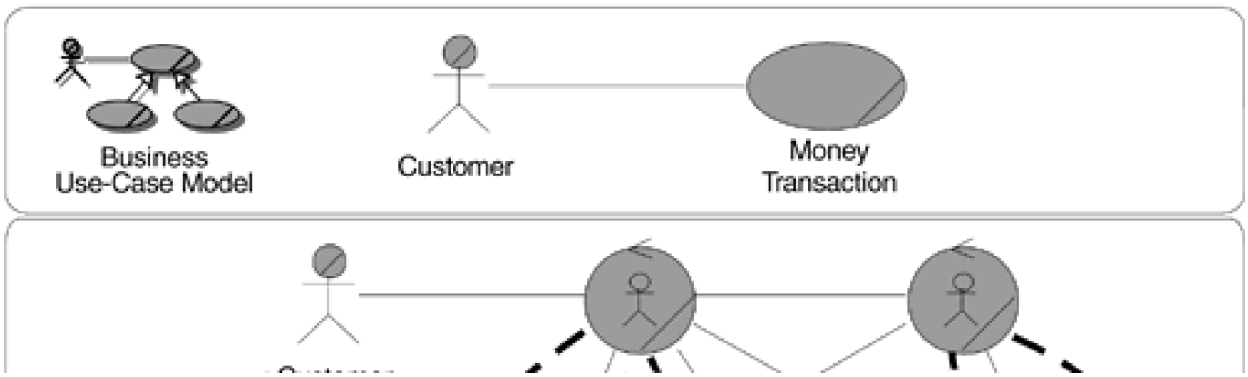
Figure 8-3. From the business models to the system models

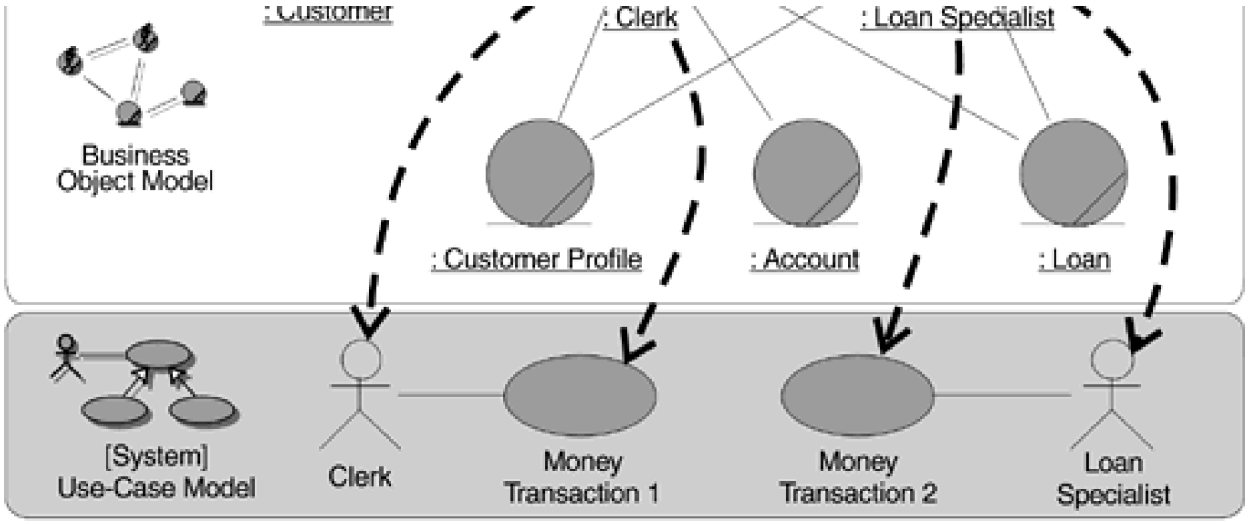


Business Models and Actors of the System

To identify information-system use cases, begin with the business workers in the business object model. For each business worker, identify a candidate system actor. For each business use case the business actor participates in, create a candidate system use case. For example, see Figure 8-4.

Figure 8-4. You can derive candidate system actors and system use cases based on business models of a bank.



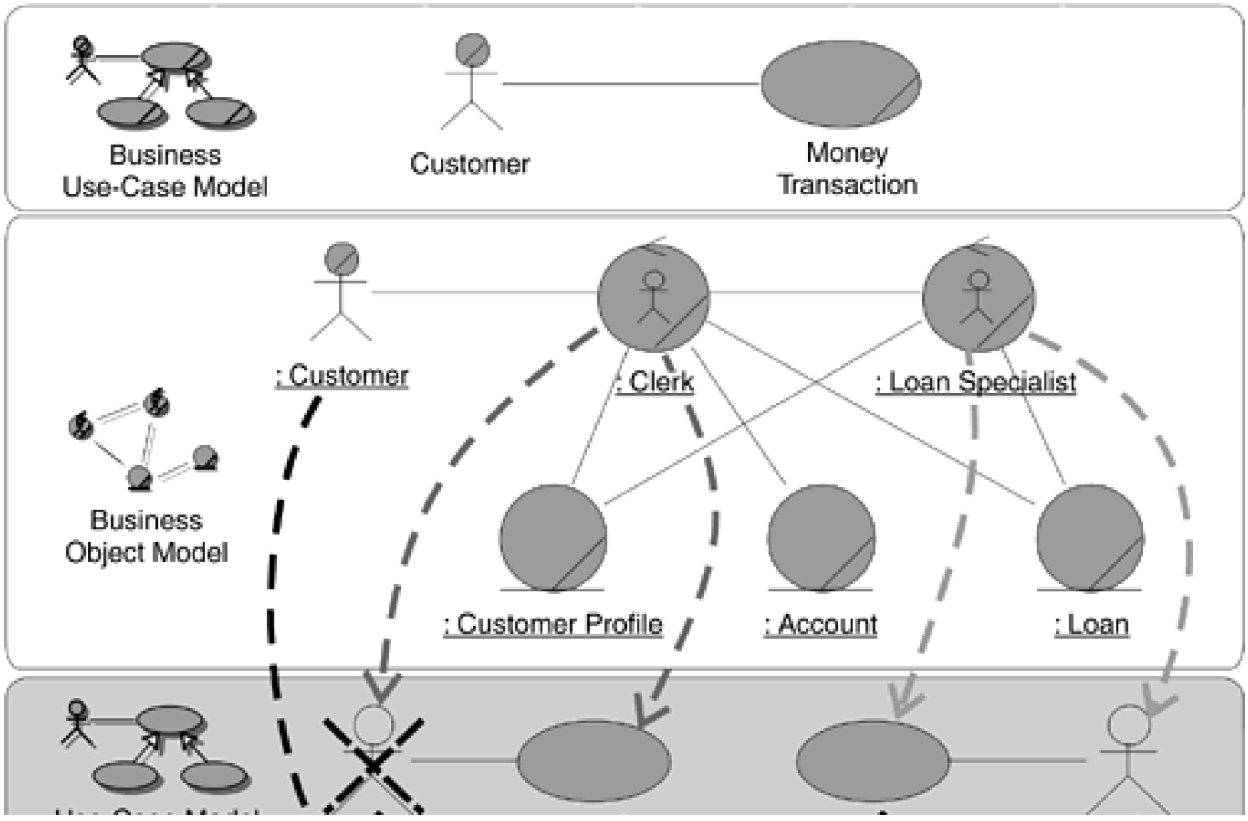


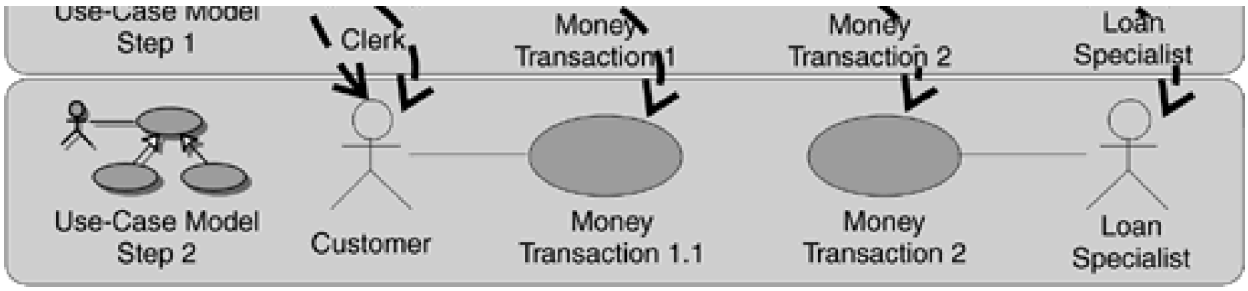
Automated Business Workers

If you are aiming to build a system that automates a set of business processes completely, which is the case if you are, for example, building an e-commerce application, it is no longer the business worker who will become the system actor. It is instead the business actor who will communicate directly with the system and act as system actor.

You are in effect changing the way business is performed when building an application of this kind. Responsibilities of the business worker will be moved to the business actor. For example, see Figure 8-5.

Figure 8-5. Completely automating business workers will change the way the business process is realized and also how you find system actors and use cases.

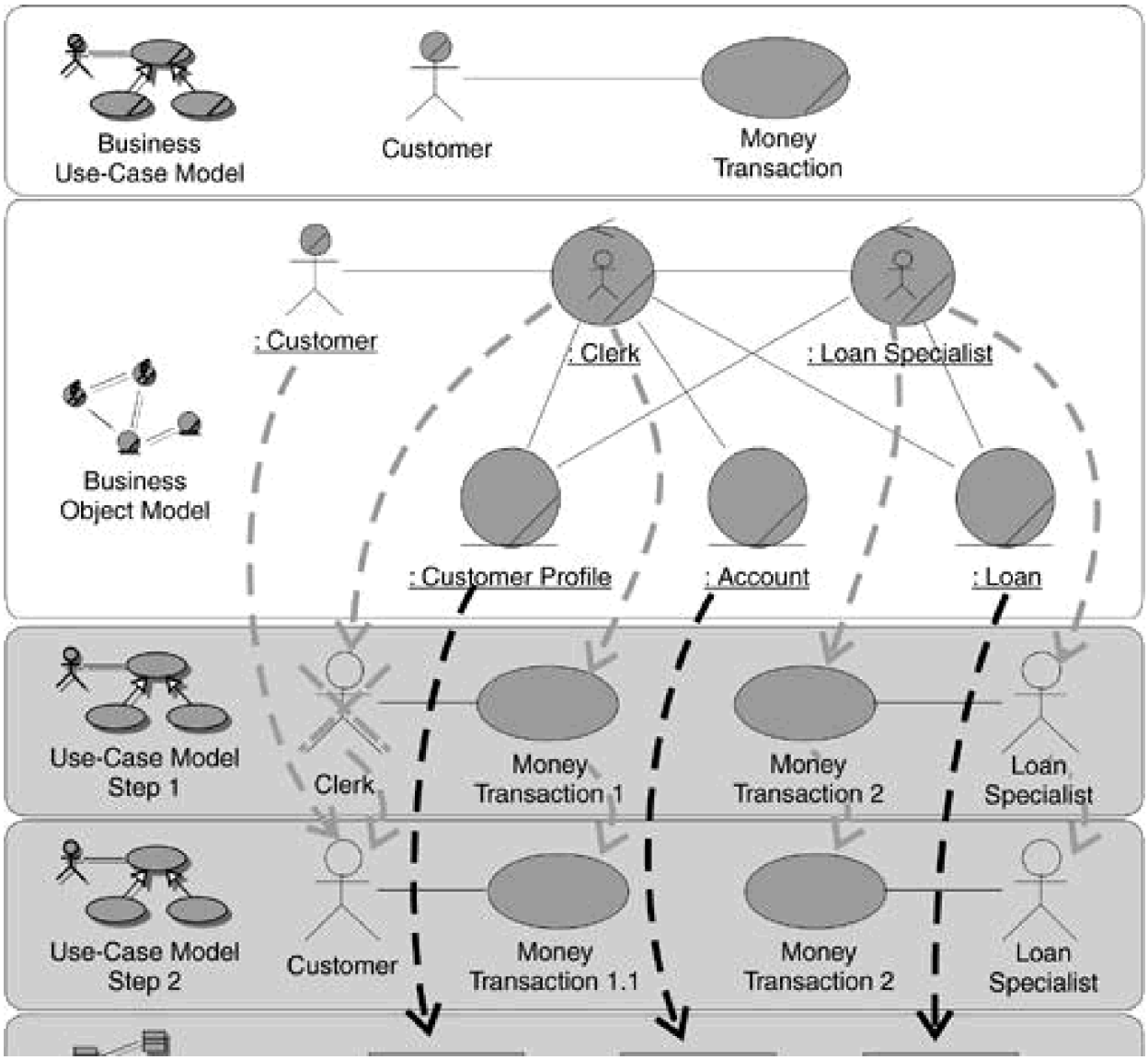




Business Models and Entity Classes in the Analysis Model

A business entity that is to be managed by an information system will correspond to an entity in the analysis model of the information system. Some attributes of the business entity might also correspond to entities in the information-system model. A business entity can be accessed by several business workers. Consequently, the corresponding entities in the system can participate in several information-system use cases. For example, see Figure 8-6.

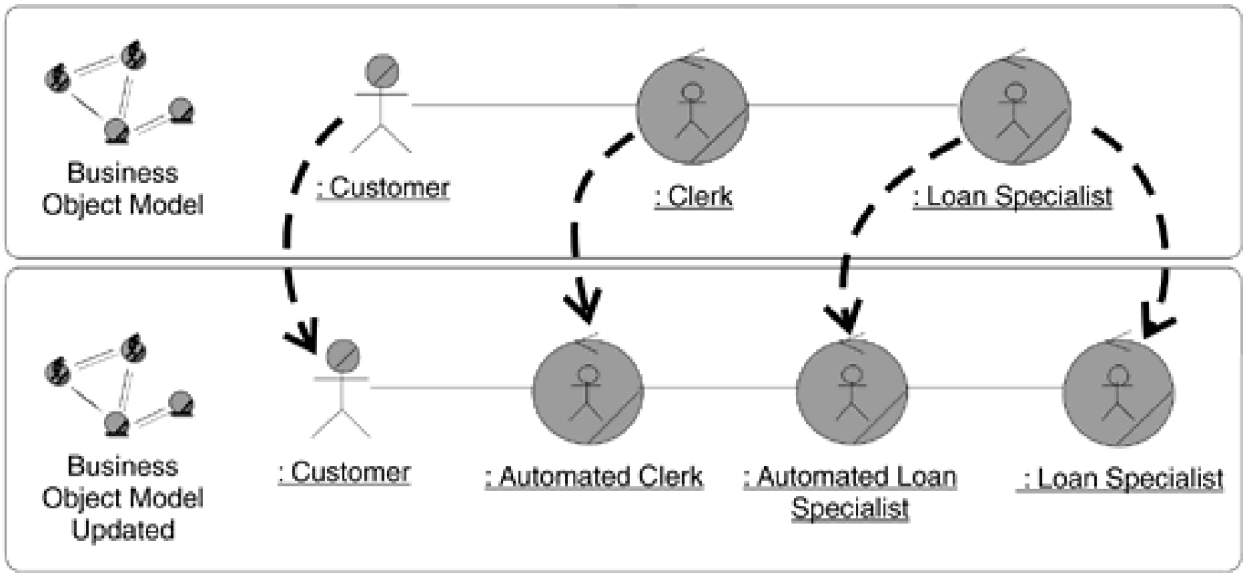
Figure 8-6. The business entities customer profile, account, and loan are all for automation.



Using the Business Analysis Model for Resource Planning

If you intend to use the business object model for resource planning or as a basis for simulation, you will need to update to reflect the type of resources used to implement the business workers and artifacts. You need to modify so that each business worker and business entity is implemented by only one type of resource (see Figure 8-7). If you are aiming at reengineering the business process, you should not consider resources in the first iteration of your business object model because that tends to make you focus on the existing solutions rather than identify problems that can be solved with new kinds of solutions. For instance, in the banking example, we decided to update the business object model to use it for resource planning.

Figure 8-7. The business workers are modified to reflect the automation.



Other Sources for Requirements on Systems

There are many sources of knowledge about—and requirements for—the information system outside the business model. Examples of sources include:

- Users of the information systems that you have not modeled in the business model— for example, the system administrator, who is a user of the information system that is (usually) not represented in the business model
- Strategies that the business as a whole has decided on— for example, regarding IT, reuse, compatibility, and quality
- Any legacy considerations
- Timing and coordination with other projects

- Trends in the business's own industry and in the IT industry

Business Models and System Architecture

From an architectural perspective, having business models in place is particularly useful if your intent is to build one of the following kinds of system:

- Customized systems for one or more companies in a particular type of industry (e.g., banks, insurance companies)
- A family of applications for the open market (e.g., order-handling systems, billing systems, air traffic control systems)

The business models would give input to the use-case view and the logical view as presented in the analysis model. You would also be able to find key mechanisms at the analysis level (analysis mechanisms).

PREVIOUS NEXT

Modeling the Software Development Business

If you were now to revisit Chapter 3, Static Structure: Process Description, you would notice its strong relationship to the concepts introduced in this chapter. That is no accident. The modeling of the Rational Unified Process is based on the same techniques: It is a model of the business of developing software.[3]

[3] See Ivar Jacobson and Sten Jacobson, "Reengineering Your Software Engineering Process," *Object Magazine*, March–April 1995, as well as Chapter 9 in *The Object Advantage*, op. cit.

Tool Support

Because we are using the same UML concepts (except for slightly different stereotypes) to model a business that we use to model a system, we can use the same tools. Rational Rose and Rational XDE provide all we need to produce the visual models described earlier.

Just as in software engineering, you can use the Rational RequisitePro tool to capture textual aspects of the business models and to maintain dependencies between and among model elements in different models.

To generate and maintain documentation of the models, you can use the Rational SoDA tool.

Summary

- You might need to model a business to understand its structure and dynamics, to ensure that all stakeholders have a common understanding of the organization, and to derive system requirements to support the organization.
- Software engineering techniques can be translated and used for business modeling.
- Depending on the character of the organization, as well as the context of the project, you may find yourself employing rather different scenarios of business modeling.
- Software requirements derive, in large part, from models of the business.
- Rational Software's design tools support business modeling.

Chapter 9. The Requirements Discipline

This chapter describes important concepts that are necessary to capture and manage the requirements of your system effectively and to design a user interface focused on the needs and goals of the users, as well as other important stakeholders. A brief overview of the requirements discipline in the Rational Unified Process is included.

Purpose

The goals of the requirements discipline are as follows:

- To establish and maintain agreement with the customers and other stakeholders on what the system should do—and why!
- To provide system developers with a better understanding of the system requirements
- To define the boundaries of (delimit) the system
- To provide a basis for planning the technical contents of iterations
- To provide a basis for estimating cost and time to develop the system
- To define a user interface for the system, focusing on the needs and goals of the users

To achieve these goals, the requirements discipline describes how to define a vision of the system and translate the vision into a use-case model that, with supplementary specifications, defines the detailed software requirements of the system. In addition, the requirements discipline describes how to use requirement attributes to help you manage the scope and changing requirements of your system.

What Is a Requirement?

We define a requirement as a condition or capability to which a system must conform.

In his efforts to establish quality criteria as a basis for measurement for evaluating software systems, Robert Grady, during his tenure at Hewlett-Packard, categorized the necessary quality attributes of a software system as functionality, usability, reliability, performance, and supportability, referred to as "FURPS."[1] We have found this acronym to be a useful reminder of the types of requirements we need to consider to fully define a quality system.

[1] Robert Grady, *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice-Hall, 1992, p. 32.

Functional Requirements

When we think about the requirements of a system, we naturally tend to think first about those things that the system does on behalf of the user. After all, we developers like to think that we have a "bias for action," and we expect no less of the systems we build. We express these actions as *functional,* or behavioral, requirements, which specify the actions that a system must be able to perform.

Functional requirements are used to express the behavior of a system by specifying both the input and output conditions that are expected to result.

Nonfunctional Requirements

In order to deliver the desired quality to the end user, a system must also exhibit a wide variety of attributes that are not described specifically by the system's functional requirements. For the system to exhibit these quality attributes, an additional set of requirements must be imposed. We refer to this set of requirements as *nonfunctional* requirements. Ultimately, they are every bit as important to the end-user community as are the functional requirements.

Using the FURPS categories, the following list summarizes considerations for defining the nonfunctional quality attributes of a system:

- *Usability*

  Usability requirements address human factors—aesthetics, ease of learning, and ease of use—and consistency in the user interface, user documentation, and training materials.

- *Reliability*

  Reliability requirements address frequency and severity of failure, recoverability, predictability, and accuracy.

- *Performance*

Performance requirements impose conditions on functional requirements—for example, a requirement that specifies the transaction rate, speed, availability, accuracy, response time, recovery time, or memory usage with which a given action must be performed.

- *Supportability*

  Supportability requirements address testability, maintainability, and the other qualities required to keep the system up-to-date after its release. Supportability requirements are unique in that they are not necessarily imposed on the system itself but instead often refer to the process used to create the system or various artifacts of the system development process. An example is the use of a specific C++ coding standard.

Frankly, it's not particularly important to divide requirements into these categories, and there is little value in a debate about whether a specific requirement is a usability or a supportability requirement. The value in these definitions is that they provide a template, or checklist, for requirements elicitation and for assessing the completeness of your understanding. In other words, if you ask about and come to understand requirements of all these categories, you will have a high degree of certainty that you have truly understood the critical requirements before you begin substantial investment in development. Often you will find, for example, that certain reliability requirements are implied by given functional requirements, and it is equally important to explore these reliability requirements. A system that fails to meet an implied reliability or performance requirement fails just as badly as a system that fails to meet an explicit functional need.

Types of Requirements

Traditionally, requirements are seen as detailed textual specifications that fit into one of the categories just mentioned, expressed in the form "The system shall . . ." To manage the full requirements process effectively, however, it is helpful to gain a comprehensive understanding of the actual user and other stakeholder *needs* that are to be fulfilled by the system being developed. This understanding arms the development team with the *why* ("Why does this system need to operate at 99.3% accuracy?") as well as the *what.* Knowing this, the team will be able to do a better job of interpreting the requirements ("Does it also need to operate at 99.3% accuracy while routine maintenance is happening?"), as well as being able to make trade-offs and design decisions that optimize the development process ("If we can achieve 92% accuracy with half the effort, is that a reasonable trade-off?").

Stakeholders: Requests versus Needs

We define a *stakeholder* as any person or representative of an organization who has a stake—a vested interest—in the outcome of a project. The most obvious stakeholder, of course, is the end user, but we must remember to consider other important stakeholders: a purchaser, a contractor, a developer, a project manager, or anyone else who cares enough or whose needs must be met by the project. For example, a nonuser stakeholder might be a system administrator whose workload will depend on certain default conditions imposed by the system software on the user. Other nonuser stakeholders might represent the economic beneficiary of the system.

Using this definition, it seems obvious that the development team must understand the specific *needs* of the key stakeholders of the system. But how do we identify those needs? It is important that we actively gather all types of *stakeholder requests* (the raw input data used to figure out the needs) throughout the development lifecycle. Early in the project, we may use interviews, questionnaires, and workshops; further on in a project we will collect change requests, defect reports, and enhancement requests. Often, these requests will be vague and ambiguous—and often stated as a need (e.g., "I need easier ways to share my knowledge of project status" or "I need to increase my productivity" or "We need to increase the performance of the system"). The *requests* from the stakeholders set a very important context for understanding the real needs of the stakeholders and provide critical input to our product requirements. This input provides a crucial piece of the puzzle that allows us to determine both the *whys* and the *whats* of system behavior.

System Features

Interestingly, when you initially interview stakeholders about their needs and requirements, they typically describe neither of the entities just defined. Typically, they tell you neither their real need (e.g., "If Joe and I don't start communicating better, our jobs are at risk" or "I want to be able to slow this vehicle down as quickly as possible without skidding") nor the actual requirement for the system (e.g., "I must be able to attach an RTF file to an e-mail message" or "The vehicle shall have a computer control system dedicated to each wheel that . . ."). Instead, they often describe an abstraction of both (e.g., "I want automatic e-mail notification" or "I want a vehicle with ABS").

We call this type of abstraction—these higher-level expressions of system behavior—the *features* of the system. Technically, we can think of a feature as "a service to be provided by the system that directly fulfills a user need." Often these features are not well defined, and they may even be in conflict (e.g., "I want ABS" and "I want lower maintenance requirements"), but nonetheless they are a representation of real needs. What is happening in this discussion is that the stakeholder has already translated the real

need ("better communication with Joe") into a behavior of the system ("automatic e-mail notification"). In so doing, there has been a subtle shift in the stakeholder's thinking from the *what* ("better communication") to the *how*—that is, how it will be implemented ("automated e-mail notification").

Because it is easy to discuss these features in natural language and to document and communicate them, they add important context to the requirements discipline. In the Rational Unified Process, we simply treat these features as another *type of requirement.* In addition, by adding to these features various attributes—such as risk, level of effort, and customer priority—you add a richness to your understanding of the system. You can then use this enhanced understanding to manage scope before substantial investments have been made in low-priority features.

Software Requirements

Neither an understanding of stakeholder needs nor an understanding of system features is, by itself, adequate to communicate to developers exactly what the system should do ("Hey, Bill, go code an automatic e-mail notification system"). You need an additional level of specificity that translates these needs and features into specifications that you can design, implement, and test to determine system conformance. At this level of specificity, you must deal with both the functional and nonfunctional requirements of the system.

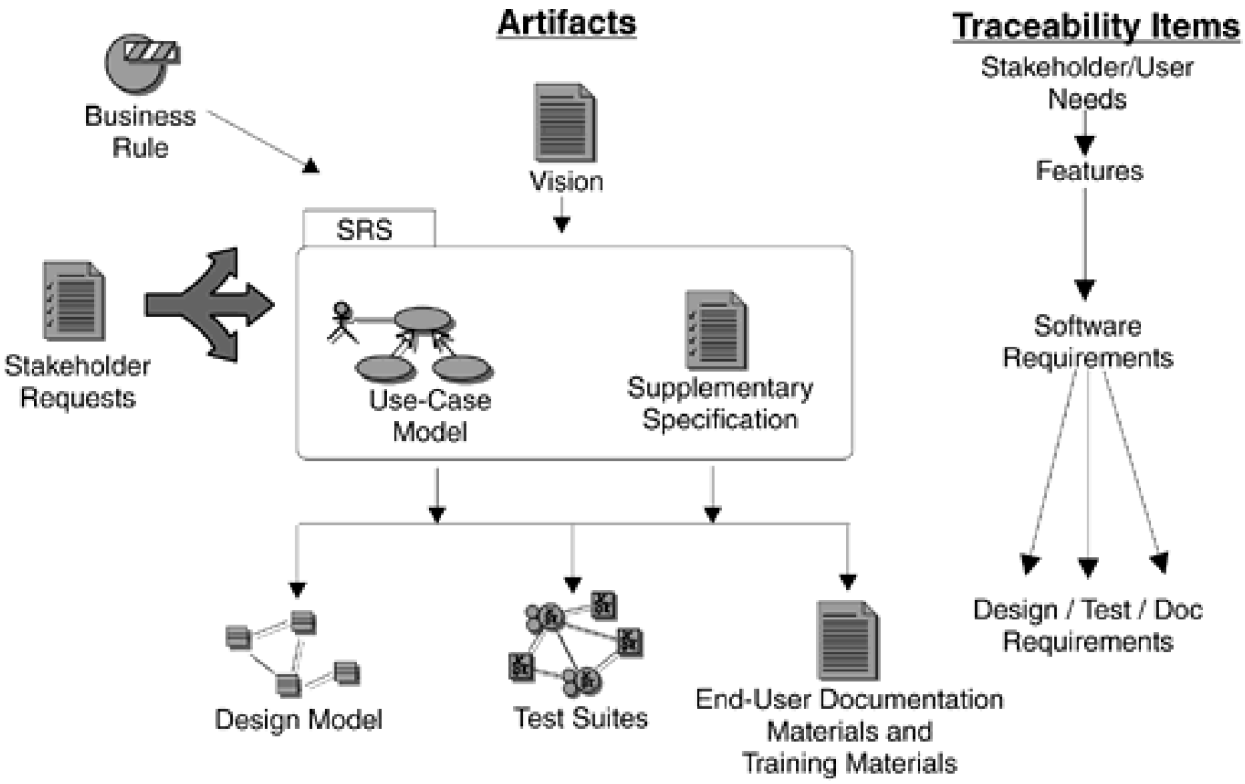
Specifying Software Requirements with Use Cases

In Chapter 6, A Use-Case-Driven Process, we learned about use-case modeling, a powerful technique that we can use to express the detailed behavior of the system via simple threads of user-system interactions that users and developers can understand easily. A use-case model is a model of the system and its intended behavior. It consists of a set of actors to represent external entities that interact with the system, along with use cases that define how the system is used by the actors. The use-case model is an effective way of expressing these more detailed software requirements.

Capturing and Managing Requirements

For each project, you can now see a common structure among these expressions of requirements, as depicted in Figure 9-1. First, you collect the *stakeholder requests,* which encompass all the requests and wish lists you get from end users, customers, marketing, and other project stakeholders.

Figure 9-1. Requirement types and relationships with artifacts



The set of stakeholder requests is used to develop a *vision document* that contains the set of key *stakeholder and user needs* and the high-level *features* of the system. These system features express services that must be delivered by the system in order to fulfill the stakeholder needs. We call this relationship *requirements traceability.* The features you decide to include in the vision document are based on an analysis of the cost of implementing a desired feature and a determination of the return on that investment. This analysis is found in the business case for the project.

Before coding begins, you must translate these features into detailed software requirements at a level at which you can design and build an actual system and identify test cases to test the system behavior. These detailed requirements are captured in the *use-case model* and other *supplementary specifications,* which capture those requirements that do not fit well in the use cases.

As you are detailing the requirements, you must keep in mind the original stakeholder needs and the system features to ensure that you are interpreting the vision correctly. This constant consideration of features and software requirements implies a traceability relationship wherein software requirements are derived from one or more features of the system, which in turn fulfill one or more stakeholder needs.

The detailed requirements are then realized in a design model and end-user documentation and are verified by a test model.

Gathering stakeholder requests and determining the needs and features specified in the vision document are started early in the

project. Throughout the project lifecycle, you can use these features, based on customer priority, risk, and architectural stability, to divide the set of requirements into "slices" and detail them in an incremental fashion. As you are detailing these requirements, you will undoubtedly find flaws and inconsistencies that make it necessary to go back to the stakeholders for clarifications and trade-offs and to update the stakeholder needs and vision. So, the requirements workflow repeats in an iterative manner until all requirements are defined, feedback is considered, and the inevitable changes are managed.

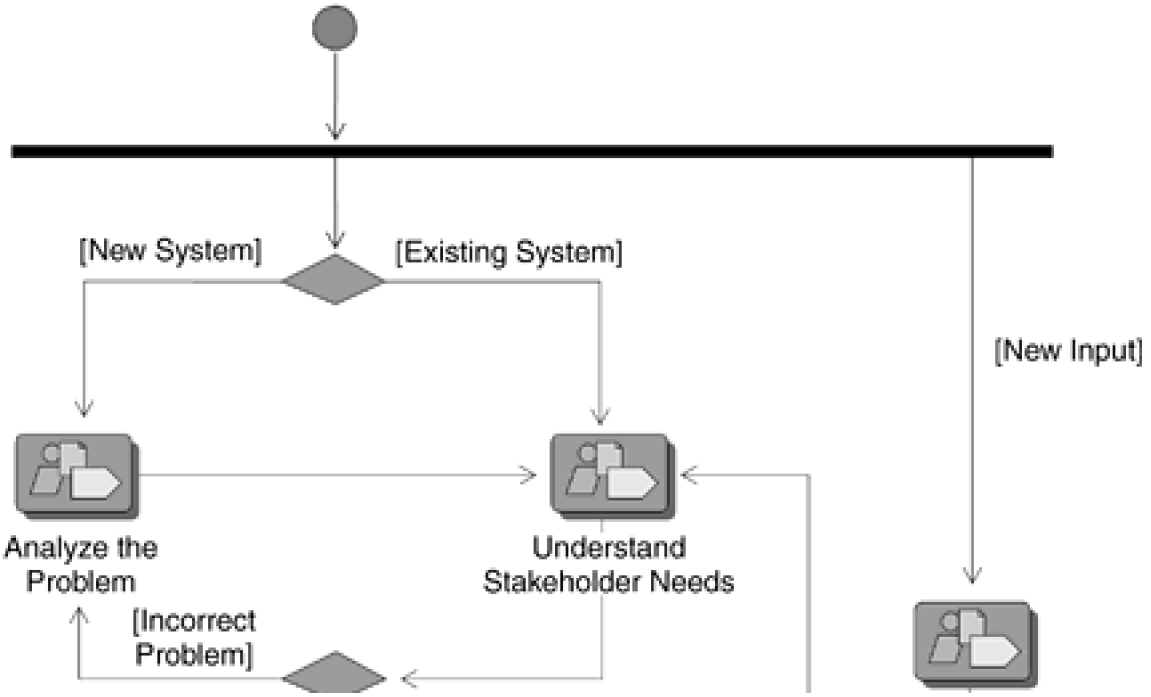
PREVIOUS NEXT

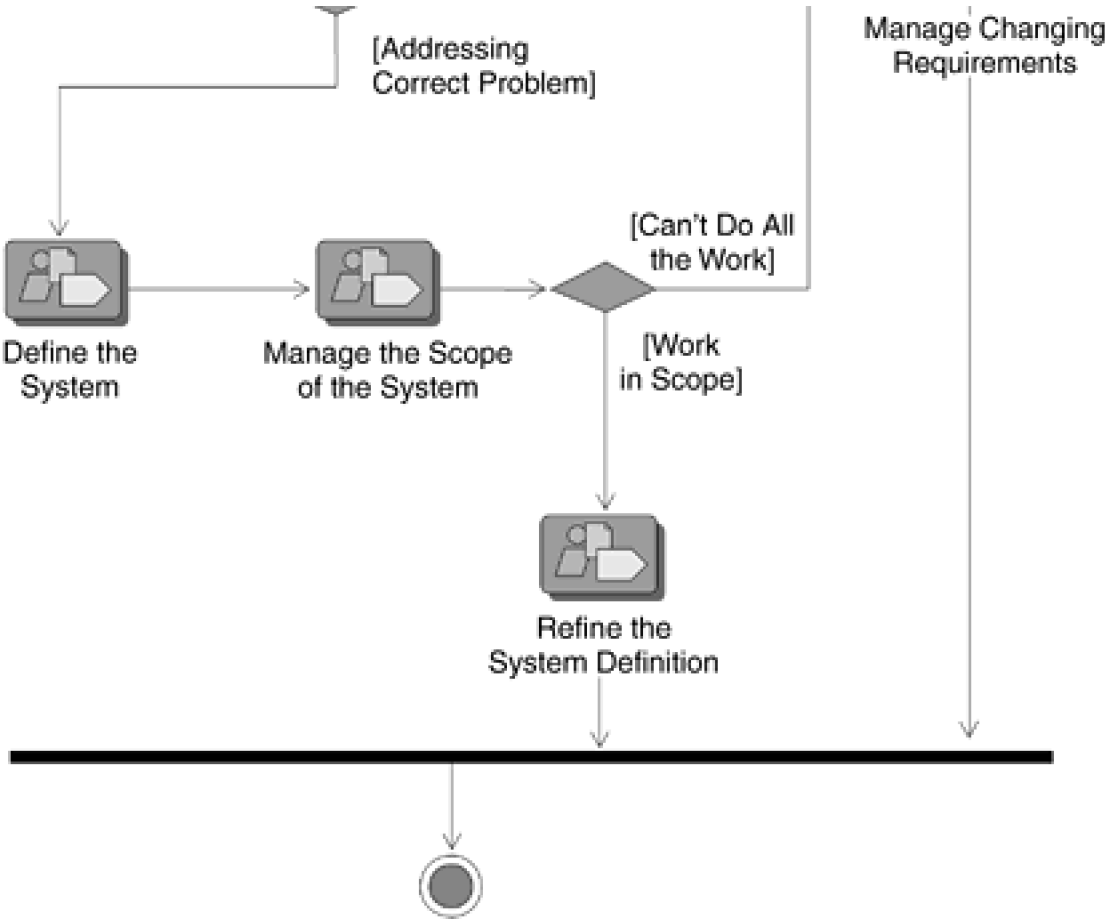
Requirements Workflow

Figure 9-2 summarizes the activities in the Rational Unified Process that constitute the requirements workflow, as shown in the workflow diagram:

- *Analyze the Problem*. Gain agreement on a statement of the problem we are trying to solve; identify stakeholders; identify the boundaries and constraints of the system.
- *Understand Stakeholder Needs*. Using various elicitation techniques, gather stakeholder requests and obtain a clear understanding of the real needs of the users and stakeholders of the system.
- *Define the System.* Based on input from the stakeholders, establish the set of system features to be considered for delivery; determine the criteria that will be used to prioritize them and begin to set realistic expectations with the stakeholders on what features will be delivered; identify actors and use cases needed for each key feature.
- *Manage the Scope of the System.* Collect important information from stakeholders in the project and maintain those—along with the requirements—as requirement attributes to be used in prioritizing and scoping the agreed-upon set of requirements so the system can be delivered on time and on budget to meet the customers' expectations.
- *Refine the System Definition*. Using a use-case model, detail the software requirements of the system to come to an agreement with the customer on the functionality of the system, and capture other important requirements, such as nonfunctional requirements, design constraints, and so forth.
- *Manage Changing Requirements.* Use requirement attributes and traceability to assess the impact of changing requirements; use a central control authority, such as a Change Control Board (CCB), to control changes to the requirements; maintain agreement with the customer and set realistic expectations on what will be delivered.

Figure 9-2. Workflow in requirements




[Addressing
Correct Problem]
Manage Changing
Requirements
[Can't Do All
the Work]
Define the
System
Manage the Scope
of the System
[Work
in Scope]
Refine the
System Definition

Roles in Requirements

How is all this translated concretely in terms of roles, artifacts, and activities? Figure 9-3 shows the main roles and artifacts in the requirements discipline:

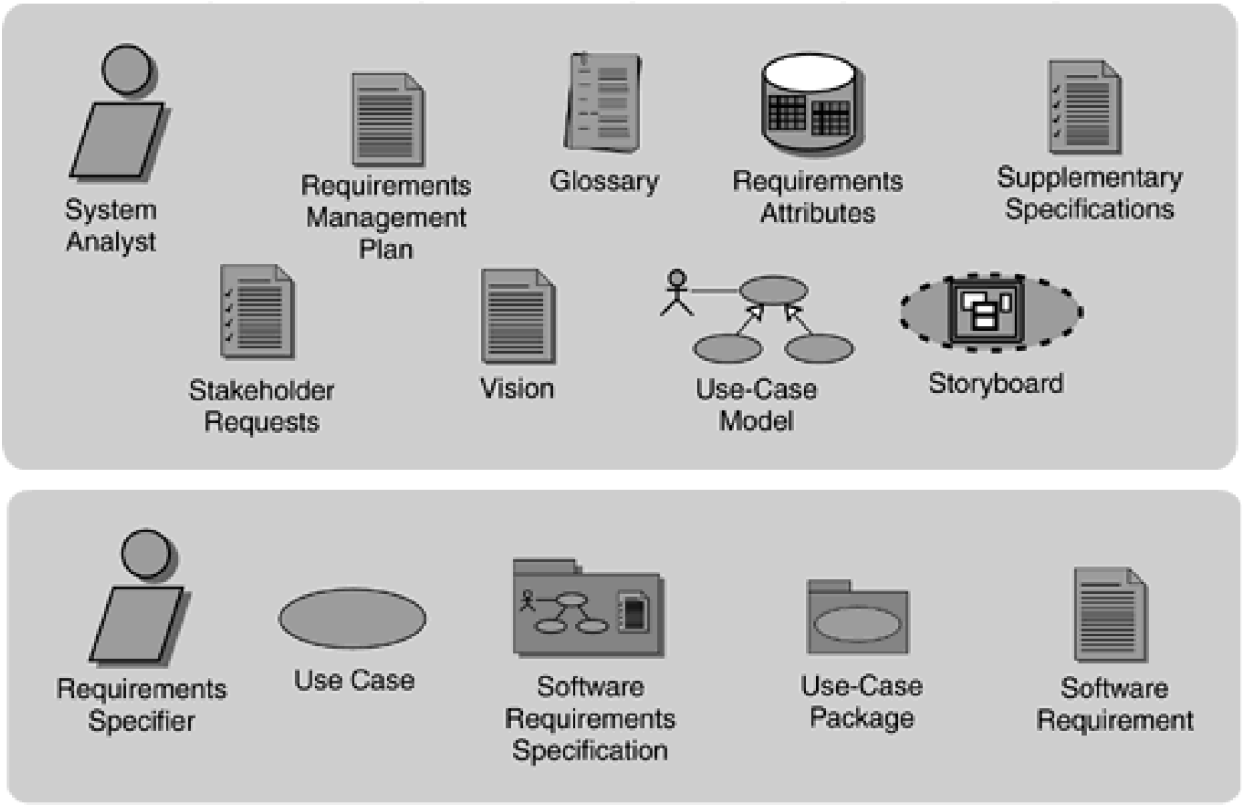
- *System Analyst*

  The System Analyst leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality and delimiting the system.

- *Requirements Specifier*

  The Requirements Specifier details all or part of the system's functionality by describing the requirements aspect of one or several use cases.

Figure 9-3. Roles and artifacts in the requirements discipline



Working with stakeholders of the project, the System Analyst analyzes the problem and works with stakeholders to understand what they need and to define what the system must do and perhaps what it should not do—also identifying nonfunctional requirements. The System Analyst can then develop a vision for the project. This vision, expressed as a set of features written from the stakeholder's perspective, is a basis for developing an outline of a Use-Case Model.

The Requirements Specifier is assigned a set of use cases and supplementary requirements, which he or she will detail and make consistent with other requirements discipline artifacts. The Requirements Specifier does not work in isolation but rather should communicate regularly with other individual Requirements Specifiers as well as with the System Analyst. Requirement Specifiers

and System Analysts work closely with the User-Interface Designers (see Chapter 10).

Another role involved in the requirements discipline is the *Software Architect,* who is involved primarily in earlier iterations and works with the System Analyst and Requirements Specifier to ensure the integrity of the architecturally significant use cases.

The *Requirements Reviewer* is another important role, representing all the different kinds of people you bring in to verify that the requirements are perceived and interpreted correctly by the development team. Because reviews have different purposes, they are performed several times during the execution of the requirements workflow by requirements reviewers with different profiles.

Together, the individuals acting in these roles form a team that exercises the requirements workflow iteratively throughout the project lifecycle.

Artifacts Used in Requirements

When considering requirements, it is important, first of all, to understand the definition and scope of the problem that we are trying to solve with this system. The business object model, developed during business modeling, will serve as valuable input to this effort. Stakeholders are identified, and Stakeholder Requests are elicited, gathered, and analyzed.

Stakeholder Requests are actively elicited and gathered to get a "wish list" of what different stakeholders of the project (customers, users, product champions) expect or desire the system to include, together with information on how each request has been considered by the project.

The key needs and features are specified in the Vision Document, and a Use-Case Model, Use Cases, and Supplementary Specifications are developed to describe what the system will do—an effort that views all stakeholders, including customers and potential users, as important sources of information (in addition to system requirements).

The Vision Document provides a complete vision of the software system under development and supports the contract between the funding authority and the development organization. Every project needs a source for capturing the expectations among stakeholders. The Vision Document is written from the customers' perspective, focusing on the essential features of the system and acceptable levels of quality. The vision should include a description of what will be included, as well as features that were considered but not included. It should also specify operational capacities (volumes, response times, accuracies), user profiles, and interfaces with entities outside the system boundary, where applicable. The Vision Document provides the contractual basis for the requirements visible to the stakeholders.

The Use-Case Model should serve as a communication medium and can serve as a contract among the customer, the users, and the system developers on the functionality of the system, which allows the following:

- Customers and users can validate that the system will become what they expected.
- System developers can build what is expected.

The Use-Case Model consists of Use Cases and Actors. Each Use-Case in the model is described in detail, showing step by step how the system interacts with the actors and what the system does in the use case. Use cases function as a unifying thread throughout the software lifecycle; the same Use-Case Model is used in system analysis, design, implementation, and testing.

The Supplementary Specifications are an important complement to the Use-Case Model, because together they capture all software requirements (functional and nonfunctional) that need to be described to serve as a complete Software Requirements Specification.

A complete definition of the software requirements, as described in the use cases and supplementary specifications, may be packaged together to define a Software Requirements Specification (SRS) for the full product, a particular "feature," or other subsystem grouping.

Complementary to the foregoing artifacts, the following artifacts are also developed:

- A Glossary
- Storyboards, which will serve as the basis of User-Interface Prototypes

The Glossary is important because it defines a common terminology that is used consistently across the project or organization. The glossary started in business modeling (see Chapter 8) can serve as a starting point, or simply be extended.

Current and potential users of the system should be involved in modeling and defining the user interface of the system in the Storyboards associated with Use Cases, which will serve as the basis of User-Interface Prototypes, which are developed in

parallel with other requirements activities (see Chapter 10).

Tool Support

To manage all aspects of your project's requirements effectively, maintaining their attribute values and traceability to other project items, it is essential to have the support of requirements management tools, in all but the very small projects. Rational RequisitePro provides support in capturing requirements and organizing them in documents and in a requirements repository along with important attributes that are used to manage requirements scope and change. Moreover, if you are using use cases, RequisitePro will help you describe the textual properties of the use cases.

For visual modeling of requirements artifacts, Rational Rose and Rational XDE provide automated support for the actors and use cases in the Use-Case Model (with automatic integration to RequisitePro for maintaining their textual properties, requirements attributes, and traceability), along with storyboards. Having requirements artifacts in Rose or XDE also allows you to maintain dependencies to elements in the design model.

Rational SoDA helps automate the generation of documentation. It allows you to define an "intelligent template" that can extract information from various sources. Rational SoDA is particularly useful if you are using several tools to capture the results of your workflow, but you must produce final documentation that pulls together that information in one place.

Summary

- Requirements management involves a team effort to establish and maintain agreement between the stakeholders and the development team on what the system should do.
- In a typical project, several types of requirements should be considered, including high-level features and more detailed functional and nonfunctional requirements and use cases.
- Maintaining requirement attributes and traceability are keys to managing the scope of the project effectively and handling changing requirements throughout the project lifecycle.
- User-interface design focuses on the key needs and goals of the users in the visual shaping of the user interface in order to build a user-centered system and meet usability requirements.
- Rational tools support the capture, visual modeling, and management of requirements, their attributes, and traceability.

Chapter 10. The Analysis and Design Discipline

This chapter describes the key aspects of the analysis and design discipline. It introduces the main artifacts used in design—classes, subsystems, and collaborations—and explains the role of analysis and design in the overall development effort.

Purpose

The purpose of the analysis and design discipline is to translate the requirements into a specification that describes how to implement the system. To make this translation, you must understand the requirements and transform them into a system design by selecting the best implementation strategy. Early in the project, you must establish a robust architecture so that you can design a system that is easy to understand, build, and evolve. Then you must adjust the design to match the implementation environment, designing it for performance, robustness, scalability, and testability, among other qualities.

Analysis versus Design

The purpose of analysis is to transform the requirements of the system into a form that maps well to the software designer's area of concern—that is, to a set of classes and subsystems. This transformation is driven by the use cases and is shaped further by the system's nonfunctional requirements. Analysis focuses on ensuring that the system's functional requirements are handled. For simplicity's sake, it ignores many of the nonfunctional requirements of the system and also the constraints of the implementation environment. As a result, analysis expresses a nearly ideal picture of the system.

The purpose of design, on the other hand, is to adapt the results of analysis to the constraints imposed by nonfunctional requirements, the implementation environment, performance requirements, and so forth. Design is a refinement of analysis. It focuses on optimizing the system's design while ensuring complete requirements coverage.

How Far Must Design Go?

Design must define only enough of the system so that it can be implemented unambiguously. What constitutes "enough" varies from project to project and company to company. In some cases, the design may be elaborated to the point that the system can be implemented through a direct and systematic transformation of the design into code.

In other cases, the design resembles a sketch, elaborated only far enough to ensure that the implementer can produce a set of components that satisfy the requirements. The degree of specification therefore varies with the expertise of the implementer, the complexity of the design, and the risk that the design might be misconstrued.

When the design is specified precisely, the code can be tied closely to the design and can be kept synchronized with the design in what we call *round-trip engineering,* thereby avoiding one transformation step and a potential source of error.

When the degree of completeness and precision in the design is very high—so high that you can execute the transformation directly by interpreting the design or by rapidly generating from it small amounts of code—the transformation is almost invisible to the designer, and the design appears to be "executable."

Roles and Artifacts

The Rational Unified Process expresses the analysis and design process in terms of roles, artifacts, activities, and workflow. Figure 10-1 shows the roles and artifacts involved. The main roles involved in analysis and design are the following:

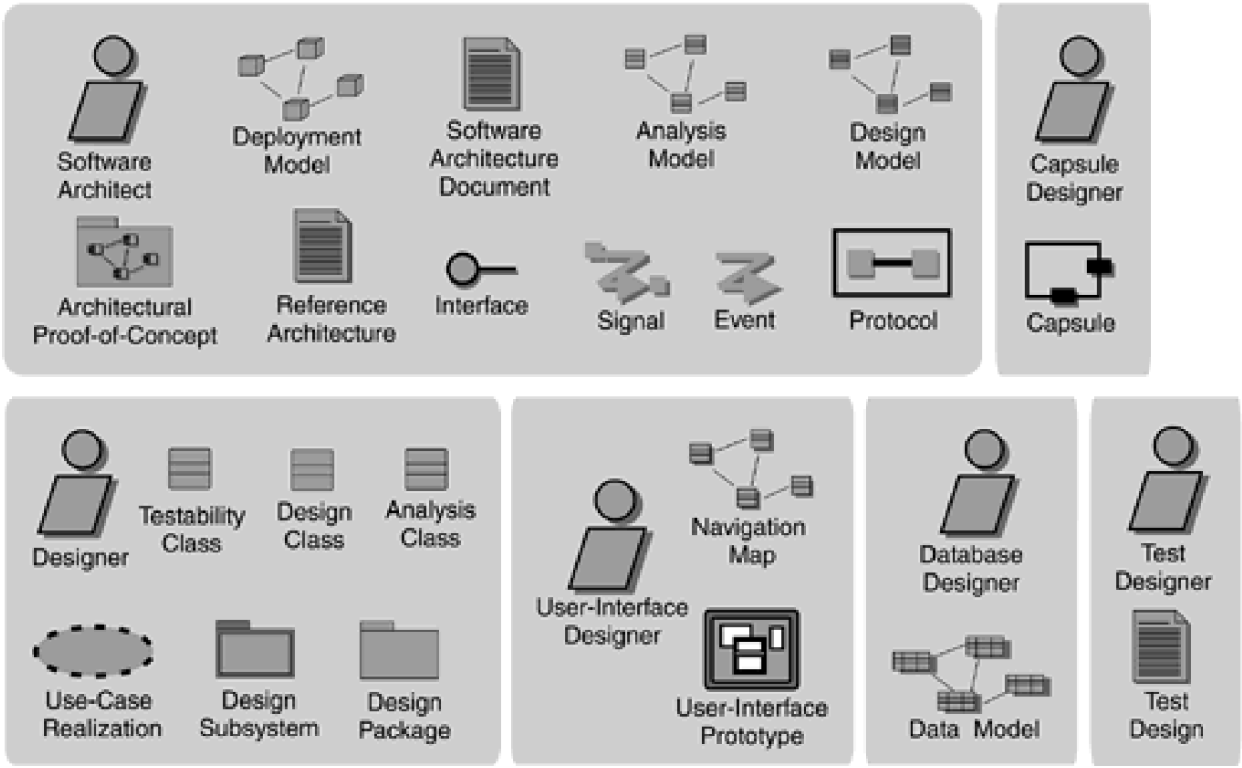
- *Software Architect*

  The Software Architect leads and coordinates technical activities and artifacts throughout the project. He or she establishes the overall structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between the major groupings. In contrast with the views of the other roles, the architect's view is one of breadth rather than depth. See Chapter 5, An Architecture-centric Process.

- *Designer*

  The Designer defines the responsibilities, operations, attributes, and relationships of one or several classes and determines how they should be adjusted to the implementation environment. In addition, the designer may have responsibility for one or more design packages or design subsystems, including any classes owned by the packages or subsystems.

Figure 10-1. Roles and artifacts in analysis and design



Analysis and design can optionally include the following roles:

- *Database Designer:* The Database Designer is needed when the system being designed includes a database.
- *Capsule Designer (for real-time systems):* The Capsule Designer is a kind of designer who focuses on ensuring that the

system is able to respond to events in a timely manner through appropriate use of concurrent design techniques.

- *Architecture Reviewer and Design Reviewer:* These specialists review the key artifacts produced through this workflow.

The key artifacts of analysis and design are as follows:

- The *Design Model,* which is the major blueprint for the system under construction
- The *Software Architecture Document*, which captures various architectural views of the system

PREVIOUS NEXT

Designing a User-Centered Interface

The expression *user-interface design* can mean one of at least the two following things:

- The visual shaping of the user interface so that it handles various usability requirements
- The design of the user interface in terms of design classes (and components such as ActiveX classes and JavaBeans) that is related to other design classes dealing with business logic, persistence, and so on, and that leads to the final implementation of the user interface

We operate under the first definition, focusing on achieving a user-centered design. In the requirement discipline, user profiles and user environments are specified in the vision document, with the main input to the design of the user interface being the use-case model, the supplementary specifications, and the storyboards, which are developed in conjunction with the users or their representatives and other key stakeholders. The results are detailed definitions of user characteristics and realizations of the user-interface-specific parts of the use cases.

Based on these results, a prototype of the user interface is built, in most cases by using a prototyping tool. We call this *user-interface prototyping.* This provides a valuable feedback mechanism for determining the final requirements of the system.

The Design Model

The primary artifact of the analysis and design discipline is the design model. It consists of a set of collaborations of model elements that provide the behavior of the system. This behavior in turn is derived primarily from the use-case model and nonfunctional requirements.

The design model consists of collaborations of classes, which may be aggregated into packages and subsystems to help organize the model and to provide compositional building blocks within the model. A *class* is a description of a set of objects that share responsibilities, relationships, operations, attributes, and semantics. A *package* is a logical grouping of classes, perhaps for organizational purposes, that reduces the complexity of the system. A *subsystem* is a kind of package consisting of a set of classes that act as a single unit to provide specific behaviors.

For a very data-centric system, the design model may be complemented by a *data model.*

The Analysis Model

Generally, there is one design model of the system; analysis produces a rough sketch of the system, which is refined further in design. The upper layers of this model describe the application-specific, or more analysis-oriented, aspects of the system. Using a single model reduces the number of artifacts that must be maintained in a consistent state.

In some companies—those in which systems live for decades or there are many variants of the system—a separate analysis model has proved useful. The analysis model is an abstraction, or generalization, of the design. It omits most of the details of how the system works and provides an overview of the system's functionality. The extra work required to ensure that the analysis and design models remain consistent must be balanced against the benefit of having a view of the system that represents only the most important details of how the system works.

The Role of Interfaces

Interfaces are used to specify the behavior offered by classes, subsystems, and components in a way that is independent of the implementation of the behavior. They specify a set of operations performed by the model elements, including the type returned and the number and types of parameters. Any two model elements that offer the same interface are interchangeable. Interfaces improve the flexibility of designs by reducing dependencies between parts of the system and therefore making them easier to change.

Artifacts for Real-Time Systems

Real-time systems are those with particular requirements for timeliness or responsiveness, often with the constraint of hard deadlines. Such systems are often found in safety-critical applications. The Rational Unified Process describes the following additional artifacts intended specifically to meet the demands of developing such systems:

- *Capsule:* A capsule is a specific design pattern, which represents an encapsulated thread of control in the system. Capsules have ports, through which they communicate with other capsules.
- *Protocol:* A protocol specifies the way a set of ports communicate, the messages exchanged, and how they are ordered.
- *Event:* An event is the specification of an occurrence in space and time, or, less formally, an occurrence of something to which the system must respond.
- *Signal:* A signal is an asynchronous event that may cause a state transition in the state machine of an object.

These additional artifacts are stereotypes of *class* and are part of the design model. They support the particular approach for real-time system development defined in the Rational Unified Process, which is based on ideas presented in the textbook *Real-Time Object-Oriented Modeling*.[1]

[1] Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.

Component-Based Design

Components are elements in the implementation model and are used to express the way the system is implemented in a set of smaller "chunks." Components offer one or more interfaces and are dependent only on the interfaces of other components. Using components lets you design, implement, deliver, and replace various parts of the system independently of the rest of the system.

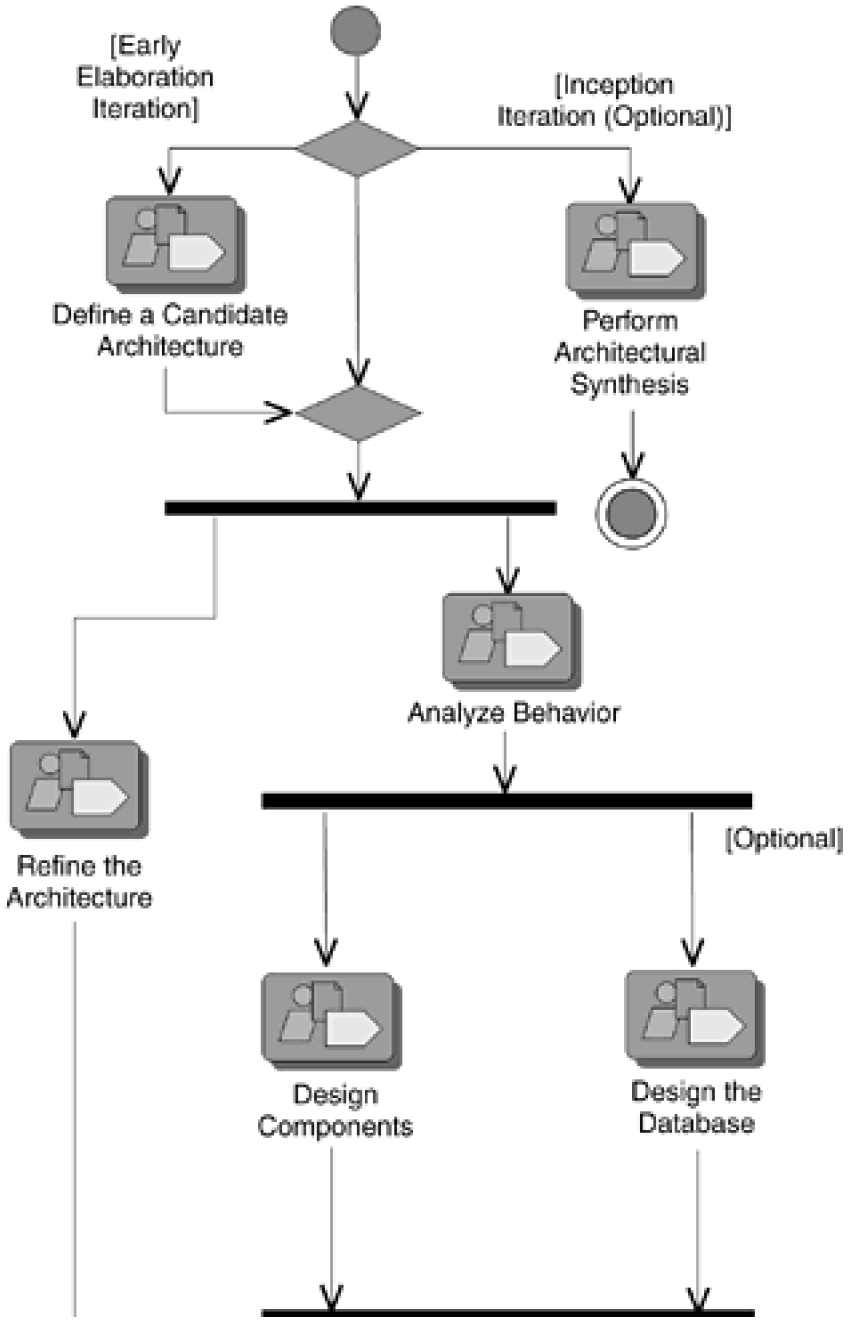
The design model element that corresponds to the component is the subsystem: Subsystems are independent units of functionality that allow different parts of the system to be designed and implemented independently. Subsystems offer interfaces and are dependent on the interfaces of other model elements.

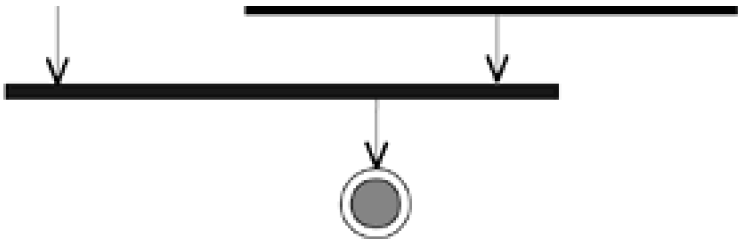
If necessary, the design can also include a data model (expressed in UML) that expresses the way the persistent objects in the system are stored and retrieved. A data model is useful when the underlying database technology imposes a significantly different physical model from that of the object-oriented application.

Workflow

Figure 10-2 illustrates an iteration in analysis and design in UML activity diagram form. Each activity state represents a workflow detail in the Rational Unified Process—for example, Define a Candidate Architecture. Each workflow detail, in turn, is composed of one or more activities. We see that some workflow details are phase-dependent: In elaboration iterations, the architecture is defined and then refined, and the more important or risky areas of the application are subjected to analysis and design.

Figure 10-2. Workflow in analysis and design





In subsequent iterations during the construction phase, the emphasis shifts to the addition of application functionality on top of what is, by then, a stable architectural platform.

Workflow Details

Define a Candidate Architecture

This workflow detail is composed of the activities Architectural Analysis, performed by the architect, and Use-Case Analysis, performed by the designer. Its purposes are:

- Create an initial sketch of the architecture of the system, by defining
    - An initial set of architecturally significant elements to be used as the basis for analysis;
    - An initial set of analysis mechanisms;
    - The initial layering and organization of the system; and
    - The use-case realizations to be addressed in the current iteration.
- Identify analysis classes from the architecturally significant use cases.
- Update the use-case realizations with analysis class interactions.

Refine the Architecture

This workflow detail is composed of the activities Identify Design Mechanisms, Identify Design Elements, Incorporate Existing Design Elements, Describe Runtime Architecture, and Describe Distribution, all performed by the architect, and Review the Architecture, performed by the architecture reviewer. The purpose of this workflow detail is to

- Provide the natural transition from analysis activities to design activities, identifying
    - Appropriate design elements from analysis elements and
    - Appropriate design mechanisms from related analysis mechanisms.
- Maintain the consistency and integrity of the architecture, ensuring that
    - New design elements identified for the current iteration are integrated with preexisting design elements

and

- Maximal reuse of available components and design elements is achieved as early as possible in the design effort.

- Describe the organization of the system's runtime and deployment architecture.
- Organize the implementation model to make the transition between design and implementation seamless.

Analyze Behavior

This workflow detail is composed of the activities Use-Case Analysis, performed by the designer, Identify Design Elements, performed by the architect, and Review the Design, performed by the design reviewer. The purpose of this workflow detail is to transform the behavioral descriptions provided by the use cases into a set of elements on which the design can be based. Note that in Analyze Behavior, we are becoming less concerned with the nonfunctional requirements levied on the system and much more with how to deliver the needed application capabilities.

Design Components

This workflow detail is composed of the activities Use-Case Design, Class Design, and Subsystem Design, performed by the designer, and Review the Design, performed by the design reviewer. The purposes of this workflow detail are:

- Refine the definitions of design elements by working out the details of how the design elements implement the behavior required of them.
- Refine and update the use-case realizations based on new design elements introduced, thus keeping the use-case realizations up to date.
- Review the design as it evolves.

In Design Components, class characteristics and relationships, subsystem interfaces and their realization by the contained classes, and use-case realizations in terms of collaborations are all completely specified.

Design Real-Time Components

This workflow detail applies to designs that will use the Capsule artifact as a primary design element, within the context of a real-time or reactive system. Design Real-Time Components has a purpose similar to Design Components, but includes the activity Capsule Design to define the concurrent threads of control in the system (i.e., *capsules*) and the *protocols* between them.

Design the Database

This is an optional workflow detail, invoked when the system involves a large amount of data in a database. Design the Database

is composed of the activities Database Design, performed by the database designer, Class Design, performed by the designer, and Review the Design, performed by the design reviewer. Its purposes are:

- Identify the persistent classes in the design.
- Design appropriate database structures to store the persistent classes.
- Define mechanisms and strategies for storing and retrieving persistent data in such a way that the performance criteria for the system are met.

Note that Figure 10-2 shows component design and database design occurring in parallel with architectural refinement: There may be many interactions between roles performing these workflows, for example, the discovery of a new class may require adjustment of the architecture, or architectural refinement for performance reasons may impact the design of persistent classes. This is the case particularly for iterations in the elaboration phase, when the architecture is still fluid.

PREVIOUS NEXT

Tool Support

The language of choice to express all these models is the UML, and the modeling guidelines associated with the various artifacts are expressed in terms of the UML. The tools of choice to capture, manage, and display the models are Rational Rose and Rational XDE. Both allow you to perform round-trip engineering with a few selected programming languages, keeping the design and the code perfectly synchronized and allowing the system to evolve from the design, from the code, or from both. The more powerful Rational XDE let us play with reusable assets and design patterns in particular.

The Rational Unified Process provides tool mentors to guide the designers in the use of UML and Rose, Rose RealTime and XDE. Rose RealTime allows the direct execution of a design model.

Summary

- Analysis and design bridge the gap between requirements and implementation. This workflow uses use cases to identify a set of objects that is subsequently refined into classes, subsystems, and packages.
- Responsibilities in analysis and design are distributed among the roles Software Architect (the big picture issues), Designer (the details), and Database Designer (the details that require specialized knowledge about handling persistence).
- Analysis and design result in a design model, which can be abstracted using three architectural views. The logical view presents the decomposition of the system into a set of logical elements (classes, subsystems, packages, and collaborations). The process view maps those elements to the processes and threads in the systems. The deployment view maps the processes to a set of nodes on which they execute.
- The design of the user interface proceeds in parallel, resulting in a user-interface prototype.
- In some cases, a separate analysis model can be useful for presenting an overview or abstraction of the system or to bootstrap the design activities.

Chapter 11. The Implementation Discipline

This chapter describes the implementation discipline. It introduces the concepts of prototypes and incremental integration.

Purpose

The implementation discipline has four purposes:

- To define the organization of the code in terms of implementation subsystems organized in layers
- To implement classes and objects in terms of components (source files, binaries, executables, and others)
- To test the developed components as units
- To integrate into an executable system the results produced by individual implementers or teams

The scope of test within the implementation discipline is limited to unit test of individual components. System test and integration test are described in the test discipline (see Chapter 12). This is because the implementer is responsible for unit test. Remember that, in practice, the workflows are interwoven.

To explain implementation in the Rational Unified Process, we introduce the following three key concepts:

- Builds
- Integration
- Prototypes

Builds

A *build* is an operational version of a system or part of a system that demonstrates a subset of the capabilities to be provided in the final product.

During iterative software development there will be numerous builds. Each build provides early review points and helps to recognize integration problems as soon as they are introduced.

Builds are an integral part of the iterative lifecycle. They represent ongoing attempts to demonstrate the functionality developed to date. Each build is placed under configuration control in case there is a need to roll back to an earlier version when added functionality causes breakage or otherwise compromises build integrity.

Typically, projects try to produce builds at regular intervals, up to one each day, but at least one per week toward the end of an iteration.

Integration

The term *integration* refers to a software development activity in which separate software components are combined into a whole. Integration is done at several levels and stages of the implementation, for the following purposes:

- To integrate the work of a team working in the same implementation subsystem before the subsystem is released to system integrators
- To integrate subsystems into a complete system

The Rational Unified Process approach to integration is that the software is integrated incrementally. *Incremental* integration means that code is written and tested in small pieces and then is combined into a working whole by the addition of one piece at a time.

The contrasting approach is phased integration. *Phased* integration relies on integrating multiple (new and changed) components at the same time. The major drawback of phased integration is that it introduces multiple variables and makes it hard to locate errors. An error could be inside any of the new components, in the interaction between the new components, or in the interaction between the new components and those at the core of the system.

Incremental integration offers the following benefits:

- Faults are easy to locate. When a new problem occurs during incremental integration, the new or changed component (or its interaction with the previously integrated components) is the obvious first place to look for the fault. Incremental integration also makes it likely that defects will be discovered one at a time, something that makes it easier to identify faults.
- The components are tested more fully. Components are integrated and tested as they are developed. This means that the components are exercised more often than they are when integration is done in one step.
- Some part of the system is running earlier. It is better for the morale of developers to see early results of their work instead of waiting until the end to test everything. It also makes it possible to receive early feedback on design, tools, rules, and style.

It is important to understand that integration occurs (at least once) during each iteration. An iteration plan defines the use cases to be designed and thus the classes to be implemented. The focus of the integration strategy is to determine the order in which classes are implemented and then combined.

Prototypes

Prototypes are used in a directed way to reduce risk. Prototypes can reduce uncertainty surrounding the following issues:

- The business viability of a product being developed
- The stability or performance of key technology
- Project commitment or funding (through the building of a small proof-of-concept prototype)
- The understanding of requirements
- The look and feel, and ultimately the usability, of the product

A prototype can help to build support for the product by showing something concrete and executable to users, customers, and managers.

The nature and goal of the prototype, however, must remain clear throughout its lifetime. If you don't intend to evolve the prototype into the real product, don't suddenly assume that because the prototype works it should become the final product. An exploratory, behavioral prototype, intended to try out a user interface rapidly, rarely evolves into a strong, resilient product.

Types of Prototype

You can view prototypes in two ways: by what they explore and by how they evolve (their outcome). In the context of the first view—what they explore—there are two main kinds of prototype:

- A behavioral prototype, which focuses on exploring specific behavior of the system
- A structural prototype, which explores architectural or technological concerns

In the context of the second view—their outcome—there are also two kinds of prototype:

- An exploratory prototype, also called a throwaway prototype, which is thrown away after it is finished and you have learned whatever you wanted from it
- An evolutionary prototype, which evolves to become the final system

Behavioral Prototypes

Behavioral prototypes tend to be exploratory prototypes; they do not try to reproduce the architecture of the system to be developed but instead focus on what the system will do as seen by the users (the "skin"). Frequently, this kind of prototype is "quick and dirty" and not built to project standards. For example, you might use Visual Basic as the prototyping language even

though you intend to use C++ for the development project.

Structural Prototypes

Structural prototypes tend to be evolutionary prototypes; they are more likely to use the infrastructure of the ultimate system (the "bones") and are more likely to evolve into the final system. If you build the prototype using the production language and tool set, you gain the advantages of being able to test the development environment and helping personnel familiarize themselves with new tools and procedures.

The Rational Unified Process advocates the development of an evolutionary structural prototype throughout the elaboration phase, accompanied by any number of exploratory prototypes. An example of an exploratory prototype is the user-interface prototype introduced in Chapter 9, The Requirements Discipline.

Exploratory Prototypes

An exploratory prototype is designed to test a key assumption that involves functionality or technology or both. This prototype might be something as small as a few hundred lines of code created to test the performance of a key software or hardware component. Exploratory prototypes also serve to clarify requirements; a small prototype might be developed to see whether the developer understands a particular behavioral or technical requirement.

Exploratory prototypes are usually intended to be throwaway prototypes created with minimal effort, and therefore they are usually tested only informally. The design of exploratory prototypes tends to be informal and is usually the work of one or two developers.

Evolutionary Prototypes

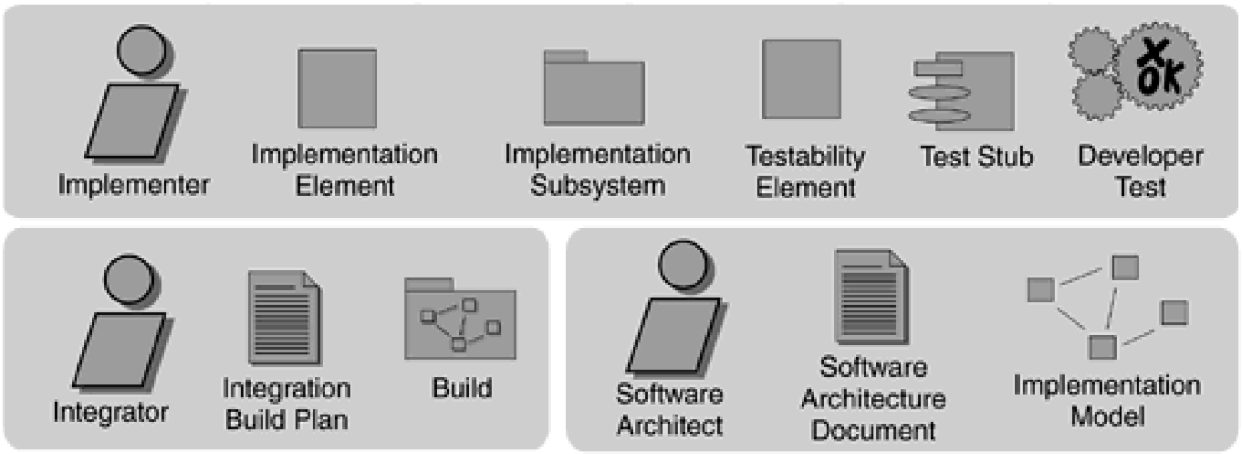
Evolutionary prototypes, as their name implies, evolve from one iteration to the next. Although initially they are not of production quality, their code tends to be reworked as the product evolves. To keep rework manageable, evolutionary prototypes tend to be designed rather formally and tested somewhat formally even in the early stages. As the product evolves, testing as well as design usually becomes more formalized.

Squeezed by time constraints, project teams often fall into the trap of keeping an exploratory or throwaway prototype, with all its shortcomings in quality, as the final system. To avoid this problem, you should clearly define the objective and scope of a prototype when starting to develop it.

Roles and Artifacts

In the Rational Unified Process, how is all this translated in terms of roles, artifacts, activities, and workflows? Figure 11-1 shows the roles and artifacts involved in the implementation discipline.

Figure 11-1. Roles and artifacts in implementation



The main roles involved in the implementation discipline are as follows:

- The *Implementer,* who develops the components and related artifacts and performs unit testing
- The *System Integrator,* who constructs a build

Other roles include:

- The *Software Architect,* who defines the structure of the implementation model (layering and subsystems)
- The *Code Reviewer,* who inspects the code for quality and conformance to the project standard

The key artifacts of implementation are as follows:

- *Implementation Subsystem*

  A collection of implementation elements and other implementation subsystems, it is used to structure the implementation model by dividing it into smaller parts.

- *Implementation Elements*

  A piece of software code (source, binary, or executable) or a file containing information (for example, a start-up file or a readme file), an implementation element can also be an aggregate of other elements—for example, an application consisting of several executables.

- *Integration Build Plan*

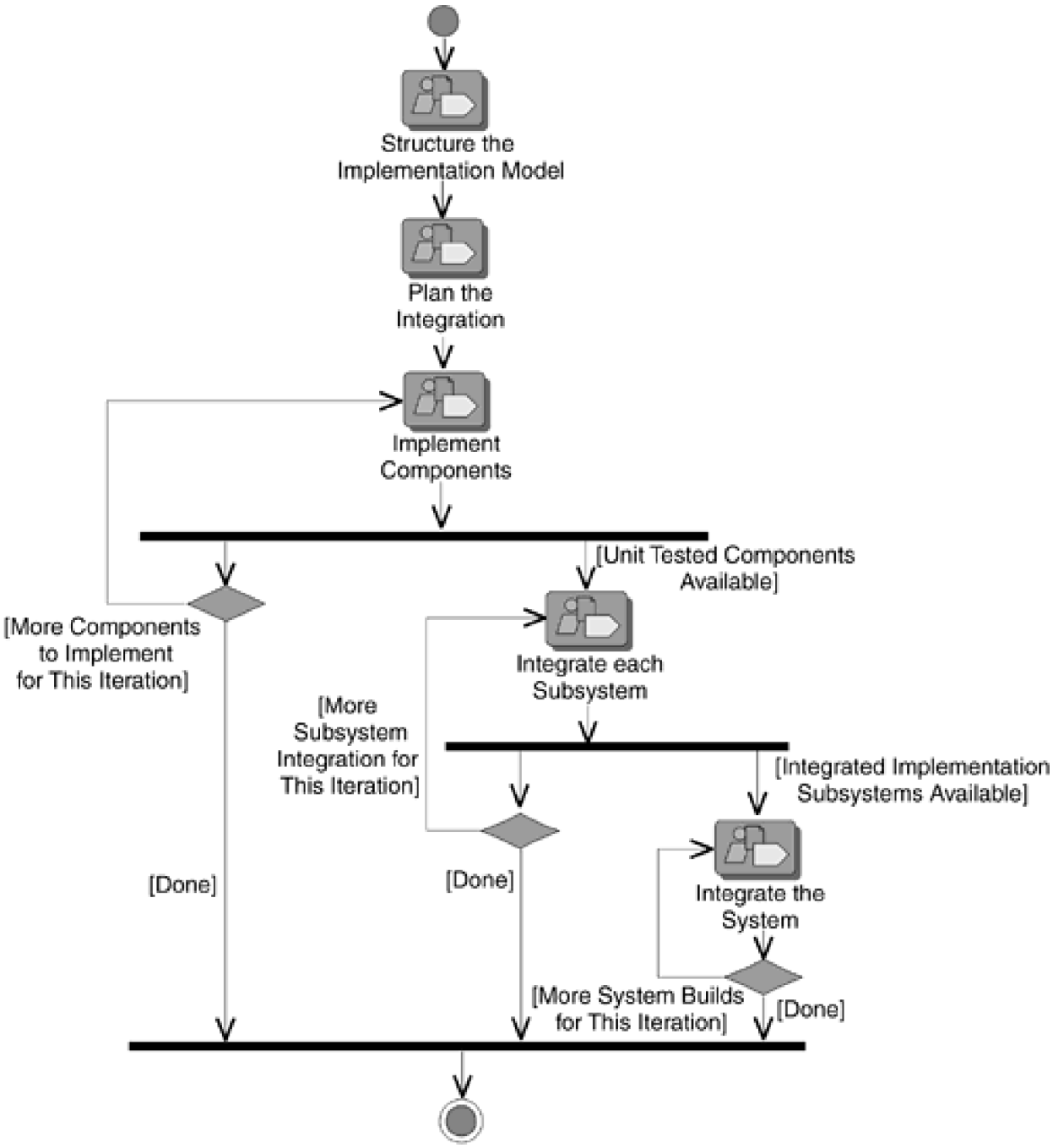
  This document defines the order in which the elements and subsystems should be implemented and specifies the builds

to be created when the system is integrated.

Workflow

Figure 11-2 shows an implementation workflow, giving an overview of all activities.

Figure 11-2. An implementation workflow



The main work to Structure the Implementation Model is done early in the elaboration phase. The goal is to ensure that the

implementation model is organized in such a way as to make the development of components and the build process as conflict-free as possible. A well-organized model will prevent configuration management problems and will allow the product to be built up from successively larger integration builds.

For each iteration, you would do the following:

- Plan which subsystems should be implemented, and the order in which the subsystems should be integrated in the current iteration. This is described in Plan the Integration.
- For each subsystem, the responsible implementer plans the integration of the subsystem, that is, defines in which order to implement the classes.
- The implementers implement the classes and objects in the design model—write source code, adapt existing components, compile, link, and execute. If the implementers discover defects in the design, they submit rework feedback on the design.
- The implementers also fix code defects and perform unit tests to verify the changes. Then the code is reviewed to ensure quality and that the programming guidelines are followed.
- If several implementers work (as a team) in the same implementation subsystem, one of the implementers is responsible for integrating the new and changed components from the individual implementers into a new version of the implementation subsystem. The integration results in a series of builds in a team integration area. Each build is then integration tested by an integration tester. The latest version of the subsystem is released for system integration.
- Following the Integration Build Plan, the System Integrator integrates the system, by adding the released subsystems into an integration area and creating builds. Each build is then integration tested by an integration tester. After the last increment, the build can be system tested by a system tester.

Implementation is tied closely to design, and there are clear tracing links from design elements to implementation elements (for example, classes to code).

Round-trip engineering, which is possible only with the use of certain programming languages and tools, such as Rational Rose, and for some types of applications, will allow a close tie between design and implementation. The person acting alternately as the designer and the implementer can either modify the design model and regenerate the corresponding code or modify the implementation code and then alter the design to match the modification. This closes a gap in the process and helps prevent errors in translating the design or a design's getting out of sync with the implementation and therefore not being trusted by the implementers.

For information about integration test and system tests, see Chapter 12, The Test Discipline.Chapter 13, The Configuration and Change Management Discipline, discusses the activity of building the product.

Tool Support

Traditionally, classic software development tools—such as editors, compilers, linkers, and debuggers—have been used in the area of implementation. Today these code-level tools are assembled into integrated development environments and share semantic knowledge. An example is the Rational Apex development environment for Ada and C++.

Rose provides support for round-trip engineering, closely tying together design and the implementation activities. Tools such as Purify and Quantify support the detection of defects in code. Rational ClearCase provides support for individual workspaces as well as "staging" workspaces for subsystem and system integration. Rational ClearQuest is the tool of choice for tracking change requests and defects and tracing them to the source code.

Summary

- A key feature of the Rational Unified Process is its approach to incremental integration throughout the lifecycle.
- During the elaboration phase, you build an evolutionary, structural prototype that becomes the final system during the construction phase.
- In parallel, you can build a few throwaway behavioral prototypes to explore issues such as the user interface.
- Round-trip engineering is a technique, supported by a tool such as Rose or XDE, that closely ties together the design and the implementation of a software development effort.

Chapter 12. The Test Discipline

This chapter introduces the concept of quality, describes the Test discipline, and discusses the relationship between Quality, Test, and the other disciplines in the process.

Purpose

In many respects, the Test discipline acts as a service provider to the other disciplines. Testing focuses primarily on evaluating or assessing product quality, using a number of core practices:

- Find and document failures in the software product: defects, problems.
- Advise management on the perceived quality of the software.
- Evaluate the assumptions made in design and requirement specifications through concrete demonstration.
- Validate that the software product works as designed.
- Validate that the requirements are implemented appropriately.

An interesting difference exists between Test and the other RUP disciplines: Essentially Test is tasked with finding and exposing *weaknesses* in the software product. To get the biggest benefit, you need a different philosophy or mindset than that used in the Requirements, Analysis and Design, and Implementation disciplines. Whereas those three disciplines focus on completeness, consistency, and correctness, the Test discipline focuses on what is missing, incorrect, or inconsistent.

A good test effort is driven by questions such as these:

- How could this software break?
- In what possible situations could this software fail to work predictably?

Test challenges the assumptions, risks, and uncertainty inherent in the work of other disciplines, and addresses those concerns using concrete demonstration and impartial evaluation. You want to avoid two potential extremes:

- An approach that does not suitably or effectively chal-lenge the software and exposes its inherent problems or weaknesses.
- An approach that is inappropriately negative or destructive—adopting such a negative approach, you may find it impossible to ever consider the software product of acceptable quality. Taking such an approach can alienate the Test effort from the other disciplines.

Information presented in various surveys and essays states that software testing accounts for 30% to 50% of software development costs. It is, therefore, somewhat surprising to note that most people believe computer software is not well tested before it's delivered. This contradiction is rooted in a few key issues:

- Testing software is very difficult. How do you quantify the different ways in which a given program can behave or misbehave?
- Typically, testing is done without a clear methodology, creating results that vary from project to project and from organization to organization. Success is primarily a factor of the quality and skills of the individuals in the test team.
- Productivity tools are either used inefficiently or not at all, which makes the laborious aspects of testing unmanageable. In addition to the lack of automated test execution, many test efforts are conducted without tools that let you effectively manage extensive Test Data and Test Results.
- Flexibility of use and complexity of software make complete testing an impossible goal. Using a well-conceived

strategy and appropriate tools can improve both the productivity and effectiveness of software testing.

High-quality software is essential to the success of safety-critical systems—such as air traffic control, missile guidance, and medical delivery systems—where a failure can harm people. These systems have to exhibit extremely high reliability and be tolerant to failure. The criticality of a typical Management Information System (MIS) may not be as immediately obvious, but it's likely that the impact of a defect could cause the business using the software considerable expense in lost revenue and possibly legal costs. In this information age, with increasing demands on providing electronically delivered services over the Internet, many MIS systems are now considered mission-critical; that is, when failures occur, companies cannot fulfill their functions and they experience massive losses.

A continuous approach to quality, initiated early in the software lifecycle, can significantly lower the cost of completing and maintaining your software. This greatly reduces the risk of deploying poor-quality software and having to fix it in a subsequent release.

PREVIOUS NEXT

Testing in the Iterative Lifecycle

Testing is not a single activity, nor is it a project phase during which we assess quality. If developers are to obtain timely feedback on evolving product quality, testing must occur throughout the lifecycle: We can test the broad functionality of early prototypes; we can test the stability, coverage, and performance of the architecture while there is still an opportunity to fix it; and we can test the final product to assess its readiness for delivery to customers. The all-too-common viewpoint (arising from the waterfall methodology) that testing is a kind of final check to make sure that everything works misses the main benefit of testing: providing feedback while there is still time (and resources) to do something about it.

Quality

The term *quality* commonly refers to a number of things: Principally it means the absence of defects, but more important, a fitness for a desired purpose—something imbued with quality does what we want it to do. A thing can be free from defects, but if it does not do what we need it to do, it is just as useless as a defective product. As Gerald Weinberg puts it, "Quality is value to some person."

The ultimate goal of testing is to assess the quality of the end product and, in doing so, to assess the quality of the components that comprise the product and the architecture that gives form to these components. By means of this assessment, we need to ensure that the product satisfies or exceeds a defined and accepted set of requirements, as assessed by defined and accepted measures and criteria.

Quality cannot be assessed completely in isolation; software is developed by organizations using engineering processes. A poor process, or one that is not followed, can be a significant cause of poor quality. As a result, quality assessments often consider process quality and organizational factors as well as direct product quality.

Product Quality Ownership

Producing a quality product is the responsibility of every member of the project team—it is not possible for a "quality assurance" organization to ensure quality. If quality is not designed and built into the product from the start, it will not be possible for it to be "added later" through an aggressive quality-assurance effort.

The role of testing is not to *assure* quality but to *assess* it and to provide timely feedback so that quality issues can be resolved in a timely and cost-effective manner. The role of the tester is to assess quality and to provide feedback, while the role of the project team is to produce artifacts or work products that meet requirements and quality expectations.

In RUP, the assessment of process quality is not in the scope of the Test discipline but a concern associated with the Environment discipline and the Process Engineering Process (PEP, see Chapter 14).

Dimensions of Testing

To assess product quality, different kinds of test, each one with a different focus, are needed. These tests can be categorized in various ways:

- *Quality dimension:* the test focuses on a major quality characteristic or attribute.
- *Level of testing:* the test strategy targets a general category of software elements.
- *Type of test:* an individual test has a specific test objective, usually limited to a single quality dimension.

Quality Dimension

There are patterns to quality problems; the same kinds of problems tend to recur in virtually every system. There are various conventions for enumerating quality problems, each designed to help you reason the most important dimension of quality in your context. The following aspects are generally assessed for most products:

- *Reliability*

  The software performs predictably and consistently; it is resistant to failure during execution: no crashing, hanging, memory leaks, and so on.

- *Functionality*

  The software executes the required use cases or desired behavior as intended.

- *Performance*

  The software and the system execute and respond in a timely manner and continue to perform acceptably when subjected to real-world operational characteristics (e.g., load, stress, and lengthy periods of operation). Performance testing focuses on ensuring that the functionality of the system can be provided while satisfying the nonfunctional requirements of the system.

- *Usability*

  The software is suitable for use by its end users; the focus here is on human factors, esthetics, and so forth.

For each quality dimension, one or more types of test should be executed at different test levels. Additionally, there are other qualities to consider whose assessment may be more subjective: supportability, maintainability, extensibility, flexibility, and so on. Whenever possible, quantitative measures should be derived to assess these qualities.

Levels of Testing

Testing is not a single activity that is executed all at once. In each iteration, testing is executed against different types of targets (targets-of-test) in one or more test cycles. In each test cycle, different targets will be tested, ranging from small elements of the system, such as components (unit testing), to completed systems (system testing). The four levels we use have the following

purposes:

- *Unit testing*

  The smallest testable elements of the system are tested, typically at the same time that those elements are implemented.

- *Integration testing*

  The integrated units (or components or subsystems) are tested.

- *System testing*

  The complete application and system (one or more applications) are tested.

- *Acceptance testing*

  The complete application (or system) is tested by end users (or representatives) to determine readiness for deployment.

Keep in mind that these levels occur throughout the lifecycle, with varying emphasis, either in sequence or in parallel. An early conceptual prototype used in the inception phase to assess the viability of the product vision will be subjected to acceptance tests, informal though they may be. For example, an architectural prototype developed during the elaboration phase will be subjected to integration and system tests to validate architectural integrity and performance of key architectural elements, even though most of the code in the system exists only as stubs. These levels are not "phases" that occur in sequence near the end of the project—in the iterative lifecycle, testing occurs early and often.

Types of Test

There are many types of test, each one focusing on a specific test objective, testing a characteristic or attribute of the software, or finding certain classes of faults or failures. Because tests are executed throughout the lifecycle, the software being tested may be a single unit of code, integrated units, or a complete application or system. Here are some of the most common types of test:

- *Benchmark test*

  Assesses the performance of a target-of-test against a known standard such as existing software or measurement(s)

- *Configuration test*

  Assesses how the target-of-test functions on different test configurations (hardware or software)

- *Functional test*

  Assesses how the target-of-test functions execute the required use case as intended

- *Installation test*

  Assesses how the target-of-test installs on different configurations or under different conditions, such as insufficient disk space

- *Integrity test*

  Assesses the target-of-test's reliability, robustness, and resistance to failure during execution

- *Load test*

  Assesses the acceptability of the target-of test's performance under varying operational conditions, such as number of

users, number of transactions, and so on, while the configuration remains constant

- *Performance test*

  Assesses the acceptability of the target-of-test's performance using various configurations while the operational conditions remain constant

- *Stress test*

  Assesses the target-of-test's performance when abnormal or extreme conditions are encountered, such as diminished resources or an extremely high number of users

Regression Testing

Regression testing is a test strategy in which previously executed tests are reexecuted against a new version of the target-of-test to ensure that the quality of the target has not regressed (moved backward) when new capabilities have been added. The purpose of the regression test is to ensure two things:

- The defects identified in the earlier execution of test have been addressed.
- The changes made to the code have not introduced new defects or reintroduced old ones.

Regression testing can involve the reexecution of any of the test types. Typically, some regression testing is performed during each iteration, rerunning tests from prior iterations, and therefore is of particular importance in our lifecycle model.

Roles and Artifacts

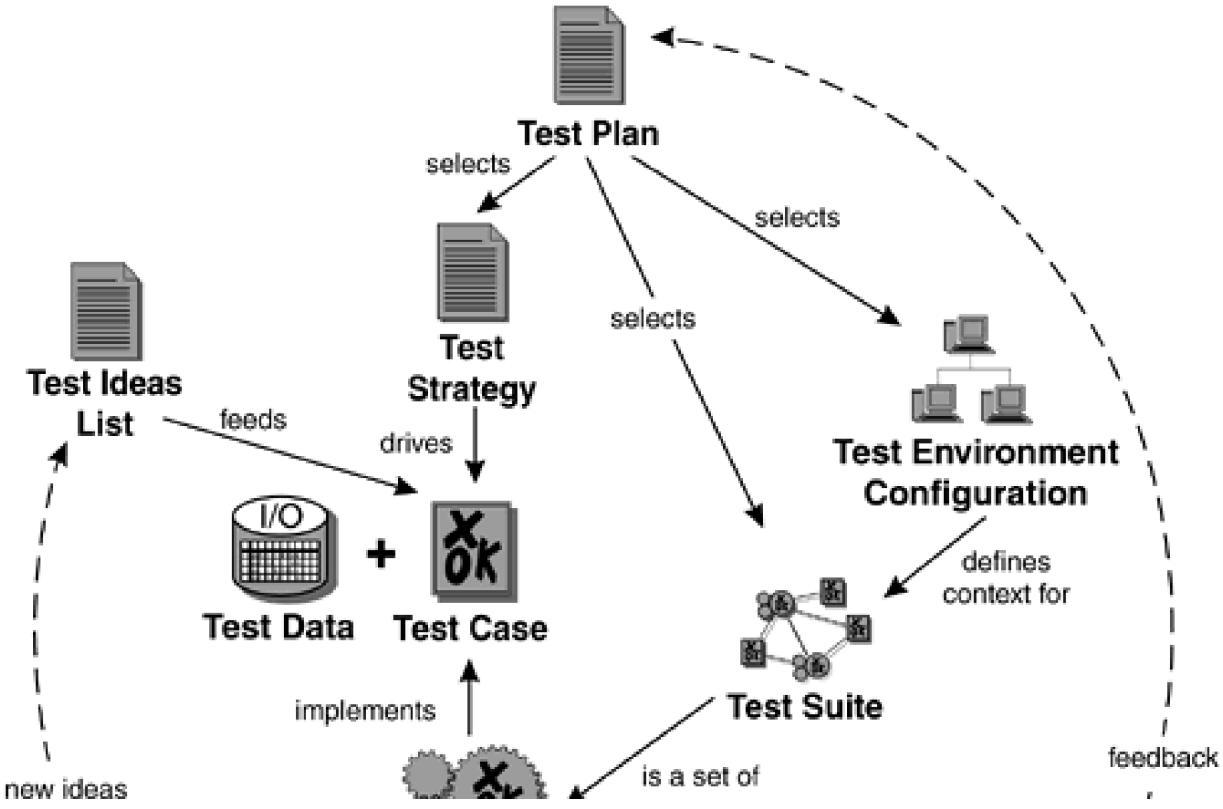
Four main roles are involved in the Test discipline:

- The *Test Manager* has the overall responsibility for the test effort's success. The role involves quality and test advocacy, resource planning and management, and resolution of issues that impede the test effort.
- The *Test Analyst* is responsible for identifying and defining the required tests, monitoring detailed testing progress and results in each test cycle, and evaluating the overall quality experienced as a result of testing activities. The role typically carries the responsibility for appropriately representing the needs of stakeholders that do not have direct or regular representation on the project.
- The *Test Designer* is responsible for defining the test approach and ensuring its successful implementation. The role involves identifying the appropriate techniques, tools, and guidelines to implement the required tests and to give guidance on the corresponding resource requirements for the test effort.
- The *Tester* is responsible for executing the system tests. This effort includes setting up and executing tests, evaluating test execution, recovering from errors, assessing the results of testing, and logging change requests.

When specific code (such as drivers or stubs) must be developed to support testing, the Designer and the Implementer are also involved in roles similar to the ones defined in Chapters 10 and 11.

Figure 12-1 shows the test artifacts and their relationships.Figure 12-2 shows the roles and artifacts in the Test discipline.

Figure 12-1. Artifacts and their relationships in the Test discipline



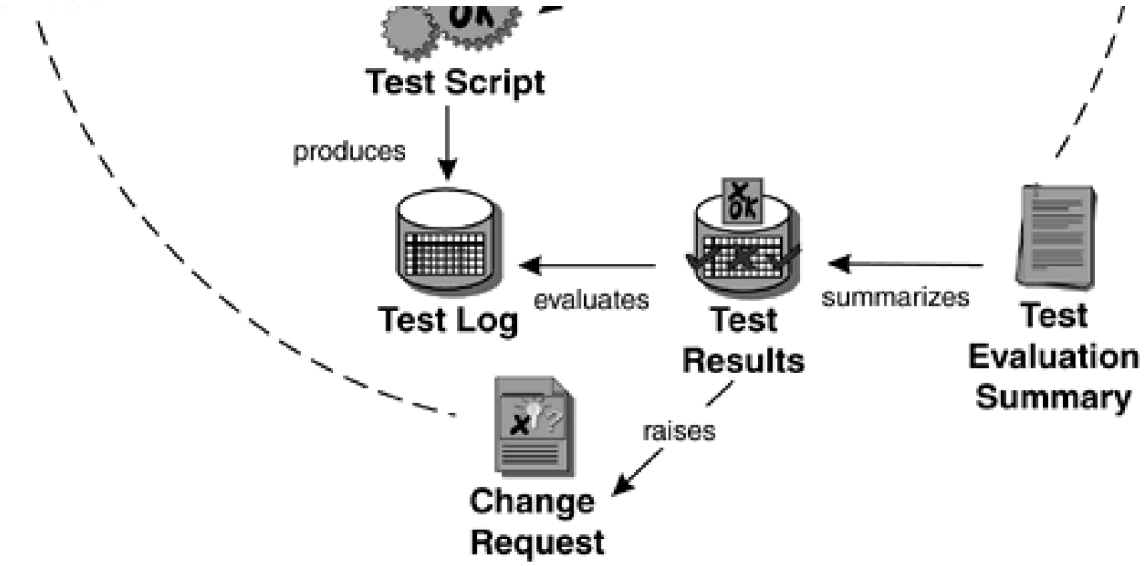
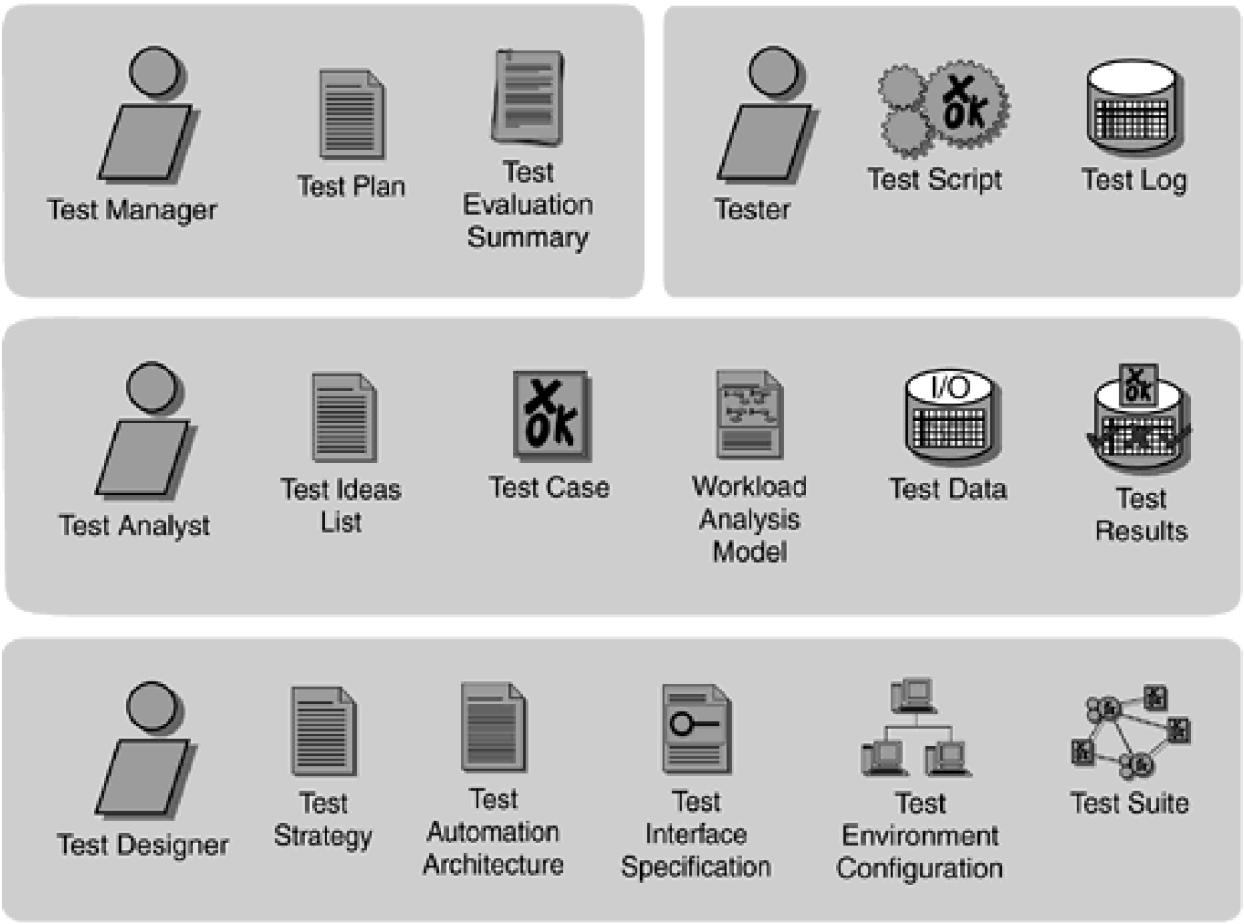


Figure 12-2. Roles and artifacts in the Test discipline



These are the key testing artifacts:

- The *Test Plan* contains information about the purpose and goals of testing for the project within its schedule. The Test Plan identifies the strategies to be used, the resources necessary to implement and execute testing, and which *Test Configurations* will be required.
- The *Test Analyst* maintains a *Test Ideas List,* an enumerated list of ideas, often partially formed, that identify potentially useful tests to conduct.

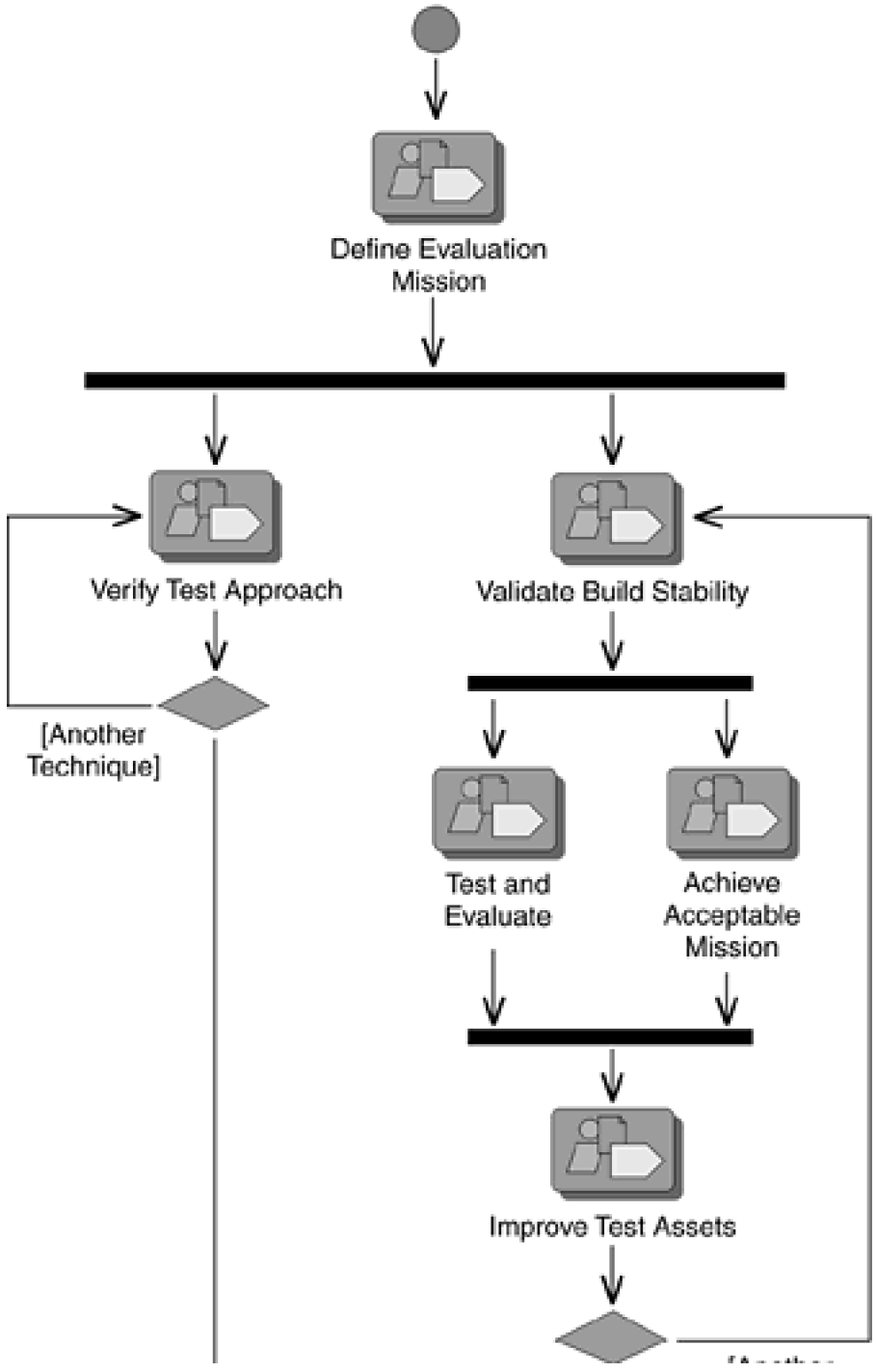
- Some test ideas will evolve into full-fledged *Test Cases,* which specify a test, its conditions for execution, and the associated *Test Data.*
- *Test Scripts* are manual or automated procedures used by the Tester to execute the tests.
- Test Scripts may be assembled into *Test Suites.*
- A *Workload Analysis Model* is a special kind of Test Case for performance testing; it identifies the variables and defines their values used in the different performance tests to simulate or emulate the actor characteristics and the end user's business functions (use cases), their load, and their volume.
- The *Test Log* is the raw data captured during the execution of Test Suites.
- Test logs are filtered and analyzed to produce *Test Results,* from which change requests are raised, documenting defects or enhancement requests (see Chapter 13). Ultimately a *Test Evaluation Summary* is produced as part of the project iteration assessment and periodic status assessment (see Chapter 7, The Project Management Discipline).

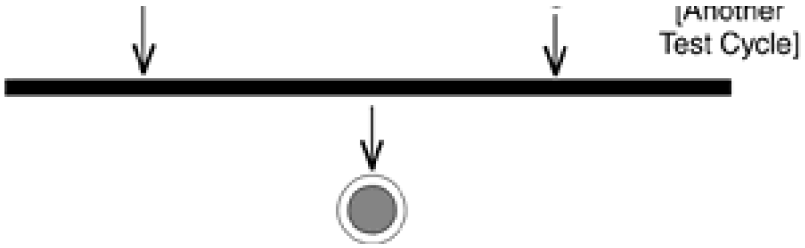
PREVIOUS NEXT

Workflow

Figure 12-3 illustrates the typical testing workflow, showing the major workflow details and their dependencies.

Figure 12-3. Typical testing workflow with details and their dependencies





Define Evaluation Mission

The purpose of this workflow detail is to identify the appropriate focus of the test effort for the iteration and to gain agreement with stakeholders on the corresponding goals that will direct the test effort.

For each iteration, this work focuses on:

- Identifying the objectives for, and deliverables of, the testing effort
- Identifying a good resource utilization strategy
- Defining the appropriate scope and boundary for the test effort
- Outlining the Test Strategy that will be used
- Defining how progress will be monitored and assessed

Verify Test Approach

The purpose here is to demonstrate that the various techniques outlined in the Test Strategy will facilitate the planned test effort. The intent is to verify by demonstration that the strategy will work, will produce accurate results, and will be appropriate for the available resources.

The objective is to gain an understanding of the constraints and limitations of each technique as it will be applied in the given project context and to either

- Find an appropriate implementation solution for each technique, or
- Find alternative techniques that can be used.

This helps to mitigate the risk of discovering too late in the project lifecycle that the approach to test is unworkable.

In each iteration, this work is focused on these tasks:

- Verify early that the intended test strategy will work and produce results of value.
- Establish the basic infrastructure to enable and support the test strategy.
- Obtain commitment from the development team to develop the software to meet the testability requirements to achieve the test strategy, and to provide continued support for those testability requirements.
- Identify the scope, boundaries, limitations, and constraints of each technique.

Validate Build Stability

Before entering a test cycle for a new Build, we validate that this Build is stable enough for detailed test and evaluation effort to begin. This work is also referred to as *smoke test, build verification test, build regression test, sanity check,* or *acceptance into testing.* This work prevents wasting resources on a futile and fruitless testing effort.

Test and Evaluate

Typically performed once per test cycle, this workflow detail involves the core tactical work of the test and evaluation effort, namely, the implementation, execution, and evaluation of specific tests and the corresponding reporting of incidents that are encountered.

For each test cycle, this work is focused on these tasks:

- Provide ongoing evaluation and assessment of the target of test.
- Record the information necessary to diagnose and resolve any identified defects or issues.
- Achieve suitable breadth and depth in the test and evaluation work to be able to provide feedback on the most likely areas of quality risk.

Achieve Acceptable Mission

The objective is to deliver a useful evaluation result to the stakeholders of the test effort, where useful evaluation result is assessed in terms of the Evaluation Mission set up at the beginning. In most cases that will mean focusing your efforts on helping the project team achieve the Iteration Plan objectives that apply to the current test cycle.

For each test cycle, this work is focused on these tasks:

- Actively prioritize the minimal set of necessary tests that must be conducted to achieve the Evaluation Mission.
- Advocate the resolution of important issues that have a significant negative impact on the Evaluation Mission.
- Advocate appropriate quality.
- Identify regressions in quality that have been introduced between test cycles.
- Where appropriate, revise the Evaluation Mission in light of the evaluation findings so as to provide useful evaluation information to the project team.

Improve Test Assets

The objective is to maintain and improve the various test assets: Test Ideas List, Test Cases, Test Data, Test Scripts, and so on. This is important especially if the intention is future reuse, in subsequent test cycles, of the assets developed in the current test cycle.

For each test cycle, this work is focused on these tasks:

- Add the minimal set of additional tests to validate the stability of subsequent Builds.
- Remove test assets that no longer serve a useful purpose or have become economic infeasible to maintain.
- Conduct general maintenance of and make improvements to the maintainability of test automation assets.
- Assemble Test Scripts into additional appropriate Test Suites.
- Explore opportunities for reuse and productivity improvements.
- Maintain Test Environment Configurations and Test Data sets.
- Document lessons learned—both good and bad practices—discovered during the test cycle.

PREVIOUS NEXT

Tool Support

Because testing is an iterative effort that is implemented throughout the development lifecycle, tool support provides an essential capability to test early and often. Manual testing is simply not productive enough to test the software sufficiently throughout the lifecycle—this is especially true for performance or load tests in which workload must be simulated and large amounts of test data gathered.

The Rational Suite of software development tools supports test automation and the Test discipline as follows:

- *Rational TestStudio*® is a suite of tools that support the implementation, execution, and evaluation of tests. The tools in TestStudio enable testers to create and execute GUI-based test scripts focused on the quality dimensions of reliability, function, and performance. TestStudio includes the following tools:

  - *Robot* supports the test implementation and execution by enabling testers to create and play back GUI test scripts and compare actual results with expected results.

  - *LogViewer* captures test results and presents reports to evaluate test execution.

  - *TestManager* supports test planning, design, and evaluation, and provides requirements-based test coverage and test status reports.

- *Rational Test Realtime*® is a cross-platform solution for component testing and runtime analysis. Test RealTime was designed specifically for embedded, real-time, and other types of cross-platform software products.

- *Rational XDE Tester*®, integrated with Rational XDE® for Java and Web testing.

- *Rational PerformanceStudio*® implements and executes virtual user (VU) test scripts for performance and limited functional tests.

- *Rational PurifyPlus*® supports the test discipline with the following tools:

  - *Rational Purify*® for locating hard-to-find runtime errors

  - *Rational PureCoverage*® to identify untested code and perform code-coverage analysis

  - *Rational Quantify*® to pinpoint performance bottlenecks

The Rational Unified Process provides tool mentors for many of these tools.

Summary

- Testing enables the assessment of the quality of the product being produced.
- Testing in an iterative process occurs in all phases of the lifecycle, enabling early feedback on product quality to be used to improve the product as it is designed and built. Testing is not just an end-of-lifecycle activity intended to approve or reject the developed system; it is a vital part of the project's ongoing feedback mechanism.
- Quality is everyone's responsibility. Quality is not produced by a quality-assurance or testing organization. In an iterative process, testing provides feedback on quality measures to the development organization in a timely manner, enabling it to improve the quality of the system.
- The test workflow provides a project feedback mechanism, enabling quality to be measured and defects to be identified and resolved before they become unworkable. Testing activities begin early in the project, with test planning and some assessment occurring even in the inception phase, and continue throughout the project.

Chapter 13. The Configuration and Change Management Discipline

This chapter introduces key concepts in software configuration management and then describes the roles, activities, and artifacts involved in the configuration and change management discipline.

Purpose

The purpose of the configuration and change management discipline is to track and maintain the integrity of evolving project assets. During the development lifecycle many valuable artifacts are created. Development of these artifacts is labor-intensive, and they represent a significant investment. As such, they are important assets that must be safeguarded and available for reuse. These artifacts evolve and, especially in iterative development, are updated again and again. Although one person is usually responsible for an artifact, we cannot rely on the individual's memory (or upper-left desk drawer) to keep track of these project assets. The project team members must be able to identify and locate artifacts, to select the appropriate version of an artifact, to look at its history to understand its current state and the reasons it has changed, and to ascertain who is currently responsible for it.

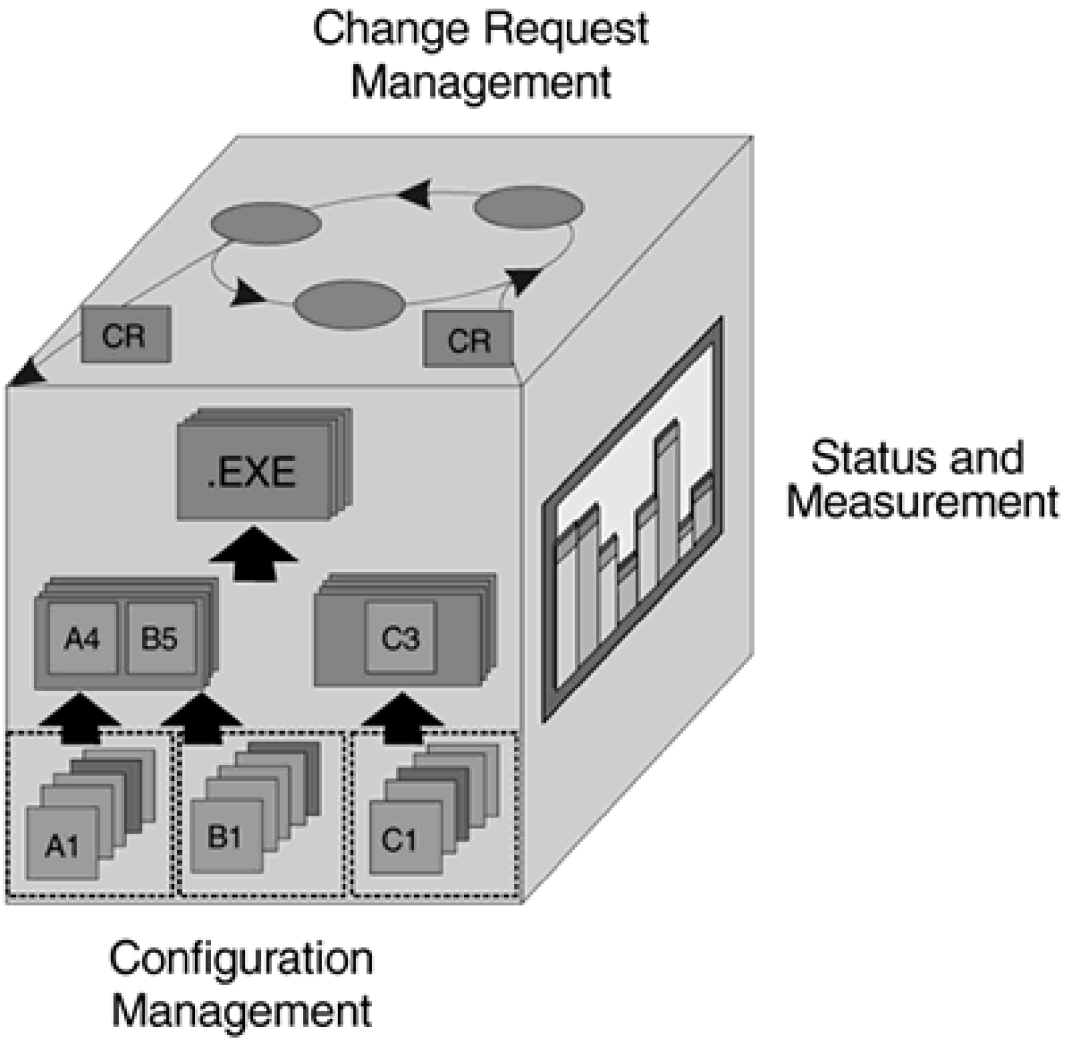
At the same time, the project team must track the evolution of the product, capture and manage requests for changes no matter where they come from, and then implement the changes in a consistent fashion across sets of artifacts.

Finally, to support the project management discipline, we must provide status information on the key project artifacts and gather measures related to their changes.

The CCM Cube

Configuration and change management (CCM) covers three interdependent functions. These key aspects can be illustrated using the "CCM cube" shown in Figure 13-1.

Figure 13-1. The CCM cube



The cube has three faces, and each face examines a different aspect of the problem. All three aspects are deeply intertwined:

- *The configuration management face is related to the product structure.*

  Configuration management (CM) deals with artifact identification, versions, and dependencies among artifacts, as well as the identification of configurations that are consistent sets of interrelated artifacts. It also deals with the issue of providing workspaces to individuals and teams so that they can develop without constantly stepping on one another's feet.

- *The change request management face is related to the process structure.*

  Change request management (CRM) deals with the capture and management of requested changes generated by internal and external stakeholders. It also deals with the analysis of the potential impact of the change and with the tracking of what happens with the change until it is completed.

- *The status and measurement face is related to the project control structure.*

  Status and measurement deals with the extraction of information for project management from the tools that support the configuration management and the change request management functions. The following information is useful for an assessment:

  - The product's status, progress, trends, and quality

  - What has been done and what remains to be done

  - The expenditures

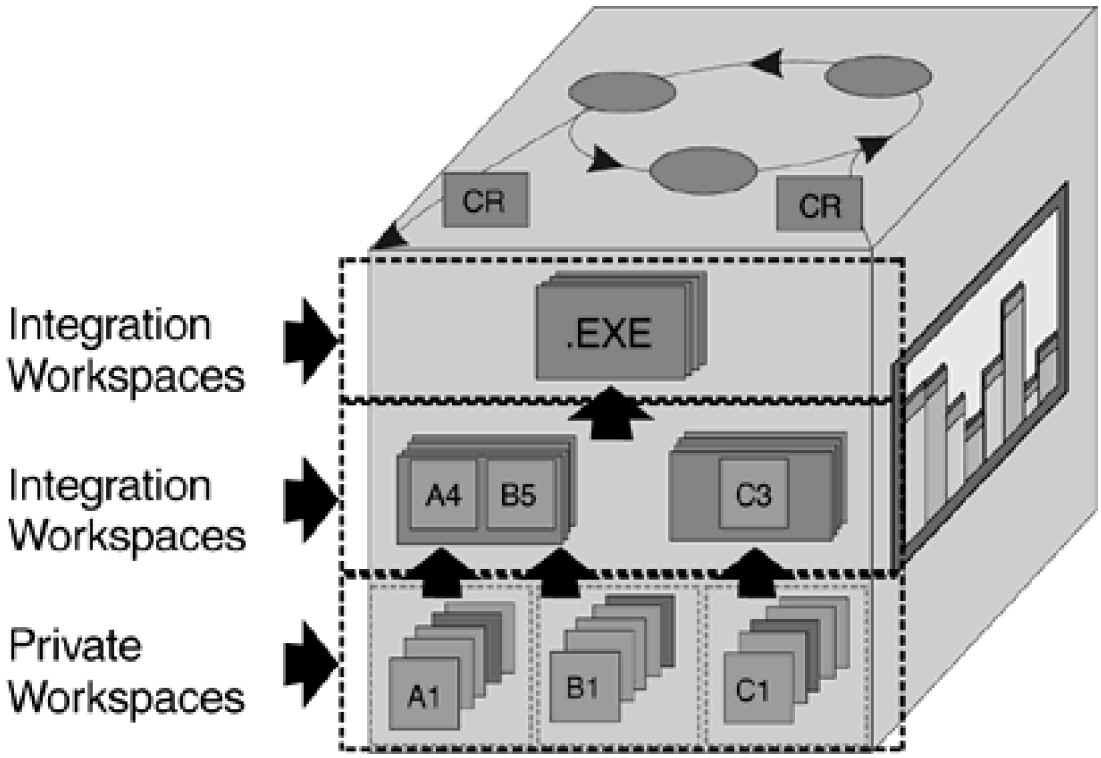
  - The problem areas that require attention

Now let's examine each of these three aspects in detail.

Configuration Management

The configuration management aspect of this discipline deals with the product structure, as shown in Figure 13-2. Important artifacts are placed under version control. As an artifact evolves, multiple versions exist, and you must identify the artifact, its versions, and its change history.

Figure 13-2. Product structure and workspaces



Artifacts depend on one another. As a result, each change causes a ripple effect. When an artifact has been modified, dependent artifacts should be revisited and possibly modified or regenerated. For example, a C++ source file depends on the header files it includes. It also depends on the class specification it implements.

The latest version of an artifact often is the best one, but there are many circumstances in which things are not that simple: Multiple developers may work in parallel on the same artifact, or variants of the same artifact may be required for different end products.

The knowledge of the dependencies among artifacts, the transformations that occur along the dependencies, and perhaps the tool used to effect these transformations can be exploited to re-create stacks of dependent artifacts. For example, a makefile can be created to trigger the recompilation and linkage of an entire application based on a few modified source files.

Build management is about making sure that components that would constitute a build are available and are assembled in the right order based on their dependency hierarchies. Effective build management requires that we can trust that constituent components have been developed and tested adequately to be included in a build. This is akin to the notion of a pedigree: An artifact is OK only if its ancestors are OK.

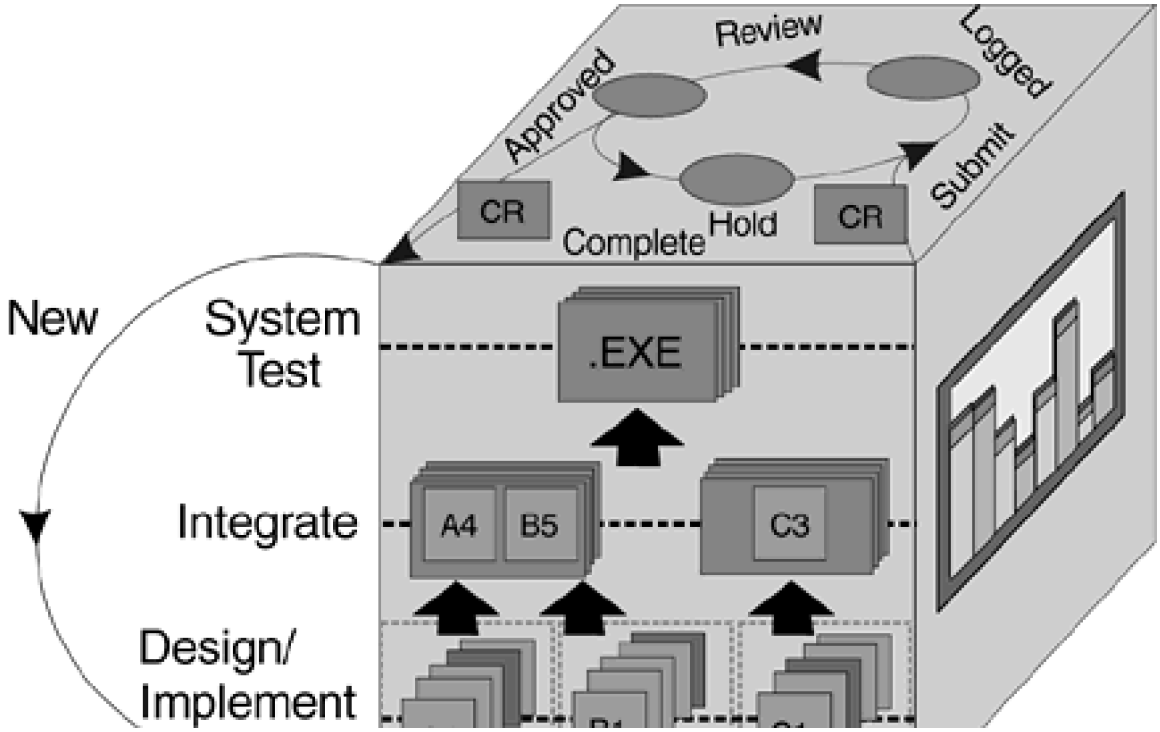
A workspace provides a "sandbox" in which individual developers or small teams work as if they were isolated. It provides access to a necessary set of artifacts. Workspaces, playing the role of staging areas, can be used for basic development or for the progressive integration of products.

The overall product structure—the organization of all artifacts—is driven by the implementation view of the architecture.

Change Request Management

Change request management deals with the process structure and is shown at the top of the cube in Figure 13-3. A change request is a documented proposal of a change to one or more artifacts. Change requests can be raised for a variety of reasons: to fix a defect, to enhance product quality (such as usability or performance), to add a requirement, or simply to document the delta between one iteration and the next.

Figure 13-3. Change requests



Change requests have a life represented as a simple state machine, with states such as new, logged, approved, assigned, and complete. As the change requests go from state to state, information about the change is added, such as the reason for the change, the motivation, the affected artifacts (artifacts that must be modified simultaneously), and the impact on the design, the architecture, the cost, and the schedule. Not all change requests are acted on. A management decision must take into account the result of the impact analysis and relative priorities to determine which changes will be implemented, when they will be implemented, and in which release they will be implemented.

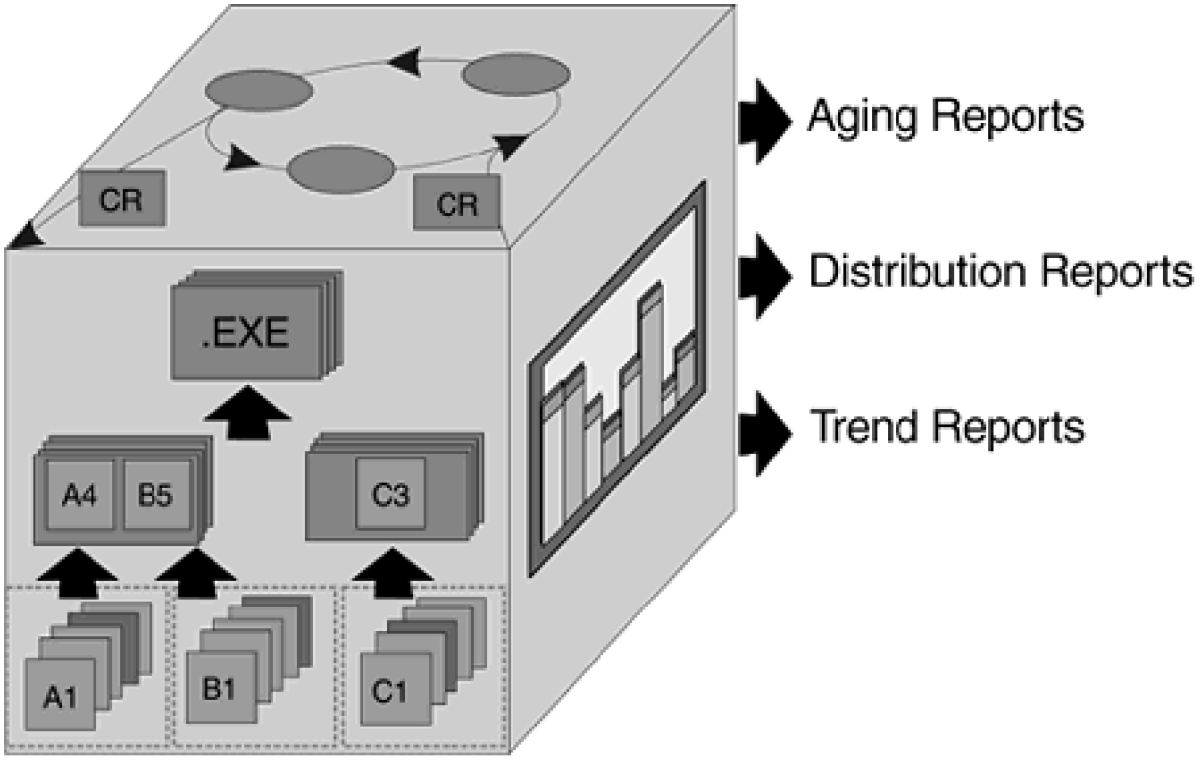
To perform a given change, the exact workflow is driven by the type of artifacts affected and their dependencies. If a change affects the design of a class, the workflow will include the Activity: Design Class and the Role: Designer. The workflow may also include any activity and role affected downstream, such as Activity: Review Class, Activity: Database Design, Activity: Implement Class, and so on. In a sense, change requests drive the actual process.

Changes to artifacts are tracked and associated with the corresponding change request. Change requests are closed when the change has been completed, tested, and included in a release.

Status and Measurement

The status and measurement aspect deals with the project control structure and is shown on the right face of the cube in Figure 13-4. Change requests have a state, as well as other attributes such as root cause, nature (such as defect or enhancement), severity, priority, and area affected (such as layer or subsystem). The various states of a change request provide useful tags as the basis for measurement reporting.

Figure 13-4. Status and measurement



Many change requests are accumulated in a change request database, and they represent the bulk of the work to be done during an iteration, especially in the construction and transition phases.

From this database, we can extract *status* information on the progress of the project:

- Overall progress of the project relative to the changes
- Number of changes made in the various states
- Age of the change requests—that is, how long they have been in a particular state

We can also extract reports on the *distribution* according to the following:

- By severity or priority
- By layer or subsystem affected
- By team
- By root cause

Trend reports look at the derivative of these measurements over time, a measure that is useful for projection.

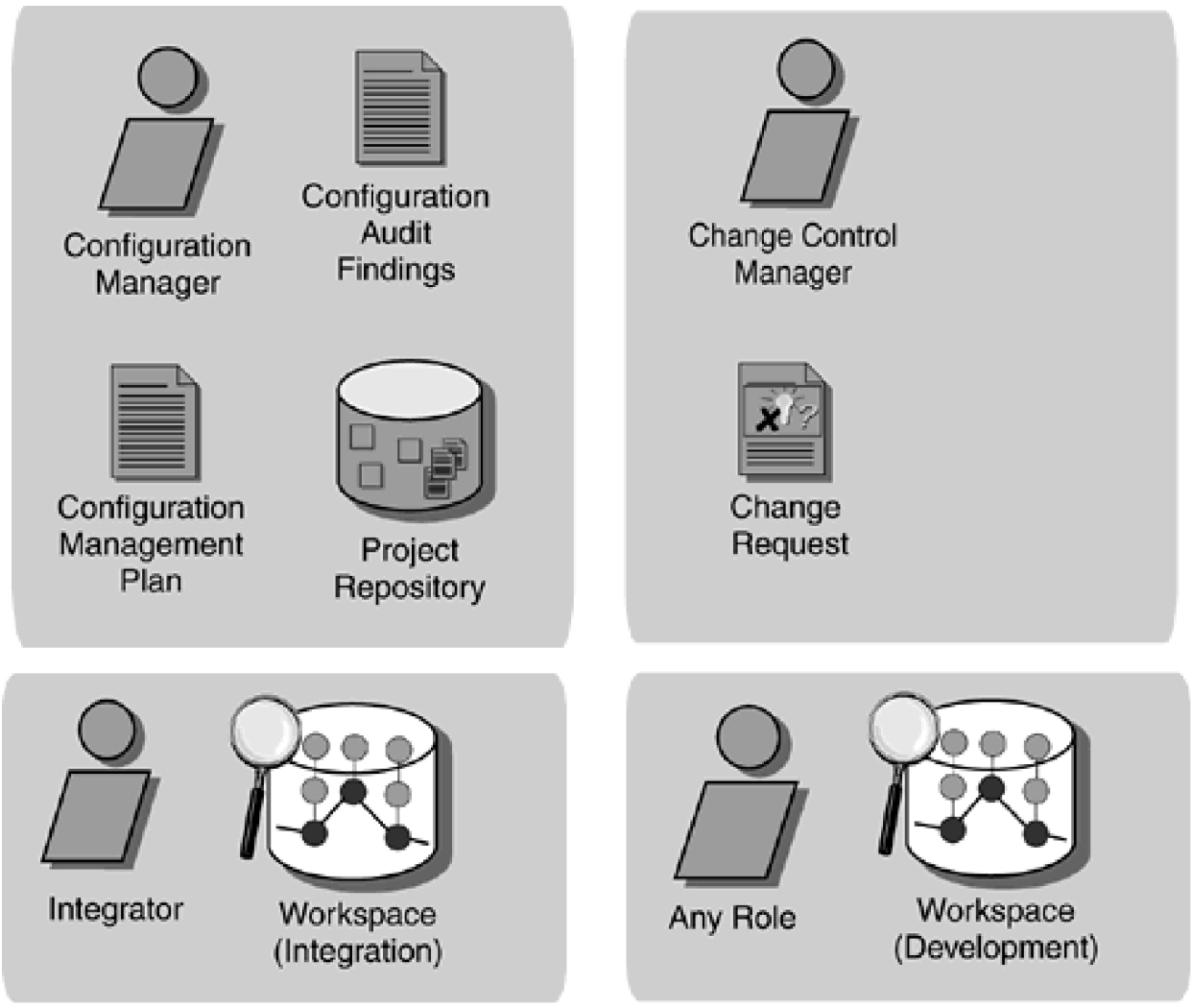
PREVIOUS NEXT

Roles and Artifacts

In the Rational Unified Process, how is all this translated concretely into terms of roles, artifacts, activities, and discipline? Figure 13-5 shows roles and artifacts in the configuration and change management discipline. The main roles involved in configuration and change management are as follows:

- The *Configuration Manager* is responsible for setting up the product structure in the CM system, for defining and allocating workspaces for developers, and for integration. The Configuration Manager also extracts the appropriate status and measurement reports for the Project Manager.
- The *Change Control Manager* oversees the change control process. This role is usually played by a Configuration, or Change, Control Board (CCB), which should consist of representatives from all interested parties, including customers, developers, and users. In a small project, a single person, such as the project manager or software architect, may play this role. The Change Control Manager is also responsible for defining the Change Request Management process, which should be documented in the CM plan.

Figure 13-5. Roles and artifacts in the configuration and change management discipline



The following roles are also involved in this discipline:

- The *Software Architect* provides input to the product structure by means of the implementation view.
- *Implementers* access adequate workspace and the artifacts they need to implement the changes for which they are responsible.
- *Integrators* accept changes in the integration workspace they manage and build the product.
- *Any role* can submit a change request and uses workspaces.

The Change Control Board is a group of various technical and managerial stakeholders, such as the Project Manager, the Architect, the Configuration Manager, and any stakeholder (customer representative, marketing personnel, and so on). The role of the CCB is to assess the impact of changes, determine priorities, and approve changes.

The key artifacts of configuration and change management are as follows:

- *The configuration management plan*

  The CM plan describes the policies and practices to be used on the project for CM: versions, variants, workspaces, and procedures for change management, builds, and releases. The CM plan defines the rules and responsibilities for the CCB. It is a part of the software development plan.

- *Change requests*

  Change requests may be of a wide variety: They may document defects, changes to requirements, or the delta from one iteration to the next. Each change request is associated with an originator and a root cause. Later, impact analysis attaches the impact of the change in terms of affected artifacts, cost, and schedule. As the change request evolves, its state changes; history items, in particular the CCB decisions, are attached to it.

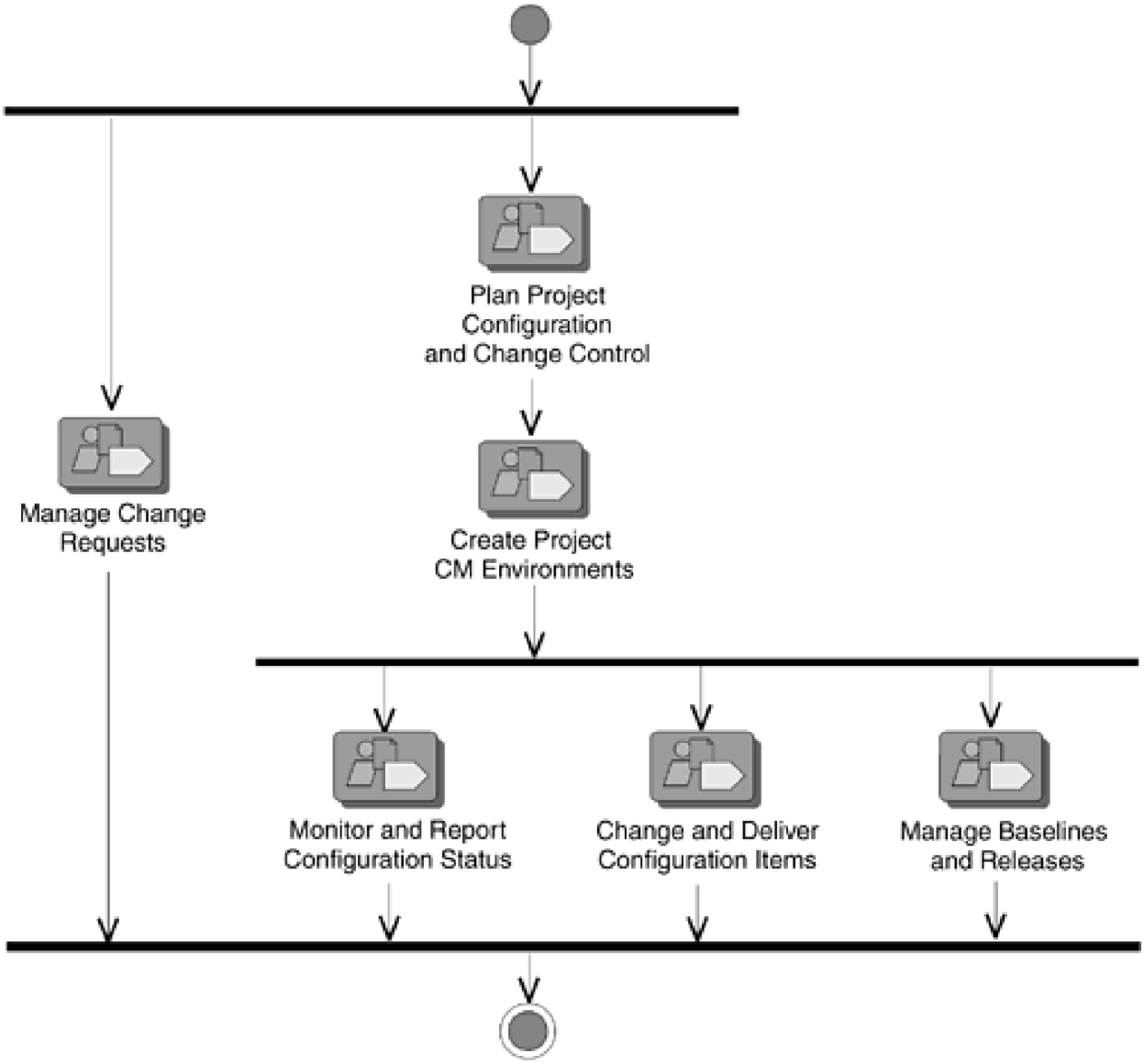
In addition, this discipline includes the following:

- The *implementation model* drives the product used by the configuration manager to set up the CM environment.
- The *Configuration Audit Findings* identify a baseline, any missing required artifacts, and incompletely tested or failed requirements.

Workflow

There are two interwoven workflows in configuration and change management: one from the perspective of the CM product structure, shown in Figure 13-6, and a second one from the perspective of the life of a change request.

Figure 13-6. The configuration and change management workflow



Plan Project Configuration and Change Control

The configuration management plan describes all CM-related activities that need to be resourced and performed during the course of a project. It describes the procedures and policies applied to identify, safeguard, and report on all artifacts that are generated during the project lifecycle. Project naming conventions are important because they facilitate communication and enable easy identification of configuration items for updating or reuse. Project artifacts are the tangible assets that need to be safeguarded. Safeguarding is achieved through the enforcement of archiving, baselining, and access control privileges.

The CM plan also describes the project change control process. The purpose of a change control process is to ensure that all changes planned or made to a project artifact are done by informing the appropriate stakeholders. Managers and co-workers need to know about the exact nature of the change and understand the impact on cost and schedule.

Create a Project CM Environment

The CM environment should facilitate product development. All project members should be able to get, and work on, the correct development artifacts as they need them.

The purpose of this activity is to create a work environment where all developmental artifacts are available and where the product can be developed, integrated, and archived for subsequent maintenance and reuse. The CM environment provides developers and integrators private and shared workspaces where they can build and integrate software.

Change and Deliver Configuration Items

This workflow detail refers to the way any role can create a workspace. Within a workspace a role can access project artifacts, make changes to those artifacts, and deliver the changes for inclusion in the overall product. Delivery of changes is made into an integration workspace that can include submissions from multiple roles. The idea is to integrate the individual contributions and make those visible to developers for the next round of development.

From the integration workspace, the Integrator builds the product, creates baselines, and then makes the baselines available to the rest of the development team.

Manage Baselines and Releases

A baseline is a description of all the versions of artifacts that make up the product at any given time. Typically, baselines are created at ends of iterations and at project and delivery milestones. A product should be baselined each time it is released to a customer. When a customer has a problem with a given release, the development or maintenance team can revert to a release baseline to re-create and fix the particular defect.

Monitor and Report Configuration Status

The CM environment provides the context for all software development work. All artifacts need to be put under configuration control, and the CM manager is responsible for creating the workspaces where developers and integrators perform their work.

As the custodian of the project repositories, the CM manager has to make sure that they are safe and have all the necessary artifacts. The CM manager is also responsible for reporting on the state of the configuration. Configuration reporting can provide

good input to project management for tracking overall project progress and trends. For example, reporting on defects and their closure rates not only can help determine where the problems lie, but also can provide information that can be factored into determining the impact on project cost and schedules.

Manage Change Requests

The purpose of a standard, documented change control process is to ensure that changes in a project are made in a consistent manner and the appropriate stakeholders are informed of the state of the product, changes to it, and impact of these changes on the cost and schedule.

Tool Support

Managing all this change is hard. The complexity of doing it right, day after day, grows exponentially with the size of the team, the size of the product (that is, the number of artifacts), the geographic distribution of the team, and the schedule pressure. It is error-prone, tedious, and labor-intensive and is therefore an obvious task that can benefit from tool support and automation.

The Rational Suite of software development tools supports the configuration and change management workflow with the following tools:

- Rational ClearCase assists with the configuration management portion.
- Rational ClearQuest assists with the change request management and status and measurement portions.

The Rational Unified Process provides tool mentors for both tools.

Unified Change Management

Unified Change Management (UCM) is Rational Software's approach to managing change in software system development, from requirements to release. UCM spans the development lifecycle, defining how to manage change to requirements, design models, documentation, components, test cases, and source code.

One of the key aspects of the UCM model is that it unifies the activities used to plan and track project progress and the artifacts undergoing change. The UCM model is realized by both process and tools. The Rational products ClearCase and ClearQuest are the foundation technologies for UCM. ClearCase manages all the artifacts produced by a software project, including both system artifacts and project management artifacts. ClearQuest manages the project's tasks, defects, and requests for enhancements (referred to generically as activities) and provides the charting and reporting tools necessary to track project progress.

Unified Change Management is described in the book by our colleagues Brian White and Geoffrey Clemm, *Software Configuration Management Strategies and Rational Clearcase: A Practical Introduction* (Boston: Addison-Wesley, 2000).

Summary

- The purpose of the configuration and change management discipline is to maintain the integrity of the project artifacts as they evolve in response to change requests.
- Configuration management deals with the product structure, identification of elements, version, valid configuration of elements, and workspaces.
- Change request management encompasses the process of modifying artifacts in a consistent fashion.
- Status and measures can be extracted from configuration and change management information to assist in status assessment.
- Tools such as Rational ClearCase and Rational ClearQuest automate the most tedious aspects of this discipline.

Chapter 14. The Environment Discipline

This chapter explains the kinds of artifacts that are produced and the activities that take place in the environment discipline. It also introduces the Process Engineering Process (PEP), a separate component of the Rational Unified Process that focuses on the improvement of software process rather than the software product.

Purpose

The purpose of the environment discipline is to support the development organization with both processes and tools. This support includes the following:

- Tool selection and acquisition
- Setup of tools and configuration of tools to suit the organization
- Process configuration
- Process improvement
- Technical services to support the process: the information technology (IT) infrastructure, account administration, backup, and so on

Some of the activities related to process implementation and configuration are described in Chapter 17, Implementing the Rational Unified Process.

The activities related to the definition, configuration and implementation of the software development process constitute effectively a process distinct from the software development process—the Process Engineering Process (PEP). It is an optional component that may not be installed or accessible to all software practitioners.

Configuring the Rational Unified Process

The RUP process framework contains a vast amount of guidance, artifacts, and roles. Because no project can use all of these artifacts, you need to specify a subset of the RUP to use for your project. This is done by selecting or producing a *RUP Process Configuration,* which constitutes a complete process from the perspective of a particular project's requirements. You can use one of the ready-made configurations "as is," use a ready-made configuration as a starting point for yours, or create a process configuration from scratch.

- A *RUP Process Component* is a coherent, quasi-independent "chunk" or module of process knowledge that can be named, packaged, exchanged, and assembled with other process components.
- A *RUP Library* is a collection of Process Components out of which a set of RUP Process Configurations may be compiled with RUP Builder. New process components can be added to a RUP Library through the means of RUP Plug-Ins.
- A *RUP Base* is a collection of RUP Process Components meant to be extended by applying plug-ins to generate RUP Process Configurations. It resides in a RUP Library.
- A *RUP Plug-In* is a deployable unit for one or several Process Components that can be readily "dropped" onto a RUP Base to extend it. A RUP Plug-In is encapsulated into a single physical file, allowing it to be moved around and added to a RUP Library with a compatible RUP Base.

To use a programming language analogy, a RUP Plug-In is a "precompiled" RUP Process Component, ready to be "linked" into a RUP Base to create a RUP Configuration.

There are two main RUP Bases:

- One for small, informal projects

- One for larger or more formal projects

And there is a large collection of RUP Plug-Ins available, addressing issues related to domain (real-time, MIS), to technologies and languages (Java, J2EE, NET, and so on). Some plug-ins are developed by IBM/Rational and are delivered with the product, some are available from the IBM Web site, and some are developed by partner companies.

A RUP Configuration also includes *Process Views*, which are role-based or personalized access to a RUP configuration.

Instantiating the Rational Unified Process

Each project team needs to decide how it is going to use its RUP Configuration, which artifacts will be used and how, which tools will be used and how, which roles will be required and who will play them. This is captured in a *Development Case*, which is the main artifact produced in the Environment discipline.

You can also create a *Project Web Site* that acts like a portal to all the actual artifacts used in the project, including the Development Case and the RUP Configuration.

Customizing the Rational Unified Process

Some organizations may need to create RUP Configurations that are different from what can simply be assembled out of RUP Bases and RUP Plug-Ins. They want to include their own process guidance, adding to the RUP, or modifying or extending existing elements of the RUP. They also want to continuously improve their process and capitalize on the lessons learned at each iteration, phase, or project by extending their configurations. They can achieve this by developing their own RUP Plug-Ins.

- A *Thin Plug-In* does not change the structure of the RUP. It merely substitutes new content to existing process elements or adds some guidance such as guidelines and tool mentors. Such a Plug-In is usually sufficient for capturing company-specific templates, adding steps to activities, or modifying an artifact outline.
- A *Structural Plug-In* does change process elements and their relationship. It is used to add process elements such as roles, artifacts, workflow, and whole disciplines.

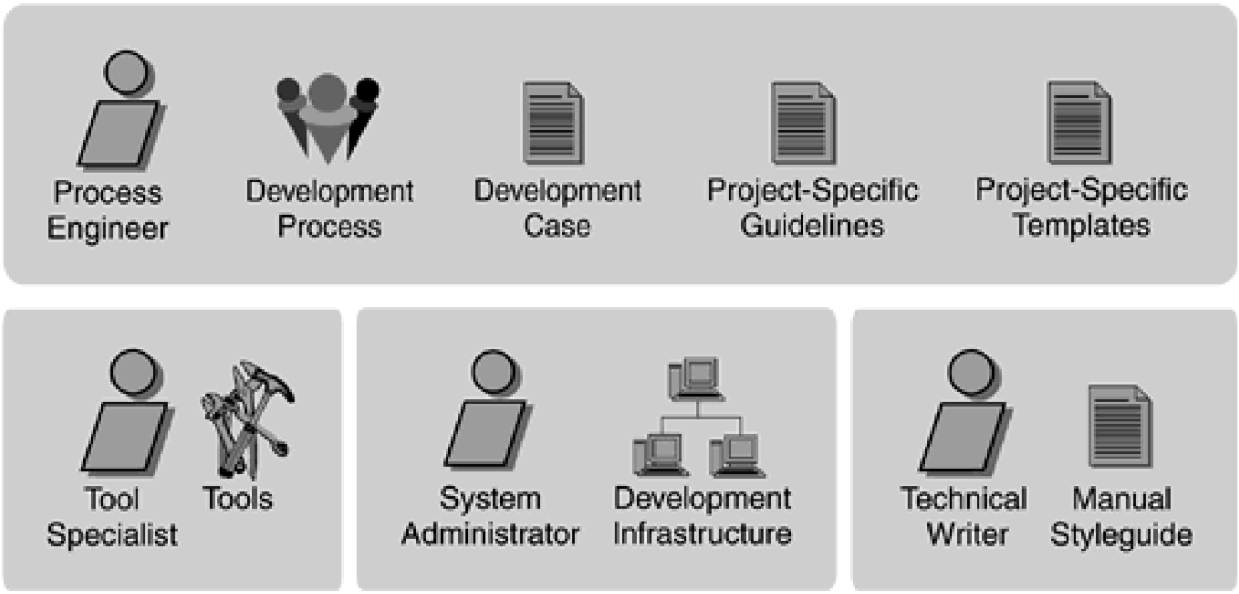
Implementing the Rational Unified Process

The most challenging task of the Process Engineer is undeniably to deploy or "roll out" the Rational Unified Process in an organization. This is often a project in itself, which includes configuring and customizing the RUP, and also involves assessing the current organization, setting up goals, and organizing training and mentoring. See Chapter 17, Implementing the Rational Unified Process.

Roles and Artifacts

Figure 14-1 summarizes the roles and artifacts involved in the Environment discipline. The main role in the process is the Process Engineer, who is responsible for the software development process itself. This includes configuring the process before project start-up and continuously improving the process during the development effort.

Figure 14-1. Roles and artifacts in the environment discipline



The main artifact is the Development Case, which specifies the tailored process for the individual project. A development case describes, for each process discipline, how the project will apply the process. For each process discipline, you decide which artifacts to use and how to use them. A Development Case should be brief and refer to the Process Configuration for details.

In certain aspects of the process, to establish the *project-specific guidelines* and *templates,* the Process Engineer needs the competence of other roles:

- A *Business Process Analyst,* for the business modeling guidelines
- A *System Analyst,* for the use-case modeling guidelines
- A *User-Interface Designer,* for the user-interface guidelines
- A *Software Architect,* for the design guidelines and programming guidelines
- A *Technical Writer,* for the *user manual styleguide*

The following main roles are involved with the tool environment:

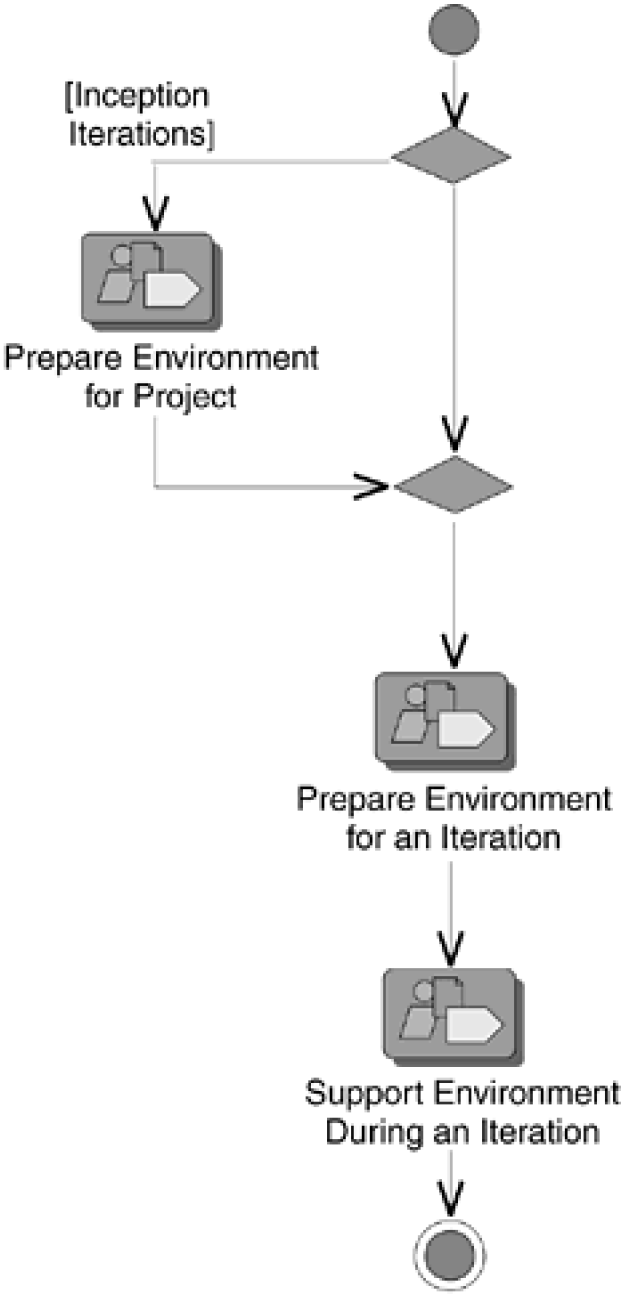
- The *Tool Specialist* selects and acquires tools to support the development. The Tool Specialist sets up tools and configures the tools to suit the project needs. There are usually several individuals acting as Tool Specialist, each responsible for one tool or a group of related tools.

- The *System Administrator* maintains the hardware and software development environment and performs system administrative tasks such as account administration, backups, and so on.

Workflow

The environment discipline is organized into three workflow details, as described in Figure 14-2.

Figure 14-2. An environment workflow



Prepare Environment for Project

The purposes of preparing the environment for a project are:

- Assess the current development organization.
- Assess the current tools support.
- Develop a first draft of the development case.
- Produce a list of candidate tools to use for development
- Produce a list of candidate project-specific templates for key artifacts.

Prepare Environment for an Iteration

The purposes of preparing the environment for an iteration include the following:

- Complete the development case to be ready for the iteration.
- Prepare and if necessary customize tools to use within an iteration.
- Verify that the tools have been configured and installed correctly.
- Produce a set of project-specific templates to use within the iteration.
- Train people to use and understand the tools and the development case.

For every iteration, it is necessary to prepare guidelines for the following activities and artifacts:

- Business modeling
- Use-case modeling
- Design
- Programming
- Manual styleguide
- User interface
- Tests
- Tools

By analyzing problems and defects in the previous iteration, you understand what needs to be done better and can take that into account when preparing guidelines for the next iteration.

Support Environment for an Iteration

The developers need support in their use of tools and process during an iteration.

Tool Support

RUP Builder is the tool used to assemble and publish a RUP Configuration out of a RUP Base and any number of compatible RUP Plug-Ins. RUP Builder also allows some limited selection of certain process elements—artifacts, guidelines, and so on—and it can create predefined Process Views.

For basic process customizations, RUP Organizer allows you to simply drag and drop your own artifacts or process examples into a RUP configuration and publish RUP configurations or Thin Plug-Ins. For more advanced process customizations and the creation of Structural Plug-Ins, RUP Modeler is an add-in to Rational XDE—Rational's most recent UML modeling tool. RUP Modeler relies on the concepts of the SPEM (introduced in Chapter 3) to support an object-oriented model of the process. Together, RUP Modeler and RUP Organizer constitute the Rational Process Workbench.

The Rational Unified Process contains tool mentors for RUP Builder and RUP Organizer. The Process Engineering Process contains tool mentors for the RUP Modeler.

Summary

- The goal of the Environment discipline is to provide adequate support for the development organization in tools, processes, and methods.
- A software development organization usually creates a Configuration of the Rational Unified Process, tailored to its context, its need, and the size and type of project.
- The actual process used by the project is described in a Development Case. A Development Case describes how the project will apply the Rational Unified Process.
- Many activities and steps of the Rational Unified Process can be automated through the use of tools, thereby removing the most tedious, human-intensive, and error-prone aspects of software development.

Chapter 15. The Deployment Discipline

This chapter gives an overview of the type of activities that take place when the software product is deployed in its operational environment and of the additional artifacts that may need to be developed for its successful use.

Purpose

Software developers sometimes declare victory too soon. They forget that it is the customer's willingness to use the finished product, rather than a clean compile, that is the mark of a successful development project.

The purpose of deployment is to turn the finished software product over to its users. The Deployment discipline involves the following types of activity:

- Testing the software in its final operational environment (beta test)
- Packaging the software for delivery
- Distributing the software
- Installing the software
- Training the end users and the sales force (product rollout)
- Migrating existing software or converting databases

The ways these activities are carried out varies widely across the software industry, depending on the size of project, the mode of delivery, and the business context.

Modes of Deployment

The Rational Unified Process can be applied across a wide spectrum of software development projects. To illustrate the kind of deployment issues that can arise for different kinds of software, let's have a look at three specific examples:

1. Deployment of software in custom-built systems
2. Deployment of shrink-wrapped software
3. Deployment of software that is downloadable over the Internet

The key differentiators of these various modes of deployment are the degree of involvement of the development organization in the way the software is packaged and distributed, and how the end user will learn to use it.

Custom-built systems are usually "one of a kind," and sometimes even have associated custom-built hardware. Typical customers for custom-built software come from the transportation and telecommunication industries, embedded systems, and large organizations such as banks, trading houses, and shipping companies. In the aerospace industry, software is almost invariably required to run on specifically built target hardware. This may also be the case in the banking world where the software must accommodate the characteristics of particular ATMs or other interface devices.

Included in the category of custom-built systems would be large-scale enterprise software systems that, although they come from a standard source, need to be tailored for each situation. This would also be true for distributed systems in which all nodes need to be brought up to date and activated in predetermined order.

There is little difference in the developer's involvement in the installation of shrink-wrapped software and software that is

downloadable from the Internet. In each of these cases, the installer will be the end user, who with the aid of installation wizards should have little or no difficulty installing and running the product software. The difference lies in the delivery mechanism and in the developer's involvement in either setting up the product Web site or creating the packaging and distributing the product.

Timing of Deployment

Deployment activities peak in the *transition phase* and represent the culmination of the software development effort. Successful deployment and indeed the overall success of the development effort are defined by the customer's willingness to use the new software.

The purpose of the transition phase, as well as the goal of deployment activities, is to ensure a smooth transition of a self-supporting user to the new software. The Rational Unified Process advocates ongoing customer engagement and includes end-user beta testing of earlier releases as part of the "ramp up" to the final delivery of the product.

During the transition phase the end user can start using the product to get a feel for how it works in the real runtime environment. Trial installations or beta tests, over a series of deployment iterations, provide an opportunity for final suggestions for adjustments to the product.

With good planning and customer involvement, by the time the final product is delivered there should be no surprises—for either the developer or the user!

Deployment planning, as with other project planning activities, can start early in the project lifecycle to account for the overall deployment and customer preparation strategy and resources needed to deliver the tested product and end-user support material. Work on deployment artifacts, such as user and training manuals, can begin in earnest following the lifecycle architecture milestone, at the end of the elaboration phase, once the architecture and requirements have been stabilized.

Roles and Artifacts

The following roles typically are involved in the deployment discipline:

- The Deployment Manager plans and organizes deployment. She is responsible for the beta test feedback program and ensuring the product is packaged appropriately for shipment.
- The Project Manager is the primary customer interface and is responsible for approving deployment based on feedback and test result evaluations and for the customer's acceptance of the delivery.
- The Technical Writer plans and produces end-user support material and the Course Developer plans and produces training material.
- The Graphic Artist is responsible for all product-related artwork.
- The Tester runs the acceptance tests and is responsible for ensuring that the product has been tested adequately.
- The Implementer creates installation scripts and related artifacts that will help the end user install the product.

Deployment Artifacts

Deployment as described in the Rational Unified Process ranges from the deployment of customized systems to the situation where the end user can download software from a specified Web site. Given this vast range, whether or not the artifacts listed below will be required depends on the deployment mode.

The key artifact is a release, packaged in one or more Deployment Units, which may consist of one or more of the following artifacts:

- The executable software, in all cases
- Installation artifacts: script, tools, files, guides, licensing information
- Release Notes, describing the release for the end user
- Support Material, such as user manual, operations and maintenance manuals
- Training Materials

In the case of the shrink-wrapped product, additional artifacts will be required for creating the "product," such as:

- Bill of Materials (the complete list of items to be included in the manufactured product)
- Product Artwork (part of the packaging to help with product branding and identification)

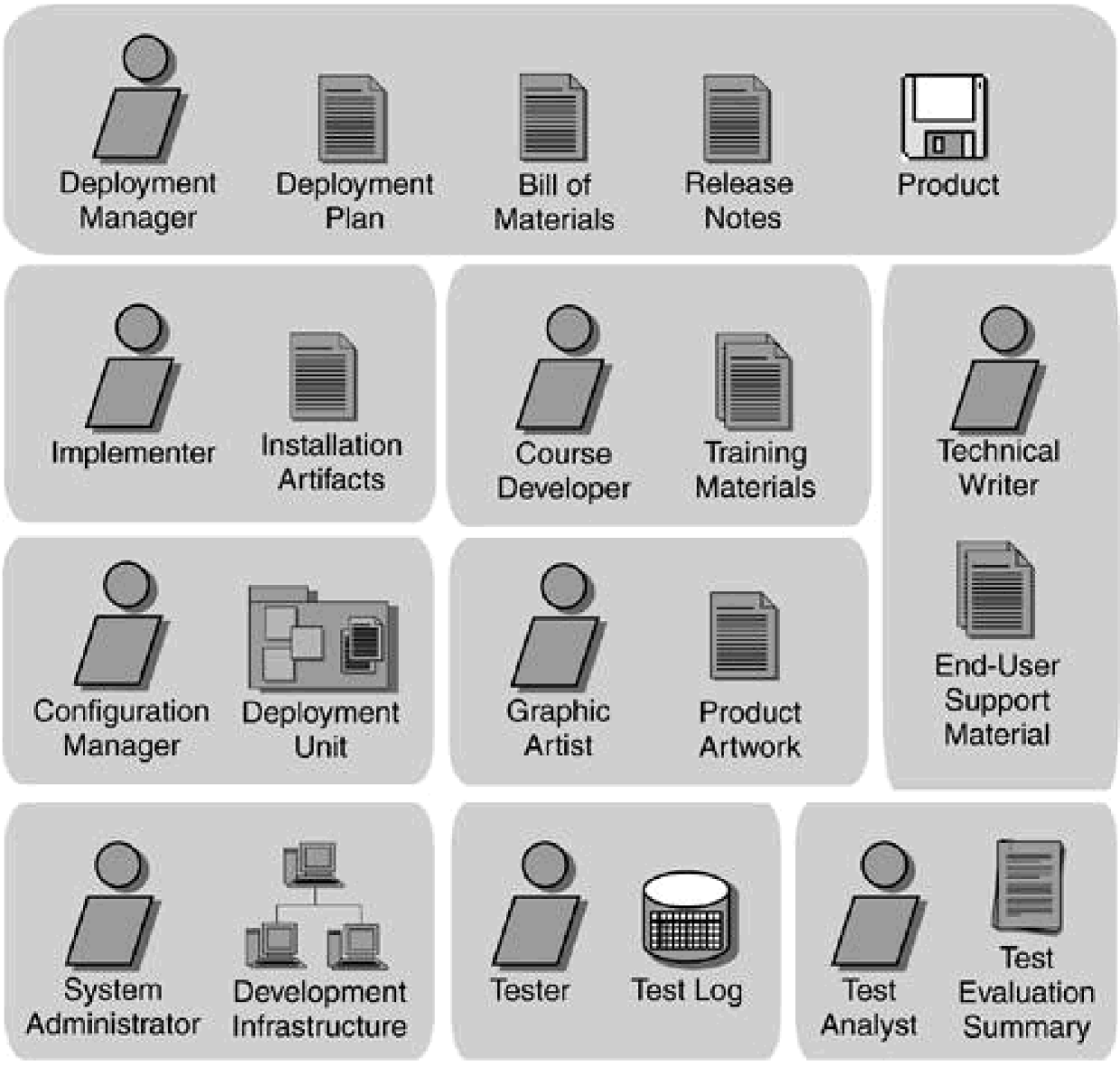
Other deployment artifacts used in the development of the product, but not necessarily released to the customer, include

- Test Results (from development site and on-site tests)

- Feedback Results (from beta testing)
- Test Evaluation Summary

Figure 15-1 shows roles and key artifacts in the Deployment discipline.

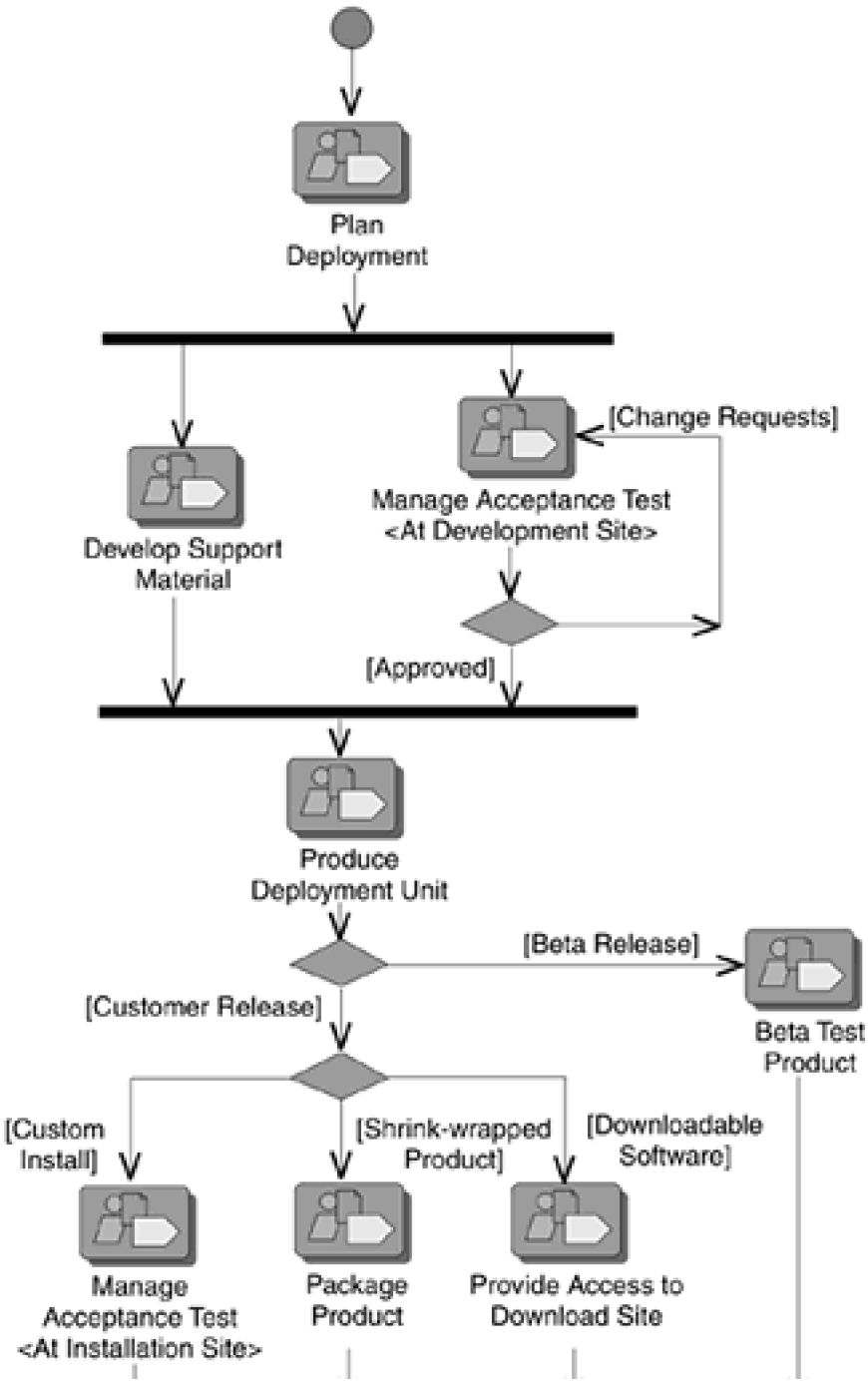
Figure 15-1. Roles and key artifacts in the deployment discipline

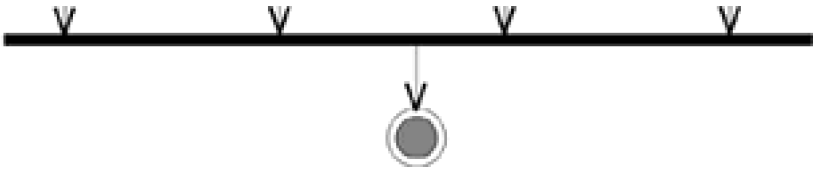


Workflow

The activity diagram in Figure 15-2 identifies the major kinds of activities that are necessary to deploy software effectively to the end user.

Figure 15-2. The deployment workflow





Plan Deployment

Given that successful deployment is defined by the customer's willingness to use the new software, deployment planning not only must take into account how and when the complete list of deliverables will be developed, but also must ensure that the end user has all the necessary information to take delivery of the new software. To ensure a smooth transition, deployment plans include the beta test program for iteratively assessing earlier versions of the software under construction.

Overall system deployment planning requires a high degree of customer collaboration and preparation. A successful conclusion to a software project can be impacted severely by factors outside the scope of software development such as the building, hardware infrastructure not being in place, and users who are ill-prepared for cut-over to the new system.

Develop Support Material

Support material covers the full range of information that will be required by the end user to install, operate, use, and maintain the delivered system. It also includes training material for all the various positions that will be required to use the new system effectively.

Product Deployment Unit

The purpose is to create a deployment unit that consists of the software and the accompanying artifacts required to install and use it successfully. The deployment unit can be created for the purposes of *beta testing,* a test deployment to the final users or, depending on its level of maturity, for the final product.

Manage Acceptance Test at Development Site

Testing at the development site helps determine whether the product is sufficiently mature for release either as the final deliverable or for distribution to beta testers.

Beta testing a packaged product is done by polling a wide range of end users for feedback. In the case of a custom-installed system, a beta test could be a pilot installation at the target site.

To reinforce the idea that testing activities and platforms at the development site should closely approximate the intended target site, Test Product activities are identical for testing at the development site and the target site, with one exception. The exception is that the testing at the development site may use test harnesses or simulators. Testing at the target site will use the actual target hardware.

Beta Test Product

Beta testing requires the delivered software to be installed by the end user, who provides feedback on its performance and usability.

In the context of iterative development, beta testing is essential to ensuring that customer expectations are met and that user feedback is factored into the next development iteration.

Manage Acceptance Test at Installation Site

Following sufficient internal and beta testing, the product needs to be installed on-site and tested by the customer. Based on the preceding iteration tests and customer engagements, the final site tests should come as no surprise, but should be only a formality for the customer to "accept" the system!

Package Product

This set of optional activities describes what needs to occur for producing "packaged software" products. In this case, the release is saved as a master for mass production and then packaged in product boxes with the Bill of Materials for shipment to the customer.

Provide Access to Download Site

The appeal of Web commerce and the Internet as a software distribution channel is obvious. The product is entirely accessible through the software environment of browsers and Web sites. The challenge for the provider is to make sure the product is available at all times (24/7) to a global marketplace even through varying loads that could choke the host hardware and communication bandwidths.

Setting up the hardware infrastructure to host the corporate Web presence is beyond the scope of a software development process. However, the Deployment Manager needs to know how to add the product to the list of products offered through the Web and communicate that the product is available for purchase or delivery on demand.

The Web is a good mechanism for enabling feedback from users, announcing upgrades, or otherwise providing access, perhaps through passwords, to other corporate services.

Summary

- The Deployment discipline takes care of all artifacts delivered to the end users or customers as well as to the supporting organizations: support, marketing, distribution, and sales.
- The Deployment discipline describes the activities associated with beta testing and delivering installable software. Installation can be done by the vendor (in the case of complex, distributed systems) or by the user (in the case of software delivered as a packaged product or over the Internet).
- This discipline is highly dependent both on the kind of product being developed and on the business context, and it must be specialized by the organization that is adopting the Rational Unified Process.

Chapter 16. Typical Iteration Plans

This chapter offers an overview of three typical iteration plans, one for each of the first three phases of the process.

The disciplines presented in sequence from Chapter 8 through Chapter 15 may wrongly give an impression of a waterfall process. Remember that the disciplines are merely logical groupings of related activities; the activities in the disciplines are revisited again and again at each iteration and partially re-executed. The actual work performed on a project varies depending on the nature of the project and where you are in the lifecycle.

In Chapter 3 we introduced the concept of the typical iteration plan, which describes the work that is performed as the project goes through one iteration. In this chapter, we give three examples of typical iteration plans:

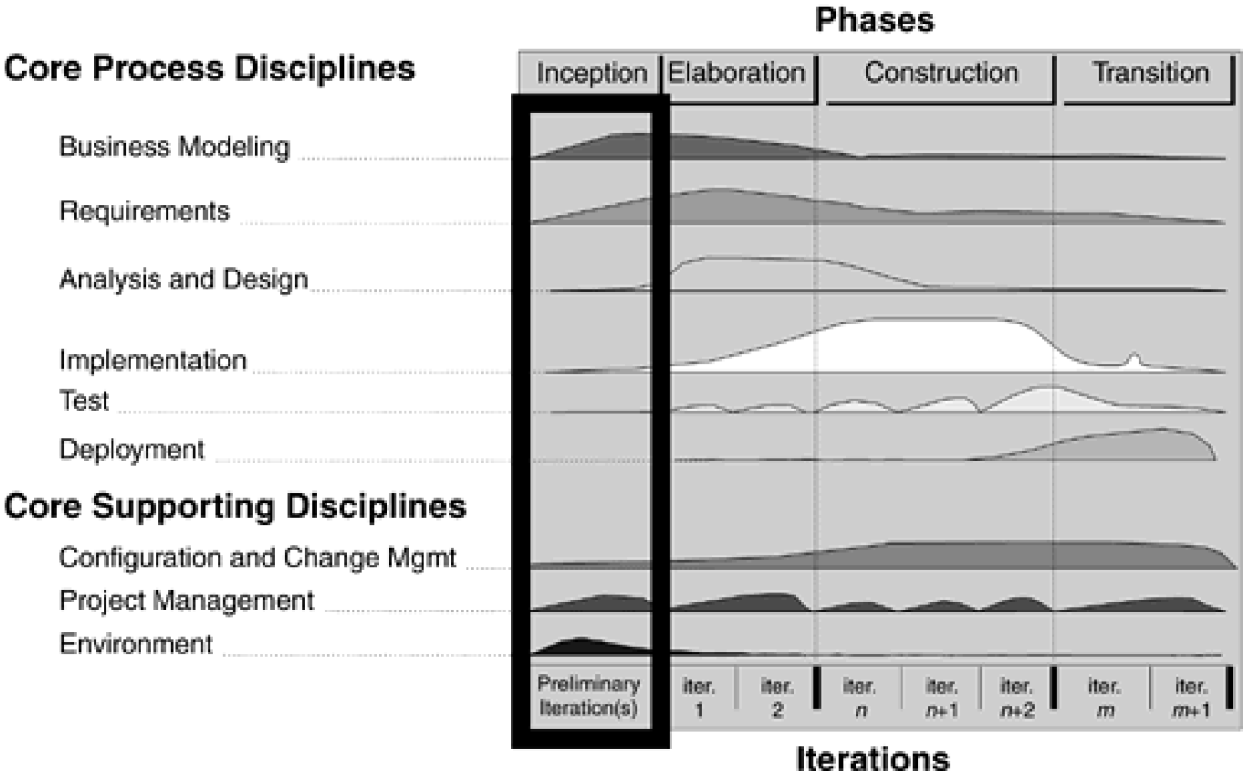
- An iteration in the inception phase to define the project vision and the business case
- An iteration early in the elaboration phase to build an architectural prototype
- An iteration late in the construction phase to implement the system

These examples are skeletal and do not involve all activities from all disciplines and workflows.

Defining the Product Vision and the Business Case

The context of this first example is the initial development cycle for a new product (see Figure 16-1). It illustrates a "greenfield" development for a small system.

Figure 16-1. An iteration in the inception phase



Start-up: Define the system's vision and scope.	The Stakeholders of the system to be developed (such as customers, managers, marketing, and funding authorities), working with System Analysts, define the vision and the scope of the project (Artifact: Vision). The driving factor to consider in this effort is the user's needs and expectations (Artifact: Stakeholders' Requests). Also considered are constraints on the project, such as platforms to be supported, and external interfaces. Based on the early sketches of the vision, the group starts to prepare the business case and initiate a risk list, enumerating hazards that could arise on the way to success (Artifacts: Business Case and Risk List).
Requirements: Outline and clarify the system's functionality.	The system analysts use various techniques, such as storyboarding and brainstorming, to conduct sessions to collect stakeholders' opinions about what the system should do. Together, they sketch an initial outline of the system's use-case model (Artifact: Use-Case Model). An initial glossary is also captured to simplify the maintenance of the use-case model and to keep it consistent (Artifact: Glossary). The main results of these sessions are a stakeholder needs document and possibly an outline of a use-case model: a use-case survey.
Project Management: Consider the project's feasibility	With input from the use-case modeling, it is time to transform the vision into economic terms by considering the project's investment costs, the resource estimates, the development environment, the success criteria (revenue projection and market recognition), and so on, and to flesh out a business

and outline the Project Plan.	case. The risk list is updated, taking into account business risks. The Project Manager builds a phase plan showing tentative dates for the inception, elaboration, construction, and transition phases along with the major milestones. The software development plan envisages the development environment and process that will be needed (Artifact: Project Plan, inside the Software Development Plan).
Project Management: Refine the Project Plan.	At this stage, the stakeholders should have a good understanding of the product's vision and feasibility. An order of priority among features and use cases is established. The Project Manager may start planning the project in more detail, basing the project plan on the prioritized use cases and associated risks. A tentative set of iterations is defined, with objectives for each iteration. The results achieved at this stage are refined in each subsequent phase and iteration and become increasingly accurate as iterations are completed. This is a key differentiator in using RUP—recognizing that initial project plan estimates are rough estimates, but that those estimates become more realistic as the project progresses and there are real metrics on which to base estimates; successive refinement of the project and iterations plans is both expected and essential.

Result

The result of this initial iteration is a first cut at the project's vision, business case, and scope, as well as the project plan. The stakeholders (organization) initiating the project should have a good understanding of the project's return on investment (ROI); that is, what is returned at what investment costs? Given this knowledge, a "go/no go" decision can be made.

Project schedules are not written in stone. A key differentiator in using this process is the recognition that initial project plan estimates are rough estimates that become more realistic as the project progresses and real measures are developed on which to base estimates. With project planning, practice makes perfect. It's important to stress continual refinement of the project plan.

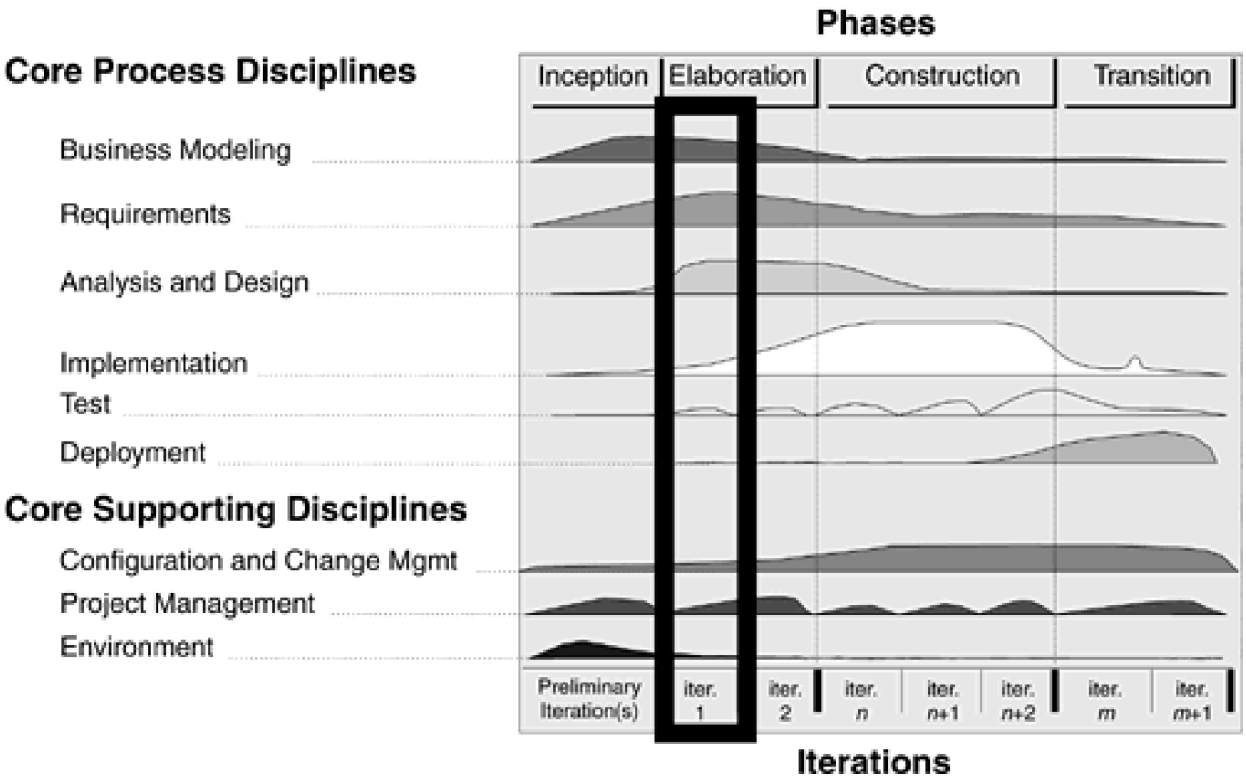
Subsequent Iterations in Inception

You can initiate subsequent iterations to enhance the understanding of the project's scope and the system to be developed. This activity might imply further enhancement of the use-case model or of the plans, staffing, environment, and risks. The need for such additional work depends on variables such as the complexity of the system, the associated risks, and the domain experience.

Building an Architectural Prototype

Figure 16-2 illustrates our second example: an iteration early in the elaboration phase. The context of this example is that the inception phase has been completed and the lifecycle objective milestone has been passed; we have an outline of the actors and use cases as well as an initial cut at a project plan. (See the iteration workflow discussed in the preceding section.)

Figure 16-2. An iteration early in the elaboration phase



Start-up: Outline the Iteration Plan, risks, and architectural objectives.	Based on the project plan outlined earlier, the Project Manager starts sketching an iteration plan for the current iteration (Artifact: Iteration Plan). Evaluation criteria for the architecture are outlined in discussions with the Architect and by considering the architectural risks to be mitigated (Artifact: Risk List). Remember that one of the goals of elaboration is to establish a robust, executable architecture; the plan for doing this must be developed in this initial elaboration iteration.
Requirements: Decide which use cases and scenarios will drive the development of the architecture.	The Software Architect continues by discussing an initial use-case view with the Project Manager to determine which use cases and scenarios should be focused on in this iteration; these use cases and scenarios will drive the development of the architecture. Note that the identification of these use cases and scenarios affects the iteration plan described in the preceding step, and the iteration plan should be updated.
Requirements: Understand this driver in detail and inspect the results.	A number of requirement specifiers describe some of the selected use cases and scenarios in detail: the higher priority, most critical, and most complex ones to be addressed in this first elaboration iteration. The System Analyst might then need to restructure the use-case model as a whole. The changes to the use-case model are then reviewed and approved (Artifacts:

	Use-Case Model and Supplementary Specification).
Reconsider use cases and risks.	The Software Architect revisits the use-case view, taking into consideration new use-case descriptions and possibly a new structure of the use-case model. The task now is to select the set of use cases and scenarios to be analyzed, designed, and implemented in the current iteration. Note again that the development of these use cases and scenarios sets the software architecture. The Project Manager again updates the current iteration plan accordingly (Artifact: Iteration Plan) and might reconsider risk management because new risks might have been made visible according to new information (Artifact: Risk List).
Requirements: Prototype the user interface.	Using the use cases that have already been fleshed out, a user-interface designer starts expanding them in storyboards and builds a user-interface prototype to get feedback from prospective users (Artifacts: Storyboards and User-Interface Prototype).
Analysis and Design: Find obvious classes, do initial subsystem partitioning, and look at use cases in detail.	To get a general sense of the obvious classes needed, the Software Architect considers the system requirements, the glossary, the use-case view (but no use cases), and the team's general domain knowledge to sketch the outline of the subsystems, possibly in a layered fashion. The architect also identifies the analysis mechanisms that constitute common solutions to common problems during analysis. The software architecture document is initiated (Artifact: Software Architecture Document). In parallel with this effort, a team of designers, possibly with the architect, starts finding classes or objects for this iteration's use cases or scenarios. This team also begins to allocate responsibilities to the identified classes and analysis mechanisms.
Analysis and Design: Refine and homogenize classes and identify architecturally significant ones; inspect results.	A number of designers refine the classes identified in the preceding step by allocating responsibilities to the classes and updating their relationships and attributes. It is determined in detail how the available analysis mechanisms are used by each class. Then the architect identifies a number of classes that should be considered architecturally significant and includes them in the logical view (Artifact: Software Architecture Document).
Analysis and Design: Consider the low-level package partitioning.	To analyze the service aspect of the architecture, the Software Architect organizes some of the classes into design packages and relates these packages to the subsystems. The subsystem partitioning might need to be reconsidered.
Analysis and Design: Adjust to the implementation environment, decide the design of the key scenarios, and define formal class interfaces; inspect results.	The Software Architect then refines the architecture by deriving the design mechanism needed by the analysis mechanisms identified earlier. The design mechanisms are constrained by the implementation environment, that is, the implementation mechanisms available: operating system, middleware, database, and so on. Designers instantiate these classes into objects when they describe how the selected use cases or scenarios are realized in terms of collaborating objects in interaction diagrams. This puts requirements on the employed classes and design mechanisms; the interaction diagrams previously created are refined. Given the detailed requirements that are then put on each object, the designers merge these into consistent and formal interfaces on their classes. The requirements put on each design mechanism are handled by the Software Architect, who updates the logical view accordingly. The resulting design artifacts are then reviewed.
Analysis and Design: Consider concurrency and distribution of the architecture.	The next step for the Architect is to consider the concurrency and distribution required by the system. The Software Architect studies the tasks and processes required and the physical network of processors and other devices. An important input to the architect here is the designed use cases in terms of collaborating objects in interaction diagrams: the use-case realizations (Artifact: Software Architecture Document).
Analysis and Design: Inspect the architectural design.	The architecture is reviewed.
Implementation: Consider the physical packaging of the architecture.	The Software Architect now considers the impact of the architectural design on the implementation model and defines the implementation view.

Implementation: Plan the integration.	A System Integrator studies the use cases that are to be implemented in this iteration and defines the order in which subsystems should be implemented and later integrated into an architectural prototype. The results of this planning should be reflected in the project plan (Artifact: Project Plan).
Test: Plan integration tests and system tests.	A Test Designer plans the system tests and the integration tests, selecting measurable testing goals to be used when assessing the architecture. These goals could be expressed in terms of the ability to execute a use-case scenario with a certain response time or under specified load. The Test Designer also identifies and implements test cases and test procedures (Artifact: Test Plan).
Implementation: Implement the classes and integrate.	A number of Implementers code and unit-test the classes identified in the architectural design. The implementations of the classes are physically packaged into components and subsystems in the implementation model. The Integration Testers test the implementation subsystem, and then the Implementers release the subsystems to integration.
Integrate the implemented parts.	The System Integrators incrementally integrate the subsystems into an executable architectural prototype. Each build is tested.
Test: Assess the executable architecture.	Once the whole system (as defined by the goal of this iteration) has been integrated, the System Tester tests the system. The Test Designer then analyzes the results to make sure that the testing goals have been reached. The Software Architect then assesses this result and compares it with the risk identified initially.
Assess the iteration itself.	The Project Manager compares the iteration's actual cost, schedule, and content with those of the iteration plan; determines whether rework is necessary and, if so, assigns it to future iterations; updates the Risk List (Artifact: Risk List); updates the Project Plan (Artifact: Project Plan); and prepares an outline of an iteration plan for the next iteration (Artifact: Iteration Plan). Other lessons learned in terms of productivity, process improvement, tool support, and training are also interesting to consider at this stage, and actions are defined for the next iteration.

Result

The result of this initial iteration is a first cut at the architecture. It consists of fairly well-described architectural views (the use-case view, the logical view, and the implementation view) and an executable architectural prototype.

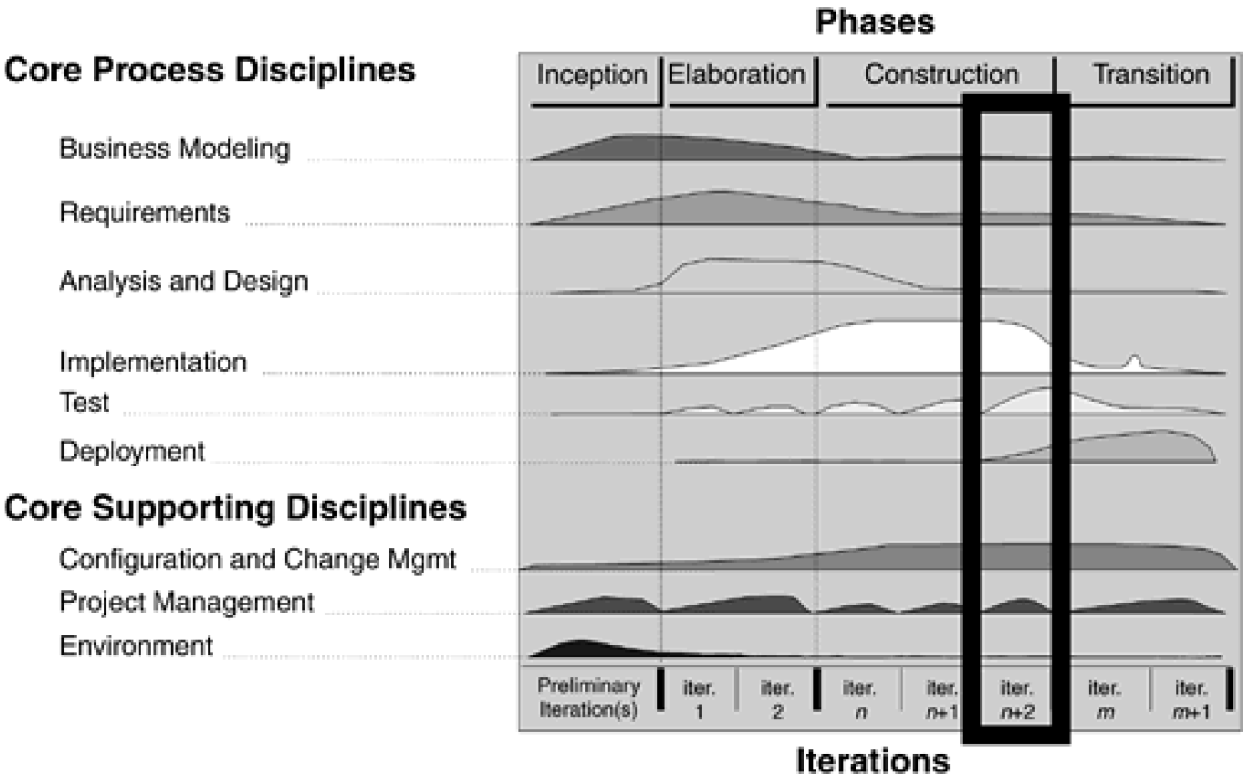
Subsequent Iterations in Elaboration

Subsequent iterations enhance the understanding of the requirements and of the architecture. This might imply a further enhancement of the design or implementation model (that is, the realization of more use cases in priority order).

Implementing the System

The context of our third example is that in this late construction phase, requirements are stable and a great deal of the functionality has been implemented and integrated in preparation for test. The project is approaching its first beta release and the initial operational capability milestone (see Figure 16-3).

Figure 16-3. An iteration late in construction



Project Management: Plan the iteration.	The Project Manager updates the Iteration Plan based on any new functionality that is to be added during the new iteration, factoring in the current level of product maturity, lessons learned from the previous iterations, and any risks that need to be mitigated in the upcoming iteration (Artifact: Iteration Plan and Artifact: Risk List).
Implementation: Plan system-level integration.	Integration planning takes into account the order in which subsystems are to be put together to form a working and testable configuration. The choice depends on the functionality already implemented and on the aspects of the system that must be in place to support the overall integration and test strategy. This is done by the System Integrator, and the results are documented in the Build Plan (Artifact: Integration Build Plan). The Integration Build Plan defines the frequency of builds and describes when given build sets will be required for ongoing development, integration, and test.
Test: Plan and design system-level test.	The Test Designer ensures that there will be an adequate number of test cases (Artifact: Test Cases) to verify testable requirements. The Test Designer must identify and describe Test Cases and identify and structure Test scripts and test suites. In general, each Test Case has at least one associated Test Script. The Test Designer also reviews the accumulated body of tests from preceding iterations, which could be modified to be reused in regression testing for current and

	future iteration builds (Artifact: Test Scripts).
Analysis and Design: Refine use-case realizations.	Designers refine the classes identified in previous iterations by allocating responsibilities to specific classes and updating their relationships and attributes. Classes may need to be added to support design and implementation constraints. Changes to classes may require a change in subsystem partitioning.
Test: Plan and design integration tests at the subsystem and system levels.	Integration tests focus on the degree to which developed components interface and function together. The Test Designer follows the Test Plan that describes the overall test strategy, required resources, schedule, and completion and success criteria. The Designer identifies the functionality that will be tested together and the stubs and drivers that must be developed to support the integration tests. The Implementer develops the stubs and drivers based on the input from the Test Designer.
Implementation: Develop code and test unit.	In accordance with the project's programming guidelines, Implementers develop code to implement the classes in the Implementation Model. They fix defects and provide feedback that might lead to design changes based on discoveries made in implementation.
Implementation: Plan and implement unit tests.	The Implementer designs unit tests that address what the unit does (the black-box test) and how it does it (the white-box test). Under black-box (specification) testing, the Implementer must be sure that the unit, in its various states, performs to its specification and can accept and produce a range of valid and invalid data. Under white-box (structure) testing, the challenge for the Implementer is to ensure that the design has been implemented correctly and that the unit can be traversed through each of its decision paths.
Implementation: Test unit within a subsystem.	Unit test focuses on verifying the smallest testable components of the software. Unit tests are designed, implemented, and executed by the Implementer of the unit. The emphasis of unit test is to ensure that white-box testing produces the expected results and that the unit conforms to the project's adopted quality and development standards.
Implementation and Test: Integrate a subsystem.	The purpose of subsystem integration is to combine into an executable build set units that may come from many developers within the subsystem (part of the implementation model). In accordance with the plan, the Implementer integrates the subsystem by bringing together completed and stubbed classes that constitute a build. The Implementer integrates the subsystem incrementally from the bottom up based on the compilation dependency hierarchy.
Implementation: Test a subsystem.	Testers execute Test Procedures developed earlier. If there are unexpected test results, Integration Testers log the defects for arbitration to decide when they are to be fixed.
Implementation: Release a subsystem.	Once the subsystem has been tested sufficiently and it is ready for integration at the system level, the Implementer releases the tested version of the subsystem from the team integration area into an area where it becomes visible, and usable, for system-level integration.
Implementation: Integrate the system.	The purpose of system integration is to combine the available Implementation Model functionality into a build. The System Integrator incrementally adds subsystems and creates a build that is handed over to the Integration Testers for overall integration testing.
Test: Test integration.	Integration Testers execute Test Procedures developed earlier. The Integration Testers execute integration tests and review the results. If there are unexpected results, the Integration Testers log the defects.
Test: Test the system.	Once the whole system (as defined by the goal of this iteration) has been integrated, the System Tester tests the system. The Test Designer then analyzes the results of the test to make sure that the testing goals have been reached.
Project Management: Assess the iteration itself.	The Project Manager compares the iteration's actual cost, schedule, and content with the Iteration Plan; determines whether rework is necessary and, if so, assigns it to future iterations; updates the Risk List (Artifact: Risk List); updates the Project Plan (Artifact: Project Plan); and prepares an outline of an Iteration Plan for the next iteration (Artifact: Iteration Plan).

Result

The main result of a late iteration in the construction phase is more functionality, and that yields a more comprehensive system. The results of the current iteration are made visible to developers to form the basis of development for the subsequent iterations.

PREVIOUS NEXT

Summary

- Each iteration may run through each of the core workflows; the emphasis on the activities and the degree to which they are performed depend on the goals of the iteration.
- Whether an activity is performed also depends in large part on the current lifecycle phase and on the artifacts that the Development Case prescribes for the project.

Chapter 17. Implementing the Rational Unified Process

This chapter describes the strategies and tactics for deploying the Rational Unified Process in an adopting software development organization.

Introduction

The Rational Unified Process could be used in whole or in part "out of the box," using it casually, simply as some kind of online Reference Manual of Software Engineering Practices. But we have found that this is not the best way to take advantage of the RUP, and that organizations using this approach tend to "sink" in the RUP, trying to do everything that it describes, using too much of RUP.

The Rational Unified Process must be configured and tailored to the specific context and needs of an organization before its full implementation.

To *configure* the Rational Unified Process means to adapt the process product to the needs and constraints of the adopting organization by modifying the process framework delivered by Rational Software. The result of configuring the Rational Unified Process is captured in a *Development Case.* This aspect is covered by the RUP's Process Engineering Process (PEP) and described in Chapter 14, the Environment Discipline.

To *implement* the Rational Unified Process in a software development organization means to change the organization's practice so that it routinely and successfully uses a configuration of the Rational Unified Process tailored to its context and its needs.

Our colleagues Stefan Bergstrom and Lotta Råberg have dedicated to this topic an entire book: *Implementing the Rational Unified Process: Success with the RUP*. Boston: Addison-Wesley, 2003.

The Effect of Implementing a Process

Process changes are difficult, and it may be a long time before you see their true effects. This is in contrast to the adoption of a new tool, something that is relatively easy and fast: You install it, read the user manual, go through an example, and maybe attend a training course. The transition to a new tool might last from a few hours to a couple of weeks. But changing the software development process often means affecting the fundamental beliefs and values of the individuals involved and changing the way they perceive their work and its value. It is a cultural change, and often political or philosophical as well.

A process change affects the individuals and the organization more deeply than a change of technology or tools. It must be planned and managed carefully. The adopting organization must identify the opportunity and the benefits, convey them clearly to the interested parties, raise their level of awareness, and then gradually change from the current practice to a new practice. Ivar Jacobson describes this as "reengineering your software engineering process."[1]

[1] Ivar Jacobson and Sten Jacobson, "Reengineering Your Software Engineering Process,'*Object Magazine*, March–April, 1995.

When implementing a process, you must address the following areas:

- *The people and their competence, skills, motivation, and attitude:* You must make sure that everyone is adequately trained and motivated. To succeed with the process-implementation project, it is important to involve people in the effort as early as possible. They are an important source of information when you are assessing the software development organization's current state. Second, you should ensure that everyone understands the current state of the organization and perceives the problems it is experiencing as well as how and where they can improve. Building up this understanding is one of the keys to success for any change project.
- *The supporting tools:* You must buy new tools, replace old ones, and customize and integrate others.
- *The software development process:* This includes the life-cycle model, the organizational structure, the activities to be performed, the practices to be followed, the artifacts to be produced, and the scope of the software development process.
- The *description* of the software development process.

Other areas also affect the way people work—for example, the physical working environment, the management structure (treatment of people, how people are involved in decision making), and the organizational structure (reporting, organizational units, workgroup relationships, etc.).

In addition to the people whose work is affected most directly by the process change, you must take into consideration the people who otherwise might feel excluded from the change process:

- Managers are responsible for the performance of the software development organization. They must understand why you are changing the process and why you are procuring new tools. It is important that they understand how (and whether) progress is being made. Any process improvement project must have executive support. You must make sure that management understands that there is a return on the investment in changing the process, and you must also manage that expectation carefully.
- Customers must be informed that your process effectively takes their needs into consideration to evolve the product as their needs change.
- Other parts of the software development organization are also affected. Sometimes you change only one part of the

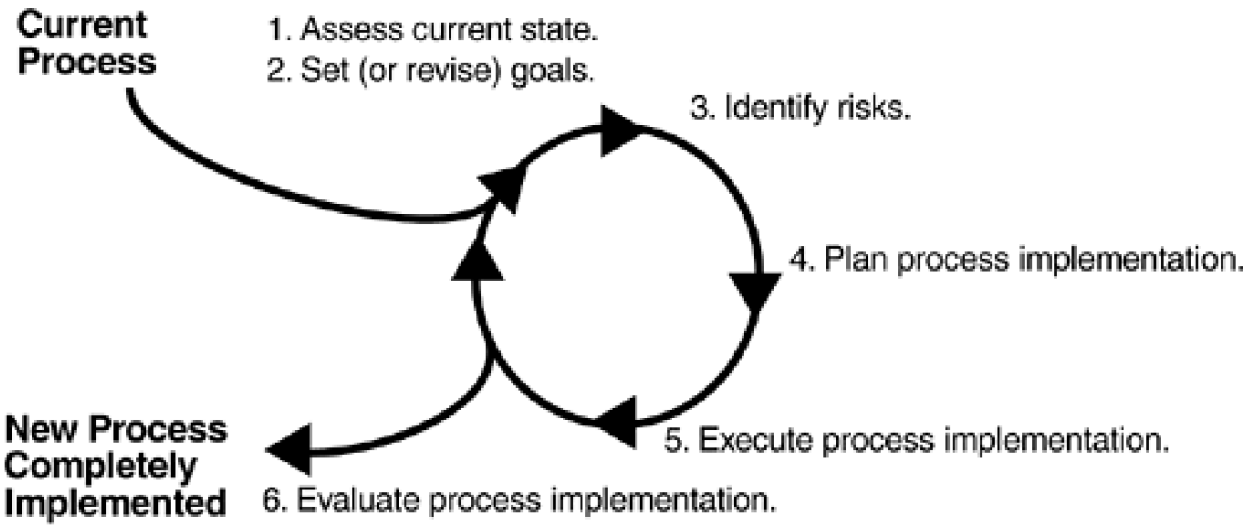
organization, and that may lead to resistance and skepticism from other parts of the organization. Often, the people in other departments do not understand what you are doing and why you are doing it. Even if they do not have a direct influence, their exclusion may cause political problems. Start early to let the entire organization know what is going on and where the process change is heading.

Implementing the Rational Unified Process Step by Step

Implementing a new process in a software development organization can be described in six steps (see Figure 17-1).[2]

[2] R. McFeeley, *IDEAL: A User's Guide for Software Process Improvement*, SEI Tech. Rep. CMU/SEI-96-HB-001, 1996.

Figure 17-1. The steps to implement a new software development process



Step 1: Assess the Current State.

You need to understand the current state of the software development organization in terms of its people, process, and supporting tools. You should also identify problems and potential areas for improvement as well as collect information about outside issues such as competitors and market trends. When this step is complete, you should know the following:

- The current state of the software development organization
- The kind of people who work here and their level of competence, skills, and motivation
- The tools currently used in the organization
- The current software engineering process and how it is described

Why should you assess the current state? Consider the following reasons:

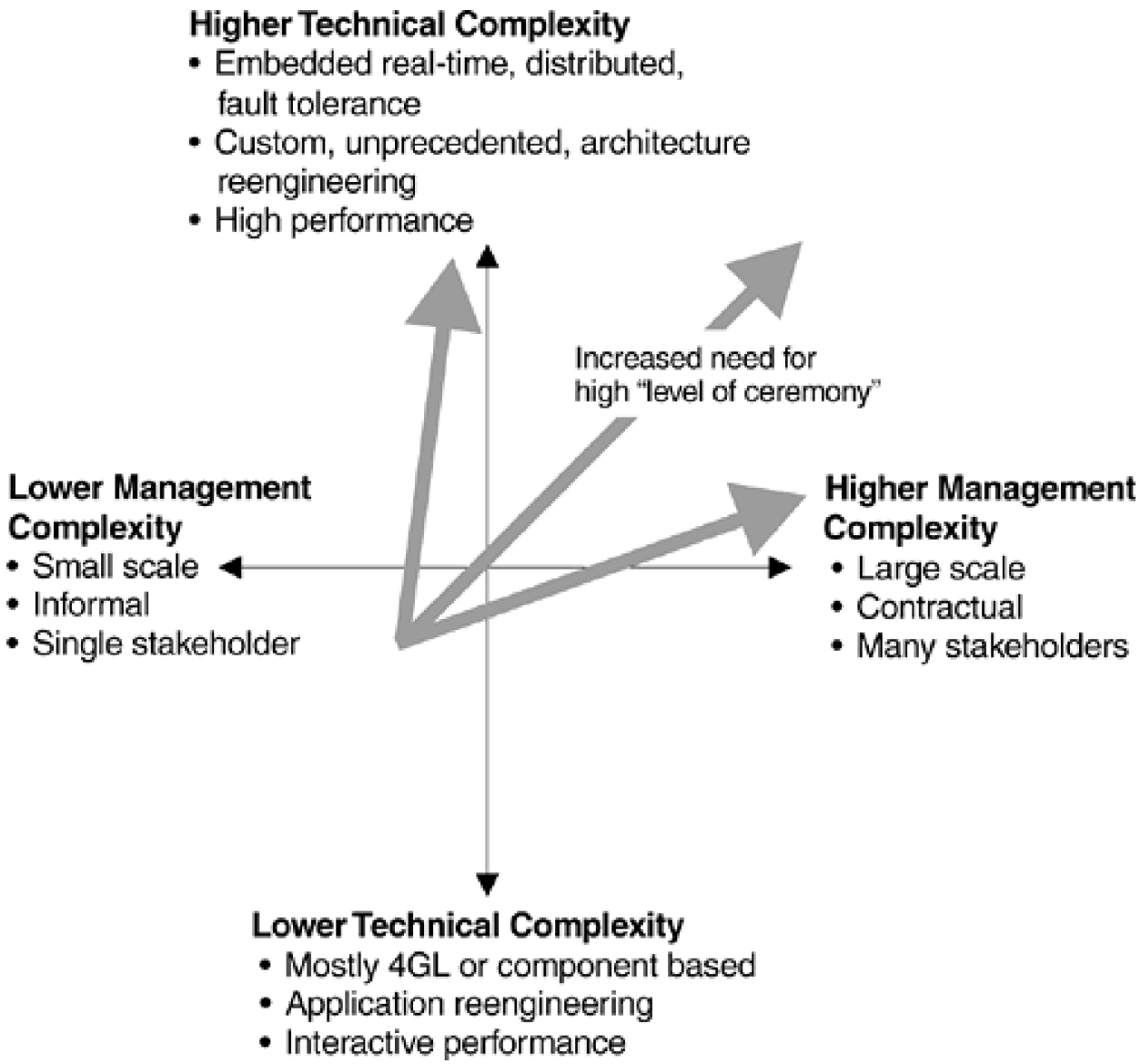
- You want to use it to create a plan to get from the current state of the organization to your goal.
- You want to identify the areas that need to be improved first. You may not want to introduce the entire process and all the

tools at once. Instead, you may prefer to do it in increments, starting with the areas that have the greatest need and the best potential for improvement.

- You want to explain to the sponsors why you need to change the process, tools, and people.
- You want to create motivation and a common understanding among the people who are affected directly or indirectly.

It is important to understand the project's level of management complexity and level of technical complexity. The more stakeholders the project has and the bigger the project, the higher the level of ceremony that is needed. More artifacts must be produced, communicated, explained, reviewed, and approved. The higher the level of technical complexity, the more effort that must be put into maintaining the artifacts and tracking the status of the project (see Figure 17-2).

Figure 17-2. Systems classified according to technical and managerial complexity



Step 2: Set (or Revise) Goals.

The second step is to set goals for the process, people, and tools, noting where you want to be when the process implementation

project is complete. You set up goals for the following reasons:

- Goals serve as an important input to planning the implementation of the process.
- Goals, combined with the result of step 1 (a description of the current state), are used to motivate the sponsors and the people in the organization.

The result should be a list of measurable goals that are expressed so that they can be understood by project members. The goals can serve as a vision of the future state of the organization.

Step 3: Identify Risks.

To succeed in implementing a new process, you must control the many risks involved. We recommend that you perform a risk analysis in which you identify possible risks, try to understand their impact, and then rank them. You should also plan how you will mitigate the risks and in what order (as described in Chapter 7).

Switching from a linear process to an iterative process is not a risk-free undertaking. Software development organizations using an iterative approach for the first time may fall into several traps, including the following:

- The first iteration is too ambitious, and people cannot focus on the most important risks.
- There is a failure to implement and test in each iteration. In order to mitigate a risk, you normally have to test what has been built.
- Some stakeholders do not buy into or understand the iterative approach. They expect the project to progress as if it were using the waterfall method, where, for example, a complete design of the system should be produced early in the lifecycle.
- You plan feature by feature instead of planning the complete project from the start (at a high level) and being prepared to change the plan along the way.
- There is too much rework between the iterations, meaning you do things in an earlier iteration that you must redo later.
- You fail to control requirements changes, thinking that changes do not matter very much because problems can be fixed in later iterations.
- You undertake too much training too early. As a result, too much time elapses before people start to produce real work, and they tend to forget what they learned earlier.
- You staff the project too quickly, and too many people are involved in the inception phase. This makes it difficult for individuals to understand their responsibilities and runs the risk of leaving some people without tasks.
- Tool support is insufficient, or people lack the skills to use them properly.
- You fail to assess the true value of artifacts and thus produce unnecessary artifacts.

Step 4: Plan the Process Implementation.

You should develop a plan for implementing the process and the tools in the organization. This plan should describe how to move efficiently from the current state of the organization to the goal state.

Do not try to do everything at once. Instead, divide the implementation into a number of increments, and for each one, implement a

portion of the new process with the supporting tools. Typically, you should focus on one of the areas where you believe the change will have the most impact. If the software development organization is weak in testing, you might start by introducing the test discipline of the Rational Unified Process with tools to automate testing. If, on the other hand, the organization is weak in capturing or managing requirements, you might start by introducing the Requirements discipline with its supporting tools.

There are different approaches to implementing a process. The one you choose depends on two things:

- *The need for change in the current organization*

  If there are a lot of problems in the organization—problems with the tools or with the way people work—the frustration level in the organization will be high. In this case, you can be fairly aggressive and employ the new process and tools, or parts of them, in real projects.

- *The risks involved*

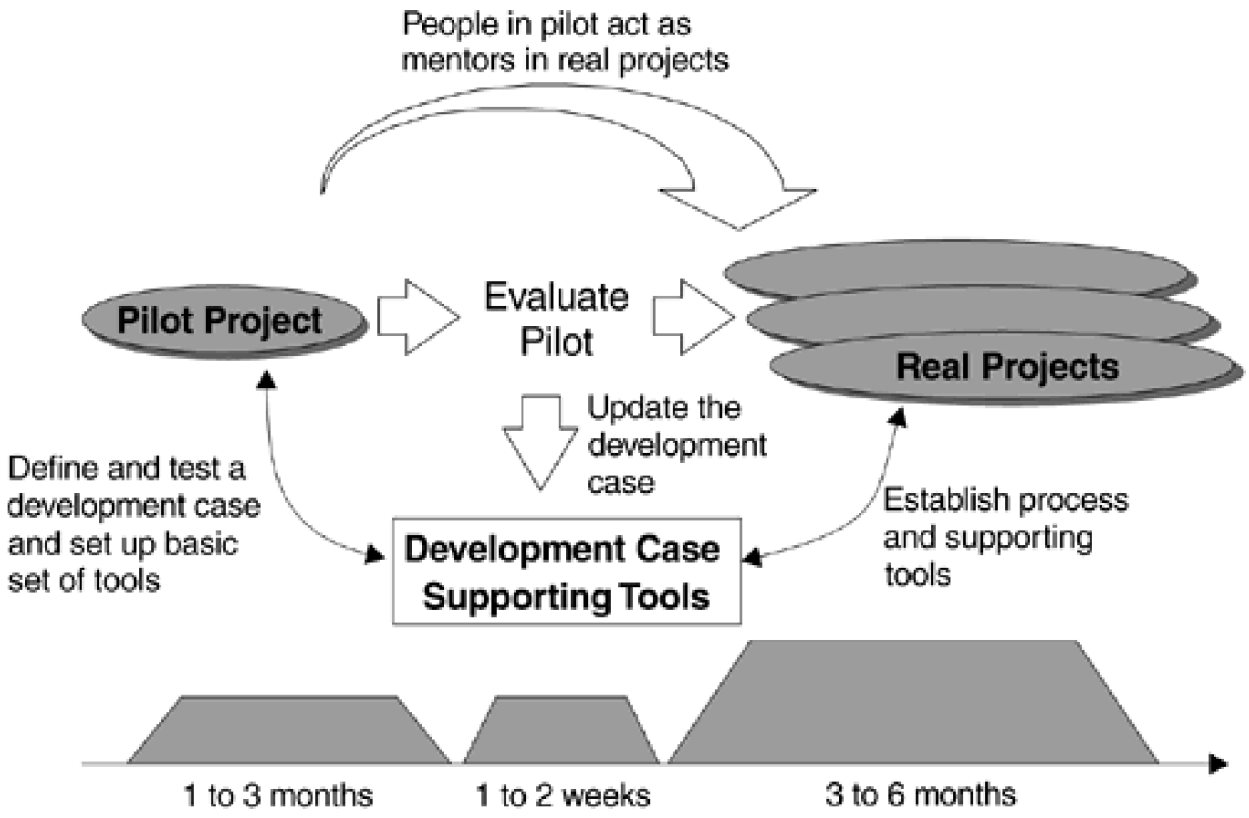
  If the risks are small, you can be fairly aggressive and quickly start using the process and tools in new projects. If the risks are great you must be careful, using pilot projects to verify the process and the tools.

We give examples of two approaches:

- The "typical" approach
- The "fast" approach

In the typical approach (illustrated in Figure 17-3), you first implement the process in a Pilot Project. In an initial step you configure the process and describe it in a Development Case. You use the process on the Pilot Project. Experience from the Pilot Project is fed back into the Development Case.

Figure 17-3. The "typical" approach to implementing a process and tools



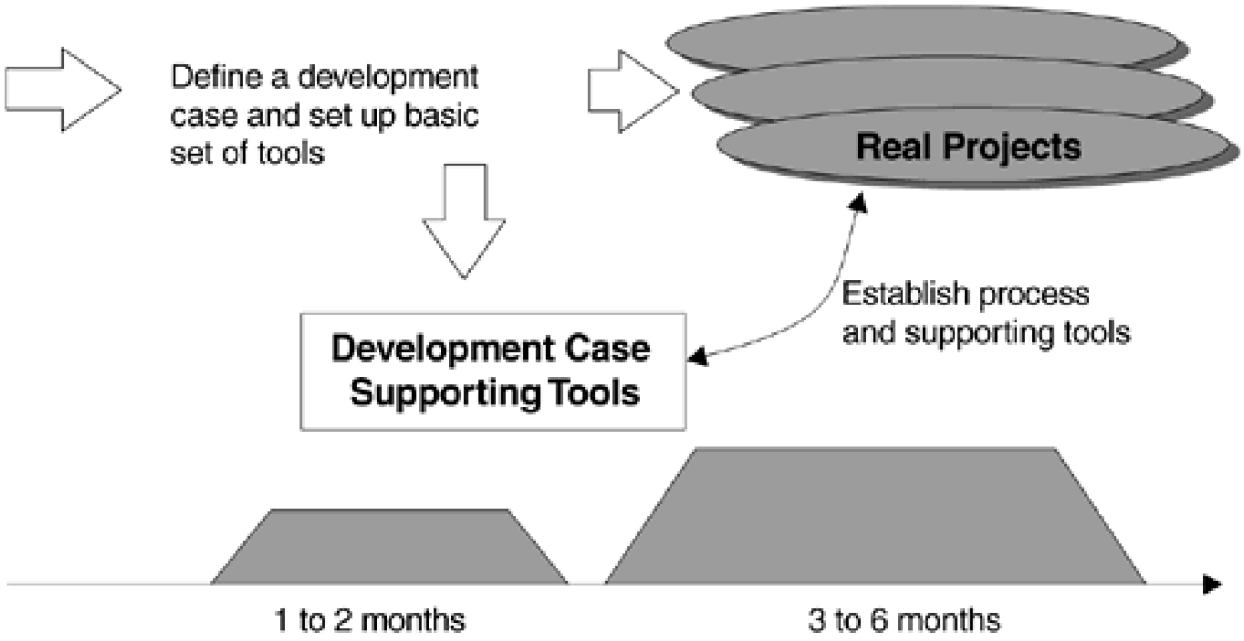
The process is then considered tested, or verified, and can be rolled out to a broader audience. As a Pilot Project you might use one of the following:

- A complete software development project that is considered to have low risk from a technical and financial perspective
- The first complete iteration of an actual software development project, with the caveat that the focus is on learning and improving the process and not on developing software

This approach is the most effective way to introduce the process and tools.

The fast approach (illustrated in Figure 17-4) is to use the process and tools directly in actual projects without first verifying that they work in a Pilot Project. This approach introduces a greater risk of failure, but there can be good reasons for taking these risks. For example, if the current process is very similar to the Rational Unified Process and if the tools are already used in the organization, it may be relatively easy and low-risk to implement the new process and tools.

Figure 17-4. The "fast" approach to implementing a process and tools



Make sure that you have chosen the right people to participate in the Pilot Project because they will pass the message on to the rest of the organization. Some of them will play important roles as mentors when the process is implemented in the rest of the organization.

Step 5: Execute the Process Implementation.

The most time-consuming step in implementing a process is to execute it according to the plan defined in step 4. Step 5 includes the following tasks:

- Develop a new development case or update an existing one.
- Acquire and adapt tools to support and to automate the process.
- Train members of the development team to use the new process and tools.
- Apply the process and tools in a software development project.

Step 6: Evaluate the Process Implementation.

When you have implemented the process and tools in a software development project, actual or pilot, you must evaluate the effort. Did you achieve the goals you established? Evaluate the people, the process, and the tools to understand which areas to focus on when you start again from step 1.

PREVIOUS NEXT

Implementing a Process Is a Project

Implementing a software development process is a complex task that should be controlled carefully. We recommend treating it as a project (external to or a subproject of your software development project) and setting up milestones, allocating resources, and managing it as you would for any project.

The process implementation project is divided into a number of phases. All six steps are performed in each phase until the project is ready and the process and tools are deployed and successfully used by the entire organization (see Figure 17-5). Table 17-1 summarizes how a project can be planned with four phases.

Figure 17-5. A process implementation project divided into phases

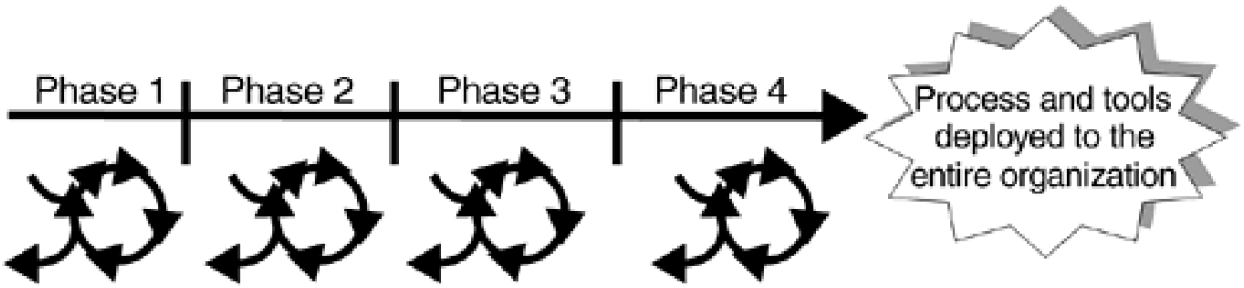
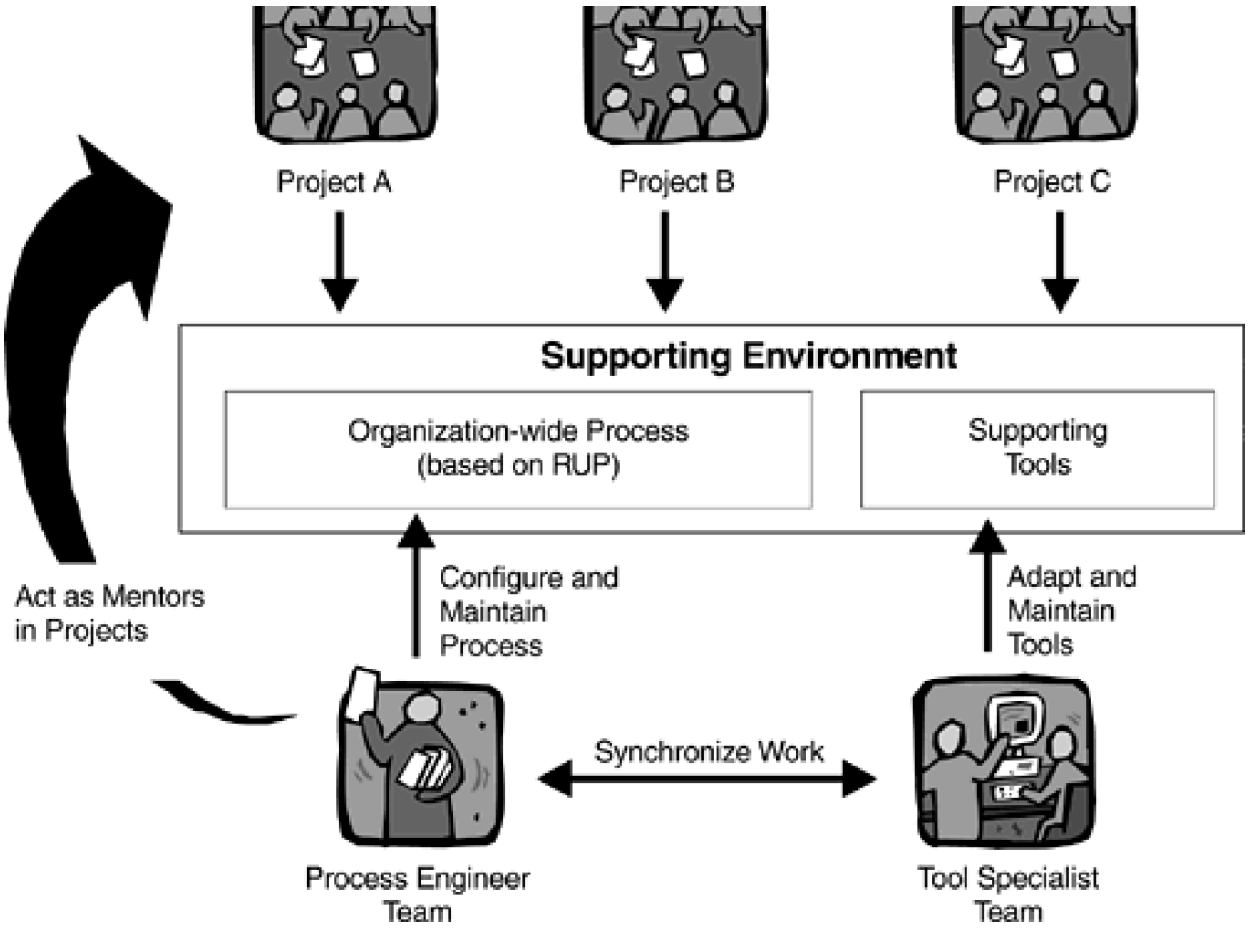


Table 17-1. The Four Phases of a Process Implementation Project

Phase	Purpose	Important Results after the Phase
1	To "sell" the process implementation project to the sponsors	Go/no go decision from the sponsors. To support the decision, the tools may be demonstrated and a development case may be exemplified.
2	To handle the major risks	Some tools ready to use; critical parts of development case ready.
3	To complete everything	All tools are ready, the development case is complete, a training curriculum is ready; mentors are ready to support real projects in next phase.
4	To deploy it to the entire organization	Process and tools are deployed to the entire organization.

The group of people working on implementing the process should be dedicated to this task. They should function as mentors in the software development project, applying the process and tools. It is also their responsibility to maintain the new process, and that includes incorporating improvements suggested by the users. Also, to make sure that the process gains credibility within the organization, this group must make clear to the rest of the organization the impact of the new process on productivity and product quality. In large organizations, you may have a process engineering team (such as the Software Engineering Process Group, SEPG) that configures and maintains the process, as well as tool specialists who adapt and maintain the supporting tools (see Figure 17-6).

Figure 17-6. Process engineers and tool specialists


Project A
Project B
Project C
Supporting Environment
Organization-wide Process
(based on RUP)
Supporting
Tools
Act as Mentors
in Projects
Configure and
Maintain
Process
Adapt and
Maintain
Tools
Synchronize Work
Process Engineer
Team
Tool Specialist
Team

Summary

- Changing the software development process used by an organization is a challenging endeavor, a project in itself, which involves more than just software engineering; it also involves social engineering and psychology.
- An organization can take various approaches to adopt the Rational Unified Process. A typical approach involves trying part of the Rational Unified Process in a Pilot Project before extending it to the entire organization.
- Adopting the Rational Unified Process involves developing a Development Case, which is a project-specific version of the process.
- The Development Case is typically a RUP configuration created with RUP Builder (and RUP Organizer) that may include a company-wide RUP plug-in that contains organization-wide process assets.
- Tools to support the process must be put in place simultaneously with the process.
- Professional education and tool training are integral parts of a carefully planned process change.

Appendix A. Summary of Roles

This appendix summarizes the 30 roles involved in the Rational Unified Process. They are grouped along five main categories of roles: Analysts, Developers, Managers, Testers, Production and Support. Remember that roles are not persons. They represent skills and competence, and they are associated with a set of activities (see Chapter 3).

The Analyst Roles

Business-Process Analyst The Business-Process Analyst is responsible for defining the business architecture, the business use cases and actors, and how they interact. The Business-Process Analyst leads and coordinates business use-case modeling by outlining and delimiting the organization being modeled, for example, by establishing what business actors and business use cases exist and how they interact. The Business-Process Analyst is responsible for the business architecture, outlining and delimiting the organization being modeled.

Business Designer The Business Designer details the specification of a part of the organization. The Business Designer specifies the workflow of business use cases in terms of business workers and business entities. It also distributes the behavior to these business workers and business entities—defining their responsibilities, operations, attributes, and relationships.

System Analyst The System Analyst leads and coordinates requirements elicitation and use-case modeling by outlining the system's functionality and delimiting the system, for example, identifying what actors exist and what use cases they will require when interacting with the system.

Requirements Specifier The Requirements Specifier specifies the details of one or more parts of the system's functionality by describing one or more of the aspects of the requirements. Different aspects of the requirements are typically documented in different types of software requirements artifacts: use cases, supplementary specifications, and so on. The Requirements Specifier may be responsible for many of those artifacts—such as one or more use cases or use-case packages—and for maintaining the integrity of the requirements within and among those artifacts.

The Developer Roles

Software Architect The Software Architect has overall responsibility for driving the major technical decisions, which are expressed as the software architecture. This typically includes identifying and documenting the architecturally significant aspects of the system, including requirements, design, implementation, and deployment "views" of the system. The Software Architect is also responsible for providing the rationale for these decisions, balancing the concerns of the various stakeholders, driving down technical risks, and ensuring that decisions are effectively communicated, validated, and adhered to.

Designer The Designer is responsible for designing a part of the system, within the constraints of the requirements, architecture, and development process for the project. The Designer identifies and defines the responsibilities, operations, attributes, and relationships of design elements. The Designer ensures that the design is consistent with the software architecture, and that it is detailed to a point where implementation can proceed.

User-Interface Designer The User-Interface Designer coordinates the design of the user interface. User-Interface Designers are also involved in gathering usability requirements and prototyping candidate user-interface designs to meet those requirements.

Capsule Designer The Capsule Designer is responsible for the design of capsules, ensuring that the system is able to respond to events in a timely manner and in accord with the concurrency requirements.

Database Designer The Database Designer defines the tables, indexes, views, constraints, triggers, stored procedures, table spaces or storage parameters, and other database-specific constructs needed to store, retrieve, and delete persistent objects.

Implementer The Implementer is responsible for developing and testing components, in accordance with the project's adopted standards, for integration into larger subsystems. When test components, such as drivers or stubs, must be created to support testing, the Implementer is also responsible for developing and testing the test components and corresponding subsystems.

Integrator Integrators are responsible for planning the integration and performing the integration of implementation elements to produce builds.

The Manager Roles

Project Manager The Project Manager allocates resources, shapes priorities, coordinates interactions with the customers and users, and generally tries to keep the project team focused on the right goal. The Project Manager establishes a set of practices to ensure the integrity and quality of project artifacts.

Change Control Manager The Change Control Manager oversees the change control process. This role is usually played by a Change Control Board (CCB), which should consist of representatives from all interested parties, including customers, developers, and users. In a small project, a single person, such as the project manager or software architect, may play this role.

Configuration Manager A Configuration Manager is responsible for providing the overall CM infrastructure and environment for the product development team. The CM function supports the product development activity so that developers and integrators have appropriate workspaces to build and test their work and that all artifacts are available as required. The Configuration Manager ensures that the CM environment facilitates product review, change, and defect tracking activities. The Configuration Manager is also responsible for writing the CM plan and reporting change-request-based progress statistics.

Test Manager The Test Manager has the overall responsibility for the test effort's success. The role involves quality and test advocacy, resource planning and management, and resolution of issues that impede the test effort.

Deployment Manager The Deployment Manager is responsible for planning the product's transition to the user community, ensuring those plans are enacted appropriately, managing issues and monitoring progress.

Process Engineer The Process Engineer is responsible for the software development process itself. This includes configuring the process before project start-up and continuously improving the process during the development effort.

Management Reviewer The Management Reviewer is responsible for evaluating project planning and project assessment artifacts at major review points in the project's lifecycle.

The Tester Roles

Test Analyst The Test Analyst is responsible for identifying and defining the required tests, monitoring detailed testing progress and results in each test cycle, and evaluating the overall quality experienced as a result of testing activities. The role typically carries the responsibility for appropriately representing the needs of stakeholders that do not have direct or regular representation on the project.

Test Designer The Test Designer is responsible for the planning, design, implementation, and evaluation of testing, including generation of the test plan and test model, implementation of the test procedures, and evaluation of test coverage, test results, and effectiveness.

Tester The Tester is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing.

The Production and Support Roles

System Administrator The System Administrator maintains the development environment—both hardware and software—and is responsible for system administration, backup, and so on.

Technical Writer The Technical Writer produces end-user support material, such as user guides, help texts, release notes, and so on.

Graphic Artist The Graphic Artist creates product artwork that is part of the product packaging and documentation.

Tool Specialist The Tool Specialist is responsible for the supporting tools of the project. This includes assessing the need for tool support and selecting and acquiring tools.

Course Developer The Course Developer develops training material to teach users how to use the product. He or she creates slides, student notes, examples, tutorials, and so on to enhance their understanding of the product.

Additional Roles

Reviewer The Reviewer is a generic role, responsible for providing timely feedback to project team members on the artifacts they have produced. See also Management Reviewer and Technical Reviewer.

Review Coordinator The Review Coordinator is responsible for facilitating formal reviews and inspections, and ensuring that they occur when required and are conducted to a satisfactory standard.

Any Role This is a generic role used to carry the activities that any project member can perform. For example, any role can be given access privileges to "check in" and "check out" any product-related artifact for maintenance in the configuration control system.

Stakeholder A stakeholder is anyone who is materially affected by the outcome of the project. This is another generic role that covers, depending on the project context, customer, end user, purchaser, product manager, and so on.

Appendix B. Summary of Artifacts

This appendix summarizes the artifacts that are produced or used during the process. They are organized in this appendix in nine artifact sets following the nine main disciplines of the Rational Unified Process and are associated with their responsible role (see Chapter 3).

Business Modeling Artifact Set

● Target-Organization Assessment	Business-Process Analyst
● Business Vision	Business-Process Analyst
● Business Glossary	Business-Process Analyst
● Business Rules	Business-Process Analyst
● Supplementary Business Specification	Business-Process Analyst
● Business Use-Case Model	Business-Process Analyst
- Business Use Case	Business Designer
- Business Actor	Business Designer
- Business Goal	Business Designer
● Business Analysis Model	Business-Process Analyst
- Business System	Business Designer
- Business Rule	Business Designer
- Business Event	Business Designer
- Business Worker	Business Designer
- Business Use-Case realization	Business Designer
- Business Entity	Business Designer
● Business Architecture Document	Business-Process Analyst

Requirements Artifact Set

Requirements Management Plan	System Analyst
Software Requirement	Requirements Specifier
Stakeholder Requests	System Analyst
Glossary	System Analyst
Vision	System Analyst
Requirements Attributes	System Analyst
Supplementary Specification	System Analyst
Software Requirements Specification	Requirements Specifier
Use-Case Model	System Analyst
- Use-Case Package	Requirements Specifier
- Use Case	Requirements Specifier
- Actor	Requirements Specifier
Storyboard	User-Interface Designer

Analysis and Design Artifact Set

● Reference Architecture	Software Architect
● Architectural Proof-of-Concept	Software Architect
● Software Architecture Document	Software Architect
● Use-Case Realization	Designer
● Analysis Model	Software Architect
- Analysis Class	Designer
● Design Model	Software Architect
- Design Subsystem	Designer
- Design Package	Designer
- Design Class	Designer
- Interface	Designer
- Use-Case Realization	Designer
- Capsule	Designer
- Protocol	Designer
- Signal	Software Architect
- Event	Software Architect
- Testability Class	Software Architect
- Test Design	Designer
● Deployment Model	Test Designer
● Data Model	Software Architect
● User-Interface Prototype	Database Designer
● Navigation Map	User-Interface Designer
	User-Interface Designer

Implementation Artifact Set

● Build	Integrator
● Implementation Model	Software Architect
- Implementation Subsystem	Implementer
- Implementation Element	Implementer
- Test Stub	Implementer
- Testability Element	Implementer
● Integration Build Plan	Integrator
● Developer Test	Implementer

Test Artifact Set

Test Plan	Test Manager
Test Evaluation Summary	Test Manager
Test Strategy	Test Designer
Test Idea List	Test Analyst
Test Case	Test Analyst
Test Data	Test Analyst
Test Results	Test Analyst
Workload Analysis Model	Test Analyst
Test Interface Specification	Test Designer
Test Script	Test Designer
Test Suite	Test Designer
Test Environment Configuration	Test Designer
Test Automation Architecture	Test Designer
Test Log	Tester

Deployment Artifact Set

- Deployment Plan
- Product
  - Bill of Materials
  - Deployment Unit
  - Product Artwork
  - Installation artifacts
- End-User Support Material
  - Release Notes
  - Training material

Deployment Manager

Deployment Manager

Deployment Manager

Configuration Manager

Graphic Artist

Implementer

Technical Writer

Development Manager

Course Developer

Configuration and Change Management Artifact Set

• Configuration Management Plan	Configuration Manager
• Project Repository	Configuration Manager
• Workspace	Any Role
• Configuration Audit Findings	Configuration Manager
• Change Request	Change Control Manager

Project Management Artifact Set

• Deployment Plan	Deployment Manager
• Business Case	Project Manager
• Software Development	Plan Project Manager
- Quality Assurance Plan	Project Manager
- Problem Resolution Plan	Project Manager
- Risk Management Plan	Project Manager
- Product Acceptance Plan	Project Manager
- Measurement Plan	Project Manager
• Risk List	Project Manager
• Issues List	Project Manager
• Iteration Plan	Project Manager
• Iteration Assessment	Project Manager
• Status Assessment	Project Manager
• Work Order	Project Manager
• Project Measurements	Project Manager
• Review Record	Reviewer

Environment Artifact Set

• Development Organization Assessment	Process Engineer
• Development Process	Process Engineer
- Development Case	Process Engineer
- Project-Specific Templates	Process Engineer
- Project Specific Guidelines	Process Engineer
• Development Infrastructure	System Administrator
• Tools	Tool Specialist

Appendix C. Acronyms

ATM	Automated Teller Machine
BPR	Business-Process Reengineering
CBD	Component-Based Development
CBT	Computer-Based Training
CCB	Change Control Board
CCM	Configuration and Change Management
COCOMO	Constructive Cost Model
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CM	Configuration Management
CR	Change Request
CRM	Change Request Management
DLL	Dynamically Linked Library
EJB	Enterprise JavaBeans
4GL	Fourth-Generation Language
ICAO	International Civil Aviation Organization (part of U.N.)
IOC	Initial Operating Capability
IEEE	Institute of Electrical and Electronics Engineers
IKIWISI	I'll Know It When I See It
IT	Information Technology
ISO	International Standards Organization
J2EE	Java 2 Enterprise Edition
LCA	Life-Cycle Architecture
LCO	Life-Cycle Objective
MTTF	Mean Time to Failure
MVC	Model-View-Controller

OMG	Object Management Group
OMT	Object Modeling Technique
ORB	Object Request Broker
PEP	Process Enginering Process
PRA	Project Review Authority
RDN	Rational Developers Network
RFP	Request for Proposal
ROI	Return on Investment
ROP	Rational Objectory Process
RPW	Rational Process Workbench
RUP	Rational Unified Process
SAD	Software Architecture Description
SDP	Software Development Plan
SEPG	Software Engineering Process Group
SPEM	Software Process Engineering Metamodel
SRSs	Software Requirements Specifications
UC	Use Case
UCM	Unified Change Management
UML	Unified Modeling Language
WBS	Work Breakdown Structure
XDE	eXtended Development Environment
XP	eXtreme Programming

PREVIOUS NEXT

Glossary

abstraction

The essential characteristics of an entity that distinguish it from all other kinds of entities and thus provide crisply defined boundaries from the viewer's perspective.

activity

A unit of work that a role performs. An activity may be decomposed into steps. Activities produce or modify artifacts.

actor

Someone or something outside the system or business that interacts with the system or business.

architectural baseline

The baseline at the end of the elaboration phase, at which time the foundation structure and behavior of the system are stabilized.

architectural pattern

A description of an archetypal solution to a recurrent design problem that reflects well-proved design experience.

architectural view

A view of the system architecture from a given perspective; focuses primarily on structure, modularity, essential components, and the main control flows.

See also **[view]**

architecture

See **[software architecture]**

artifact

A piece of information that is produced, modified, or used by a process, defines an area of responsibility, and is subject to version control. An artifact can be a model, a model element, or a document.

base

See **[RUP Base]**

baseline

A reviewed and approved release of artifacts that constitutes an agreed-on basis—a reference—for evolution or development and that can be changed only through a formal procedure, such as change and configuration control.

build

An operational version of a system or part of a system that demonstrates a subset of the capabilities to be provided in the final product.

builder

See **[RUP Builder]**

change control board (CCB)

The role of the CCB is to provide a central control mechanism to ensure that every change request is considered, authorized, and coordinated properly.

change management

The activity of controlling and tracking changes to artifacts.

change request (CR)

A request to change an artifact or process. Documented in the CR is information on the origin and impact of the current problem, the proposed solution, and its cost.

class

A description of a set of objects that have the same responsibilities, relationships, operations, attributes, and semantics.

component

A nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

See also **[process component]**

component-based development (CBD)

The creation and deployment of software-intensive systems assembled from components, as well as the development and harvesting of such components.

configuration

(1) General: The arrangement of a system or network as defined by the nature, number, and chief characteristics of its functional units; applies to both hardware and software. (2) The requirements, design, and implementation that define a version of a system or system component.

configuration management (CM)

A supporting process workflow whose purpose is to identify, define, and baseline items; control modifications and releases of these items; report and record status of the items and modification requests; ensure completeness, consistency, and correctness of the items; and control storage, handling, and delivery of the items. (ISO)

construction

The third phase of the Rational Unified Process, in which the software is brought from an executable architectural baseline to the point at which it is ready to be transitioned to the user community.

cycle

One complete pass through the four phases—inception, elaboration, construction, and transition—that results in a product release.

defect

A product anomaly; examples include omissions and imperfections found during early lifecycle phases and symptoms of faults contained in software sufficiently mature for test or operation. A defect can be any kind of issue you want tracked and resolved.

deliverable

An output artifact of the process that has a value, material or otherwise, to a customer.

deployment

A core discipline in the software engineering process whose purpose is to ensure a successful transition of the developed system to its users. Included are artifacts such as training materials and installation procedures.

deployment view

An architectural view that describes one or several system configurations; the mapping of software components (tasks, modules) to the computing nodes in these configurations.

design

The part of the software development process whose primary purpose is to decide how the system will be implemented. During design, strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design model

An object model that describes the realization of use cases; serves as an abstraction of the implementation model and its source code.

development case

The software engineering process used by the performing organization on a given project. It is developed as a configuration of the Rational Unified Process product adapted to the project's needs.

discipline

A logical grouping of roles, activities, artifacts, and other elements of guidance in the description of a process.

domain

An area of knowledge or activity characterized by a family of related systems.

elaboration

The second phase of the process, in which the product vision and its architecture are defined.

environment

A supporting discipline in the software engineering process whose purpose is to define and manage the environment in which the system is being developed; includes process descriptions, configuration management, and development tools.

evolution

The life of the software after its initial development cycle; any subsequent cycle during which the product evolves.

extended help

A RUP feature that allows associated software development tools to provide a context to the RUP, which will make the RUP display appropriate topics based on the given context.

framework

A micro-architecture that provides an incomplete template for applications within a specific domain.

implementation

A discipline in the software engineering process whose purpose is to implement and unit test the classes.

implementation model

A collection of components and the implementation subsystems that contain them.

implementation subsystem

A collection of components and other implementation subsystems; used to structure the implementation model by dividing it into smaller parts.

implementation view

An architectural view that describes the organization of the static software elements (code, data, and other accompanying artifacts) of the development environment, in terms of packaging, layering, and configuration management (ownership, release strategy, and so on). In the Rational Unified Process, it is a view on the implementation model.

inception

The first phase of the Rational Unified Process, in which the initial impetus for developing software is brought to the point of being funded (at least internally) to enter the elaboration phase.

increment

The difference (delta) between two releases at the end of subsequent iterations.

integration

The software development activity in which software components are combined into an executable whole.

iteration

A distinct sequence of activities with a plan and evaluation criteria that result in a release (internal or external).

layer

A specific way of grouping packages in a model at the same level of abstraction.

logical view

An architectural view that describes the main classes in the design of the system: major business-related classes and the classes that define key behavioral and structural mechanisms (persistency, communications, fault tolerance, and user interface). In the Rational Unified Process, the logical view is a view of the design model.

management

A supporting discipline in the software engineering process whose purpose is to plan and manage the development project.

measure

A numeric attribute of an entity.

method

(1) A regular and systematic way of accomplishing something; the detailed, logically ordered plans or procedures followed to accomplish a task or attain a goal. (2) UML 1.2: The implementation of an operation; the algorithm or procedure that affects the results of an operation.

metrics

(obsolete) Replaced by *measure* or *measurement* in RUP 2003.

milestone

The point at which an iteration or phase formally ends; corresponds to a release point.

model

A semantically closed abstraction of a system. In the Rational Unified Process, a complete description of a system from a perspective—"complete" meaning that you don't need additional information to understand the system from that perspective; a set of model elements.

model element

An element that is an abstraction drawn from the system being modeled.

modeler

See **[RUP Modeler]**

MyRUP

The RUP browser used to view the process, search, use the index, navigate graphically, and create personalized process views of a RUP Process Configuration.

node

A runtime physical object that represents a computational resource, generally having at least a memory and often processing capability. Runtime objects and components may reside on nodes.

object

An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, and behavior is represented by operations and methods. An object is an instance of a class.

object model

An abstraction of a system's implementation.

operation

A service that can be requested from an object to affect behavior.

organizer

See **[RUP Organizer]**

phase

The time between two major project milestones during which a well-defined set of objectives is met, artifacts are completed, and decisions are made to move or not to move into the next phase.

plug-in

A RUP Plug-In is a deployable unit for one or several RUP Process Components that can be readily "dropped" onto a RUP Base to extend it. A RUP Plug-In can be compiled into a physical file, allowing it to be moved around and added to a RUP Library with a compatible RUP Base.

process component

A RUP Process Component is a coherent, quasi-independent "chunk" or module of process knowledge that can be named, packaged, exchanged, and assembled with other process components.

process configuration

A RUP Process Configuration is a browsable set of RUP Process Components that constitute a complete process ("complete" from the perspective of a particular user). It is a Web site that sits on the user's machine or a server. RUP Process Configurations are compiled with RUP Builder.

Process View

A feature of MyRUP that allows you to customize the parts of the RUP Process Configuration, as well as external links, you want to see in your MyRUP tree browser. Process views can be role-based (for example, analyst, developer, tester) or personalized for the needs of a specific user.

prototype

A release that is not necessarily subject to change management and configuration control. A prototype may be throwaway or may evolve into becoming the final system.

quality

The characteristic of an artifact that satisfies or exceeds a defined and accepted set of requirements, is assessed using defined and accepted measures and criteria, and is produced using a defined and accepted process.

Rational Process Workbench (RPW)

A process authoring toolkit. It is a combination of tools, processes, and other assets that allow a process engineer to develop RUP Plug-Ins. The RPW includes **RUP Modeler, RUP Organizer,** a **RUP Library,** and process guidance for process authoring.

release

A subset of the end product that is the object of evaluation at a milestone. A release is a stable, executable version of the product, together with any artifacts necessary to use this release, such as release notes or installation instructions. A release can be internal or external. An internal release is used only by the development organization, as part of a milestone or for a demonstration to users or customers. An external release (or delivery) is delivered to end users.

report

An automatically generated description that describes one or several artifacts. A report is not in itself an artifact. A report is in most cases a transitory product of the development process and a vehicle to communicate certain aspects of the evolving system; it is a snapshot description of artifacts that are not themselves documents.

requirement

A description of a condition or capability of a system; either derived directly from user needs or stated in a contract, standard, specification, or other formally imposed document.

requirements discipline

A core discipline in the software engineering process whose purpose is to define what the system should do. The most significant activity is to develop a use-case model.

risk

An ongoing or impending concern that has a significant probability of adversely affecting the success of major milestones.

role

A definition of the behavior and responsibilities of an individual or a set of individuals working together as a team.

RPW

See **[Rational Process Workbench]**

RUP Base

A collection of RUP Process Components meant to be extended by applying plug-ins to generate RUP Process Configurations. It resides in a RUP Library.

RUP Builder

A tool within the RUP product used to create RUP Process Configurations out of a RUP Base and any number of RUP Plug-Ins.

RUP Exchange

A place on the Rational Developer Network where RUP Plug-Ins together with other process-related material are made available to the user community.

RUP Library

A collection of Process Components out of which a set of RUP Process Configurations may be compiled with RUP Builder. New process components can be added to a RUP Library through the means of RUP Plug-Ins.

RUP Modeler

A component of Rational Process Workbench. It allows a process engineer to visually model process elements such as activities, artifacts, roles, disciplines, and tool mentors, and their relationships; assemble them into RUP Process Components; and compile them into RUP Plug-Ins. RUP Modeler is an add-in to Rational XDE and works in conjunction with RUP Organizer.

RUP Organizer

A component of Rational Process Workbench. It allows you to associate content files with process elements such as activities, artifacts, roles, disciplines, and tool mentors and to compile these RUP Process Components to create a RUP Plug-In with the new or modified files. The files can be examples, guidelines, or reusable assets. RUP Organizer also allows you to modify Extended Help.

scenario

A described use-case instance or a subset of a use case.

sequence diagram

A diagram that describes a pattern of interaction among objects arranged in chronological order; it shows the objects participating in the interaction by their "lifelines" and the messages that they send to one another.

software architecture

Software architecture encompasses the following:

- The significant decisions about the organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements
- The composition of these elements into progressively larger subsystems; the architectural style that guides this organization, these elements, and their interfaces, their collaborations, and their composition

Software architecture is concerned with not only structure and behavior, but also usage, functionality, performance, resilience, reuse, comprehensibility, economic and technologic constraints and trade-offs, and aesthetic issues.

stakeholder

Any person or representative of an organization who has a stake—a vested interest—in the outcome of a project or whose opinion must be accommodated. A stakeholder can be an end user, a purchaser, a contractor, a developer, or a project manager.

step

A subset of an activity. Not all steps needs to be performed each time an activity is invoked.

stub

A dummy or skeletal implementation of a piece of code used temporarily to develop or test another piece of code that depends on it.

test

A core discipline in the software engineering process whose purpose is to integrate and test the system.

tool mentor

A description that provides practical guidance on how to perform specific process activities or steps using a specific software tool.

transition

The fourth phase of the process in which the software is turned over to the user community.

use case

A sequence of actions a system performs that yields an observable result of value to a particular actor. A use-case class contains all main, alternate, and exception flows of events related to producing the observable result of value. Technically, a use case is a class whose instances are scenarios.

use-case model

A model of what the system is supposed to do and the system environment.

use-case realization

A description of the way a particular use case is realized within the design model, in terms of collaborating objects.

use-case view

An architectural view that describes how critical use cases are performed in the system, focusing on architecturally significant components (objects, tasks, nodes). In the Rational Unified Process, it is a view of the use-case model.

version

A variant of an artifact; later versions of an artifact typically expand on earlier versions.

view

A simplified description (an abstraction) of a model that is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

See also **[architectural view]**

vision

The user's or customer's view of the product to be developed.

worker

(obsolete) Renamed **role** starting with RUP 2000.

workflow

The sequence of activities performed in a business that produces a result of observable value to an individual actor of the business. Core workflows as containers of process descriptions were renamed **disciplines** starting with RUP 2000.

Bibliography

This highly selective bibliography records the Rational Unified Process Development group's favorite books (and a few articles)—books that have had a major impact on the Rational Unified Process as it stands today, along with books that are complementary to the process and books published recently by our colleagues from Rational Software, and now IBM. Note that some books that could have appeared in several categories are listed only once

General

Brooks, Frederick P., Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering*, *Anniversary Edition*. Reading, MA: Addison-Wesley.
A classic that should be read and reread by everyone involved in software development. I recommend this edition rather than the original 1975 edition.

Davis, Alan. 1995. *201 Principles of Software Development* New York: McGraw-Hill.
Full of good advice for every worker.

Glass, Robert L. 2003. *Facts and Fallacies of Software Engineering* Boston: Addison-Wesley.

Katzenbach, Jon R., and Douglas K. Smith. 1993. *The Wisdom of Teams.* New York: HarperBusiness.
The secret of effective teams.

Yourdon, Edward. 1997. *Death March: Managing "Mission Impossible" Projects* Upper Saddle River, NJ: Prentice-Hall.
An interesting view of project troubles, which gives many justifications to our iterative and risk-driven approach.

Software Development Process

Boehm, Barry W. 1996. "Anchoring the Software Process." *IEEE Software*, July, pp. 73–82.
This article defines the four phases and the corresponding milestones.

Boehm, Barry W. 1998. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, May, pp. 61–72.
This seminal article defines the principles and motivations of iterative development.

Humphrey, W.S. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.
A classic book on the software process and the Capability Maturity Model developed at the Software Engineering Institute.

ISO/IEC 12207. 1995. *Information Technology—Software Life-Cycle Processes*; and ISO 9000-3. 1991. *Guidelines for the Application of ISO 9001 to the Development, Supply, and Maintenance of Software*. Geneva: ISO.
Two key standards for software process definition and assessment.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.
This textbook is a more detailed description of the principles of the Unified Process and is a useful companion to the Rational Unified Process. Also provides examples of UML modeling.

Jacobson, Ivar, Martin Griss, and Patrik Jonsson. 1997. *Software Reuse: Architecture, Process, and Organization for Business Success*. Reading, MA: Addison-Wesley.
This textbook on software reuse is a great complement to the Rational Unified Process. It also features some great chapters on architecture.

Kruchten, Philippe. 1991. "Un processus de développement de logiciel itératif et centré sur l'architecture." *Proceedings of the 4th International Conference on Software Engineering*. Toulouse, France, EC2, December.
The Rational iterative process explained in French.

Kruchten, Philippe. 1996. "A Rational Development Process." *CrossTalk* 9(7), pp. 11–16.
Developed with Walker Royce, Sue Mickel, and a score of Rational consultants, this article describes the iterative lifecycle of the Rational Process.

McFeeley, Robert. 1996. *IDEAL: A User's Guide for Software Process Improvement*. Pittsburgh, PA: Software Engineering Institute.
This book describes a software process improvement program model: IDEAL, a generic description of a sequence of recommended steps for initiating and managing a process implementation project.

Paulk, Mark, et al. 1993. *Capability Maturity Model for Software, Version 1.1*. Pittsburgh, PA: Software Engineering Institute.
The original reference for the capability maturity model.

Object-Oriented Technology

Booch, Grady. 1994. *Object-Oriented Analysis and Design with Applications, Second Edition*. Reading, MA: Addison-Wesley.

Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software* Reading, MA: Addison-Wesley.

Jacobson, Ivar, et al. 1992. *Object-Oriented Software Engineering: A Use-Case-Driven Approach*. Wokingham, UK: Addison-Wesley.

Larman, Craig. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Upper Saddle River, NJ: Prentice-Hall.
An alternate set of artifacts, activities, and guidelines in a process framework not too different from ours.

Rumbaugh, James, et al. 1991. *Object-Oriented Modeling and Design* Englewood Cliffs, NJ: Prentice-Hall.
These three books, the original roots of the object-oriented analysis and design workflow, were written by "the three amigos" just before the advent of the UML and the Rational Unified Process. Despite the use of their original notations, these books are still the key references for the OO designer.

Rumbaugh, James. 1996. *OMT Insights*. New York: SIGS Books.
A complement to the original OMT book, this dives into special topics: inheritance, use cases, and so on.

Selic, Bran, Garth Gullekson, and Paul Ward. 1994. *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons.
The reference work on using object technology for the design of reactive systems by the people who developed Rose for Real-Time.

Modeling and the Unified Modeling Language

Booch, G., J. Rumbaugh, and I. Jacobson. 1998. *The Unified Modeling Language*. Documentation set, version 1.3. Cupertino, CA: Rational Software.
The latest OMG standard on the UML at the time of this writing. Available online at www.rational.com.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
Published at the same time as the Rational Unified Process 5.0, this manual is an excellent user's guide on UML by its main authors.

Eriksson, Hans-Erik, and Magnus Penker. 1997. *UML Toolkit*. New York: John Wiley & Sons.
A comprehensive book on UML as seen from Sweden by another pair of Rational friends.

Fowler, Martin. 1999. *UML Distilled: Applying the Standard Object Modeling Language*, *Second Edition*. Reading, MA: Addison-Wesley.
A very nice little introduction to UML if you are in a hurry.

Muller, Pierre-Alain. 1998. *Instant UML*. Chicago: Wrox, Inc.
Another short introduction to UML by a former colleague of ours.

Quatrani, Terry. 1998. *Visual Modeling with Rational Rose and UML* Reading, MA: Addison-Wesley.
Provides step-by-step guidance on how to build UML models. At the same time, it follows the Rational Unified Process for which this book provides, in effect, a small-scale example.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.
Certainly more digestible than the OMG standard, this book is an in-depth examination of UML by its main authors.

Project Management

Boehm, Barry W. 1991. "Software Risk Management: Principles and Practices." *IEEE Software*, Jan., pp. 32–41.
Still the best little introduction to risk management.

Booch, Grady. 1996. *Object Solutions: Managing the Object-Oriented Project* Reading, MA: Addison-Wesley.
A pragmatic book for managers of object-oriented projects; one of the sources of the underlying philosophy of the Rational Unified Process.

Cantor, Murray. 2002. *Software Leadership: A Guide to Successful Software Development* Boston: Addison-Wesley.
A nice little book on the topic of driving software projects full of concrete and wise advice.

Carr Marvin J., et al. 1993. *Taxonomy-Based Risk Identification*. Technical report CMU/SEI-93-TR- 6, SEI, June.
A source of inspiration to get started on your own list of risks.

Charette, Robert. 1989. *Software Engineering Risk Analysis and Management* New York: McGraw-Hill.
Practical perspective on risk management.

Fairley, Richard. 1994. "Risk Management for Software Projects." *IEEE Software* 11(3), May, pp. 57–67.
Straightforward strategy for risk management if you have never done it.

Gilb, Tom. 1988. *Principles of Software Engineering Management*. Harlow, UK: Addison-Wesley.
A great book by a pioneer of iterative development, full of pragmatic advice for the project manager.

Jones, Capers. 1994. *Assessment and Control of Software Risks* Englewood Cliffs, NJ: Yourdon Press.
An indispensable source of risks to check to make sure your list is complete.

McConnell, Steve. 1998. *Software Project Survival Guide* Redmond, WA: Microsoft Press.

O'Connell, Fergus. 1994. *How to Run Successful Projects*. Upper Saddle River, NJ: Prentice-Hall International.
A real gem. Everything you really need to know to manage your first project, in 170 pages.

Royce, Walker. 1998. *Software Project Management: A Unified Framework* Reading, MA: Addison-Wesley.
An indispensable companion to the Rational Unified Process, this book describes the spirit and the underlying software economics of the Rational Unified Process. It is full of great advice for the project manager.

Requirements Management

Bittner, Kurt, and Ian Spence. 2003.*Use Case Modeling*. Boston: Addison-Wesley.
Our colleagues' thorough and masterful treatise of use cases.

Cockburn, Alistair. 2001.*Writing Effective Use Cases*. Boston: Addison-Wesley.
For an alternative, and often complementary, view to that of Bittner and Spence's.

IEEE Std. 830-1998. 1998.*Recommended Practice for Software Requirements Specifications*. New York: Software Engineering Standards Committee of the IEEE Computer Society.

Leffingwell, Dean, and D. Widrig. 1999.*Managing Software Requirements: A Unified Approach* Reading, MA: Addison-Wesley.
Grounded in years of practical experience with numerous projects in a wide range of application, this book extends the requirements workflow with practical tips, examples, case studies, and war stories.

Configuration Management

Berlack, H. 1992. *Software Configuration Management.* New York: John Wiley & Sons.

Buckley, F. 1993. *Implementing Configuration Management: Hardware, Software and Firmware*. Los Alamitos, CA: IEEE Computer Science Press.

White, Brian. 2000. *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*. Boston: Addison-Wesley.
Provides the philosophy and history of configuration management and describes a simple but powerful usage pattern adaptable to a large set of project situations.

Whitgift, David. 1991. *Methods and Tools for Software Configuration Management.* New York: John Wiley & Sons.

Testing and Quality

Beizer, Boris. 1995. *Black Box Testing*. New York: John Wiley & Sons.
A treasure of strategies to develop test cases for the functional testing of software. Dr. Beizer's writing style (and wit) makes this book easy and fun to read, with excellent, understandable examples.

Binder, Robert V. 2000. *Testing Object-Oriented Systems: Models, Patterns, and Tools* Boston: Addison-Wesley.

Black, Rex. 1999. *Managing the Testing Process*. Redmond, WA: Microsoft Press.

IEEE 829-1983. 1983. *Standard for Software Test Documentation*. New York: Software Engineering Standards Committee of the IEEE Computer Society.

Kaner, Cem, James Bach, and Bret Pettichord. 2001 *Lessons Learned in Software Testing* New York: John Wiley & Sons.

Kaner, Cem, Jack Falk, and Hung Quoc Nguyen. 1999. *Testing Computer Software*, *Second Edition*. New York: John Wiley & Sons.

Perry, William E. 1995. *Effective Methods for Software Testing*. New York: J. Wiley/QED Press.

Software Architecture

Bass, Len, Paul Clements, and Rick Kazman. 1998. *Software Architecture in Practice*. Reading, MA: Addison-Wesley.
A handbook of software architecture, with numerous case studies.

Buschmann, Frank, Régine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stahl. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: John Wiley & Sons.
This book makes an inventory of a wide range of design patterns at the level of the architecture.

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. 2002. *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley.

Dikel, David M., David Kane, and James R. Wilson. 2001. *Software Architecture: Organizational Principles and Patterns*. Upper Saddle River, NJ: Prentice-Hall.
They capture the role and activities of the architect in the VRAPS model: Vision, Rhythm, Anticipation, Partnering, and Simplification.

Hofmeister, Christine, Robert Nord, and Dilip Soni. 2000. *Applied Software Architecture*. Boston: Addison-Wesley.
The architectural design approach these authors recommend is very similar to that of the Rational Unified Process and is based on multiple coordinated views.

IEEE 1471-200 Recommended Practice for Architectural Description.
This standard recommends architectural description based on the concept of multiple views.

Kruchten, Philippe. 1995. "The 4+1 View Model of Architecture." *IEEE Software* 12(6).
The origin of the 4+1 views used for architectural description in the Rational Unified Process.

Rechtin, Eberhardt. 1991. *Systems Architecting: Creating and Building Complex Systems*. Englewood Cliffs, NJ: Prentice-Hall; and Rechtin, Eberhard, and Mark Maier. 1997. *The Art of System Architecting*. Boca Raton, FL: CRC Press.
Although not directed specifically to software engineers, these two books are extremely valuable for software architects: They introduce an invaluable set of heuristics and many examples of architecture.

Business Engineering

Eriksson, Hans-Erik, and Magnus Penker. 1999. *Business Modeling with UML* New York: John Wiley & Sons.
An alternative approach to that of the RUP, but the business patterns they propose are a very valuable complement to the business engineering workflow.

Hammer, Michael, and James Champy. 1993. *Reengineering the Corporation: A Manifesto for Business Revolution*. New York: HarperBusiness.
The book that popularized the movement of business (re-)engineering. An excellent complement to the Jacobson book.

Jacobson, Ivar, Maria Ericsson, and Agneta Jacobson. 1994. *The Object Advantage: Business Process Reengineering with Object Technology*. Reading, MA: Addison-Wesley.
The basis of the business modeling workflow, this is the first book that applied object technology to the field of business engineering.

Others

Ambler, Scott. 1998. *Process Patterns*. New York: SIGS Books/CUP.

Ambler, Scott. 1999. *More Process Patterns*. New York: SIGS Books/CUP.

Bergstrom, Stefan, and Lotta Råberg. 2003. *Implementing the Rational Unified Process: Success with the RUP*. Boston: Addison-Wesley.
An entire book dedicated to the delicate topic of process change and process improvement using the RUP. How to assess the organization and plan a successful process implementation project.

Cockburn, Alistair. 2002. *Agile Software Development*. Boston: Addison-Wesley.

Constantine, Larry, and Lucy Lockwood. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Reading, MA: Addison-Wesley.
A practical guide to the models and methods of usage-centered design, or how user-interface design connects with use cases.

DeGrace, Peter, and Leslie Stahl. 1990. *Wicked Problems, Righteous Solutions: A Catalog of Modern Software Engineering Practices*. Englewood Cliffs, NJ: Yourdon Press.
An insightful book on various process lifecycles, their origins, their flaws, and their strengths; useful for an understanding of the importance of process.

Eeles, Peter, Kelli Houston, and Wojtek Kozaczynski. 2003. *Building J2EE Applications with the Rational Unified Process*. Boston: Addison-Wesley.
The application of RUP for the development of large Java application; this book provides simultaneously an example of RUP, RUP artifacts, and an introduction to J2EE.

Graham, Ian, Brian Henderson-Sellers, and Houman Younessi. 1997. *The Open Process Specification*. Harlow, UK: Addison-Wesley.
Another process model, this one coming from down under, that shares some principles with the Rational Unified Process.

Highsmith, James A. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.

IBM. 1997. *Developing Object-Oriented Software: An Experience-Based Approach*. Upper Saddle River, NJ: Prentice-Hall.
As with the Rational Unified Process, this book describes an iterative, incremental, object-oriented, scenario-driven, risk-aware process developed by IBM's Object Technology Center.

Kettani, Nasser, et al. 1998. *De Merise à UML*. Paris: Editions Eyrolles.
Merise, a popular software development methodology in France, has been upgraded to use UML. It has some similarity to the Rational Unified Process.

McCarthy, J. 1995. *Dynamics of Software Development*. Redmond, WA: Microsoft Press.
Fifty-three rules of thumb by a Microsoft development manager.

McConnell, Steve. 1993. *Code Complete: A Practical Handbook of Software Construction*. Redmond, WA: Microsoft Press.
A great book for the implementer and test workers, this looks at the implementation, integration, and test aspects of the development process.

Robillard, Pierre, and Philippe Kruchten. 2003. *Software Engineering Process with the UPEDU*. Boston: Addison-Wesley.
This textbook describes the unified process for education, a simple derivative of the RUP used in education. See also the Web sites www.upedu.org and www.yoopeedoo.org.

Schwaber, Ken, and Mike Beedle. 2002. *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice-Hall.

Stapleton, Jennifer. 1997. *The Dynamic System Development Method*. Reading, MA: Addison-Wesley.
At 15,000 feet, the DSDM approach could be seen as an introduction to the Rational Unified Process. Although they use different terminologies, the two processes are very close to each other, and you can see the Rational Unified Process as an instance or an

implementation of DSDM.

Brought to You by