

Real-Time UML Workshop for Embedded Systems

Second Edition

by Bruce Powel Douglass, Ph.D.



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, UK

Second edition 2014

Copyright © 2014 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publisher. Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone (+44) (0) 1865 843830; fax (+44) (0) 1865 853333; email: permissions@elsevier.com. Alternatively you can submit your request online by visiting the Elsevier web site at <http://elsevier.com/locate/permissions>, and selecting Obtaining permission to use Elsevier material.

Notice

No responsibility is assumed by the publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein. Because of rapid advances in the medical sciences, in particular, independent verification of diagnoses and drug dosages should be made.

Library of Congress Cataloging-in-Publication Data

A catalog record for this book is available from the Library of Congress

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-12-407781-2

For information on all Newnes publications visit
our web site at store.elsevier.com

Printed and bound in USA

14 15 16 17 18 10 9 8 7 6 5 4 3 2 1



Working together
to grow libraries in
developing countries

www.elsevier.com • www.bookaid.org

*This book is dedicated to my family – two boys who make me proud,
a wife any man would envy, and a stepdaughter who is so bright,
she's scary. It's good to be me – thanks guys!*

Preface

Most books on UML or real-time systems, even those that I have written, are what I call “lecture books.” They might have exercises at the end of the chapters, but the focus and organization of the books are “Here, let me tell you about that” This book is fundamentally different. It is meant to be a workbook, a set of guided exercises that teach by practice and example, not by exposition. As such, we won’t discuss, in detail, various aspects of UML, or MDA, or even process. We will mention such things in passing, but the expectation is that the reader already has a reasonable understanding of UML and wants practice and experience in the application of UML to embedded and real-time systems development. This book is meant as a companion to my other book, *Real-Time UML: Advances in the UML for Real-Time Systems*.¹ That book is a lecture book and explains, in some detail, the structural, behavioral, and functional aspects of UML, particularly how it applies to the development of real-time and embedded systems.

The present book is organized primarily around two example problems, whose detailed problem statements can be found in the appendices. To provide some breadth of experience in the exercises, both small-scale and large-scale applications are provided. The small-scale example is an intersection traffic light control system. This is a small-scale system, but is still complex enough to be a rich source of work for the reader.² The larger-scale example is an unmanned aerial vehicle (UAV), although it is more properly called an unmanned combat air vehicle (UCAV). This system is highly complex and offers good experience in doing requirements analysis and architectural design for large-scale systems, something the traffic light control system cannot. The point of these examples is to provide a context for the reader to explore the application of UML to the specification, analysis, and design of such systems. It is most assuredly *not* to provide a complete example of such systems. Such systems are created by teams (or, as in the case of the UCAV, teams of teams). If the purpose of the book was to provide complete examples of such systems, the book would be somewhere between 20 and 100 times its present size.

¹ Douglass, Bruce Powel. *Real-Time UML: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004).

² One of the Laws of Douglass is “Anything is simple if you don’t have to do it!” This certainly applies in spades to the development of real-time systems, since even “simple” systems can be very complex.

This book can be thought of as being composed of several parts. First is a brief overview of the basics – UML and process – provided in Chapters 1 and 2, followed by a brief description of some of the more common industry standards with which our systems designs and process must comply. The next section is a series of chapters that walk through the phases in an incremental spiral process – requirements analysis, object analysis, architectural design, mechanistic design, and detailed design – posing problems that occur in the context of the problem. The next section provides solutions for the problems posed in the previous section. Lastly, there is some end matter – the appendices that provide the problem statements, and a brief notational guide to UML.

All the UML diagrams in this book are created with the Rhapsody tool from IBM Rational, and a time-limited demo copy of the book can be download from IBM at <http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/>. There is even a reduced functionality but free version of the tool, known as Rhapsody Modeler, that can be downloaded at <http://www.ibm.com/developerworks/downloads/r/modeler/>. The full version offers validation, model execution, and production-quality code generation facilities, as well as interfaces to other tools, such as requirements management, XMI import/export, etc. While these features are enormously helpful in actual development, they are not required to the purpose of the book. If you prefer to use another tool for the exercises, or even (gasp) pen and paper, you can do that as well.

The solutions are, of course, the “pointy end” of the stick. They are meant, not so much as the absolute truth or measure of goodness in UML modeling, but instead as examples of solutions to the problem. If your solution differs from the one provided and still meets the solution criteria, then that’s perfectly fine. There are always many good solutions to any problem.³ If your solution differs from the provided solution, I recommend that you study the differences between the two and under which conditions one or the other would be preferable.

Audience

The book is oriented towards the practicing professional software developer and the computer science major, in the junior or senior year. This focuses on practical experience by solving posed problems from the context of the examples. The book assumes a reasonable proficiency in UML and at least one programming language (C++ is used in this book, but C, Java, Ada, or any equivalent 3GL can be used). This book is meant as a companion to the author’s *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems*, but that book is not required to use this workbook. Other books that may help the reader by providing more depth in various topic areas include the author’s *Doing Hard Time: Developing Systems with UML, Objects, Frameworks and Patterns* (Addison-Wesley, 1999) *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2002), *Real-Time*

³ There are always even more *bad* solutions to any problem as well!

Agility (Addison-Wesley 2009) and Design Patterns for Embedded Systems in C” (Elsevier Press 2011).

Goals

The goal of this book is simply to give the reader practical experience in modeling different aspects of real-time and embedded systems. While there is some introduction early in the first two chapters of the book and at the beginning of the problem chapters, the goal is to pose problems that the reader will go off and model.

By the time you’ve reached the end of the book, you will hopefully have the expertise to tackle the real modeling issues that you face in your life as a professional developer of real-time and embedded systems.

Where to Go after the Book

If you’re interested in tools, training, or consulting, see www.ibm.com. Bruce teaches advanced classes and consults worldwide on UML, MDA, DoDAF, architectural design, design patterns, requirements modeling, use cases, safety-critical development with UML, behavioral modeling, development process improvement, project management and scheduling, and quite a bit more. You can contact him for training or consulting services at Bruce.Douglass@us.ibm.com. He also runs a (free) Yahoo! group forum as well at <http://groups.yahoo.com/group/RT-UML> - come on down! Bruce’s IBM Thought Leader page (<http://www-01.ibm.com/software/rational/leadership/thought/brucedouglass.html>) also has many white papers available for downloading on different topics that may be of interest.

Bruce Powel Douglass, PhD
Summer 2013

Acknowledgements

I wish to express my deepest thanks to my wife Sarah for supporting me while burning the midnight electrons working on this book.

About the Author

Bruce was raised by wolves in the Oregon wilderness. He taught himself to read at age 3 and calculus before age 12. He dropped out of school when he was 14 and traveled around the United States for a few years before entering the University of Oregon as a mathematics major. He eventually received his MS in exercise physiology from the University of Oregon and his PhD in neurocybernetics from the USD Medical School, where he developed a branch of mathematics called autocorrelative factor analysis for studying information processing in multicellular biological neural systems.

Bruce has worked as a systems engineer and software developer in real-time systems for over 30 years and is a well-known speaker, author, and consultant in the area of real-time embedded systems. He has been on the Advisory Board of the *Embedded Systems* and *UML World* conferences where he has taught courses in software estimation and scheduling, project management, object-oriented analysis and design, communications protocols, finite state machines, design patterns, and safety-critical systems design. He develops and teaches courses and consults in real-time object-oriented analysis and design and project management and has done so for many years. He has authored articles for many journals and periodicals, especially in the real-time domain.

He is the Chief Evangelist (being a Chief Evangelist is much like being a Chief Scientist, except for the burning bushes and stone tablets) for IBM Rational (formerly I-Logix), a leading producer of tools for real-time systems development. Bruce worked with Rational and the other UML partners on the specification of the UML and SysML standards. He developed the first UML Profile for DoDAF for the Rhapsody™ modeling tool as well as other profiles such as a Safety Analysis Profile and a Security Analysis Profile. He has been a co-chair of the Object Management Group's Real-Time Analysis and Design Working Group. He is the author of several other books on software, including *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns* (Addison-Wesley, 1999), *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2002), *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004), *Real-Time Agility* (Addison-Wesley, 2009), *Design Patterns for Embedded Systems in C* (Elsevier, 2011), and several others, including a short textbook on table tennis.

About the Author

Bruce enjoys classical music and has played classical guitar professionally. He has competed in several sports, including table tennis, bicycle racing, running, and full-contact Tae Kwon Do, although he currently only fights inanimate objects that don't hit back. He has recently gotten back into racing triathlons and ultramarathon cycling with plans to do an Ironman triathlon in 2014.

Bruce does extensive consulting and training throughout the world. If you're interested, contact him at Bruce.Douglass@us.ibm.com.

Introduction to UML

What You Will Learn

- Basic modeling concepts and overview of UML
- Class and object models
 - What are classes and objects? How do classes and objects work together in collaborations? Collaborations are the realizations of use cases. This includes the packaging of logical elements.
- Component and deployment models
 - Representing run-time artifacts and localizing them on processor nodes.
- State machines and behavioral models
 - What are state machines and how do they model behavior?
- Use case and requirements models
 - Capturing black-box behavior without revealing internal structure.

1.1 UML Basic Modeling Concepts

The Unified Modeling Language (UML) is a third-generation object modeling language standard, owned by the Object Management Group (OMG). The initial version of the OMG UML standard, 1.1, was released in November 1997. Since then, a number of minor revisions and one major revision have been made. As of this writing, the current version of the standard is 2.4.1 and it is available from the OMG at www.omg.org.

UML is a rich language for modeling both software and systems, and is the *de facto* standard for software modeling. There are a number of reasons for this, and it is the totality of all of them which, I believe, accounts for the phenomenal success of UML. UML is first of all, relatively easy to learn, and once learned, relatively intuitive. Secondly, UML is well defined, and models written in UML can be verifiable (if care is taken to be precise), so not only can the models be directly executed (with appropriate tools, such as Rational Rhapsody™), but production-quality code can be generated from them. Third, there is *great* tool support; not only are there many vendors, but the vendors have distinguished themselves in the market by emphasizing different aspects of modeling and development.

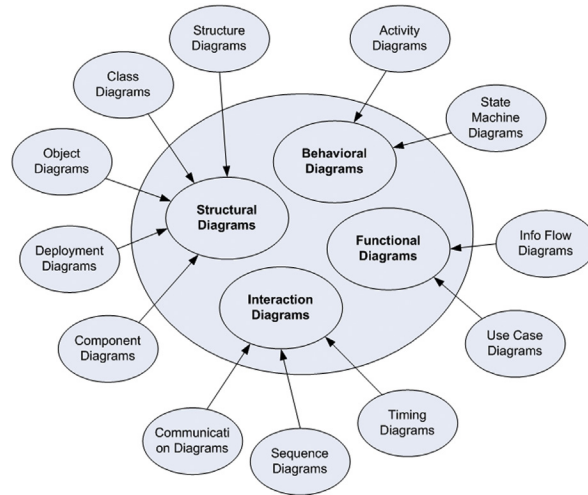


Figure 1.1
UML diagram types.

The notation used by UML is graphical in nature, easy to master and, for the most part, simple to understand.¹ Although some people claim that UML has *too many diagrams*, in reality there are only four basic types (see [Figure 1.1](#)). First there are structural diagrams, including class, structure, object, package, component, and deployment diagrams. Really these are all basically differentiated not on the contents of the diagram but on their *purpose*. Functional diagram emphasize functionality but not structure or behavior; functional diagrams include use case and information flow diagrams. Interaction diagrams focus on how elements collaborate together over time to achieve functional goals; interaction diagrams include sequence, communication (formerly known as “collaboration” diagrams), and timing diagrams. Lastly, behavioral diagrams focus on specification of behavior; these include state machine and activity diagrams. Although the breadth of the notation can be a bit overwhelming to newcomers, in reality, complex system can be easily developed with three core diagrams – class diagrams, state diagrams, and sequence diagrams. The other diagrams can be used to model additional aspects of the system (such as capturing requirements, or how the software maps onto the underlying hardware). The additional diagrams certainly add value, but in practice, you only need the three basic diagram types to develop systems and software.

UML has a well-defined underlying semantic model, called the UML metamodel. This semantic model is both broad (covering most of the aspects necessary for the specification and design of systems) and deep (meaning that it is possible to create models that are both *precise and executable* and can be used to generate source-level code for compilation). The upshot is that the developer can fairly easily model any aspect of the system that he or she

¹ Although like all useful languages, it has both complex and subtle points as well.

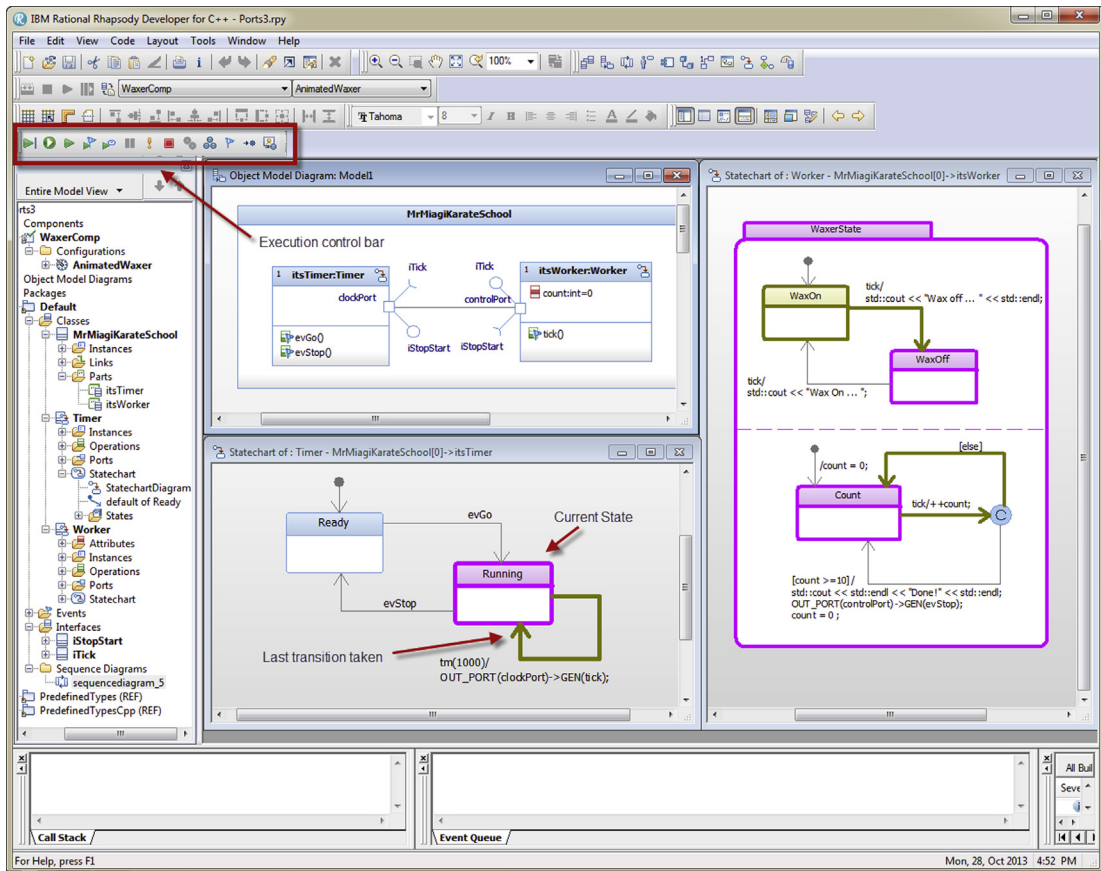


Figure 1.2
Model execution.

needs to understand and represent. [Figure 1.2](#) shows a screenshot of a model. Rhapsody uses color-coding to depict aspects such as the current state, but color-coding doesn't show up well in a black-and-white image. You can see the execution controls in the figure for step-into, step-over, setting breakpoints, inserting events, and so on. Rhapsody can also dynamically create sequence diagrams that show the history of the interaction of specified objects as they run.²

UML is a *standard*, unlike most other modeling languages, which are both proprietary and single-sourced. Using a standard modeling language means that the developer can select both tools and services from many different sources. For example, there are at least a couple of dozen different UML modeling tools. Since the tools focus on different aspects of modeling and have different price points, developers can find and select tools that best meet their own

² The models in this book were created in Rational Rhapsody but other tools can be used as well.

and their project's needs. For example, Rhapsody™ from IBM Rational emphasizes the deep semantics of UML, allowing the validation and testing of the user's models via execution and debugging using UML notation. This execution can take place on the host development machine or on the final target hardware, and the generated code can then be used in the final delivered system. Other tools emphasize other aspects, such as perhaps drawing the diagrams, but permit more flexibility for a lower price point. The availability of so many different tools in the market gives the developer a great deal of latitude in tool selection. It also encourages innovation and improvement in the tools themselves. Because UML is such a well-adopted standard, many companies provide training in the use and application of UML. I spend a great deal of my time training and consulting all over the world, focusing on using UML in embedded real-time systems and software development.³

UML is *applicable* to the development of software and systems in many different application domains. By now, UML has been used in the development of virtually every kind of software-intensive system from inventory systems to flight control software. Being a maintained standard, the standard itself evolves over time, repairing defects, adopting good ideas, and discarding ones that didn't pan out. UML is used today to model and build systems that vary in scope from simple one- or two-person projects up to ones employing literally thousands of developers. UML supports *all* the things necessary to model timeliness and resource management in real-time and embedded systems. That means that the developer need not leave UML to design the different aspects of their system, regardless of how complex or arcane those things might be.

In this chapter, we'll introduce the basics of UML. *This is intended more as a refresher than a tutorial.* Those desiring a more in-depth treatment of UML itself should pick up the companion book to this volume – *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004) by Bruce Powel Douglass. Additionally, there are many white papers available on my IBM Thought Leaders page, <http://www-01.ibm.com/software/rational/leadership/thought/brucedouglass.html>.

1.2 Structural Elements and Diagrams

UML has a rather rich set of structural elements, and provides diagrammatic views for related sets of them. Rather arbitrarily, we'll discuss modeling small, simple elements first, then discuss large-scale elements and model organization afterwards.

1.2.1 Small Things: Objects, Classes, and Interfaces

There are a number of elementary structural concepts in UML that show up in user models: object, class, data type, and interface. These structural elements form the basis of the

³ Interested readers looking for professional services can contact me at bruce.douglass@us.ibm.com.

structural design of the user model. In its simplest form, an *object* is a data structure that also provides services that act on that data. An object only exists at runtime, that is, while the system is executing, an object may occupy some location in memory at some specific time. The data parts of an object are stored in its *attributes* – whether they are simple, primitive variables or complex objects themselves. The services that act on that data are called *methods*; these are the services invoked by clients of that object (typically other objects) or by other methods existing within the object. State machines and activity diagrams may enforce specific sequencing of these services when pre- and postconditions must be met.

A class is the specification of a set of objects that share a common structure and behavior. Objects are said to be *instances* of the class. A class may have many instances in the system during runtime but each object is an instance of only a single class. A class may also specify a state diagram, which coordinates and manages the execution of its primitive behaviors (called *actions*, which are often invocations of the methods defined in the class) into allowable sets of sequences.

If you look at [Figure 1.3](#), you can see a basic class diagram. Note that objects are not shown – that is because when you show objects you are showing a snapshot of a running system at an instant of time. Class diagrams represent a set of possible object structures. Most often, when

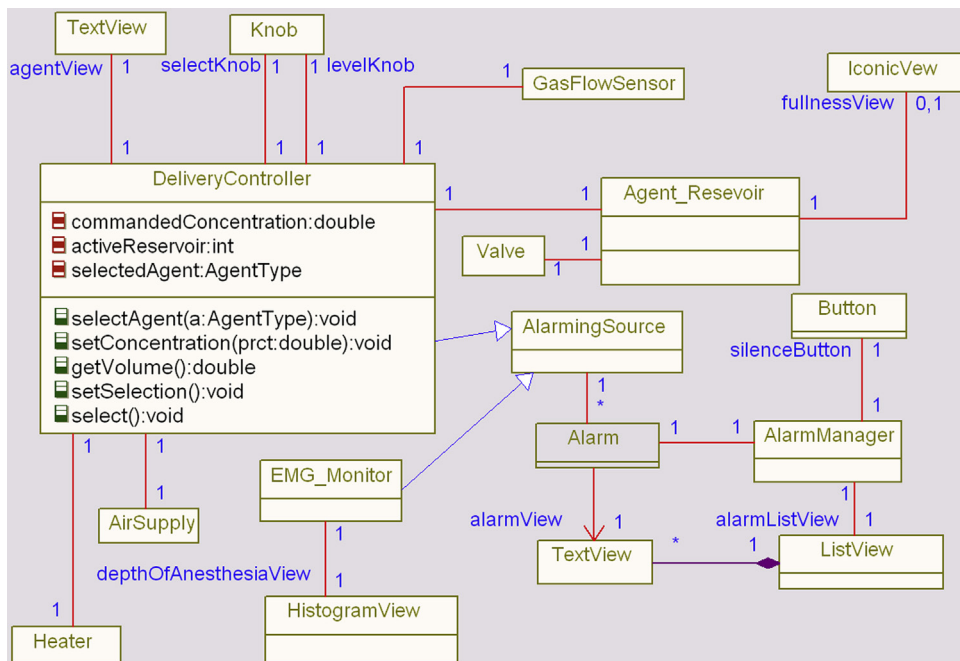


Figure 1.3
Basic class diagram.

you create structural views, you will be usually more interested in creating class, rather than object, diagrams.

The `DeliveryController` object in the figure is an example class. It contains attributes, such as `commandedConcentration` (which is of type `double`) and the selected agent (i.e., drug) type. It provides methods, such as the ability to select an agent, to get the amount of drug remaining, and to set the commanded drug concentration. The `DeliveryController` class is shown as a standard box with three segments. The top segment holds the name of the class. The middle segment holds a list of the attributes. Note, however, that this list need not be complete – not all of the attributes owned by the class must be shown. The bottom compartment shows the methods of invocable services provided by the class.

Figure 1.3 shows other classes as well, and lines (called *relations* – more on that later) connecting them. Some of these are shown in a simple, nonsegmented box, such as the `TextView` class. The attributes and methods – collectively known as *features* of the class – need not be shown. They are contained within the model and easily visible in the model browser view, and may be exposed on the diagram if desired. But you have control over which features are shown on which diagrams.

An *interface* is a named collection of services. Services come in two basic flavors. *Operations* are synchronous services that are invoked, or *called* by clients. *Event receptions* are asynchronous services that are invoked by sending an asynchronous signal to the object accepting that event. An interface can contain either or both operations and event receptions. Both operations and event signals can contain data, called *parameters*, passed with the invocation. The set of parameters, along with the name of the service, is called the *signature* of the service. A class that is compliant with an interface provides a method for every operation and an event reception for every signal specified in the interface.

Interfaces allow you to separate the specification of services that may be called on a class from the implementation of those services. A class defines methods (for operations) and event receptions (for signals, specified on the state machine). Both methods and event receptions include the lines of code that implement the service. An operation or signal is a specification only, and does not include such implementation detail.

Interfaces may not have implementation (either attributes or methods) and are not directly instantiable. A class is said to *realize* an interface if it provides a method for every operation specified in the interface, and those methods have the same names, parameters, return values, preconditions, and postconditions of the corresponding operations in the interface.

Interfaces may be shown in two forms. One looks like a class except for the keyword `interface` placed inside guillemots, as in `«interface»`. This form, called a *stereotype* in UML, is used when you want to show the service details of the interface. The other form, commonly referred to as the “lollipop” notation, is a small named circle on the side of the class. Both forms are shown in

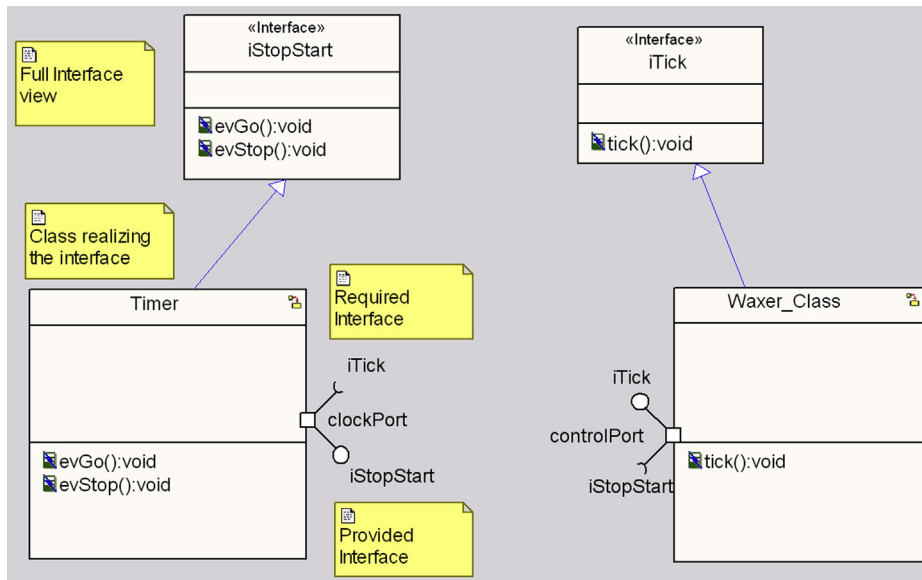


Figure 1.4
Interfaces.

Figure 1.4. When the lollipop is used, only the name of the interface is apparent. When the stereotyped form is used, a list of operations of the interface may be shown.

In **Figure 1.4**, two interfaces are shown, both of which happen to, in this case, only provide asynchronous event signals. The generalization arrow indicates the class that realizes the interface, that is, provides the services. Also shown are ports on the classes (more on ports later). The port is typed by the interfaces that are either offered or required across that port. In this case, the `clockPort`, a feature of the `Timer` class, offers the `iStopStart` interface services and requires the `iTick` interface. That means that to connect to that port, the other class must provide the `iTick` interface and require the `iStopStart` interface. The lollipop notation shows the provided (also known as “offered”) interface while the socket notation indicates the required interface. This kind of modeling allows the developer to specify contracts that enable compatible elements to be linked together and is important, especially in architectural and component design.

In summary, an object is one of possible many *instances* of a class. A class has two notable features – attributes (which store data values) and methods (which provide services to clients of the class). Interfaces are named collections of operations that are *realized* by classes. Interfaces need not be explicitly modeled but are intentionally exposed in some styles of modeling. Many useful systems have been designed solely with classes, but there are times when the addition level of abstraction is useful, particularly when more than a single implementation of an interface will be provided.

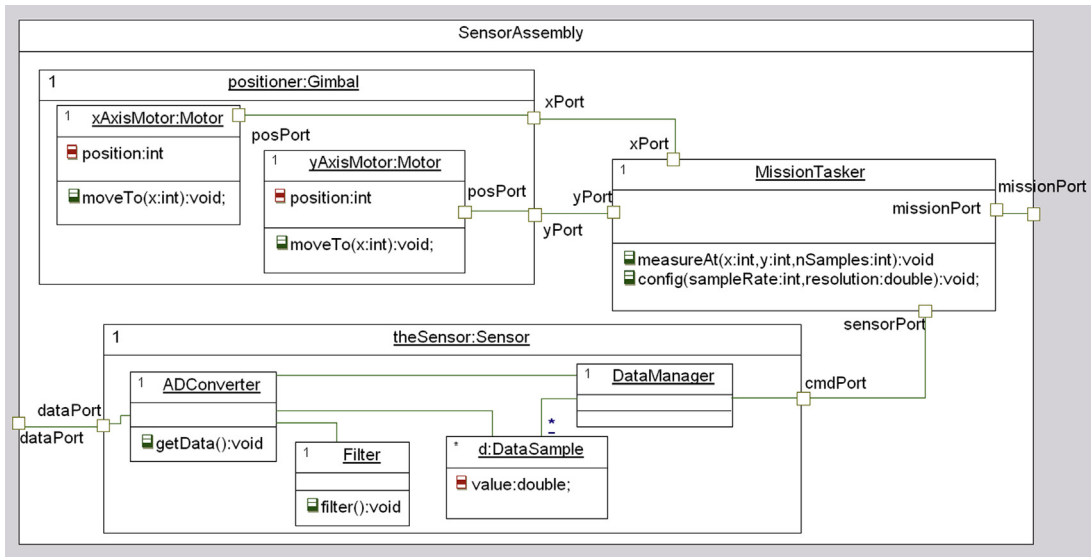


Figure 1.5
Structured classes.

Structured Classes

One of the significant extensions in UML 2.0 is the notion of a *structured class*. A structured class is a class that is composed of *parts* (object roles) that are themselves typed by classes. Structured classes still have all the features and properties (such as attributes and methods) of normal classes, but they internally own and orchestrate their internal parts. This is a huge advancement in the UML standard because it allows the explicit specification of architectural containment structures. For example, systems contain subsystems as parts; these subsystems can contain sub-subsystems, and so on down to primitive (nonstructured) parts. In addition, since the common means for representing concurrency units (e.g., threads or tasks) is with active objects, these active objects can contain the passive (nonconcurrent) objects as parts, allowing the clear specification of the concurrency architecture.

Figure 1.5 shows a typical structured class called `SensorAssembly`. This class contains internal parts, such as a `positioner` part (typed by class `Gimbal`), a `MissionTasker` part (its class isn't exposed on the diagram) and a `theSensor` part (typed by class `Sensor`). Some of these parts are themselves structured classes and contain subparts. In this way, we can construct arbitrarily complex containment hierarchies.

A couple of things should be noted in Figure 1.5. First of all, the relation between a class and its parts is not a relation between classes – it definitely implies a composition relation between the classes, but the parts are not themselves classes. The class of the part might define different parts in different structured classes as well. Neither are parts exactly objects.

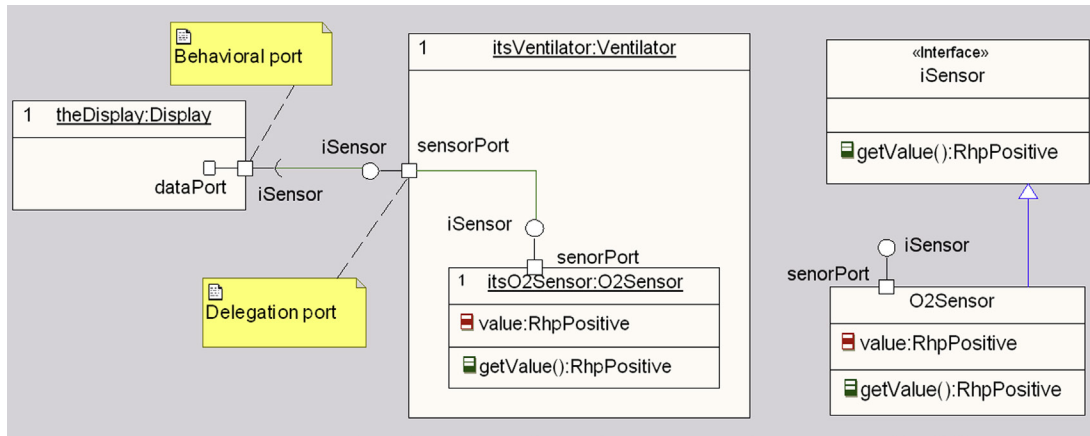


Figure 1.6
Ports.

Every `SensorAssembly` has this same structure, so it's not referring to a specific singular object for its positioner. When we actually create an instance of the `SensorAssembly`, a specific singular instance of class `Gimbal` will play the role that this part specifies. But the part is not itself an instance. It is really a role that some instance will play when the system runs, and is typed by a class specification. UML provides the term *part* for exactly this kind of object role.

The second thing to note is the presence of small squares on various class or part boxes. These small boxes are called *ports* and are discussed next.

Ports

Classes may also have features called *ports* (see Figure 1.6). Ports are named connection points of classes, and as mentioned before, are typed by the services offered by and required from them. *Behavioral ports* terminate on the part that actually supplies the service, while *delegation ports* pass the request on to another element for fulfillment. Ports have the advantages of making the interfaces supplied by the parts clear and explicit while at the same time enforcing the encapsulation of the services. That is, a client of a service provided by a port does not know (or care) which internal object (or even class) provides the service. This means that the provider of that service may undergo significant modification or even replacement with a different class, so long as the interface is retained. This can significantly improve reusability and maintainability in complex systems.

1.2.2 Relations

To achieve system-wide behavior, many objects must work together, and to work together, they must relate in some way. As part of structural modeling, UML defines different kinds of relations among structural elements. We will focus primarily on the relations between

classes, but these design-time relations between specifications have well-defined relations between instances of those classes.

Associations

UML defines a number of different kinds of relations between classes. The most important of these are associations, generalizations, and dependencies. The most basic of these is the *association*. An association is a design-time relation between classes that specifies that, at runtime, instances of those classes may have a *link* (navigable relation between objects through which services may be invoked). Said another way, a link is an instance of an association.

UML identifies three distinct kinds of associations: association, aggregation, and composition. An association between classes means simply that at some time during the execution of the system, those objects may have a link that enables them to invoke services from each other. Nothing is stated about *how* that is accomplished, or even whether it is a synchronous method call (although this is most common) or an asynchronous event signal. Associations are the specification of conduits that allow objects to send messages to each other at runtime. Associations are shown as lines connecting classes on class diagrams.

There are a number of aspects of an association between two classes that can be specified. For example, the ends of the associations may have *role names*. These name the instances with respect to the other class. The most common implementation of associations is a pointer (or reference), and it is common to name the pointer with the role name at the opposite end of the association. For example, in [Figure 1.7](#), the `Target` class might contain a pointer named `itsTargetingSystem` that would be dereferenced at runtime just as the `TargetingSystem` class might contain a pointer named `itsTarget` that allows the `TargetingSystem` to send messages to the `Target`. I use the term “might” because while pointers are a common implementation option, there are other ways to implement associations as well.

Although somewhat less common, *association labels* may also be used, such as between the `MissileTransaction` and `Missile` classes. The label is normally used to help explain why the association exists between the two classes. In this case, the label “Controls delivery of >” indicates the purpose of the relation. The “>” is a common way to indicate the speaking perspective of the label; in this case, it is from the `MissileTransaction` to the `Missile`.

The *multiplicity* is probably the most important property of an association end. The multiplicity of an association end indicates the number of instances that can participate in the association role at runtime. This may be any of the following:

- A fixed number, such as “1” or “MAX_DATA_SOURCES”
- A comma-separated list, such as “0,1” or “3,5,7”
- A range, such as “1 .. 10.”

Figure 1.7
Class relations.

In the figure, we see multiplicities on all of the association ends with the exception of the directed association between the `MissileTransaction` and the `Target`. A normal line with no arrowheads means that the association is bidirectional; that is, an object at either end of the association may send a message to an object at the other end. If only one of the objects can send a message to the other and not vice versa, then we add an *open arrowhead* (we'll see later that the type of arrowhead matters) pointing in the direction of the message flow. The multiplicity at the nonnavigable end (the end without the arrowhead) is not normally specified. This is because the multiplicity at the "pointy end" of an association has an implication on the implementation of the association in the class; you'll implement a 1.multiplicity with a pointer, but with * you might need an array of pointers. However, if the association end is nonnavigable (as is the case at the nonarrowed end of a directed association), then there *is* no implementation to change; hence its multiplicity is normally omitted.

All of these association adornments, except for multiplicity, are optional and may be added as desired to further clarify the relation between the respective classes.

An association between classes means that at some point during the lifecycle of instances of the associated classes, there *may be* a link that enables them to exchange messages. Nothing is stated or implied about which of these objects comes into existence first, which other object creates them, or how the link is formed. Composition, the strong form of aggregation, *does* allocate creation and destruction responsibilities, as we shall see in a moment.

Aggregation

An aggregation is a specialized kind of association that indicates a “whole-part” relation exists between the two objects. The “whole” end is marked with a white diamond, as in [Figure 1.7](#). For example, consider the classes `FlightPath` and `Position`. The `FlightPath` class is clearly a “whole” that aggregates possibly many `Position` elements. The diamond on the aggregation relation shows that the `FlightPath` is the “whole.” The “*” on the association end indicates that the list may contain zero or more `Position` elements. If we desired to constrain this to be no more than 100 waypoints we could have made the multiplicity “0..100.”

Since aggregation is a specialized form of association, all of the properties and adornments that apply to associations also apply to aggregations, including navigation, multiplicity, role names, and association labels.

Aggregation is a relatively weak form of “whole-part,” as we’ll see in a moment. No statement is made about lifecycle dependency or creation/destruction responsibility. Indeed, aggregation is normally treated in design and implementation identically to association. Nevertheless, it can be useful to aid in understanding the model structure and the relations among the conceptual elements from the problem domain. Sometimes it may not be clear which is appropriate – association or aggregation. In such cases, don’t worry about it. Aggregation is typically implemented in exactly the same way as association, so it doesn’t change the implementation. If you can’t decide, pick one and change it downstream if necessary.

Composition

Composition is a strong form of aggregation in which the “whole” (also known as the “composite”) has the explicit responsibility for the creation and destruction of the part objects. We’ve seen the structured class notation in the previous section between a class and its parts. Composition is the relation between the structured class and the class of those parts. There is a crucial distinction between aggregation and composition; in composition, you are explicitly assigning creation and destruction responsibility to the composite. For this reason, the multiplicity at the whole end of a composition relation is *always* 1. While an object can have multiple aggregation owners, it can only have at most one composite owner.

Because of object lifecycle responsibility, the composite exists before the parts come into existence and it exists after they are destroyed. If the parts have a fixed multiplicity with respect to the composite, then it is common to create those parts in its constructor (a special operation that creates the object) and destroy them in its destructor. With nonfixed multiplicities (e.g. “*”), the composite dynamically creates and destroys the part objects during its execution. Because the composite has creation and destruction responsibility, each part object can only be owned by a *single* composite object, although the part objects may participate in other association and aggregation relations. Composition is also a kind of association so it can likewise have all of the adornments available to ordinary associations.

Figure 1.7 shows the filled diamond form, while Figure 1.5 shows the nested part (structured class) form. As a quick exercise, redraw Figure 1.5 to use filled diamonds and redraw Figure 1.7 to use the nested part form. To be semantically correct, the filled diamond must be a relation between *classes* only and the nested part form is a relation between a class and parts (object roles).

Generalization

The generalization relation in UML means that one class defines a set of features that are either specialized or extended in another. Generalization may be thought of as an “is a type of” relation and therefore only has a design-time impact, rather than a runtime impact.

Generalization has many uses in class models. First, generalization is used as a means to ensure interface compliance, much in the same way that interfaces are used. Indeed, it is the most common way to implement interfaces in languages that do not have interfaces as a native concept, such as in C++ or C. Also, generalization can simplify your class models because a set of features common to a number of classes can be abstracted together into a single superclass, rather than redefining the same structure independently in many different classes. In addition, generalization allows for different realizations to be used interchangeably; for example, one realization subclass might optimize worst-case performance while another optimizes memory size while yet another optimizes reliability because of internal redundancy. Different subclasses can be substituted in for an instance of the superclass and the collaboration will still “work.”

Generalization in the UML means two things – inheritance and substitutability. First, it means *inheritance* – that subclasses have (at least) the same attributes, operations, methods, and relations as the superclasses they specialize. Of course, if the subclasses were *identical* to their superclasses, that would be unexciting, so subclasses can differ from their superclasses in either or both of two ways – specialization or extension.

Subclasses can *specialize* operations or state machines of their superclasses. Specializing means that the same operation (or action list on the state diagram) is implemented differently than in the superclass. This is commonly called *polymorphism*. In order to make this work,

when a class has an association with another that defines subclasses, at runtime an instance of the first can invoke an operation declared in the second, and if the link is actually to a sub-class instance, the operation of the subclass is invoked rather than that of the superclass.

Figure 1.8 shows a simple example. The `Track` class has a `timeOfMeasurement` attribute that identifies when the track was measured, and has composition relations with both the `Velocity` and `Position` classes. This is the base class for technology-specific tracks, for FLIR (forward-looking infrared), optical, and radar sensors. Each of these has technology-specific additional data. The `TrackFuser` integrates data from up to three kinds of sensors monitoring the same aircraft into a multisensor track. Because these subclasses inherit features from the base class, they also have compositions with the `Velocity` and `Position` classes as well.

The second thing that generalization means is *substitutability*. This means that an instance of a subclass is freely substitutable wherever there is an instance of the superclass. In Figure 1.8, a `Track` may provide services (such as returning its position, velocity, or combat ID) to a client. Because `MultisensorTrack` is a subclass of `Track`, at runtime the client of the `Track` class may actually have a link to an instance of the `MultisensorTrack` class. The client neither knows nor cares, because `MultisensorTrack` inherits all the capabilities of its superclass.

Dependency

Association, in its various forms, and generalization are the key relations defined within UML. Nevertheless, there are several more relations that are still useful. They are put under

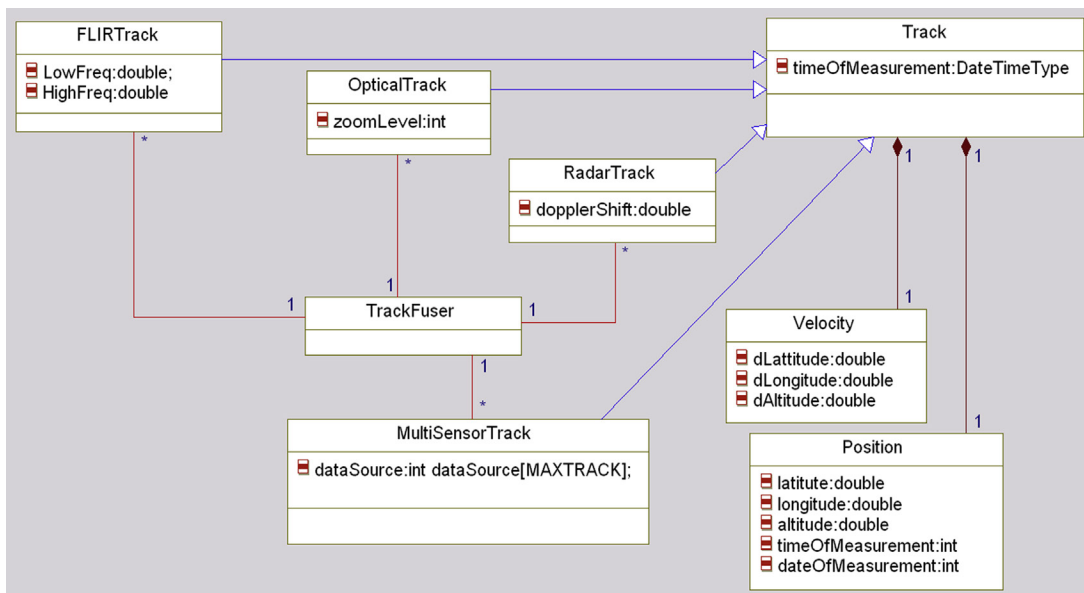


Figure 1.8
Class generalization.

the umbrella of *dependency*. UML defines four different primary kinds of dependency – abstraction, binding, usage, and permission. Each of these may be further stereotyped. For example, «refine» and «realize» are both stereotypes of the abstraction relationship and «friend» is a stereotype of permission. All of these special forms of dependency are shown as a stereotyped dependency (dashed line with an open arrowhead).

Arguably, the most useful stereotypes of dependency are «bind», «usage», and «friend». Certainly, they are the most commonly seen, but there are others as well. The reader is referred to the UML standard⁴ for the complete list of “official” stereotypes.

The «bind» stereotype binds a set of actual parameters to a formal parameter list. This is used to specify parameterized classes (templates in C++-speak or generics in Ada-speak). This is particularly important in patterns because patterns themselves are parameterized collaborations, and they are often defined in terms of parameterized classes.

A *parameterized class* is a class that is defined in terms of more primitive elements, which are referred to symbolically without the inclusion of the actual element that will be used. The symbolic name is called a *formal parameter* and the actual element, when bound, is called an *actual parameter*. In Figure 1.9, `NumericParameter` is a parameterized class whose attributes and methods are defined in terms of an unspecified type, called `BaseType`. If we bind an actual parameter, say *double*, to be used whenever `BaseType` is used, we now have a class (`NumericParameter_double`) from which we can create instances; a fully bound parameterized class is called an *instantiable class*, because we now have enough information to create instances of it. The «bind» dependency binds the actual parameter, *double*, to the formal parameter, `BaseType`.

Parametric classes can be subclassed, as they are in the figure. A `SafeNumericParameter` stores the data twice, once normally and once bit-inverted. It’s not yet an instantiable class because we haven’t specified what to use for `BaseType` yet. We do that when we create the `SafeNumericParameter_double`.

The diagram also shows the use of the «friend» dependency. This means that the `SafetyMonitor` class has access to all the features of the `SafeNumericParameter` regardless of their stated visibility (private, protected, or public). It’s like the old saying goes “You only show your friends your private parts.”

1.2.3 Big Things: Subsystems, Components, and Packages

Classes, objects, and interfaces are, for the most part, little things. It takes collaborations of many of them to provide system-wide behavior. In today’s complex systems, it is important to think about larger-scale structures. UML does provide a number of concepts to manage

⁴ Available at www.omg.org.

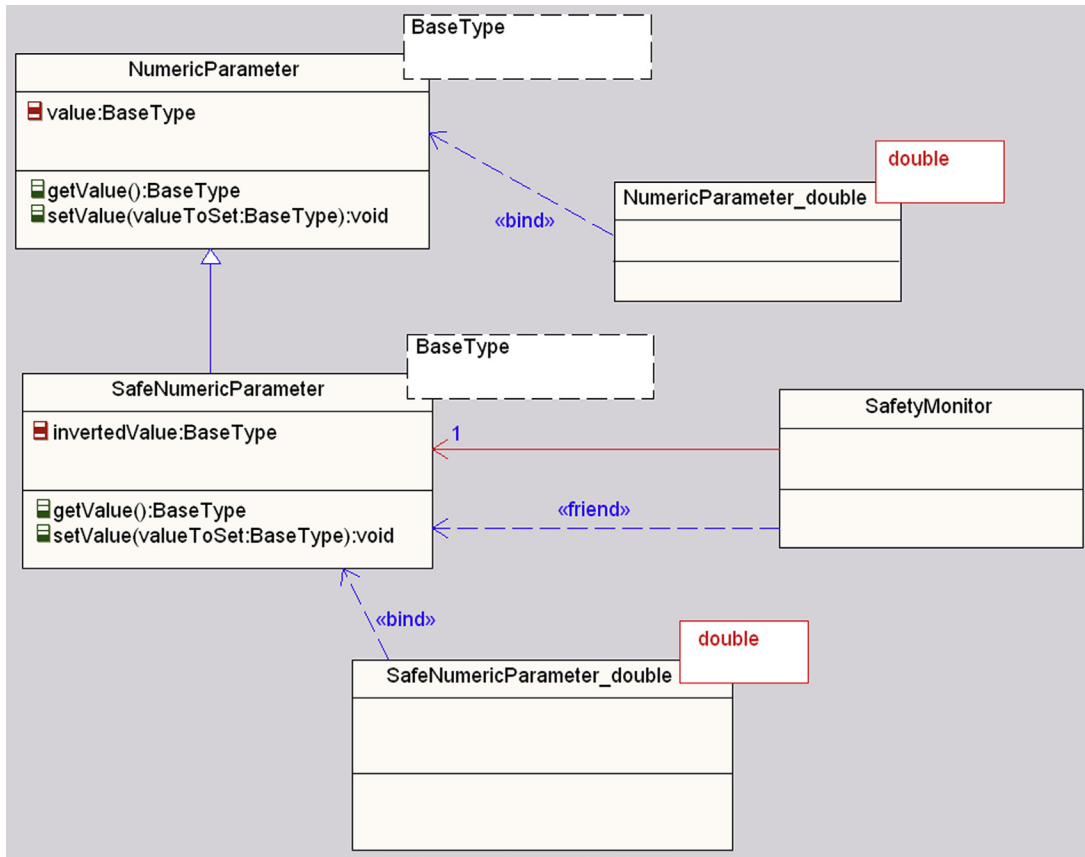


Figure 1.9
Dependency.

systems at a large scale, although most of the literature has not effectively explained or demonstrated the use of these features. And, to be honest, the UML specification does not explain them and how they interrelate very well either.

Packages are one such element. Packages are used *exclusively* to organize UML models – they are not instantiable and do not appear in any way in the running systems. Packages are model elements that can contain other model elements, including other packages. Packages are used to subdivide models to permit teams of developers to manipulate and work effectively together. They define a namespace for the model elements that they contain, but no other semantics. They are the primary model organizational units for configuration management – that is, in general packages are the primary CIs (configuration items). When you check in or out a package from configuration management (CM), you check in all the elements contained within that package. Rhapsody™ allows you to change the level of a CI down to the level of the individual class, but most developers find that to be too low level when looking for all the relevant items to check out.

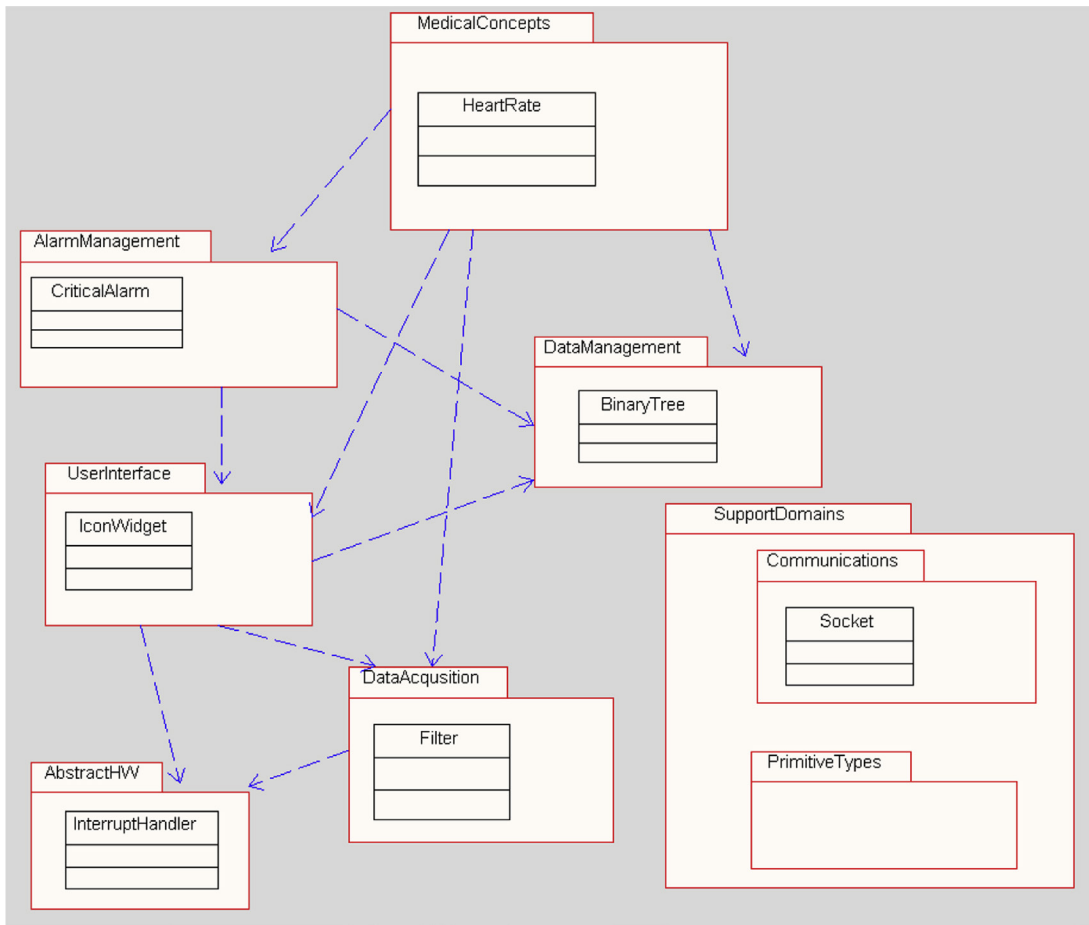


Figure 1.10
Packages.

A package normally contains elements that exist only at design-time – classes and data types – but may also contain use cases and diagrams, such as sequence and class diagrams. These design pieces are then used to construct collaborations that realize system-wide functionality. [Figure 1.10](#) shows that packages are drawn to look like a tabbed folder and may optionally show the elements that they semantically contain. It shows a single class within most packages, but they usually hold dozens or more elements. The dependency relations are optional but can be used to show compilation dependency.

Subsystems are nothing more or less than classes – structured classes, specifically – and they serve as large-scale architectural building blocks. Unlike packages, subsystems are instantiable, meaning that you can create an instance of the type that occupies memory at runtime. A subsystem is used to organize the runtime system consisting of instances; the criterion for inclusion in a subsystem is “common behavioral purpose.” The real work of a subsystem is

implemented by the parts contained within the subsystem; the subsystem offers up the collaborative behavior of those elements. Subsystems organize and orchestrate the behavior of their internal parts, but the “real work” is done by what is sometimes called the *semantic objects* of the system – the primitive objects that actually perform the bottom-level functionality. A subsystem is at a higher level of abstraction of the system than these primitive semantic objects and this level of abstraction allows us to view and manipulate the structure and behavior of complex systems much more easily.

In UML 2.0, components are basically also structured classes that provide opaque language-independent interfaces. There are some subtle metamodeling distinctions between components and subsystems, but in use, they are practically identical.

1.3 Behavioral Elements and Diagrams

What we’ve discussed so far is the definition of structural elements of the system – classes and objects (in the small) and systems, subsystems, and components (in the large). As developers, we are usually even more concerned about how these structural elements behave dynamically as the system runs. Behavior can be divided up into two distinct perspectives – how structural elements act in isolation and how they act in collaboration.

In the UML metamodel, *model elements* are the primary structural elements that exhibit behaviors. *Classifiers* (which are types of model elements) also have *behavioral features*, specifically *operations*, and the realization of operations, *methods*. In practice, we are primarily concerned with the specification of the reactive behavior of only certain classifiers (classes, objects, subsystems, components, and use cases) and certain other model elements (actions, operations, and methods).

1.3.1 Actions and Activities

An *action* is a “is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a message to an object or modifying a link or a value of an attribute.” That is, it is a primitive thing, similar in scope to a single statement in a standard source-level language, such as “++X” or “a=b+sin(c*PI).” UML identifies a number of different kinds of actions, such as creating an instance, invoking a method, or generating an event.

While UML specifies what is called *action semantics*, it doesn’t specify a language syntax to use.⁵ Some tools use proprietary abstract action languages that are converted to a target

⁵ Although there is work underway to add such a standard action language.

language during the model compilation process. This has the (claimed⁶) advantage of portability of the application from one source language to another. The downside of the approach is that debugging at the code level is usually very difficult and if you *ever* modify the source code, then the connection between the model and the code is irretrievably broken. By far, most developers prefer to specify actions in their intended source language – whether it is C, C++, Java, Ada, or something else. For one thing, *they already know* that language. For another, these languages have standards (as opposed to proprietary action languages) and multiple compilers to support them on many different platforms. Debugging is far simpler, and bringing code changes back into the model can be relatively easy, depending on the tool.

Actions have “run-to-completion” semantics, meaning that once an action is started it will run until it is done. This does not mean that an action cannot be preempted by another action running in a higher-priority thread, merely that when the context executing that action returns from preemption, it will continue executing that action until it is complete. This means that if an object is executing an action, that action will run to completion even if that object receives events directing it to do something else. The object will not accept the incoming events until the action has completed.

An *activity* is an action that runs when a classifier is in a state and is terminated either when it is complete or when the classifier changes state. That is, activities do not have run-to-completion semantics. An object executing an activity may receive an event that triggers a transition, exiting the state and terminating the activity. Thus, UML allows the modeling of, at a primitive level, both interruptible and noninterruptible behaviors.

1.3.2 Operations and Methods

An operation is a specification of an invocable behavior of a classifier, whereas a method is the implementation of an operation. That is, an operation is a specification of a method. Operations are synchronously invoked and are logically associated with call events in the UML metamodel. Operations have typed parameter lists, as you might expect, and can return typed values. It is common to use an operation call as an action on a state behavior.

Modeling of the behavior of a method is done primarily in two ways. First, and most common, is to simply list, in the selected action language, all the actions to implement the method. This is clearly the best approach for simple methods. The second approach, which will be described shortly, is to model the operation with an activity diagram.

⁶ I say “claimed” because in practice, the generation of a concrete language with these tools is not as simple and obvious as it could be, and most of the tools that work this way produce some really ugly and inefficient code for the target language.

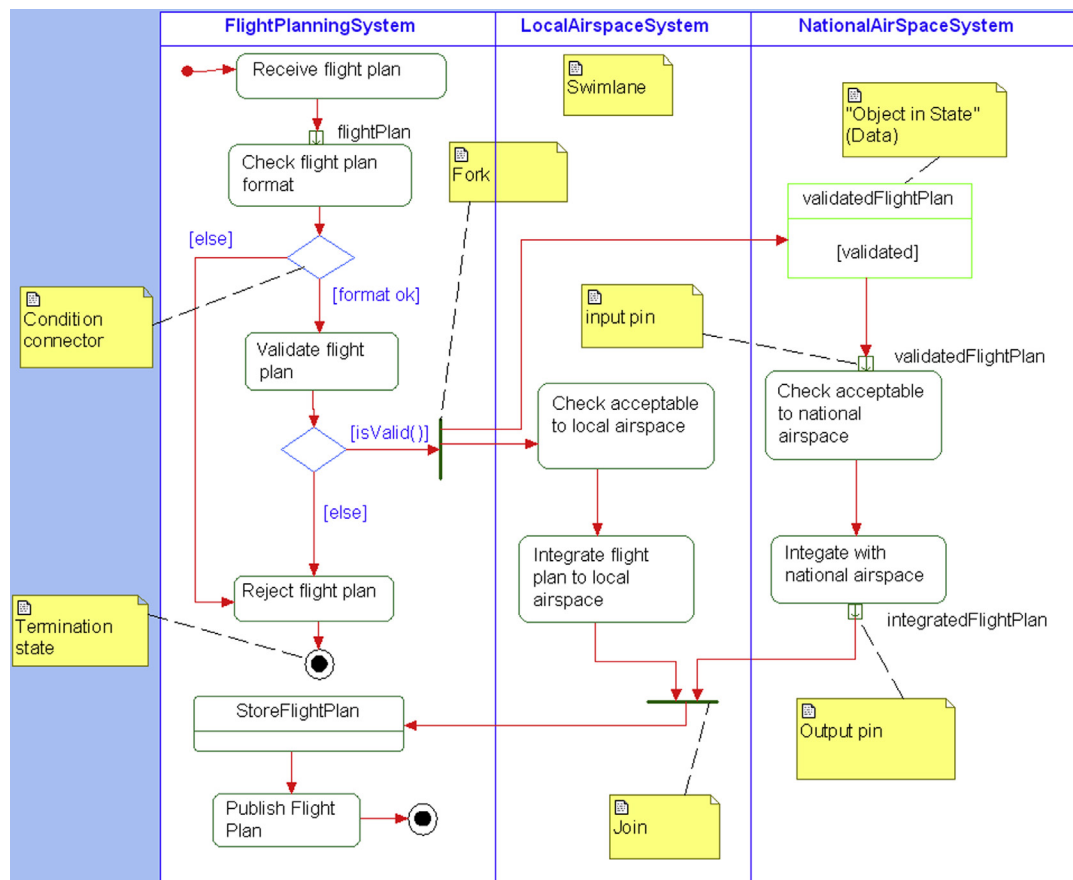


Figure 1.11
Activity diagram.

1.3.3 Activity Diagrams

In UML 1.x, activity diagrams were nothing more (or less) than a state machine with a different syntax. That is, both activity diagrams and state machines had a common underlying behavioral metamodel. In UML 2.0, however, that's been changed.

In UML 2.0, activity diagrams execute based on *token execution semantics*; that is, when an activity receives the run token from the activity that precedes it, it can run. When it is done executing, it passes the token on to the next activity in the sequence. The transition to the next activity (or action) is taken when the predecessor activity completes – no events are used. In [Figure 1.11](#) the rounded rectangles are actions or activities. The arrows indicate the flow transitions. The transition with a ball on one end is called the *initial pseudostate* and indicates where execution begins when the behavior starts.

Activity diagrams contain operators. The diamonds indicate selection based on the execution of guards (Boolean expression shown within square brackets). These guards should have no side effects. Transitions are taken when the guard evaluates to `TRUE`. The `else` transition is taken only if all other transitions are false. The bar with a single input transition and multiple exiting transitions is called a *fork*. It differs from selection in that with selection, at most one outgoing transition is taken; with a fork, *all* outgoing transitions are taken. That is, a fork indicates the presence of logical threads of execution running simultaneously.⁷ A *join*, a bar with multiple incoming transitions, collapses multiple logical threads into one. The *termination state* indicates the end of the behavior.

Actions and activities can be owned by various objects – they can invoke methods from other objects with whom they have links. The allocation of the actions and activities is most commonly done with *swimlanes*. In the figure, swimlanes are the named rectangles containing the actions and activities. The name of each swimlane usually identifies a class that provides the invoked method.

Another new feature added in UML 2.0 is the notion of *pins*. Just as an activity (or action) corresponds to a function, a pin corresponds to a parameter of a function. Thus, pins can be shown as either the origin of a transition (indicating the passing of the data to the parameter specified by the pin) or its termination (indicating the reception of data by the parameter). Pins at the origin of a transition are called *output pins* and pins at the termination of a transition are called *input pins*. Input and output pins are indicated in [Figure 1.11](#). Data can also be indicated with *object in state*, a notational hang over from UML 1.4. `validatedFlightPlan` is such an element; its state or condition is shown in square brackets inside the box. In UML 2.0, it is more common to use pins.

Activity diagrams are like flowcharts on steroids. The most common use for activity diagrams is to model algorithms. As such, their most common applications are to represent the behavior of a method of a class or the behavior of an algorithmic use case.

1.3.4 State Diagrams

A finite state machine (FSM) is a machine specified by a finite set of conditions of existence (called “states”) and a likewise finite set of transitions among the states triggered by events. A FSM differs from an activity diagram in that the transitions are primarily triggered by events rather than being triggered when the work done in the previous state is complete. State diagrams are primarily used to model the behavior of *reactive* elements, such as classes and use cases. These are elements that wait in a state until an event of interest occurs. At that

⁷ I say “logical” because they don’t have to be implemented with OS threads. They can be executed sequentially but they are logically concurrent so the developer cannot know which is executed first. That is, they are execution-order independent with respect to each other.

point, the event is processed, actions are performed, and the element transitions to a new state.

Actions may be simple in-line actions, invocation of operations, or event generations. Action may be specified to be executed when a state is entered or exited, or when a transition is taken. The order of execution of actions is

1. Exit actions of the predecessor state
2. Transition actions
3. Entry actions of the subsequent state

UML uses state diagrams as their formal FSM representation, which are significantly more expressive and scalable than “classical” Mealy-Moore FSMs. UML state machines, based on Dr. David Harel’s state diagram semantics and notation⁸ have a number of extensions beyond Mealy-Moore state machines, including

- Nested states for specifying hierarchical state membership
- And-states for specifying logical independence and concurrency
- Pseudostates for annotating commonly needed specific dynamic semantics

Figure 1.12 shows the some of the basic elements of a state diagram for dialing a number with a telephone. It includes basic or-states and transitions, as well as a few less-elementary concepts, including nested states and conditional and initial pseudostates.

Transitions are arrowed lines coming from a predecessor state and terminating on a subsequent state. Transitions usually have an optional *event signature* with an *action list*. The basic form of an event signature is

```
event-name ('parameter-list')' ['guard']' '/' action-list
```

The event-name is simply the logical name of the event class that may be sent to an instance of the classifier at runtime, such as `evOnHook` or `tm` in Figure 1.12. UML defines four distinct kinds of events that may be passed or handled:

- Signal event – an asynchronously sent event
- Call event – a synchronously sent event
- Time event – an event due to the passage of an interval of time (most common) or arrival of an epoch
- Change event – a change in a state variable or attribute of the classifier

Asynchronous event transfer is always implemented via queuing of the event until the element is ready to process it. That is, the sender “sends and forgets” the event and goes on about its

⁸ Dr. David Harel, inventor of state diagrams, was one of the founders of I-Logix (now integrated into IBM Rational) and was instrumental in the creation of the Statemate and Rhapsody tools.

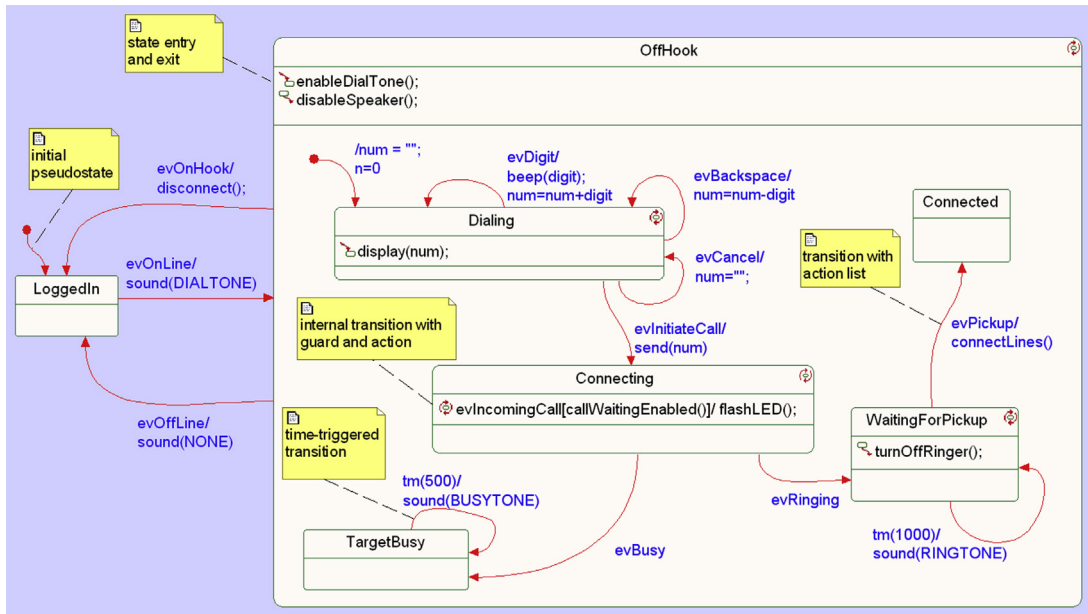


Figure 1.12
Basic state machine.

business, ignorant of whether or not the event has been processed. Synchronous event transfer executes the state processing of the event in the thread of the sender, with the sender blocked from continuing until that state processing is complete. This is commonly implemented by invoking a class method called an *event handler* that executes the relevant part of the state machine, returning control to the sender only when the event processing is complete. Rhapsody refers to call events as *triggered operations*; they are like normal operations in that they are synchronous, but don't have a standard method body. Their method body is the action list on the state machine.

Events may have parameters, which are typed values accepted by the state machine that may then be used in the guards and actions during the processing of an event. The state diagram specifies the formal parameter list, while the object that sends the event must provide the necessary actual parameters to bind to the formal parameter list. Rhapsody uses a slightly peculiar syntax for passing events. Rhapsody automatically creates a struct named `params` that contains the named parameters for every event that carries data. To use two parameters, `x` and `y`, of an event `e` in a guard, for example, you would dereference the `params` struct to access their value. So a transition triggered by event `e` with a guard that specified that to take the transition `x` must be greater than 0 and `y` must be less than or equal to 10 would look like the following:

```
e[params->x>0 && params->y<=10]
```

Time events are almost always relative to the entry to a state. A common name for such an event (and what we will use here) is `tm(interval)`, where `interval` is the time interval parameter for the timeout event.⁹ If the timeout occurs before another specified event occurs, then the transition triggered by the timeout event will be taken; if another event is sent to the object prior to the triggering of the timeout. Then the timeout is discarded. If the state is later reentered, the timeout interval restarts from zero.

If a transition does not provide a named event trigger, then it is activated by the “completion” or “null” event. This event occurs either as soon as the state is entered (which includes the execution of entry actions for the state) or when the state activities complete.

A *guard* is a Boolean expression contained within square brackets that follows the event trigger. The guard should return only `TRUE` or `FALSE` and should not have any side effects. If a guard is specified for a transition and the event trigger (if any) occurs, then the transition will be taken if and only if the guard evaluates to `TRUE`. If the guard evaluates to `FALSE`, then the triggering event is quietly discarded and the actions are not executed.

The action list for the transition is executed if and only if the transition is taken; that is, the named event is received by the object while it is in the predecessor state and the guard, if any, evaluates to `TRUE`. The entire set of exit actions—transition actions—entry actions is executed in that order and using run-to-completion semantics, as noted previously. Figure 1.12 shows actions on a number of different transitions, as well as actions on entry for the `Dialing` state, on exit for the `WaitingForPickup` state, and on both entry and exit for the `OffHook` state.

In addition to entry and exit actions, states may also have reactions, also known as *internal transitions*. These are actions taken when the object is in the specified state and the triggering event is received, but in this case, the state is not changed. The UML comments in Figure 1.12 indicate some entry actions, exit actions, and internal transitions. Rhapsody uses small icons beside the action to indicate when the actions inside the state are executed (on entry, exit, or reaction).

If you look at Figure 1.12, the rounded rectangles are states. These are called *or-states* because at any level of state machine abstraction, the object must be in only one of these states. In the figure, the `Telephone` object can either be `LoggedIn` or `Offhook`. These are or-states at the same level of abstraction. The initial pseudostate indicates which state you start in when the object is created. Rhapsody provides an `IS_IN(state)` macro for all state machines; `IS_IN(LoginIn)` will return `TRUE` if and only if the specified `Telephone` object is in the `LoginIn` state.

The `OffHook` state has a set of nested states, meaning that when the object is in the `OffHook` state, then it also must be in exactly one of those nested states as well. Within the `OffHook`

⁹ Another common name is `after(interval)`.

state, its nested states are or-states; that is, if an object is in the `OffHook` state, then it must be in *exactly one* of the nested states. If the `Telephone` object is in the `WaitingForPickup` state, then not only does `IS_IN(WaitingForPickup)` return `TRUE`, so does `IS_IN(OffHook)`. That is a basic characteristic of nested states; if you're in the bathroom, then yes, you are also in your house, and you can only be in one room of the house at a time and in one house at a time.

The `evOnLine` event enters the `OffHook` state and therefore must enter a nested state as well, but which one? The answer is indicated with the initial pseudostate inside the `OffHook` state – it indicates which of the nested states is the default. The default can be bypassed by drawing a transition directly to a different nested state, if desired. Note that there are two transitions from the `OffHook` state back to the `LoggedIn` state. They come from the composite, so they apply whenever `OffHook` is the current state, regardless of which nested state in `OffHook` is currently active. In a Mealy-Moore state machine, which lacks nesting, those two transitions would each have to come from every one of the nested states and go to the `LoggedIn` state to provide the same behavior.

Figure 1.13 shows an important additional concept in state diagrams – *and-states*. While or-states are disjoint and exclusive, and-states are disjoint but not nonexclusive. These are

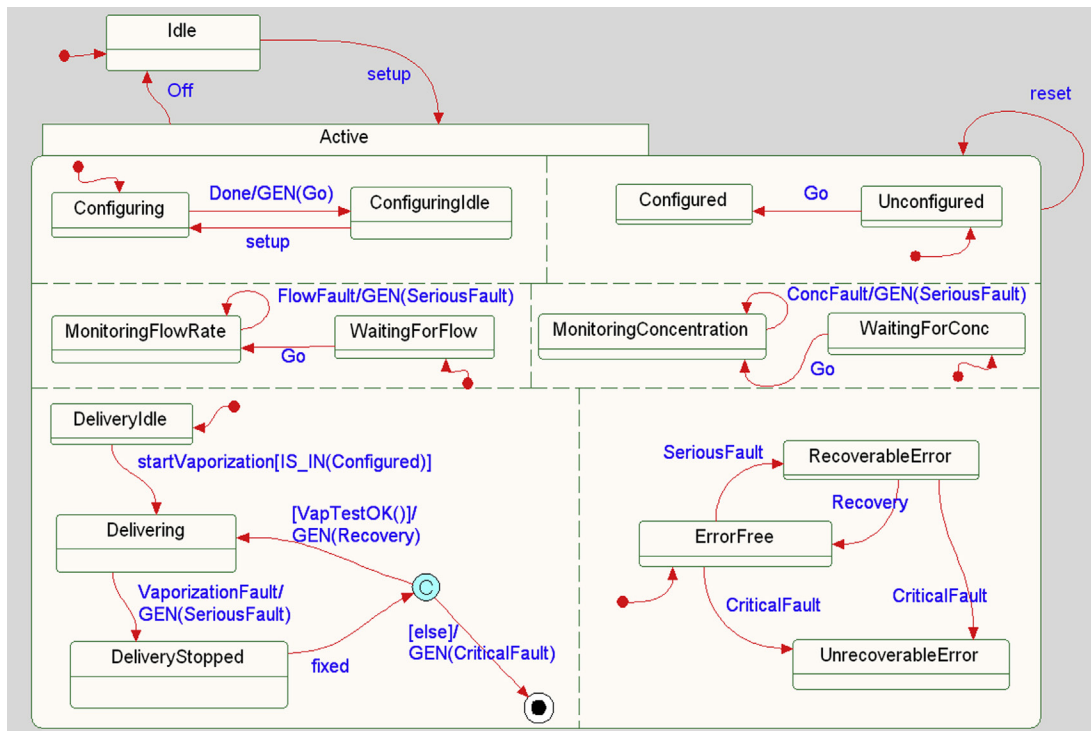


Figure 1.13
And-states.

equivalent to the logical threads in activity diagrams; they are usually not actually modeled as separate OS threads, but the order of execution between the and-states is not inherently knowable. Event receptions have object-wide scope. This means that if an object has multiple active and-states and receives an event, *all* active and-states receive a copy of the event and are free to independently act on that event, or discard it as appropriate. Furthermore, the order in which those and-states process their copies of that event is not known. If you want to specify a particular sequencing, then use or-states rather than and-states. To see some idioms for synchronizing action execution across and-states, and for a more detailed discussion of UML state machines, refer to Chapter 3 of *Real-Time UML 3rd Edition*¹⁰ and Chapters 7 and 12 of *Doing Hard Time*.¹¹

1.3.5 Interactions

UML models the collaborative behavior of multiple entities working together as *interactions*. UML provides three primary diagrams to represent interactions: communication diagrams (known in UML 1.x as “collaboration diagrams”), sequence diagrams, and timing diagrams. Communication diagrams are basically object diagrams with messages shown with numbers. Sequence diagrams show object roles as vertical lifelines with message sequences going down the page. Timing diagrams depict changes in state or value over linear time. Neither communication nor timing diagrams are widely used, so we will limit our discussion here to sequence diagrams.

Sequence diagrams depict object roles – which might represent objects, subsystems, systems, or even use cases – interacting over time. The vertical lines in [Figure 1.14](#) are called *lifelines*, and represent the object (or object role). Objects (or object roles) can both send and receive messages. Messages are shown by the arrowed lines going from one lifeline to another.

The sequence diagram shows an *exemplar* or “sample execution” of some portion of the system under specific conditions. Such an exemplar is commonly called a *scenario*. The messages may be synchronous (shown with a solid arrowhead) or asynchronous (shown with an open arrowhead). Sequence flows, more or less, from the top of the page downwards. Additionally, the state or condition of the lifeline can be shown, as can constraints. In [Figure 1.14](#), a timing constraint is shown on the right of the screen. A more general constraint is shown anchored to `alarm(Gas_Supply_Fault)`, limiting the type of faults that are reported using this value.

UML 2.0 significantly extends sequence diagrams over previous versions. Diagrams may contain, essentially, subdiagrams called *interaction fragments*. Each interaction fragment can

¹⁰ Douglass, Bruce Powel. *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004).

¹¹ Douglass, Bruce Powel. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999).

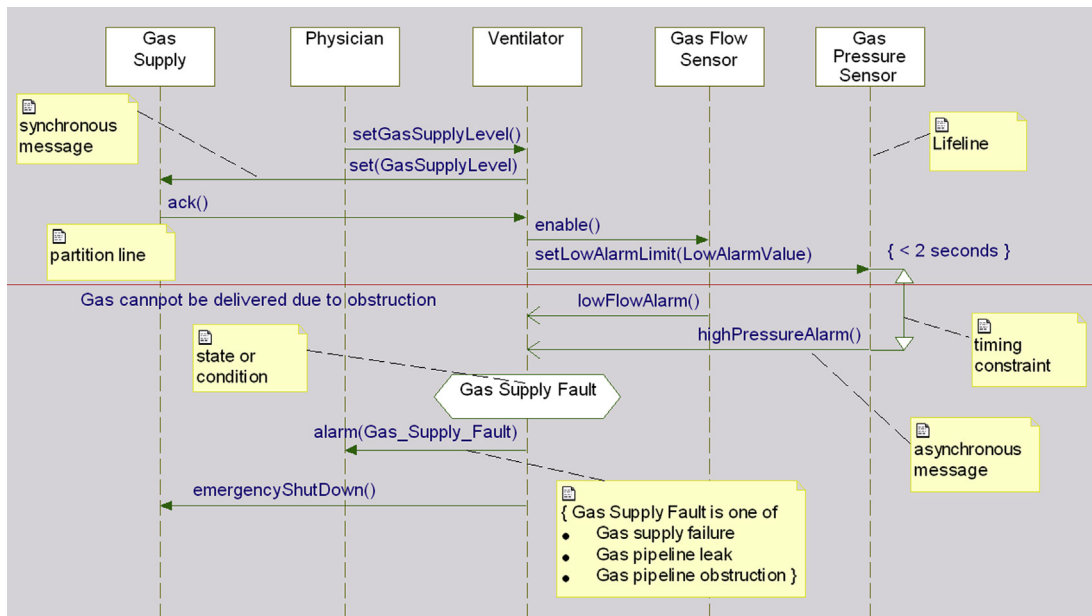


Figure 1.14
Sequence diagram.

have an operator, such as `loop`, `opt` (for “optional”), `alt` (for “alternative”), `ref` (for “reference”), `para` (for “parallel”), and so on. These interaction fragments and operators greatly enhance the ability of sequence diagrams as specification tools. Figure 1.15 shows three interaction fragments. One has a parallel operator indicating that it contains regions that execute concurrently. Within that interaction fragment two more are nested, with `loop` operators indicating that they repeat until some termination condition is reached.

Arguably the most significant extension to sequence diagrams in UML 2.0 is the ability to formally decompose them. This can be done either “horizontally” by using an interaction fragment with the `ref` operator, or “vertically” by setting a reference from one lifeline to a separate sequence diagram that shows the same scenario at a more detailed level of abstraction. The next three figures show how these decomposition mechanisms can be used.

Figure 1.16 shows the “high-level” sequence diagram for an industrial robot system. The user sets up the system and, based on the task plan, the controller commands the robot to achieve the tasks. The robot itself has internal parts – two angular joints (called the knee and the elbow) and a rotating manipulator – which can grab and control tools. This high-level sequence diagram contains two references to more detailed interactions. The first of these is the `Setup System` referenced interaction fragment. If we open up that diagram (a right-click in Rhapsody), we see the details shown in Figure 1.17.

Even more valuable is the ability to decompose the lifeline. This mechanism allows the same scenario to be viewed at many different levels of abstraction without overwhelming the

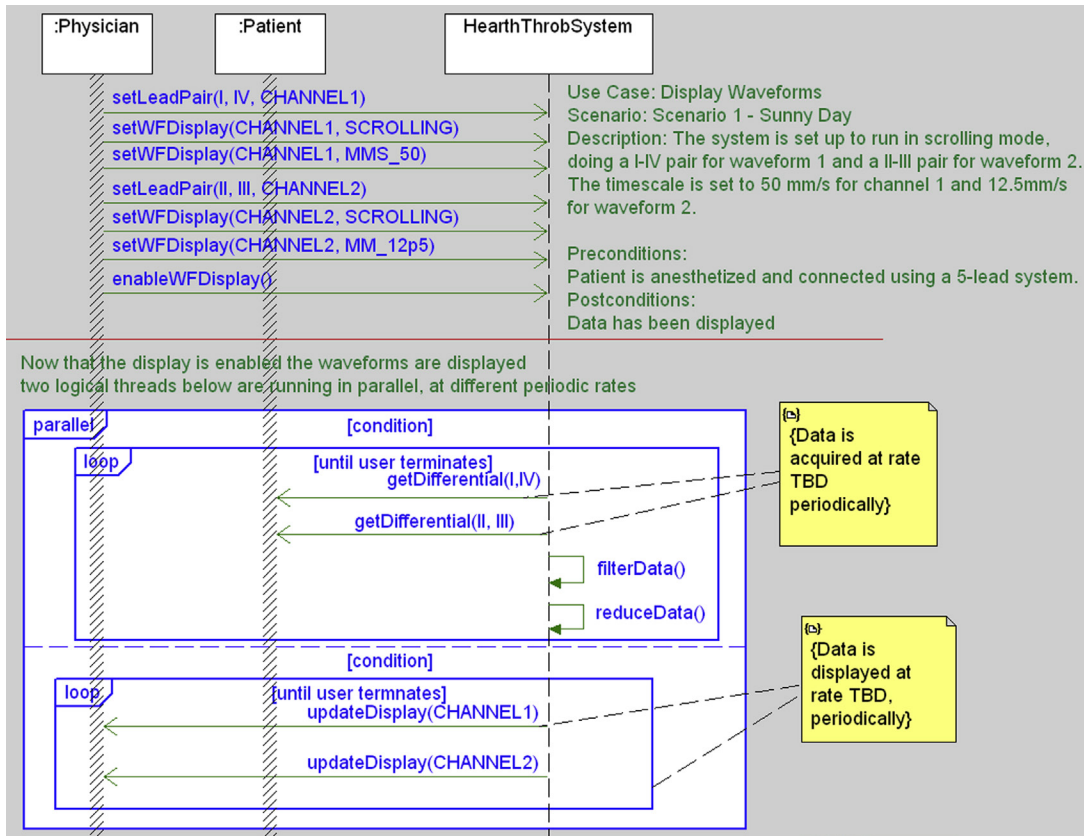


Figure 1.15
 Interaction fragments.

viewer by putting everything on a single, huge diagram. Figure 1.18 shows the details of how the internals of the robot interact to achieve their roles in this same scenario. The `ENV` lifeline is the connection between the high- and low-level interactions. At the high level, a message going to the `Robot` lifeline comes *out* of the `ENV` lifeline on the more detailed diagram. Conversely, a message going *into* the `ENV` lifeline on the more detailed diagram comes out of the `RobotController` lifeline on the higher-level sequence diagram.

1.4 Use Case and Requirements Models

A *use case* is an explicitly named capability of a system, subsystem, or component. It is a collection of operational and related quality of service requirements around a functional aspect of the system. It should return a result to one or more actors, and should not reveal or imply anything about the internal implementation of that functionality. A medium-sized system with 200 pages of requirements might have 20 use cases, each of which represent 8-10 pages of detailed requirements.

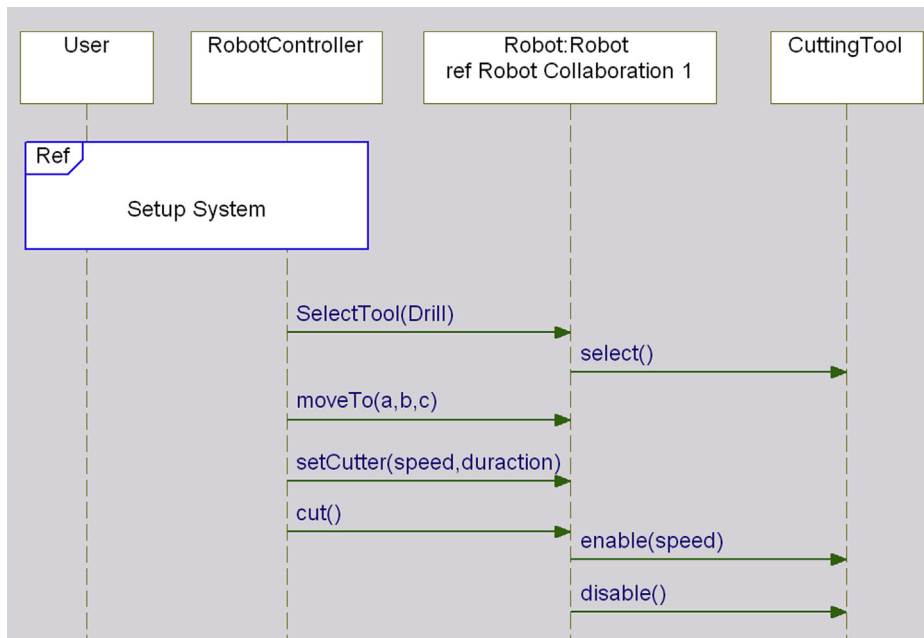


Figure 1.16
High-level sequence diagram.

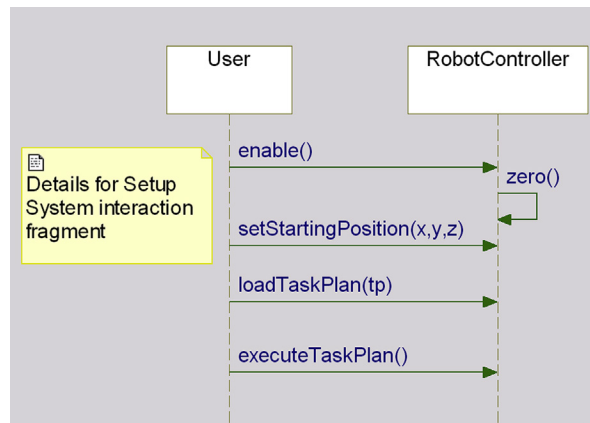


Figure 1.17
Referenced interaction fragment.

There are two different kinds of requirements typically applied against a system or system element: functional requirements and so-called “nonfunctional” or quality of service (QoS) requirements. Functional requirements refer to what the system needs to do, as in “The system shall deliver the anesthetic drug Halothane in inhalant form.” QoS requirements refer to *how well* the functional aspects are to be achieved, as in “The system shall deliver the

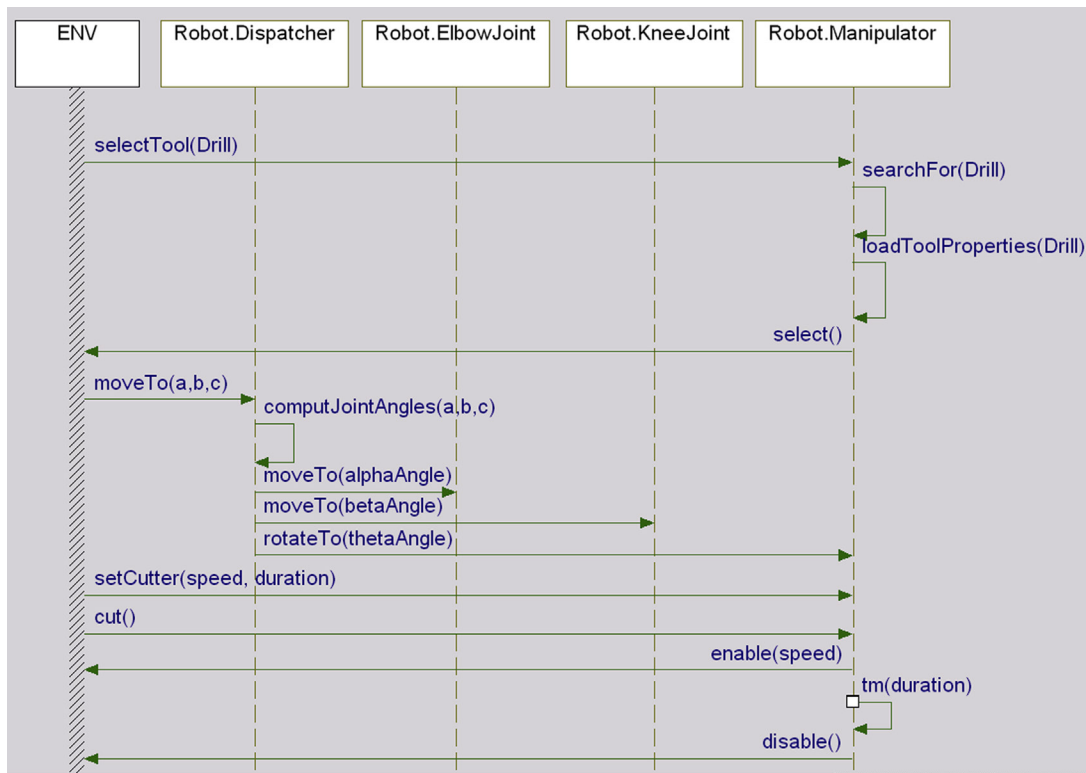


Figure 1.18
Decomposed lifeline.

anesthetic drug Halothane in inhalant form *and maintain the commanded gas concentration within 0.5% by volume. From a drug-free gas concentration, a maximum of 10% by volume shall be achieved in no less than 10 minutes at a breathing rate of 10 breaths/minute and a tidal volume of 600ml.*”

Use cases are drawn as ovals that associate with actors, indicating that the realizing collaborations interact in meaningful ways with those specified actors. In addition, use cases may relate to other use cases, although the novice modeler is cautioned against overuse of these relations.¹² The three relations among use cases are *generalization* (one use case is a more specialized form of another), *includes* (one use case includes another to achieve its functional purpose), and *extends* (one use case may optionally add functional aspects to another). An example use case diagram for a medical anesthesia system is shown in [Figure 1.19](#).

¹² In the author’s experience, it is all too easy to use the use case relations in a misguided attempt to functionally decompose the internals of the system, which is *not the point*. The purpose of use cases is to define the functional behavior of a system, subsystem, or other large-scale classifier in an implementation-free way.

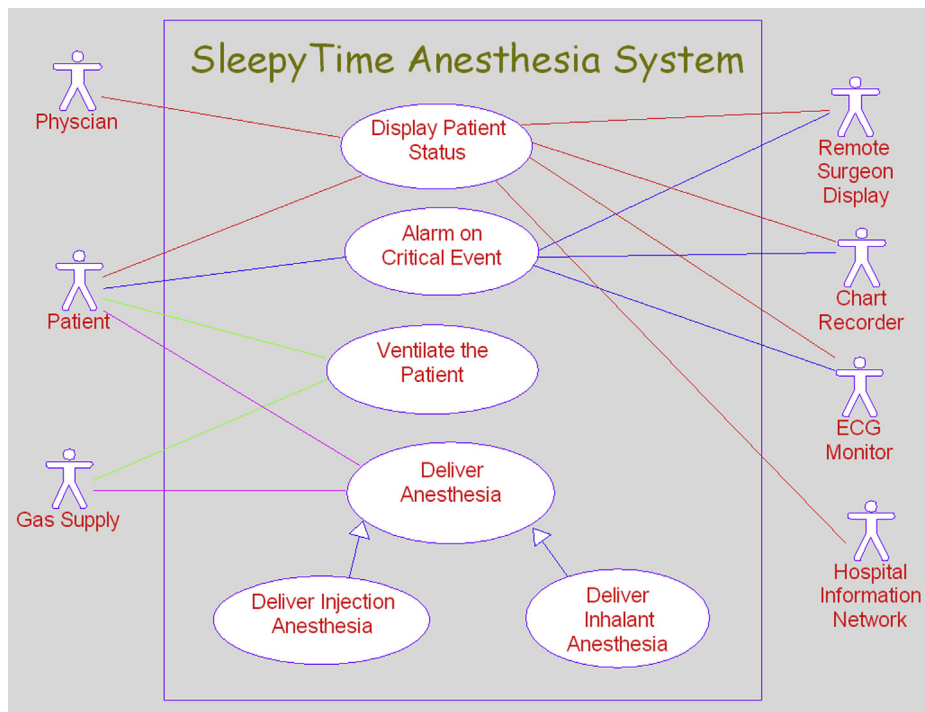


Figure 1.19
Use case diagram.

Since a use case provides little more than a name and relations to actors and other use cases, the detailed requirements must be captured somewhere. The Harmony process refers to this as “detailing the use case.” There are two complementary approaches to detailing use cases: by example and by specification. In both cases, however, the internals of the structure of the system cannot be referred to, since the requirements should be captured in an implementation-free manner.

Use cases may be thought of as “bags” that contain related detailed requirements. These requirements, as mentioned above, may be a combination of functional and QoS requirements. These details are captured in one or both of two ways: scenario modeling with sequence diagrams or specification, especially with state machines.

Scenario modeling of use cases involves the creation of sequence diagrams that capture different scenarios of the use case. Each scenario captures a very specific system–actor interaction. These scenarios capture messages sent to the system from the actors and from the system to the actors, as well as the allowable set of sequences of such messages.

When doing scenario modeling, it is important to remember the purpose is to capture requirements, not to functionally decompose the internals of the system. Therefore for requirements

scenarios, the only classifiers that should appear in the diagram are the system actors and the system or the use case, not pieces internal to the system. In the case of subsystem use cases, peer subsystems are treated as actors to the subsystem of concern.

The other approach to requirements capture is via *specification*. The specification may be done in hundreds or thousands of textual statements but text is best used in conjunction with a more formal language, such as state or activity diagrams, to define all possible scenarios. When state diagrams are used in this way, messages from the actors are represented as events on the state diagram while messages from the system to the actors are shown as actions on the state diagram.

1.5 Summary

That's it – your brief refresher on UML! Classes and objects are the usual things that people first consider when they think about UML, but clearly, UML is more expansive and expressive than this. UML also includes primitive behaviors – actions and activities – and ways to specify the allowable sequences of their execution with activity diagrams and state machines. UML provides means for showing example interactions as well, primarily with sequence diagrams. Finally, use cases cluster requirements into usable, coherent functional units. This chapter was not meant to be your *only* introduction to UML. Although we hit the highlights, we didn't go into much depth, and many useful features of UML weren't discussed at all. As mentioned previously, this book is meant to be a companion volume to the author's *Real-Time UML 3rd Edition*, and the interested reader is referred there for more detail on UML.

UML is a language and is intentionally process-agnostic. Any reasonable process can be used effectively with UML. However, not all processes are equally effective. In the next chapter, we'll discuss one process – called the *Harmony process* – that has been developed by the author, in collaboration with others, over the years and applied very effectively in the development of real-time and embedded systems. The problems and answers that come later in this book will follow, more or less, the Harmony process, so the next chapter will provide a brief introduction.

The Harmony Process

What You Will Learn

- The Harmony development process
 - Why a process?
- Harmony process overview
 - Key enabling technologies
 - Process timescales
- Harmony process variants
- Harmony development iteration in detail
 - Analysis
 - Design
 - Translation
 - Verification and validation
 - Party!

2.1 Introduction

A *methodology* consists of a *language* to specify elements and relations of interest and a *process* that tells the developer what parts of the language to use, how to use them, and when to use them. The Harmony process¹ uses UML and variants as the language. This includes profiles such as the Systems Modeling Language (SysML) or the UML Profile for DoDAF and MODFAF (UPDM) for Department of Defense Architecture Framework models. The Harmony Process also specifies an integrated set of workflows to guide the developer so that they can use UML to its fullest advantage in developing robust, capable, and safe systems.

There is a very broad range of development processes in use today, from “We don’t need no stinking process” to very formal rigorous processes. This chapter begins the discussion with

¹ The Harmony process is basically the next revision in the ROPES (Rapid Object-oriented Process for Embedded Systems) process. Harmony is an agile approach to systems and software development, and is requirements-centric and architecture focused. For more details on the Harmony process, see *Real-Time Agility* (Addison-Wesley, 2009), also written by the author.

an overview of the two major variants of the Harmony process, and then gives detailed workflows for each of the phases in the process.

Beginning with the next chapter, problems will be presented for you to work on. The set of problems is presented, roughly, in the order in which they appear when you follow the Harmony process. The order of these problems may vary from how they would appear to you if you follow a different process.

2.2 The Harmony Development Process

A process is an integrated set of workflows. Each workflow takes some aspect, typically a phase in the process, and elaborates what activities are necessary for the workers to accomplish, when and how they are going to accomplish it, and what work products they generate.

A good process provides guidance on an effective way to develop high-quality systems at a minimal cost. Far too many processes are either completely nonexistent or waste valuable developer time and resources doing the wrong things, such as generating reams of paperwork. A good process usually produces some paper work products, but only those that add value, and even then in a cost-effective manner.

2.2.1 Why a Process?

The basic reason why we, as software and system developers, should be concerned about and use a good process is to improve our lives and our products. Specifically, a good process

- Provides a project template to guide workers through the development and delivery of a product
- Improves product quality in terms of
 - Decreased number of defects
 - Lowered severity of defects
 - Improved reusability
 - Improved stability and maintainability
- Improves project predictability in terms of
 - Total amount of effort
 - Length of calendar time required for completion
- Communicates project information appropriate to different stakeholders in ways that they can use effectively

If you have a process that doesn't achieve these goals, then you have a *bad* process and should think about changing it for the better. These goals *can be* achieved with a good process or they can be *inhibited* by a bad process.

So what's a process? In Harmony, we define a process in the following way:

A **process** is the specification of sequenced set of activities performed by a collaborating set of worker roles resulting in a coherent set of project products, one of which is the desired system.

A process consists of *worker roles*, the “hats” worn by *workers* while doing various project *activities*. Each activity results in the creation or modification of one or more *work products* (also known as *artifacts*). For example, most processes have requirements capture (activity) somewhere early on before design occurs. This is performed by a requirements analyst (worker) acting as a software modeler (a worker role), and might result in a work product, such as a portion of the software model from which code will be generated. Figure 2.1 depicts these fundamental aspects and relations inherent in a development process.

The activities are the tasks that the worker does in performance of his or her duty. The activities are grouped together into workflows focused around a common thread, such as

- The work done in a development phase
- To achieve a specific goal
- To create a specific work product.
- The work done by a particular worker role

A process is normally organized into *phases*, which might be thought of as the largest-scale work activities. Each phase is specified with one more workflows. Each workflow is a sequenced set of activities – simple tasks performed by workers – with resulting work products. A common way to represent workflows is with UML activity diagrams. Another is to use a tool specifically designed to create and manage process content, such as Rational Method Composer² (RMC), the approach that will be used here.

Work products are the things created or modified during activities. The singularly most important work product is the system being produced, but there are many others, such as the source code, the software model, the requirements specification, hazard analysis, safety analysis, test vectors, and so on. Generally speaking, every activity results in the creation or modification of at least one work product.

The Harmony process, described in more detail in the next section, is applicable to (and in current use in) projects of widely different scale. Harmony achieves this scalability in a couple of different ways. First, the process is viewed at multiple timescales – macro (project timescale), micro (iteration timescale), and nano (work item timescale). Smaller projects will give much more attention to the micro- and nanocycles, but as the projects grow in size, more attention is shifted to the macro scale to organize and orchestrate the entire

² <http://www-01.ibm.com/software/awdtools/rmc/>.

development process. Secondly, a number of work products are optional and created during the process only as needed. Hazard analysis, for example, is only used for safety-critical applications. The subsystem architecture view, for another example, is only created when systems are large enough to profit from such decomposition. In fact, the Harmony process has two major variants: one, called the Full Harmony process, includes a detailed systems engineering process that precedes subsystem development, and another, called Harmony/ESW (Embedded SoftWare), focuses on software development only.

Despite its well-known problems, the waterfall lifecycle is still by far the most common way to schedule and manage projects, especially for embedded systems. Nevertheless, the most

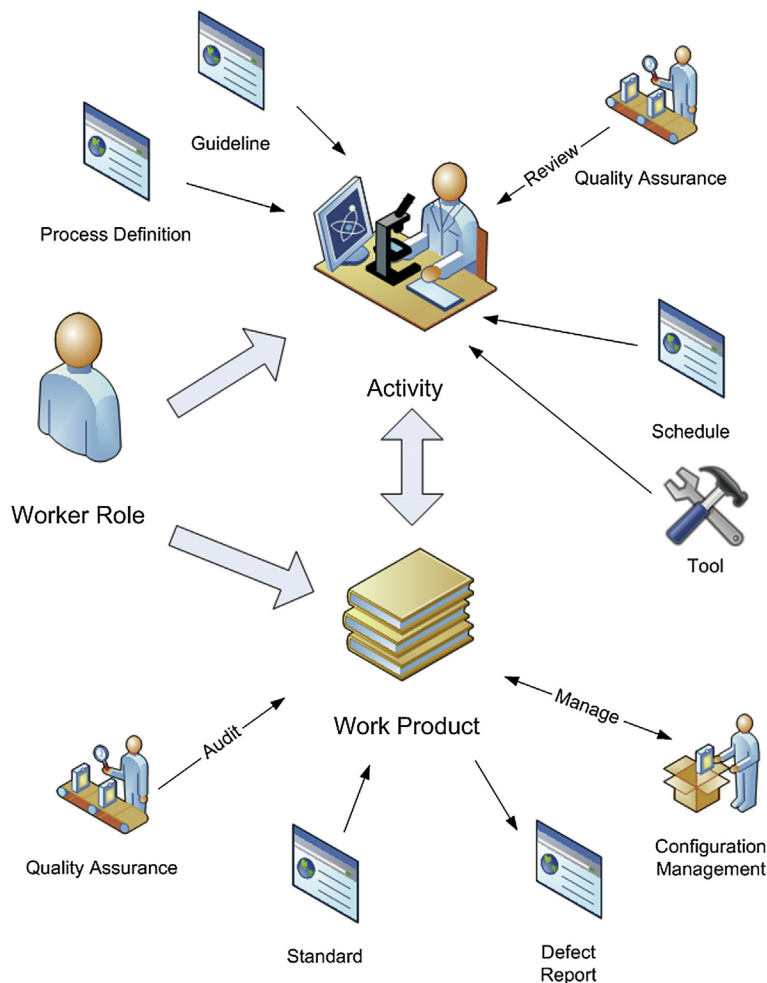


Figure 2.1
Basic elements of a process.

fundamental issue with the waterfall lifecycle is that defects introduced early in the process are not identified or fixed until late in the process. Certain kinds of strategic defects – requirements and architectural defects, specifically – are three or four orders of magnitude more expensive to repair in the waterfall lifecycle because they have broad sweeping implications. This is inherent in the waterfall lifecycle *because testing comes at the end*. The longer you wait to identify and repair defects, the more they have become entrenched and the greater the number of dependencies on the flawed aspects. Put another way, the problem with the waterfall lifecycle is that it fundamentally assumes each step in the process can be completed without making mistakes, but in fact, this is demonstrably untrue. When the defects are finally identified and repaired, the cost is very high because of the cost of defect localization and rework.

The spiral (also known as the iterative) lifecycle has become popular to address the concerns associated with the waterfall lifecycle. The basic advantage of the spiral lifecycle is that the system is tested far earlier and far more often. This results in the identification and repair of defects much earlier and at a significantly reduced cost. The spiral lifecycle essentially divides the development project into a set of smaller projects, and incrementally adds capabilities to the system, but not before verifying and validating the ones already present. The effort to introduce each additional set of capabilities is called a *spiral* or *increment*. Each of these subprojects is more limited in scope, is produced with much greater ease, and has a much more targeted focus than the entire system. In the Harmony process, the “spiral” is known as the *development iteration*.³ The primary outcome of each iteration is what Harmony calls an *iterative prototype* – a functional, high-quality system that is not as complete as the final system, except for the last iteration. Nevertheless, the prototype does correctly implement and execute some portion of the requirements and/or reduce some set of risks and contains the actual code that will ship with the product, once complete. The purpose of the work within a single iteration is identified in its *mission statement*, a clear delineation of the functionality to be developed, the risks to be mitigated, the architectural concepts to be implemented, the set of existing defects to be repaired, and the target platforms to be supported.

The Harmony process can be conceptualized as occurring simultaneously in three different scales or time frames (see [Figure 2.2](#)). The *macrocycle* occurs over the course of many months to years and guides the overall development from concept to final delivery. The Harmony macrocycle has four primary, but overlapping, phases. Each macrophase actually contains multiple iterations, as we will see shortly, and the primary outcome of each development iteration is the production of an iterative prototype.

³ This is formerly known as the Harmony *microcycle*. I changed to name to align with more common terminology to avoid confusion.

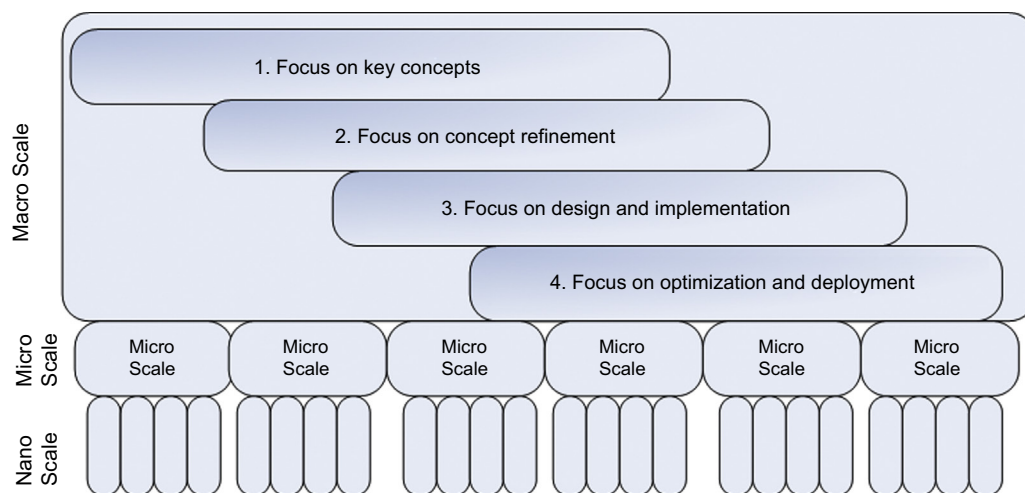


Figure 2.2
Harmony process time frames.

The macrophases are a way to show that the missions of the prototypes tend to evolve over time in a standard way. The early prototypes tend to focus on key concepts, such as requirements, architecture, or technology. The next several prototypes introduce and focus on the secondary concepts of requirements, architecture, and technology. After that, the focus shifts to design and implementation concerns. The last set of prototypes emphasizes optimization and deployment (in the target hardware and in the customer’s environment). The shift in focus of the prototypes tends to be gradual, hence the overlapping nature of the macrophases.⁴

Each macrocycle contains several development iterations (microcycles). Each iteration is fairly short, usually completing within four to six weeks. Each iteration is focused around the production and delivery of a single incremental prototype with limited, but high-quality, functionality. This is most commonly focused around one or a small number of use cases, but may also include specific risk-reduction activities.

Within the iteration, the developers work to produce the high-quality object collaborations that realize the use cases of the prototype. During this process, the increasingly complete collaborations are executed dozens to hundreds of times. This very short execution cycle – in the order of minutes to hours (at the long end) – is called the nanocycle. If an object collaboration will ultimately consist of 100 objects, experience has shown – clearly – that the best way is NOT to put down all 100 objects and say “Oh God, I hope this works,”⁵ but

⁴ If required, Preliminary Design Reviews (PDRs) and Critical Design Reviews (CDRs) are easily incorporated into the process. Usually a PDR occurs at or near the end of the first macrophase and a CDR occurs at or near the end of the second macrophase.

⁵ I’m a strong advocate of the separation of prayer and engineering. Let’s use *science*, kids.

instead to start with one (incomplete) object and get that to work in isolation, through model execution. Then add another object and get them to work together. Then refine the objects, or add a third; and so on, making small enhancements to the capabilities supported in the collaboration but validating, through execution, each small incremental step. Executable modeling tools, such as Rhapsody™, make this process highly efficient. The basic premise of the nanocycles is to make tiny incremental steps and demonstrate through execution that they are right before adding the next. In fact, this is a model-based version of Test-Driven Development (TDD), a key practice of agile methods.⁶

2.2.2 Harmony Process Overview

The Harmony process is a general systems development process that, while emphasizing the real-time and embedded software development aspects, includes the steps to produce general-purpose software and systems. The Harmony process has been used effectively on very small one to three person projects as well as large teams consisting of hundreds of team members. Harmony is, in fact, a disciplined approach to developing embedded systems that is both agile and rigorous.

The Harmony process comes in two generic forms. The first is intended for projects that are larger in scale and have significant hardware-software co-development. This form of the process is known as the *Full Harmony process* or the *Harmony Hybrid process*, and is depicted schematically in [Figure 2.3](#). This version of the process supports a continuum of interactions between systems engineering and downstream software development. In some projects, the hardware is already designed or uses commercial off-the-shelf (COTS) hardware. In this case, the work products from systems engineering (notably system requirements and architectural specifications) can be handed off to downstream engineering incrementally. This allows software development to begin sooner than in a more traditional V-cycle process ([Figure 2.3A](#)). At the other end of the continuum, many projects require complete new designs for mechanical and electrical hardware. In this case, the systems engineering work products must be more complete prior to the hand-off ([Figure 2.3B](#)) because of the long lead times for hardware manufacturing and the high cost of hardware refactoring. All projects fall somewhere in this continuum and the degree of completeness before systems engineering work products can be handed off depends upon where they fall within it.

The general form of the Harmony process shown in [Figure 2.3](#) has an up-front effort – referred to generically as “systems engineering” – in which the requirements are fully specified and organized into use cases, the subsystem architecture is defined, and the requirements are allocated to the subsystems. In the hand-off activity, requirements are allocated to the engineering disciplines of mechanical, electronic, chemical, and software.

⁶ See my book *Real-Time Agility* (Addison-Wesley 2009) for more information.

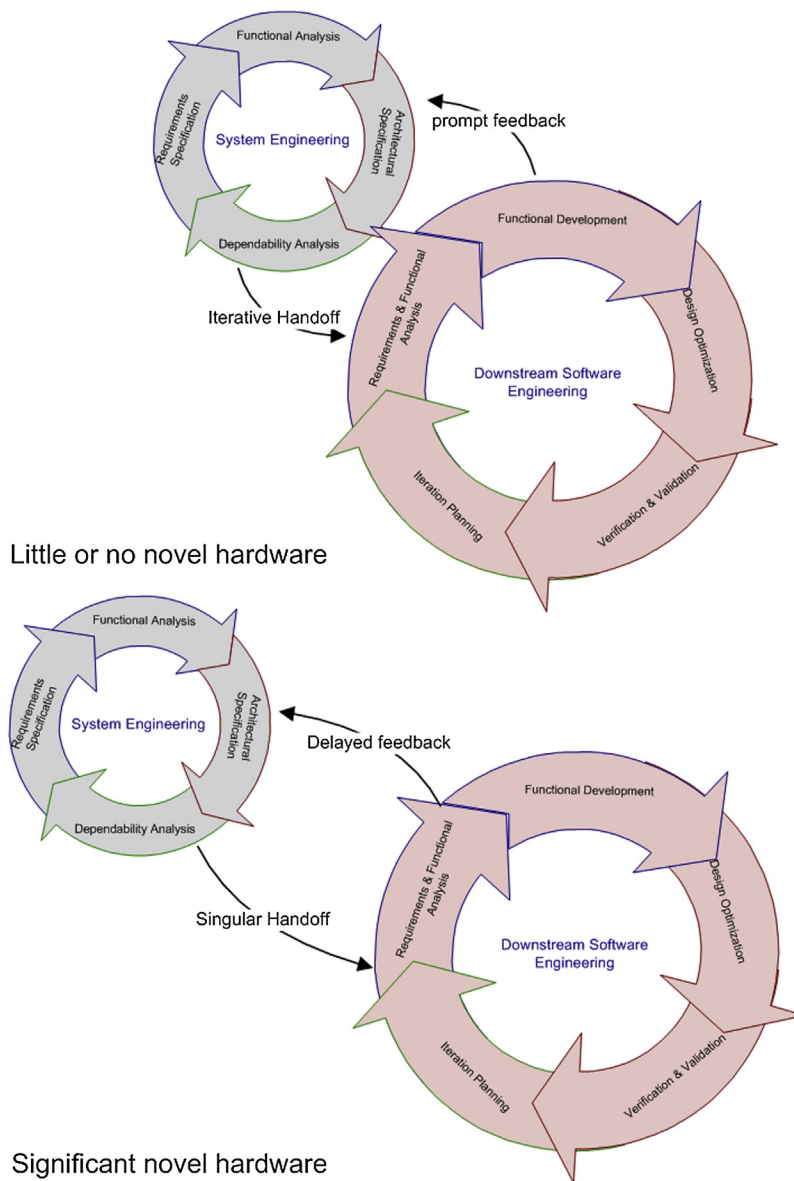


Figure 2.3
Full Harmony (or Harmony Hybrid) workflow.

The primary phases in the system engineering part of the process are shown in [Figure 2.4](#) and will be discussed in more detail in the next section.

After the engineering phases, the incremental development cycle can begin. At this point, the two variants of the process – Full Harmony and Harmony/ESW – are remarkably similar. There is an important difference in the analysis phases – in the general process, previously

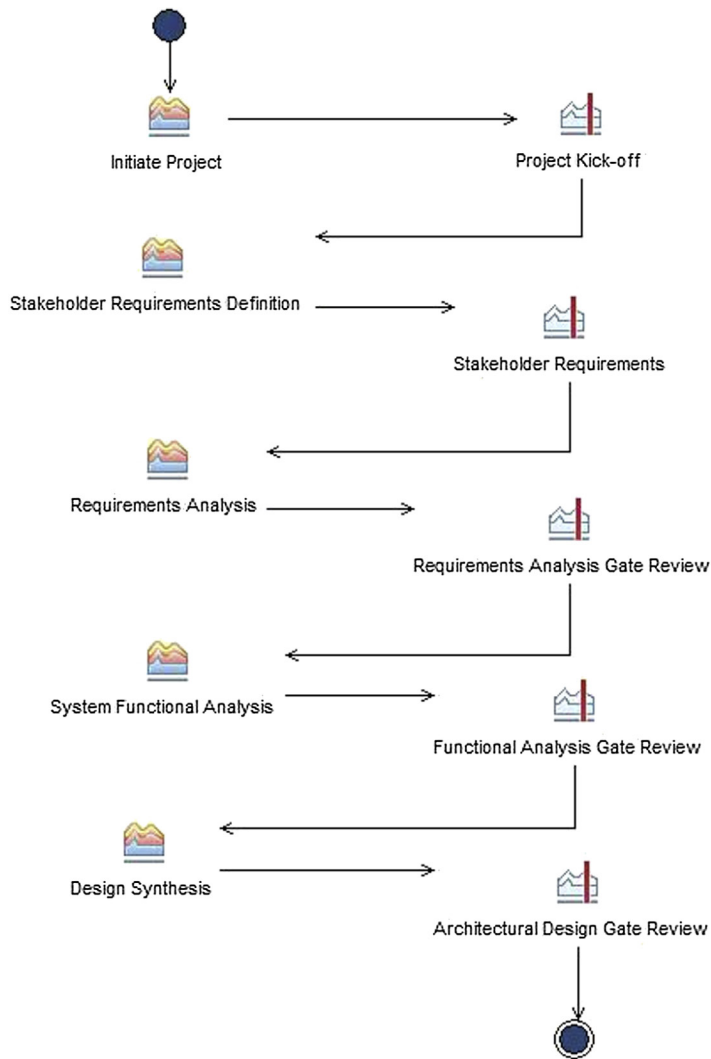


Figure 2.4

Systems engineering activities of the Full Harmony process.

detailed use cases are selected and used as the basis for the development, while in the software-only process, the as-yet-unspecified use cases must be detailed as a part of the iteration.

The iterative part of the downstream engineering development process is shown in [Figure 2.5](#).

In the Full Harmony process, the Prototype Definition phase is simply a matter of selection of the use cases previously specified in the systems engineering work and focusing on the refinement of the requirements allocated to software. In the software-only Harmony process, these use cases have been identified (named and given a single paragraph mission statement) but the detailed requirements for those use cases have not yet been specified.

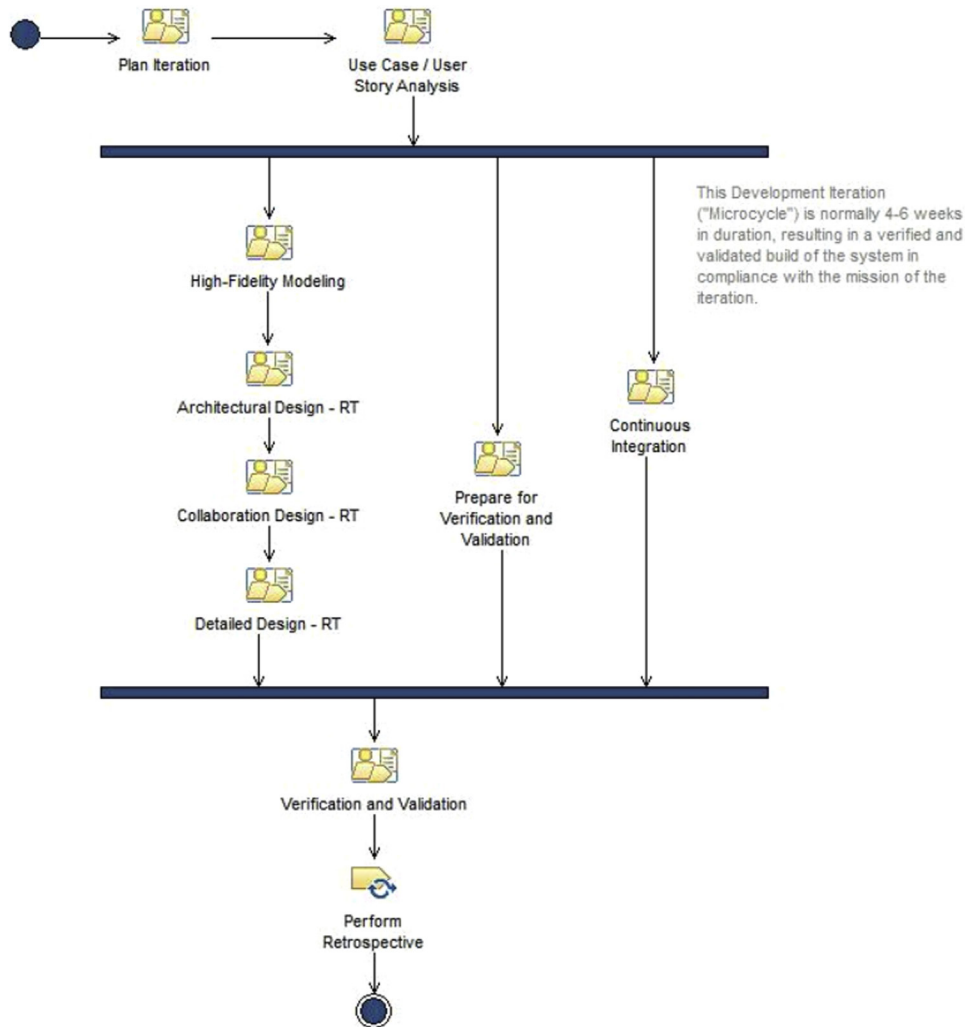


Figure 2.5
Harmony development iteration overview.

In this latter case, the first part of the development iteration details the use cases so that they are fully specified.

The development iteration is the primary iterative activity in the Harmony/ESW process.

Although we will focus on the development iteration activities later in this chapter, I'd like to identify the purpose of the other workflows in [Figure 2.6](#).

Pre-Iteration Planning: This activity focuses on getting ready to start the software work and includes tasks to

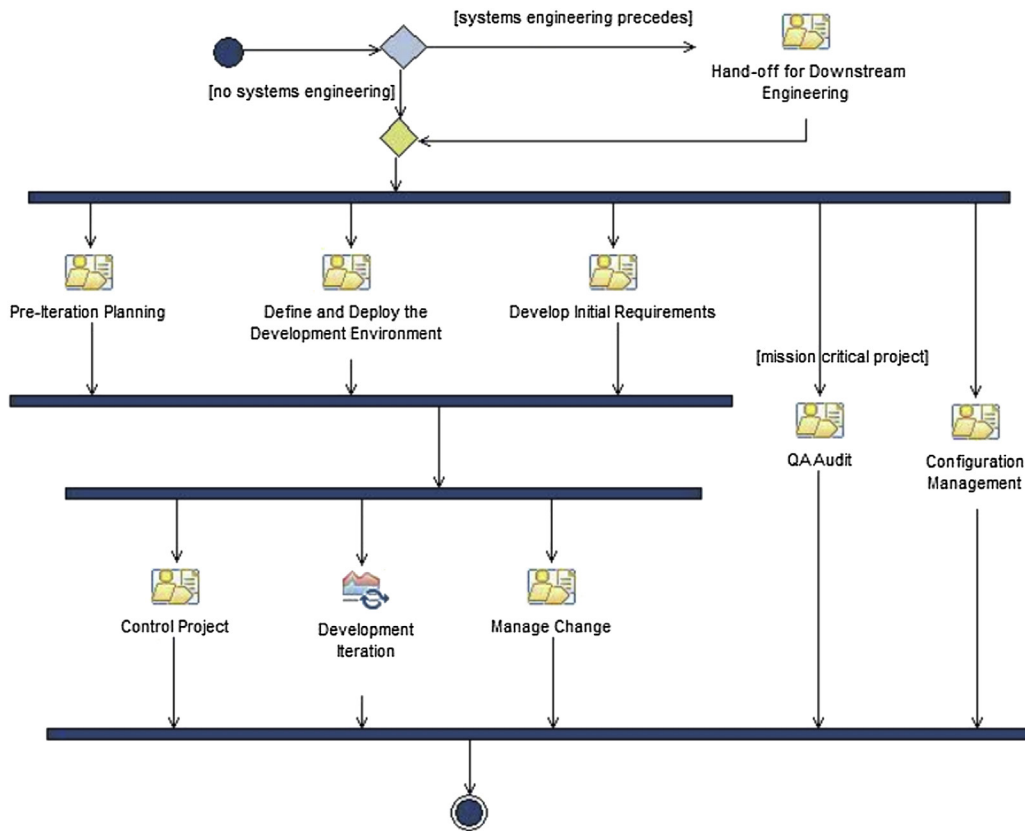


Figure 2.6
Delivery process for Harmony/ESW.

- Create the overall software schedule
- Create the software team structure
- Plan to reuse software or create reusable software
- Identify and plan to reduce project risks
- Specify the overall model and configuration management (CM) structure
- If appropriate, perform initial safety and reliability analysis

Define and Deploy the Development Environment: Here, the goal is to set up project guidance, such as standards and guidelines from the organization’s development process, and tailor the process steps to the project’s needs. This includes setting up the tooling environment.

Develop Stakeholder Requirements: It is common to work with the customer to identify written textual requirements to meet their needs. Stakeholder requirements are statements of stakeholder needs that are converted to system requirements (statements of what the system does) in the Use Case/User Story Analysis activity. Note that this usually needn’t be done in the Full Harmony process, because that work has already been done by the system engineers.

Control Project: This activity operates in parallel with the iteration and includes tasks such as updating the risk management plan, monitoring project progress and updating the schedule, and – where appropriate – performing ongoing safety assessment of the development work and updating the hazard analysis.

Manage Change: This activity manages changes, such as requirements changes and problem (defect) reports, usually resulting in updated requirements, designs, code, and tests.

Configuration Management: Just like it says – all work products must be under configuration management, including specifications, models, code, test cases, test results, quality assurance records, etc.

Quality Assurance (QA) Audit: For safety-critical and high-reliability projects, it is common to conduct QA activities (audits and reviews) to ensure the syntactic correctness of the work. Audits look at the execution of some percentage of the work tasks and ensure that those tasks are being performed properly, as specified by the process definition. QA reviews, on the other hand, look at work products to ensure that they meet the project work product standards, such as requirements, design, and coding standards.

2.3 The Systems Engineering Harmony Workflows in Detail

The Harmony system engineering has been discussed from an overview perspective ([Figure 2.4](#)), but in order to understand how to use the process, it is necessary to understand in more detail the work activities and work products produced.

2.3.1 System Functional Analysis

[Figure 2.7](#) shows the System Functional Analysis workflow. It basically specifies that the use cases are analyzed, whether one at a time or simultaneously, and a use case model is constructed for each. Each use case is specified with an *executable model* that is constructed using semantically complete modeling provided by the UML/SysML and is linked to the requirements it represents. The details of how this is done are discussed in the next section. Each use case is validated via execution to ensure that it is complete, correct, consistent, and accurate.

It is important that the use cases are independent, that is, that each requirement is mapped to a single use case. The use case analysis can be done incrementally and then handed off to the waiting software development team or the set of use cases can be fully analyzed before being handed off to downstream development, which includes multiple engineering disciplines.

The reader should note that we will use objects to represent the use cases, detail their interactions with sequence diagrams, and specify their behavior with state machines and/or activity diagrams. The fact that we are using these semantically precise languages for modeling *does not mean that we are doing design!* This is a common misunderstanding by many people.

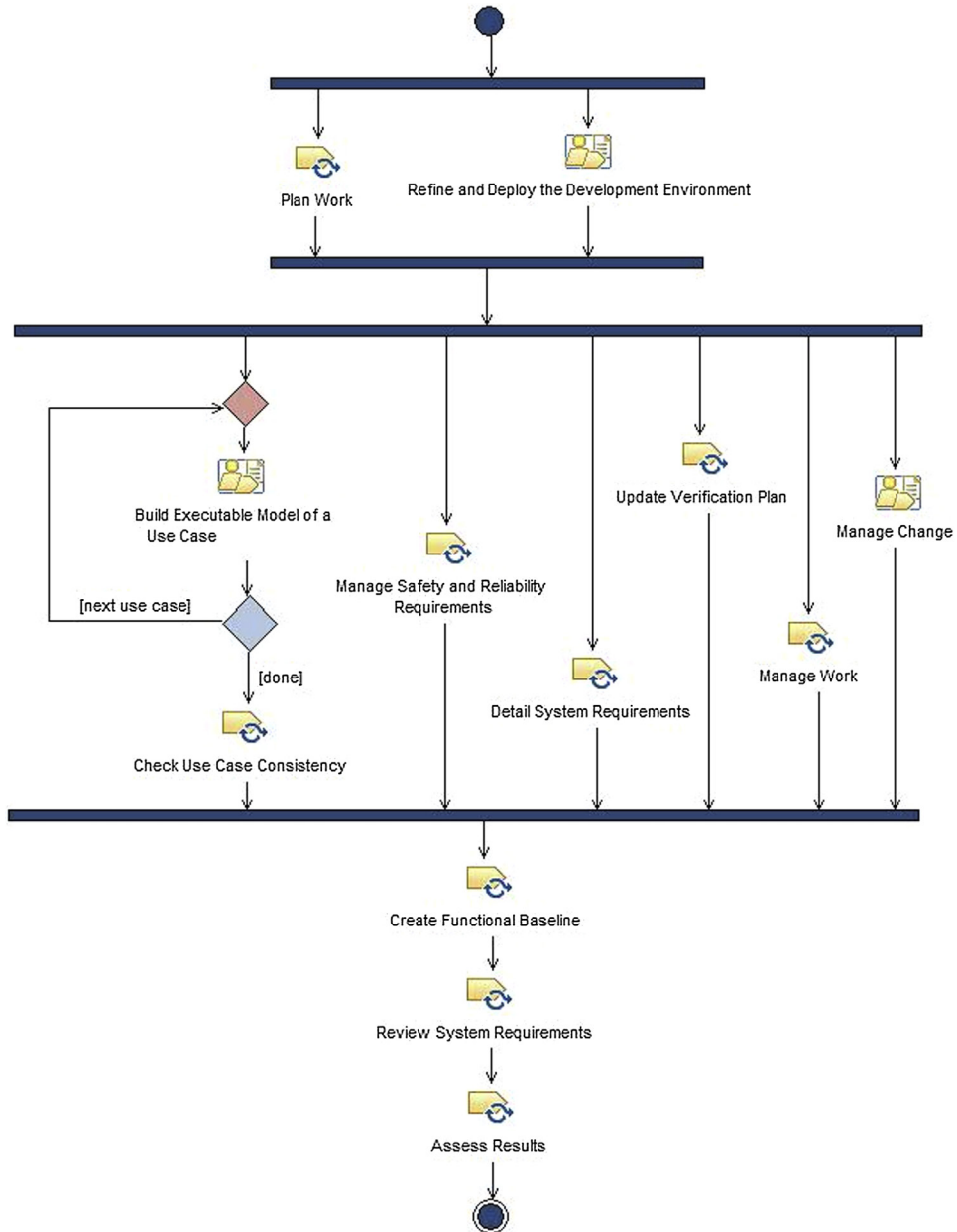


Figure 2.7
System Functional Analysis workflow.

The use of a semantically precise language, such as state machines, has nothing to do with *what* we are saying, merely that we are using a precise language to *say it*. In this context, we use semantically precise languages to specify the requirements but say nothing (yet) about design or implementation concerns. Textual requirements are still useful, even though they lack the precision of a more formal language, and must trace to their more precise counterparts (model elements on state, activity, and sequence diagrams).

2.3.2 Build Use Case Model Workflow

One of the steps in the previous workflow ([Figure 2.7](#)) is “Build the Use Case Model.” That step is detailed in the next workflow ([Figure 2.8](#)). The workflow shows three alternative approaches that represent personal preferences. By the end of the workflow, you’ll have created sequence diagrams showing the typical and exception interactions of your system with its environment, a summary of those sequences in an activity diagram, and a state machine providing an executable behavioral model of an object that represents the system use case.

Why represent the use case as an object? Use cases are themselves classifiers in UML and can have behavioral dynamics, such as state machines. However, you can show neither interfaces nor ports on use cases. Thus, we find it convenient for technical reasons only to model the use case with an object for this purpose. The object is nothing more than a notational convenience and can be thought of as a “use case” with a different notation.

This analysis is called “black box” because the internal structuring of the system isn’t known or used at this time. In the next workflow, we’ll go “open box,” identifying subsystems and allocating functionality to them.

2.3.3 Design Syntheses: System Architectural Design

The next workflow is specifies the overall system architecture. This is done by identifying coherent functional blocks – represented as subsystem objects – along with their connection points (ports) and interfaces. The operational contracts are allocated to these subsystems. At this point, each subsystem is still “mixed discipline” – that is, it contains elements from various engineering domains, such as electronics, mechanical, and software. These subsystems are validated by executing them together and showing how they collectively reproduce the very same “black box” scenarios specified in the previous workflow. This workflow is shown in [Figure 2.9](#).

2.4 The Hand-off from Systems Engineering

The hand-off from systems engineering is not a zero-time milestone activity. It is an activity that allocates subsystem specifications (including requirements) to the cross-discipline engineering teams that are to implement them. The workflow, illustrated in [Figure 2.10](#), shows two primary flows.

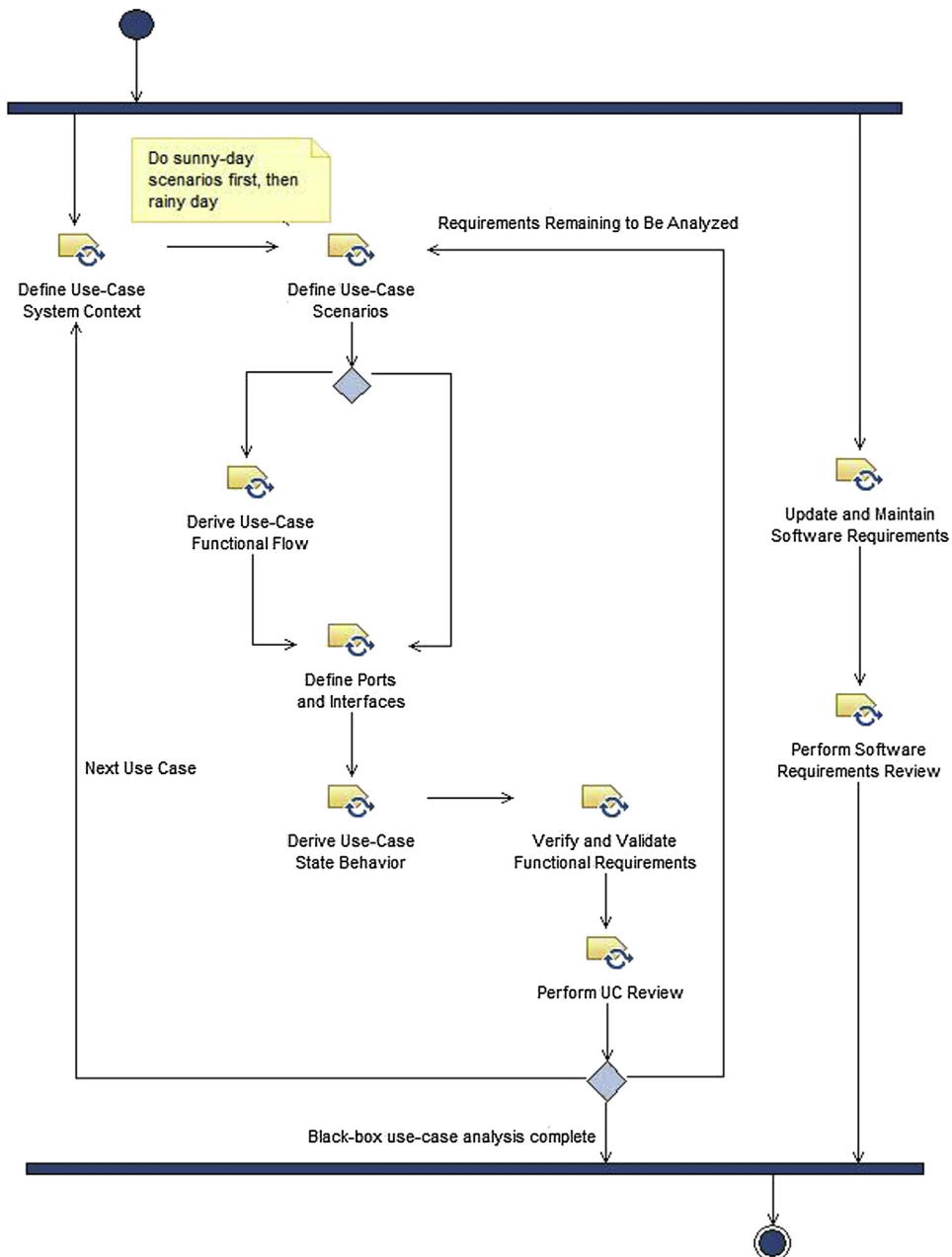


Figure 2.8
Build Use Case Model workflow.

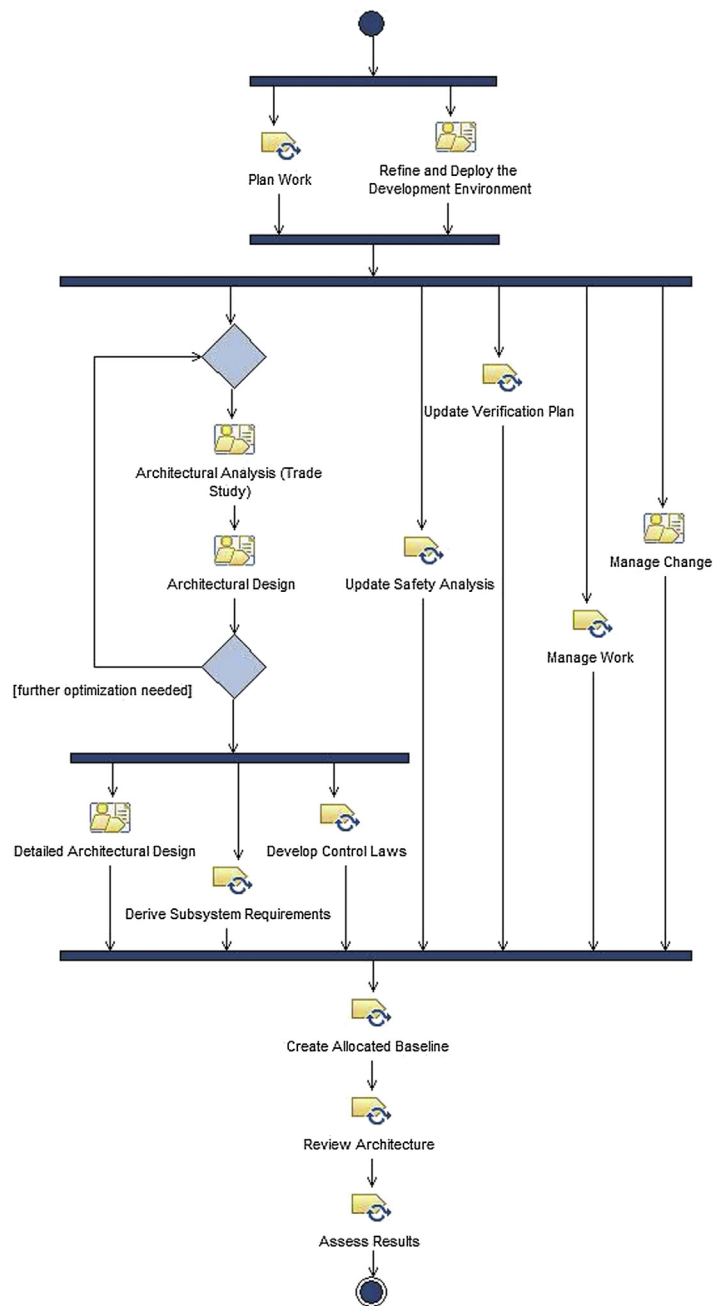


Figure 2.9
System Architecture Design workflow.

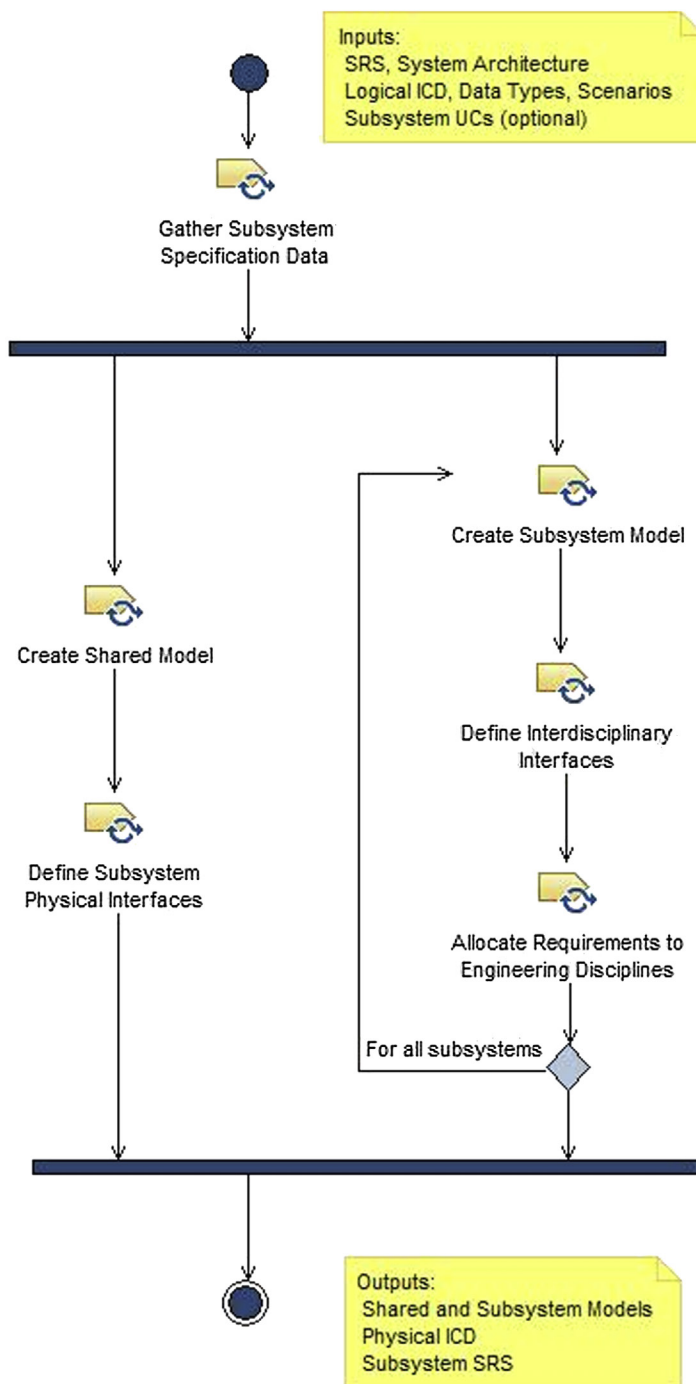


Figure 2.10
Hand-off from systems engineering.

The left flow is about allocating specification items shared by more than one subsystem into the *shared model*, a repository for shared elements among the subsystem teams. These shared specification items include subsystem interfaces and the data and interface types that they use.

The right flow is repeated for each subsystem. An Interdisciplinary Project Team (IPT) is formed with representatives from all involved engineering disciplines (e.g., systems, software, electronic, and mechanical engineering) that both allocates requirements to these disciplines and also defines the interdisciplinary interfaces. In this way, relevant engineers have input into not only the allocation of responsibilities among the engineering disciplines but also how the software will invoke the electronics, how the electronics will interact with the mechanical parts, etc.

At the end of the hand-off activity, each subsystem has their set of requirements allocated into the engineering disciplines, the interfaces between the subsystems are defined, and the interfaces within the subsystem between disciplines are defined. This doesn't mean that these specifications and allocations will never change, but they are put under configuration control and are managed, as all crucial work products must be.

2.5 The Software Workflows in Detail

At the end of the workflows in the previous section, the model is handed off to the interdisciplinary subsystem teams and work enters the development iteration. In this section, we detail only the software workflows of the development iteration, but the reader should understand that in the Full Harmony process, engineers of other disciplines are concurrently working in an incremental fashion as well. Integration occurs in the testing phase of the development iteration, and this integration may include mechanical, chemical, electronic, and software disciplines. There may not be hardware components to integrate in any specific iteration instance, but there often is. It might be breadboard or wire-wrapped electronics, with mock up or hand-built mechanicals, or first-run factory electronic and mechanical components. The point is to avoid the shotgun integration at the end of the project and plan for incremental integration as early as possible.

Figure 2.11 shows an abstract view of the Harmony/ESW process more common in the agile world. This view emphasizes the incremental iterative nature of the software development effect and focuses on two time frames – the (monthly or so) iteration and the (hourly or so) nanocycle. From an overview perspective, I prefer this view, but from a detailed technical perspective, I prefer Figure 2.5.

2.5.1 Analysis with the Harmony Process

The purpose of analysis is to define the essential properties of the system to be developed. “Essential” properties are those that will make the system *wrong* or *incomplete* if they are

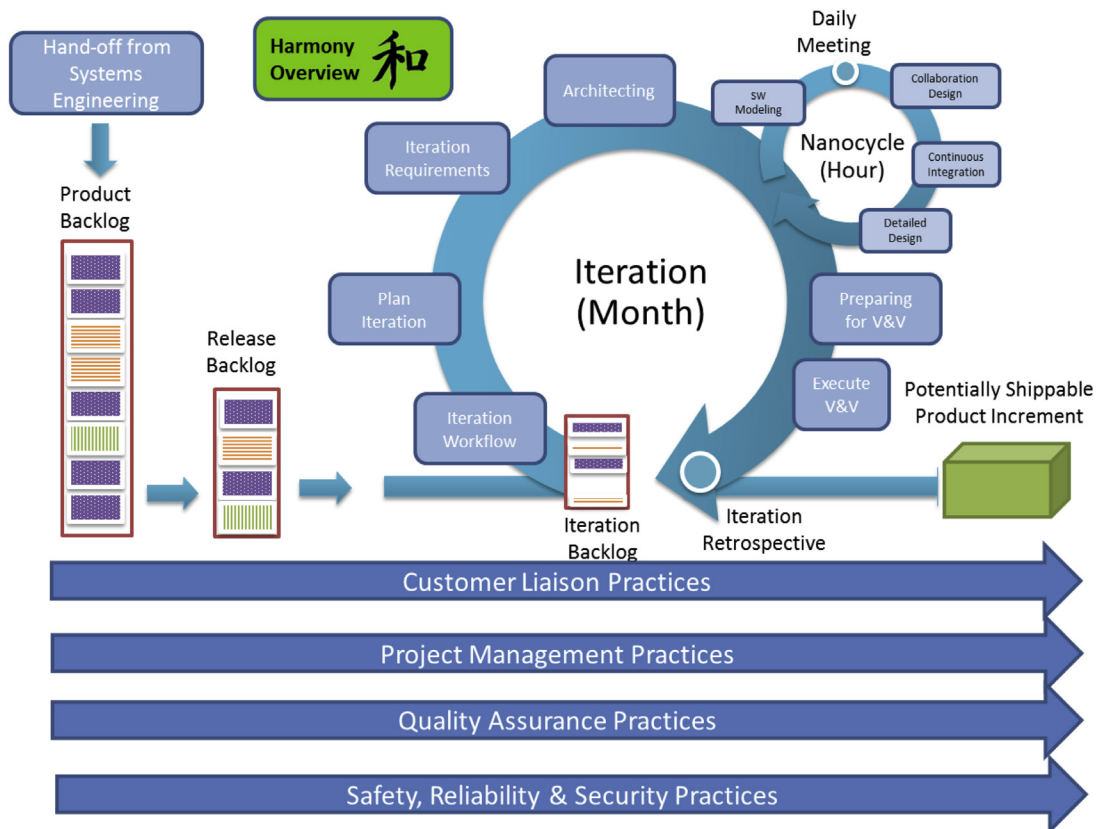


Figure 2.11
Harmony software process: agile view.

missing. In Model-Driven Architecture (MDA)⁷ terms, we are constructing a Computation-Independent Model (CIM) and a Platform-Independent Model (PIM) of the prototype capabilities in the analysis phase. The two primary workflows in the analysis phase of the spiral are Prototype Definition (which details the CIM) and High-Fidelity Modeling⁸ (which constructs the PIM).

Iteration Planning

Iteration planning includes specification of the intent of the effort in the development iteration and production of the Iteration Mission Statement. The Iteration Mission Statement includes work items in the release backlog allocated to the iteration backlog, including

⁷ See www.omg.org for a set of standards that defines MDA.

⁸ Previously this activity was known as *Object Analysis*, but it has recently been changed to High-Fidelity Modeling to reflect the fact that this step works with functionally based software modeling as well as object-oriented.

- Use cases to be realized
- Architectural intent to be developed
- Existing defects to be repaired
- Known risks to be mitigated
- Target platforms to be supported
- Any additional tasks or efforts to be performed by the team during iteration

Use Case/User Story Analysis workflow

This workflow specifies the software requirements to be realized in the current iteration. In the Full Harmony form, this effort refines the requirements allocated to software from systems engineering. This may range in effort from essentially no effort to a complete software-specific requirements specification effort, depending on how well the systems engineering has been done and how much more detail is needed by the software developers. On the other hand, in the Harmony/ESW form, the use cases have been identified but not detailed, and so at this point, the use cases being realized in this prototype must be completely specified (the use cases that will be realized in later prototypes are ignored for now). We will focus just on the Harmony/ESW form in this section. The use cases for the prototype have already been identified, but the detailed specification of what the use cases contain has not yet been specified. The workflow for this effort is provided by [Figure 2.12](#).

There are two primary ways to detail a use case: by example and by specification. By “by example,” we mean that a (possibly large) set of scenarios is created that illustrates typical and exceptional uses of the system for the use case in question. The advantages of scenarios are that they are easy for nontechnical stakeholders to understand and they can serve as a basis for the set of test vectors to be applied later to the completed prototype. The disadvantages of scenarios are that requirements of a use case are spread out over possibly dozens of different sequence diagrams rather than being in a single place, and the requirements may be difficult to represent concisely. Scenarios are almost always represented with UML sequence diagrams, but UML timing diagrams might be preferred in some cases.

The other approach to detailing use cases is “by specification.” This specification is usually a combination of informal text (usually formulated as “shall” statements) and a formal behavioral specification in UML state or activity diagrams. Traceability links match up the informal text with the more precise language of UML. The advantages of detailing use cases by specification are that it is concise, it can typically be made more precise than scenarios, and it is easy to represent requirements that are difficult to show in scenarios. The disadvantages are that it is more difficult for nontechnical personnel to understand, and directly relating the requirements to the design may also be more difficult. For situations where continuous and piecewise continuous behavior is required, we recommend using different control law diagrams or activity diagrams to represent the continuous behavior of these individual states within the use case specification.

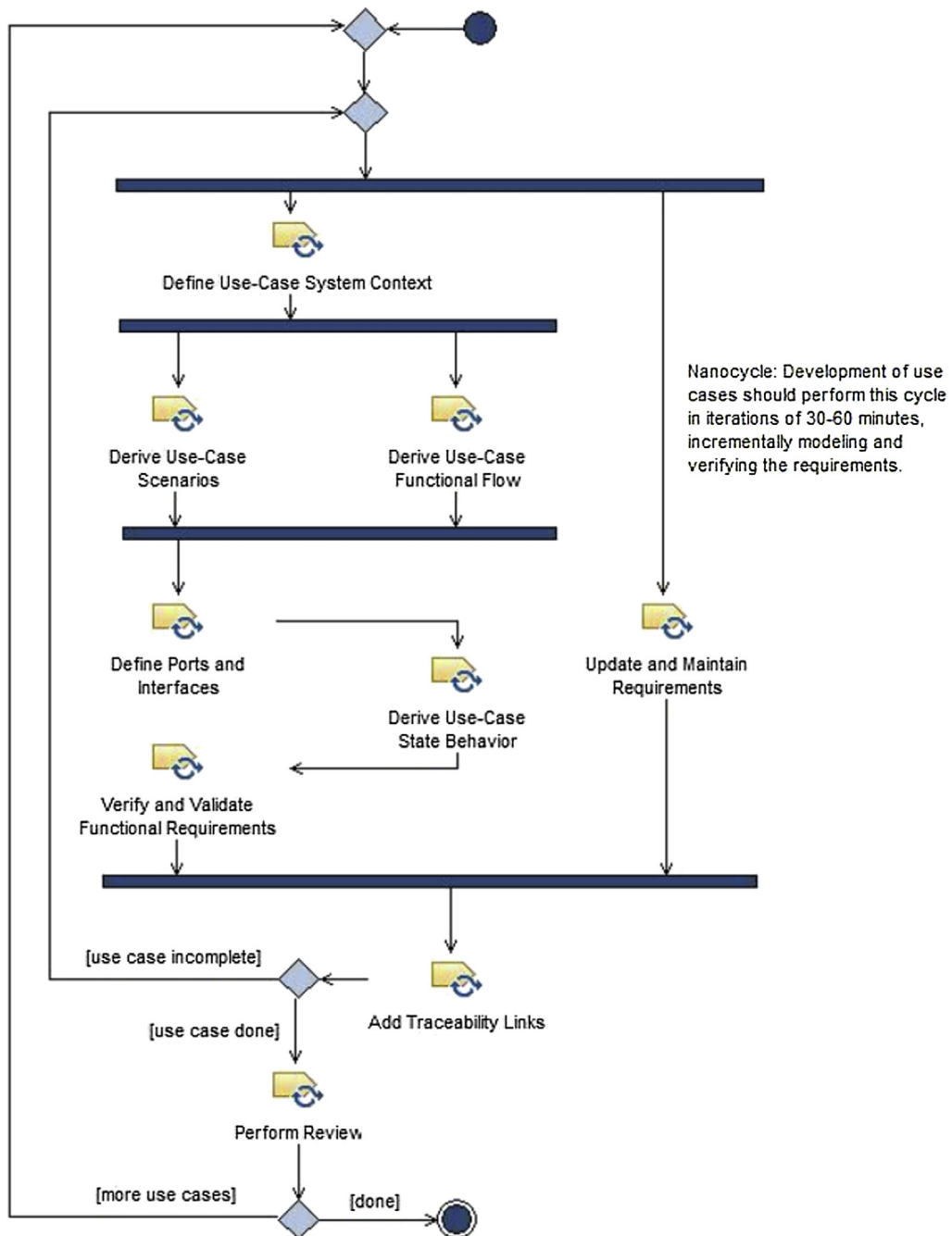


Figure 2.12

Use Case/User Story Analysis in the Harmony/ESW process variant.

We find it best to use both informal text and formal languages together for the use case specifications. Natural language is excellent at explaining “why” because it is both rich and expressive. However, it is also vague, ambiguous, and imprecise. Formal languages excel in precise statements about “what” is needed. A combination of a precise formal description, such as with a state machine, coupled with explanatory text is the best of both worlds.

Both exemplar and specification approaches are useful, and in fact the Harmony process recommends that *both* are used together. A formal specification using state machines or activity diagrams captures the requirements concisely, while scenarios derived from the formal specification can aid the nontechnical stakeholders in understanding the system. Further, the scenarios derived from the formal specification may be used to generate the test vectors for validation at the end of the development iteration. At the end of this activity, the use cases to be realized in the current iteration are fully specified and understood even though use cases and requirements for future iterations may not be.

Requirements⁹ are detailed using a combination of

- Sequence diagrams
- State machines
- Activity diagrams
- Control law diagrams (non-UML)
- Textual descriptions
- Quality of service (QoS) constraints
- (SysML) requirements diagrams

A good way to represent requirements associated to a use case is to, for each use case, create a use case diagram populated with requirements and their link to that use case. This is a good way to add the links and visualize them in a limited context (e.g., a single use case), but summary tables can be generated once the links are added into the model repository.

High-Fidelity Modeling Phase

A use case can be thought of as a bag that contains a set of detailed requirements relating to a single system capability or operational usage. The realization (implementation in UML-speak) of a use case is a collaboration – a set of objects working together to achieve this coherent set of requirements.

High-Fidelity Modeling in the Harmony process constructs this collaboration of *essential* software elements, and is performed one use case at a time. This means that for the current prototype, one collaboration is constructed for each use case implemented by the prototype.

⁹ UML does not have the notion of a “requirement” as a first-order concept. In the Systems Modeling Language (SysML), a UML profile specialized for systems engineering, requirements elements are explicitly representable. For more detail, see the SysML specification: <http://www.omg.org/cgi-bin/doc?formal/12-06-02.pdf>.

In MDA terms, the essential model is called the Platform-Independent Model (PIM). The Harmony process constructs the PIM in an incremental fashion, one (or a few) use case(s) at time. At the very same time, the software baseline from the PIM is created and unit tested so that at the end of the High-Fidelity Modeling activity, both a PIM and functionally correct (but not optimized) source code is available. The High-Fidelity Modeling workflow is shown in Figure 2.13.

Strategies for identification of software elements from requirements statements are always a popular topic among developers. The Harmony process provides a set of common ones from which the developer may choose (see Table 2.1). We have found these strategies to be a remarkably effective way to identify the essential classes, objects, functions, and data types within a collaboration. The application of these strategies will be the subject of Chapter 7.

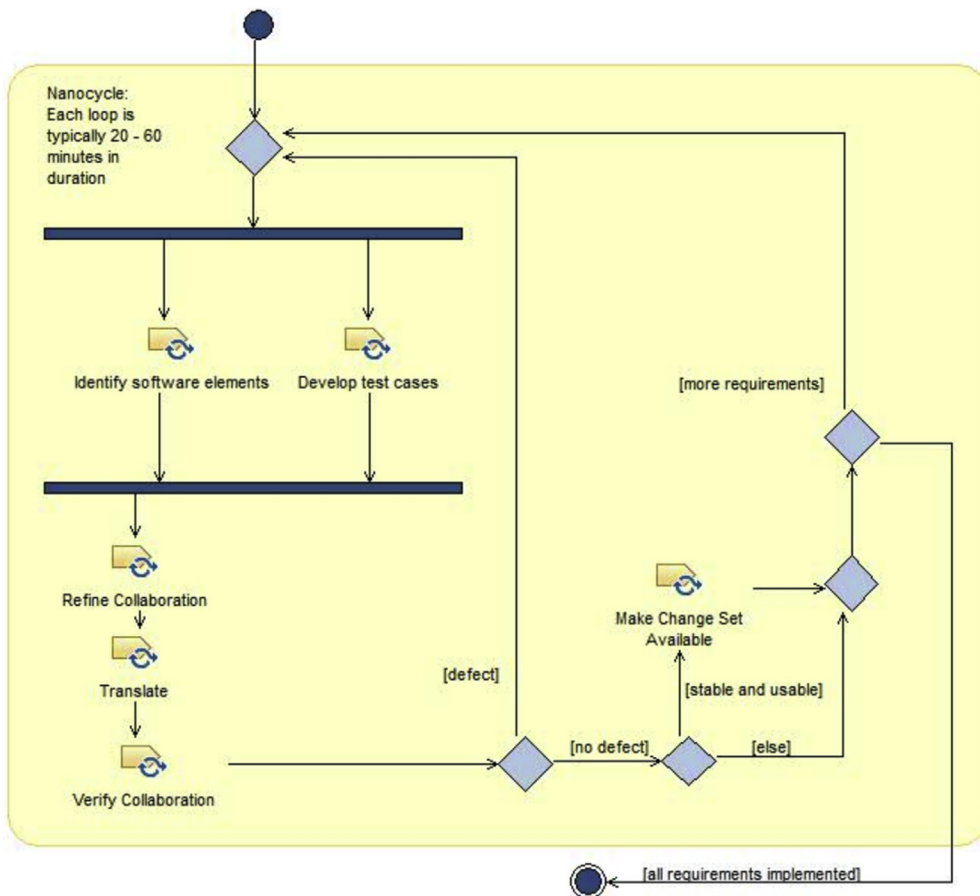


Figure 2.13
High-Fidelity Modeling workflow.

Table 2.1 Essential Software Element Discovery Strategies.

Strategy	Description
Underline the noun	This is used to gain a first-cut object list; the analyst underlines each noun or noun-phrase in the problem statement and evaluates it as a potential object.
Identify causal agents	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify messages and information flow	Messages must have an object that sends them and an object that receives them as well as, possibly, other objects that process the information contained in the messages.
Identify real world items	Real world items are entities that exist in the real world, but are not necessarily electronic devices. Examples include objects such as respiratory gases, air pressures, forces, anatomical organs, chemicals, vats, etc.
Identify physical devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. In the internal architecture, they are processors or ancillary electronic “widgets.” This is a special kind of the previous strategy.
Identify key concepts	Key concepts may be modeled as objects. Bank accounts exist only conceptually, but are important objects in a banking domain. Frequency bins for an online autocorrelator may also be objects.
Identify transactions	Transactions are finite instances of associations between objects that persist for some significant period of time. Examples include bus messages and queued data.
Identify persistent information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify visual elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify control elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects.

Care should be taken to minimize the introduction of optimization decisions (design elements) during the High-Fidelity Modeling phase. Limit the collaboration at this point to elements that clearly must be present in the high-fidelity model for it to be functionally correct. For example, if the collaboration is to model the use case “Manage Account” for a banking system, then if the collaboration does not contain objects such as *Customer*, *Account*, *Debit Transaction*, and *Credit Transaction*, you’d say it was *wrong*. In a navigation system, you would expect to see concepts, represented by objects or their attributes, such as *Position*, *Direction*, *Thrust*, *Velocity*, *Attitude*, *Waypoint*, and *Trajectory*. The goal is to include only the objects, classes, functions, data structures, and relations that are essential for functional correctness and not to include design optimizations. If anything, the PIM should be optimized for simplicity to ease the later optimization that occurs during design.

It is crucial to the process to understand that in High-Fidelity Modeling we are not only creating a model – specifically the PIM – but also producing a source code baseline. As we develop these two key work products, a key question arises - “Is it *right*?” Are the concepts properly represented? Are the relationships among those concepts correct? Do they behave appropriately? The answer to these questions is answered rapidly during the nanocycle. You can see the Harmony High-Fidelity Modeling Nanocycle workflow in [Figure 2.13](#). The idea is to make tiny incremental changes and then quickly execute the collaboration to make sure that you got it right. You can really only evaluate the correctness of an object model via *execution and unit test*. With executable modeling tools such as Rhapsody, this is very fast and easy. If you are using less capable tools, the workflow is the same, but you will manually write the code yourself. Our experience is clear; most developers wait far too long before executing and ensuring the correctness of their models and then pay for it with long, painstaking efforts to get the code finally compiled and correct. It is better to avoid defects in the first place with a continuous execution and test approach, as exemplified with the Harmony nanocycle.

The nanocycles consists of generating and executing the high-fidelity model as it is being developed, rather than waiting until the end. Testing becomes a continuous process rather than something done only at the end, resulting in higher quality systems with less effort and in less time. Take the sequence diagrams used to show requirements scenarios, *elaborate* them with the objects just created, and demonstrate, via execution, that they fulfill the expected roles within that scenario realization. This is the key concept behind the agile method – make tiny steps and verify them before you move on.

2.5.2 Design with the Harmony Process

An analysis model defines a coherent set of required properties of the system under development. These are represented first as a CIM – a set of use cases and its details (e.g., sequence diagram, state machine, activity diagrams, quality of service constraints) – and then as a PIM – a collaboration of essential object roles whose correctness is verified through execution and test. The high-fidelity model is driven primarily by the functional requirements and is demonstrated to be functionally correct. It is *not*, in general, optimal. That optimality is introduced in design.

A design model, known as a Platform-Specific Model (PSM) in MDA, is a concrete blueprint for exactly *how* the essential properties will be realized. An analysis model may be implemented by many different designs with different optimization characteristics. While an analysis model presents a set of possible solutions, a design is a particular solution to the problem. Design should always be an optimization of an analysis model. The design process is basically

- Identify the design (optimization) criteria.
- Rank the design criteria in order of criticality.
- Identify design patterns or technical solutions that optimize the more important of the design criteria at the expense of those of lesser importance.

Many of the design criteria are the quality of service constraints from the analysis. There may be others as well, such as reliability and safety level, reusability, maintainability, simplicity, time to market, and so on.

In the Harmony process, we perform design this at three levels of abstraction. The architectural level of abstraction optimizes the system at an overall gross level. As we will see, there are five key views of the architecture that are optimized more-or-less independently. The mechanistic or collaboration level of abstraction focuses on collaboration-wide optimizations, where a collaboration is a set of software elements and relations that together realize a single use case. The scope of concern here is an order of magnitude smaller than the architectural scope. Lastly, the detailed design is at the level of the individual class, object, function, or data type. At this level, individual objects are optimized and focus is centered on the 5% (or so) of the objects that have special optimization concerns.

Architectural Design Phase

As mentioned earlier, the Harmony process recognizes five important views of architecture (as shown in [Figure 2.14](#)):

- Subsystem and Component view
- Concurrency and Resource view
- Distribution view
- Dependability view
- Deployment view

In the Architectural Design phase of the development iteration, zero or more of these views is elaborated, depending on the mission of the current iteration. This is done primarily via the application of architectural design patterns (see the author's *Real-Time Design Patterns*¹⁰ or *Design Patterns for Embedded Systems in C*¹¹ for an in-depth presentation of patterns in each of these architectural views). These patterns are large in scope, affecting most or all of the system.

Architectural design representation uses the same UML diagrams as in systems architecture and High-Fidelity Modeling: class diagrams to represent the structure, sequence diagrams to represent collaborative behavior, and state machines to model the behavior of individual elements.

The *Subsystem and Component view* identifies the large-scale architectural pieces of the system, their responsibilities, and their interfaces among each other and to the external actors. The *Concurrency and Resource view* identifies the concurrency units (modeled as

¹⁰ Douglass, Bruce Powel. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2002).

¹¹ Douglass, Bruce Powel. *Design Patterns for Embedded Systems in C* (Elsevier Press, 2010).

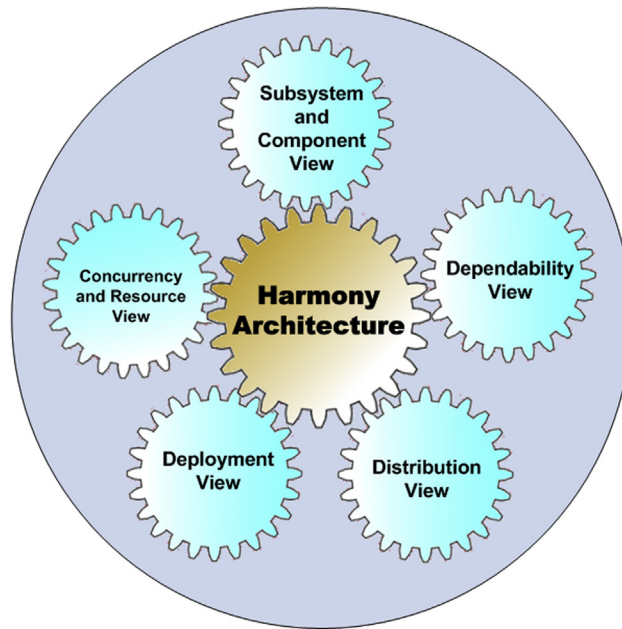


Figure 2.14
Harmony architectural views.

«active» objects), the policies for scheduling those concurrent units, and how they synchronize and share resources. The *Distribution view* identifies how objects are distributed across multiple address spaces, how they find each other, and the means and protocols they use to interact and collaborate. The *Deployment view* shows how the products from the different engineering disciplines work together, including the cross-discipline interfaces.¹² Lastly, the *Dependability view* focuses on how the system ensures safety, reliability, and security during runtime.

Figure 2.15 shows the Architectural Design workflow. You can see in the figure how the design criteria are identified and ranked and then used to select appropriate architectural design patterns. The parallel nature of the applications of the patterns in the different aspects of architecture emphasizes that (1) the order in which they are introduced is a personal preference and (2) not all the aspects of architecture must be realized in the same prototype. It is common to put off some aspects of the architecture until later prototypes when they are low risk and concentrate on the high-risk aspects in earlier prototypes.

¹² While UML provides deployment diagrams, SysML prefers to use class diagrams to represent the deployment architecture because they are richer and more expressive than deployment diagrams.

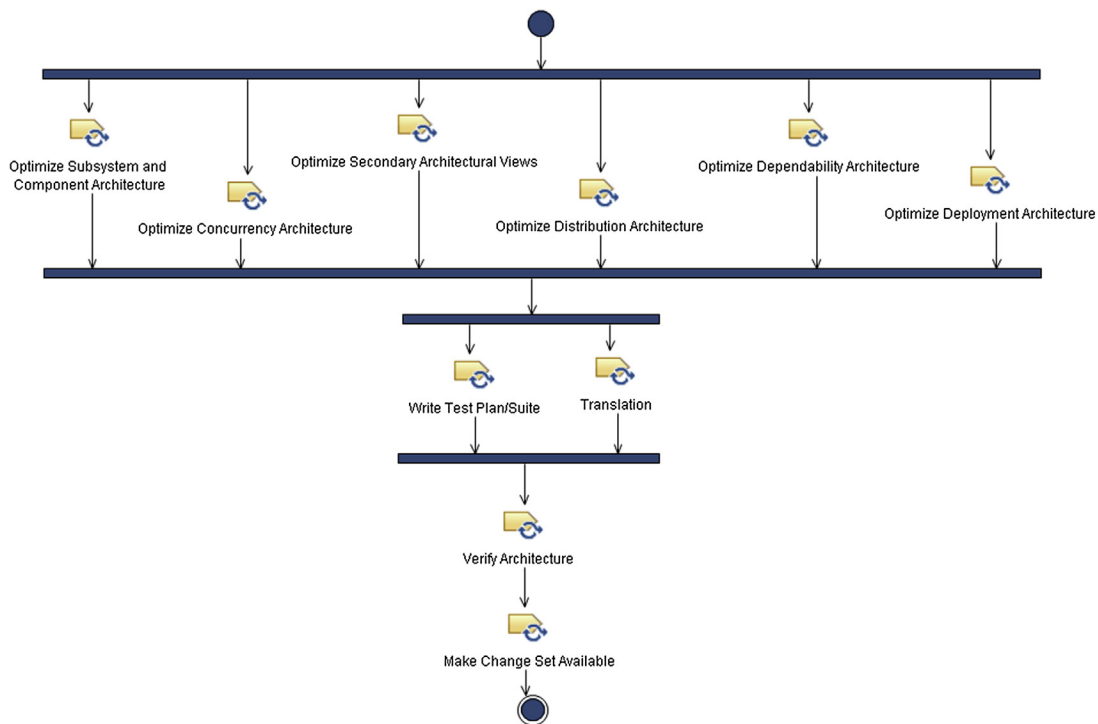


Figure 2.15
Architectural Design workflow.

Mechanistic Design Phase

The Mechanistic Design phase (also known as “Collaboration Design”) is concerned with the optimization of individual collaborations, each realizing a single use case. The scope of mechanistic design decisions is generally an order of magnitude smaller than those found in architectural design, since a system typically consists of one to several dozen use cases. Similar to architectural design, mechanistic design largely proceeds via the application of design patterns, although the scope of the patterns is much smaller than those found in architectural design. This is where the classic “Gang of Four” patterns¹³ and other more fine-grained patterns are applied.

The Mechanistic Design view is an elaboration of the High-Fidelity Modeling view and uses the same graphical representation – class and sequence diagrams for collaboration structure and sequence, activity, and state diagrams for behavior.

¹³ Gamma, et al. Design Patterns: Elements of Reusable Object Oriented Architecture (Addison-Wesley, 1995).

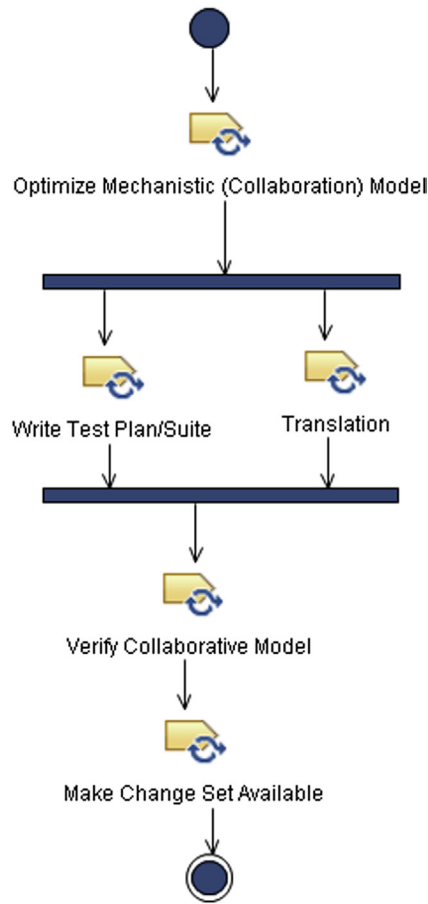


Figure 2.16
Mechanistic (Collaboration) Design workflow.

The Mechanistic Design workflow is shown in [Figure 2.16](#). The flow is similar to architectural design in that the first step is to identify what you want to optimize (the design criteria) and how important each of the criteria is, and then find design patterns that optimize the more important of these at the expense of the least. The resulting design collaboration is tested, not only to make sure that you haven't broken the functionality in the original analysis collaboration, but also to ensure that you have achieved the desired optimization.

Detailed Design Phase

The Detailed Design phase elaborates the internals of objects, classes, functions, and data structures, and has a highly limited scope – the individual software element. Most of the optimization in detailed design focuses on the issues of

- Data structuring (space or time optimization)
- Algorithmic decomposition
- Optimization of an object's state machine
- Object implementation strategies
- Association implementation
- Visibility and encapsulation concerns
- Ensuring compliance at runtime with preconditional and postconditional invariants (such as ranges on method parameters)

There are many rules of thumb, guidelines, and practices for detailed design, although these commonly fall under the title of “idioms” rather than “patterns.” For most objects, detailed design is little more than a trivial detail, but there is usually a small (5% is typical), but important, set of objects that require special attention during detailed design.

Figure 2.17 shows the Detailed Design workflow.

2.5.3 Verification and Validation (V&V)

The V&V test phase constructs the prototype from the architectural elements and ensures that the system produced during the iteration meets its mission statement (validation). The High-Fidelity Modeling and Design workflows have been performing incremental integration (a practice known as “continuous integration”) to ensure that incremental changes made to the source code base have resulted in a system that is internally consistent. The next step is to demonstrate that the produced system meets all of the requirements verified from previous iterations (regression test) and that all of the new requirements for this iteration have been properly and correctly implemented.

It should be noted that the development of the work products to be used in V&V testing – test procedures, test cases, and test harnesses – have been developed in parallel with the system development work. This is done in the *Prepare for V&V Testing* activity shown in Figure 2.5.

It should be noted that the Functional and QoS test vectors are primarily derived from the use case sequence diagrams specified in the Analysis phase (Harmony/ESW) or the Systems Engineering phase (Full Harmony). The Regression test vectors are a subset of the test vectors from previous development iterations. Additional tests, such as stress, volume, coverage, and fault-seeding tests, are added manually. The V&V workflow is shown in Figure 2.18.

Following a successful verification, the prototype may be demoed to the customer or released to them for use. This can provide valuable feedback. The verification of the prototype ensures that it meets the requirements. However, requirements are often incomplete, inconsistent, inaccurate, or just plain wrong. Allowing the customer to provide feedback greatly improves the chance that the system will actually meet the needs of the customer.

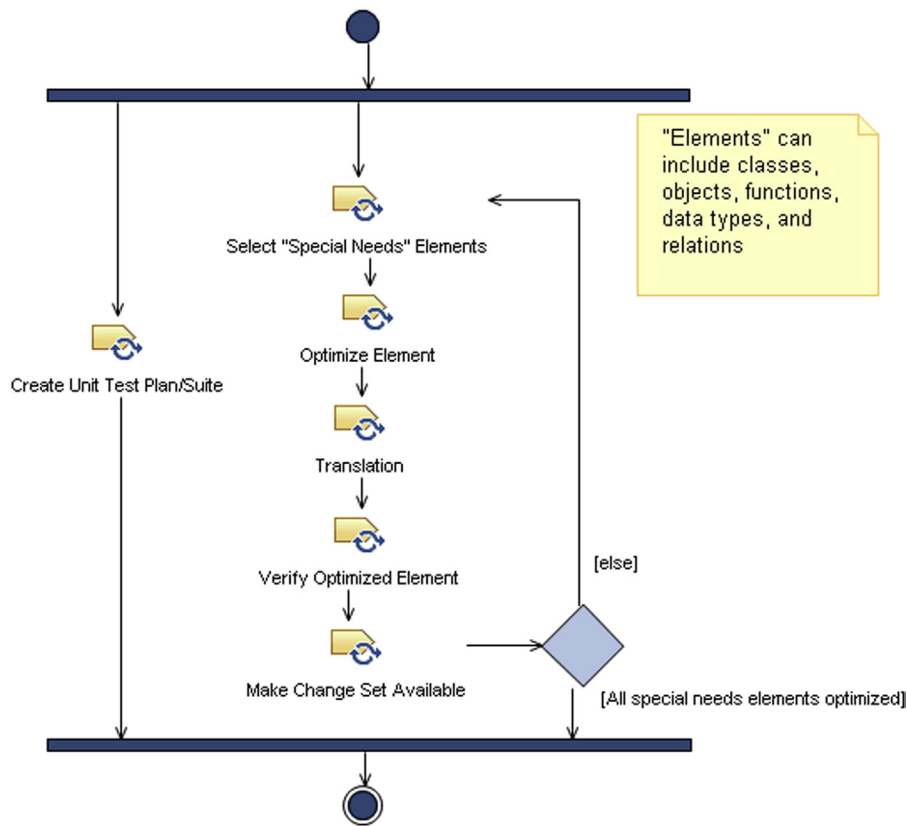


Figure 2.17
Detailed Design workflow.

2.5.4 Increment Review (Party!) Workflow

The development iteration ends with the increment review activity (also known as the Party phase).¹⁴ This activity performs an iteration retrospective to see how well the project is proceeding.

This task assesses both the project against the project and iteration plans and updates the plans as necessary. The primary work products assessed during the Party phase are

- Overall project plan, including the overall schedule
- Iteration project plan, including the iteration schedule
- Architecture stability, robustness, and optimality

¹⁴ The Party phase corresponds to both the Initial Concept and Post-Mortem Assessment phases in some other development process models. The use of the term “Party” is to reinforce the notion of “celebration of ongoing success” rather than a post-mortem “figure out what went wrong” analysis.

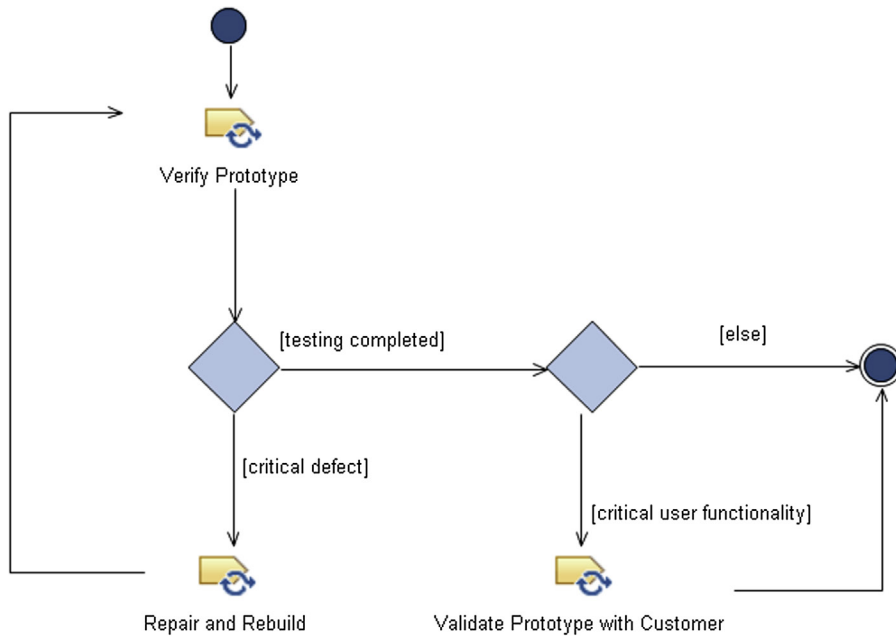


Figure 2.18
Verification and Validation workflow.

- Process efficiency and effectiveness
- Risk mitigation activity outcomes, their impact on the project, and looking ahead for upcoming risks
- Addressing other project concerns raised by staff or management

One of the more serious project management mistakes made is inadequate assessment and adjustment of projects during their execution. As DeMarco and Lister note “You cannot control what you do not measure.”¹⁵ It is equally important that you apply the measured information to make adjustments. In terms of schedule, such adjustments will be things like reassignment of resources, reordering activities, deletion of activities, reductions (or enhancements) of scope and/or quality, rescheduling subsequent activities, and so on.

Because the selection and implementation of a good architecture is crucial to the long-term success of a project and product, the Party phase evaluates architecture on two primary criteria. First, is the architecture adequately meeting the needs of the qualities of services that are driving architecture selection? Second, is that architecture scaling well as the system evolves and grows? The process of reorganizing the architecture is called *refactoring* the system. If the project team finds that the architecture must be significantly refactored on each

¹⁵ DeMarco, T. and T. Lister. *Peopleware: Productive Projects and Teams* New York, New York, Dorset House Publishing Company, 1987.

prototype, then this is an indication that the architecture is not scaling well, and some additional effort should be given to the definition of a more scalable architecture.

Early on in the project, selections are made about how to manage the project – what tools will be used, where they and their data are located and how they are accessed, security procedures, work product review and quality assessment procedures, work and work product guidelines, etc. The Party phase seeks to improve the efficiency of the process during the project by actively looking for and correcting problems and issues. It is important to remember that a *plan* is a theory and theories need to be tested and updated so that they reflect reality.

In my experience, the biggest single reason for project failure is ignoring risks. To manage risks, we recommend each project maintain a risk management plan. In this plan, each identified risk is identified and ranked and, where appropriate, a risk mitigation strategy is described. Most of these will be activities to be done in the spirals to explore, reduce, or handle the risk. In the Party phase, the risk management plan is reviewed and newly identified risks are added.

2.6 Summary

This chapter briefly described the Harmony process, in both the Full form (which contains a systems engineering phase) and software-only form. The Harmony process consists of a set of integrated workflows for each phase in the process. In the general form, the development spiral is preceded by a systems engineering phase that fully elaborates all the requirements of the system and constructs a well-formed and executable use case model. Then an internal subsystem architecture is created and the requirements, represented primarily as “operational contracts,” are allocated to these subsystems (usually grouped together into subsystem-level use cases). The subsystems themselves are decomposed into high-level engineering discipline-specific components for electronic, mechanical, chemical, and software aspects and the logical interfaces among these components are defined. At this point, the models are handed off to the subsystem teams and their engineering disciplines for further development.

In the Harmony/ESW variant, some process optimizations are enabled by the lack of significant hardware-software co-development. Rather than fully specify all system requirements up front and then allocate them to subsystems and then to software, in Harmony/ESW, use cases are identified but not detailed until the iteration in which they are to be developed. With the development spiral, a small set of use cases are selected and, *at that time*, are detailed with sequence diagrams, state machines, activity diagrams, etc. Then a high-fidelity model (and attendant source code) is created that correctly captures the functional requirements of the use case, and the resulting model and code are verified via model execution through a continuous, incremental unit test. As different engineers work on different parts of the system, their code is integrated together continuously to ensure a coherent resulting product.

This analysis model requires optimization and the introduction of specific realization technologies, which is done in design. Design is driven by the optimization criteria, such as the quality of service requirements. Design occurs at three levels of abstraction: architectural design optimizes the system at an overall level, mechanistic design optimizes collaborations of objects, and detailed design optimizes individual objects. The optimizations made to the model result in updates to the source code baseline, and these work products are likewise continuously verified with unit and integration testing.

Verification and validation demonstrates the correctness of the prototype against its mission (requirements, risks to be reduced, defects to be repaired, etc.). Defects identified during the V&V are either fixed in the testing phase of the current spiral (for critical defects) or allocated to a future iteration. If appropriate, the verified prototype is shown to or released to the customer to solicit feedback as to its adequacy to meet their needs.

The next development iteration continues where this one left off, adding and refining the next few use cases and risk mitigation items. The incremental prototype becomes increasingly more capable and complete as time goes on, until finally all the requirements have been realized and validated. At this point, the prototype is released to manufacturing or the customer.

In the next several chapters, we will introduce problems and exercises for you to gain experience in applying these workflows to real-life applications. Two applications are presented. Appendix A provides a problem statement for a traffic light control system. This is a small-scale application and demonstrates how the process workflows apply to systems of moderate size. Appendix B presents a much larger-scale system – an unmanned aerial vehicle (UAV) – complete with ground and airborne aspects. This larger system allows the exploration of problems of scale and architecture that don't arise in small systems.

In both cases, the purpose of the problems is to give you a context in which to gain experience by *doing*. It is *not* to construct a fully complete and validated model of those systems. This is especially true of the UAV system. To create and explain a full UAV system would make this book thousands of pages, many times larger than it currently is. However, the scope of the problems provides rich fodder for using model-based approaches to development.

Meeting Industry Standards

What You Will Learn

- The importance of industry standards
- Some important standards
 - DoDAF and UML Profile for DoDAF and MODAF (UPDM)
 - DO-178B/C (Avionics)
 - IEC 61508 and ISO 26262 (Safety)
 - IEC 62304 (Medical)
 - CMMI-DEV (Systems and Software Development)

3.1 Overview

Many industries that heavily employ embedded systems have standards for various aspects, such as interoperability, safety, and the development process. These provide a framework in which the engineers must operate and thus provide real value – both to the engineering organization (since they can sell their devices or get them certified) and for the user (who has some notion as to the applicability of the device to their industry-specific needs). While these standards add value, they also add complexity. They all levy additional constraints on the work that is done, how the work should be performed, or properties the produced systems must possess. This brief chapter will talk about some of these standards and how they specifically influence how models are produced, as well as the properties of models that will aid in their certification.

3.2 On the Importance of Being Standard

Software engineering is, relatively speaking, an immature discipline. In fact, many have suggested that the term *engineering* is inappropriate when applied to the development of software and that the term *craft* is more appropriate. The term “software engineering” was first used in 1968 at a NATO conference. There still rather few software engineering courses of study at universities – computer science is still far more common. This means that students training to be software developers take basic classes on topics such as database design, algorithms, and building web pages, but have little engineering training. They come out of school (usually) with lots of enthusiasm but little in the way of knowledge of configuration

management, architectural design, software program management, specification of requirements, or other topics one needs to reasonably “engineer” software. These topics instead are learned on the job in an almost apprentice relationship (although in most cases, it’s a matter of throwing them into deep water and seeing who can stay upright at the surface). There is virtually no professional certification in general, although there are technology-specific certifications that may be obtained.

The outcome of this is that the skill level among software developers in most organization is very uneven. Standards provide value in that they specify minimum levels for process (steps taken to achieve a product result), work product content, or measures of completeness and correctness. The need for standards grows with the complexity and criticality of the system involved.

Of course, different standards specify different things. Standards may be internal (specific to the company or project), such as modeling or coding standards. Or they may be specific to an industry, such as how to develop safety-critical software for avionics; specific to a topic, such as a standard for developing safety-critical software in general; or general, such as providing process objectives for systems development organizations. Standards may specify technology, processes, or work product structure and organization.

In this chapter, we’ll look at just a few common standards. The Department of Defense Architectural Framework (DoDAF) is a standard that specifies the viewpoints, views, and semantic content for enterprise architecture. International Electrotechnical Commission (IEC) 61508 is a standard specifying how to develop safety systems, including safety assessment, process objectives, and technical measures. DO-178B and DO-178C are used for avionics software development and verification and define process objectives and required evidence to show compliance. IEC 62304 is a somewhat similar standard used for the development of medical systems. Capability Maturity Model Integration for Development (CMMI-DEV) is a general standard for software development that identifies different levels of development maturity and the objectives that must be met to achieve them.

3.3 Architectural Framework Standards (*I’m looking at you UPDM*)

There are a number of AFs (architectural frameworks) out there – DoDAF, MODAF (Ministry of Defence Architectural Framework), TOGAF (The Open Group Architectural Framework), NAF (NATO Architectural Framework), and so on. The intent of these standards is *interoperability* of systems at the architectural level. Because of the similarity in intent and purpose, if not specific content, I’ll confine my discussion going forward to DoDAF, the standard with which I’ve had the most experience.¹

¹ In fact, I wrote the first commercial UML profile for DoDAF and implemented it for the Rhapsody tool, well before UPDM became an OMG standard.

The US Department of Defense (DoD) was faced with the problem of increasing difficulty in integrating complex warfare systems. The problem was architecture. Everybody did it. Everybody meant something different by it. And everybody did it differently. Deciding whether or not systems were potentially compatible was a huge problem and stovepiped systems plagued the DoD.

To address this, the DoD came up with a standard with the awkward name C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance), which identified a set of documents – known as products – that represented different aspects of an architecture in different ways. Of course, there still was a problem in that while each such product had an expected semantic content, there was no standard notion for them. This paved the path for the DoDAF standard in 2003. Version 1.5 added SOA (service-oriented architecture) views and version 2.0 is a complete rewrite of the semantic underpinnings of the standard. Meanwhile the British developed their own MODAF standard, which was similar but had some additional viewpoints and more products. This led to work in the Object Management Group (OMG) to create the UML Profile for DoDAF and MODAF (UPDM), which is the standard everyone supports today. UPDM revisions trail behind those of DoDAF and MODAF, but this is the format that most DoD primes and subcontractors use today.

The current version of DoDAF is 2.0. It has a number of primary viewpoints, each of which contains a set of products. It is based on an ontology known as IDEAS (International Defence Enterprise Architecture Specification).² IDEAS is not specified in UML, but UML is a general purpose modeling language and so can be used to represent, among other things, other modeling languages. UPDM was an industry effort to provide a means by which the rich and (relatively) vast set of UML tools could be leveraged to support DoDAF and MODAF modeling. UPDM provides a standardized means of capturing architectural aspects of enterprise and systems of systems and mapping these to standard UML and SysML elements.

The current revision of UPDM (1.1) defines a set of viewpoints:

- All viewpoint
- Data and Information viewpoint
- Standards viewpoint
- Capability viewpoint
- Operational viewpoint
- Service viewpoint
- System viewpoint
- Project viewpoint

² <http://www.ideasgroup.org/>

Within each viewpoint, there are a set of views (products) that are produced. The All viewpoint is special in that it provides the integrated dictionary for all the other viewpoints.

The Capability viewpoint acts as the initial set of stakeholder requirements and the drivers for the Operational viewpoint. It consists of a set of seven views:

- CV-1 Vision
- CV-2 Capability Taxonomy
- CV-3 Capability Phasing
- CV-4 Capability Dependencies
- CV-5 Capability to Organizational Development Mapping
- CV-6 Capability to Operational Activities Mapping
- CV-7 Capability to Services Mapping

The Capability viewpoint is generally of less interest to the readers of this book (systems and embedded software developers), since it drives the defense acquisition process, rather than engineering processes.

The Operational viewpoint describes the elements and flows necessary to conduct military operations. This view is used to derive to-be architectures, stakeholder requirements, and future concepts; to understand how existent architectures are used operationally; and to perform operational planning. It is my view that the views within the Operational viewpoint should be developed by the government customer, but in my experience it often happens that they pawn this off on the primes and subcontractors.

There are six views within the Operational viewpoint:

- OV-1 High-Level Operational Concept Graphic
- OV-2 Operational Resource Flow Description
- OV-3 Operational Resource Flow Matrix
- OV-4 Organizational Relationships Chart
- OV-5a Operational Activity Decomposition Tree
- OV-5b Operational Activity Model
- OV-6a Operational Rules Model
- OV-6b State Transition Description
- OV-6c Event Trace Matrix

The astute reader will note that they came close to running out of fingers to count these “6” views. OV-1 is “semantic-free” and is often done with a loosely defined drawing. OV-2 presents information or materiel flow among operational nodes. OV-3 represents the summary matrix of resource flows from a set of OV-2 diagrams. OV-5 represents the operational activities that produce, consume, or manipulate resources. The OV-6 views show how such behavior unfolds.

A key concept in the Operational viewpoint is that of the operational node. This may be a person, organization, or system that plays a role within an operation. Some of these will be realized by systems (that we develop) but others will not. The Operational viewpoint focuses on how operations are conducted without worrying about how the operational elements are realized.

The System viewpoint is most relevant to the readers of this book. For those operational elements that are realized by systems, these provide a high-level, pre-system engineering perspective on how the systems must interact and the kinds of information and materiel that they produce or consume.

There are 10 views in the System viewpoint:

- SV-1 Systems Interface Description
- SV-2 Systems Resource Flow Description
- SV-3 Systems-Systems Matrix
- SV-4 Systems Functionality Description
- SV-5a Operational Activity to Systems Functionality Traceability Matrix
- SV-5b Operational Activity to Systems Traceability Matrix
- SV-6 Systems Resource Flow Matrix
- SV-7 Systems Measures Matrix
- SV-8 Systems Evolution Description
- SV-10a Systems Rules Model
- SV-10b Systems State Transition Description
- SV-10c Systems Event-Trace Description

It is important to note that any given architecture will not have all viewpoints and all views represented. It is important as a part of project planning to identify which viewpoints and views are appropriate for the system under development and focus there. By the way, the numbering of the views within the viewpoints is essentially random and should not be taken to imply the order in which views should be created.

A systems or software engineer can imagine how SV-1 and SV-2 might be represented with class diagrams, SV-3 would be a summary matrix, SV-7 represents performance metrics and constraints, SV-10b uses state machines to describe system behavior, and SV-10c uses sequence diagrams to represents the interaction of systems. The SV-5 views are key in relating the operational architecture to the systems domain.

The Service views describe services and their interconnections. In this context, a service is a mechanism to allow access to one or more capabilities via a defined interface. The set of views for the Service viewpoint is the same as for the System viewpoint (except substitute “Service” for “System”). DoDAF 2.0 services implemented in UML are defined using SoaML (Service oriented architecture Modeling Language), if you want to proceed down that path.

Project views capture the relation between the project, portfolios, organizations, and programs. The Standards viewpoint identifies which standards are relevant to the architecture. Neither the Project nor the Standards viewpoints are a primary concern of this book's intended audience.

The Data and Information viewpoint is important and in DoDAF 2.0 include views that were previously in the Operational and System viewpoints. There are three Data and Information views:

- DIV-1 Conceptual Data Model
- DIV-2 Logical Data Model
- DIV-3 Physical Data Model

In this context, the logical data model portrays the important aspects of information (such as range for a radar system) but not how it is implemented, whereas the physical data model includes the presentational means (bit map) of the data.

The All viewpoint includes

- AV-1 Overview and Summary Information
- AV-2 Integrated Dictionary

Of these, AV-2 is typically stored and managed by the UML modeling tool being used.

3.3.1 UPDM

The representation of the various views isn't defined within the DoDAF specifications. Many people use non-UML representations for some or all of these views. However, UML is a powerful and well-supported language, so it is natural that a standard mapping of these views and elements from the DoDAF and MODAF standards to UML would be created. For the most part, elements such as operational nodes map to classes and most of the views map to specialized forms of class diagrams. Event trace descriptions are easily shown via sequence diagrams and state transition descriptions are, unsurprisingly, best shown as UML state diagrams.

The good thing about the UPDM standard is that it provides a standard way to specify the various views needed to support a DoDAF- or MODAF-compliant architecture. While the DoDAF itself is not very restrictive on how the views can be represented, UPDM makes good (if somewhat obvious) choices and is easily supported by tools. DoDAF (and UPDM) also doesn't state in which order these views should be created – that is the job of *process* – and the Harmony process described in the previous chapter has been adapted to the creation of UPDM views.

3.4 IEC 61508

The IEC 61508 standard exists specifically to classify safety risk in order to identify, avoid, detect, and control systematic and random faults to avoid safety-related accidents. Originally published in 1997, it was updated in 2010. IEC 61508, titled *Functional Safety of Electrical/*

Table 3.1 Reliability for Different SILs.

Safety Integrity Level	High Demand Rate (dangerous failures/hr)	Low Demand Rate (probability of failure on demand)
4	$\geq 10^{-9}$ to $< 10^{-8}$	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-8}$ to $< 10^{-7}$	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-7}$ to $< 10^{-6}$	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-6}$ to $< 10^{-5}$	$\geq 10^{-2}$ to $< 10^{-1}$

Electronic/Programmable Electronic (E/E/PES) Safety-related Systems is a seven-part generic safety standard for electronic systems, including those that contain software. These parts are

1. General requirements
2. Requirements for electrical/electronic/programmable electronic safety-related systems
3. Software requirements
4. Definitions and abbreviations
5. Examples of methods for the determination of safety integrity levels
6. Guidelines for the application of IEC 61508-2 and IEC 61508-3
7. Overview of techniques and measures

We'll discuss these sections in a bit. First, though, I want to mention Safety Integrity Levels (SILs). SILs range from level 1 (low requirement) to level 4 (high requirement). Reliability of the system is important with respect to SILs. In this context, *reliability* refers specifically to the frequency of a safety-related failure (a failure that leads to the manifestation of a hazardous condition (see [Table 3.1](#)).

High demand systems are those in use continuously or at a high rate, such as an automotive braking system. Low demand systems (invoked or used less than one use per year) are systems that are invoked only rarely, such as an automotive air bag.

If the intrinsic (unmitigated) reliability of the system doesn't meet these reliability standards for the system SIL, then safety management methods (often called "safety methods") must raise the reliability to at least those levels.

Safety must be actively managed. Safety management includes specific techniques to reduce the risk³ throughout the entire product lifecycle, from initial concept through analysis, specification, design, installation, maintenance, and disposal. Specific techniques for safety are listed and briefly discussed in Part 7 of the standard. Part 2 Annex A relates specific techniques and measures for hardware and system safety management, while Part 3 Annex A lists software-specific measures and techniques. Part 3 Annex B is particularly useful because it provides detailed recommendations of measures and techniques as a function of the SIL.

³ The value of *risk* is the product of the severity of a safety-related accident and its likelihood.

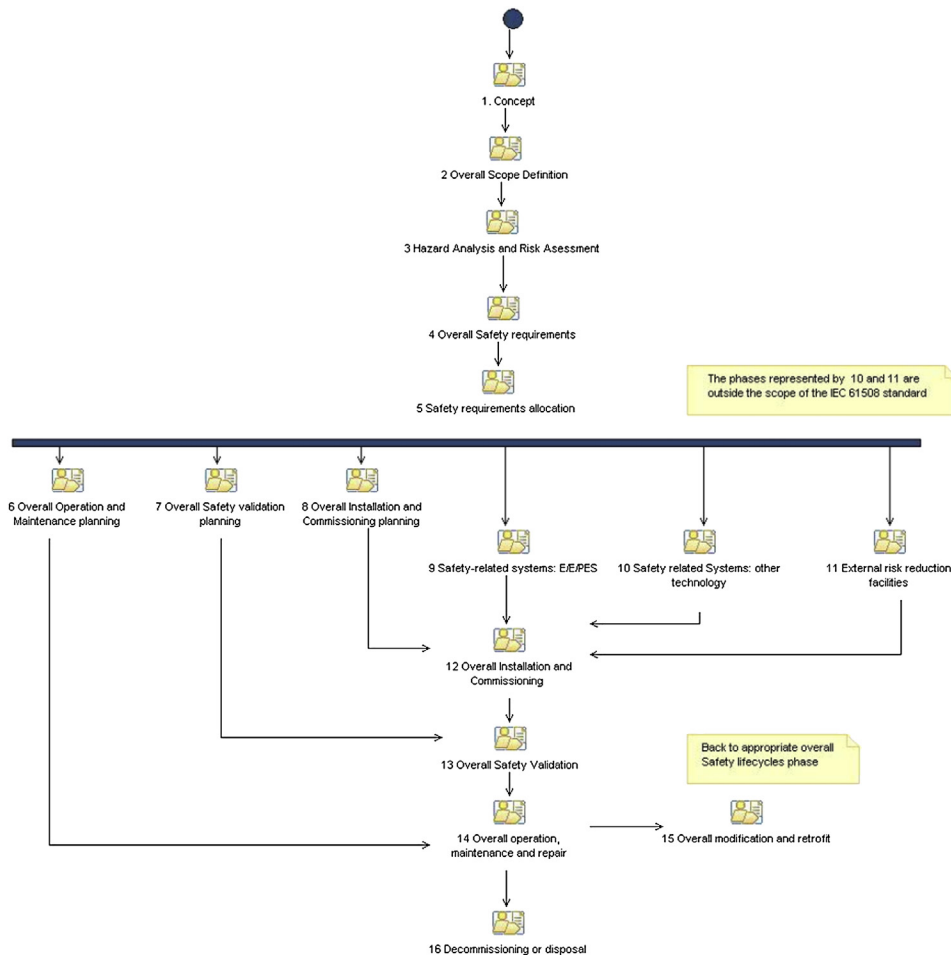


Figure 3.1
IEC 61508 safety lifecycle.

3.4.1 Process Requirements

The overall process for the IEC 61508 lifecycle is shown in [Figure 3.1](#). You can see that the lifecycle is much more inclusive than simply hardware and software development. It certainly includes those activities, but also initial safety and hazard analysis, integration and verification, safety assessment (prior to release), production, operation, service, and decommissioning.

The systems engineering lifecycle is perhaps a bit oversimplified (perhaps due to the sole focus on safety rather than product requirements). The software lifecycle is shown as a standard V-cycle development ([Figure 3.2](#)).

Although the standard discusses software process in terms of a V-cycle, it doesn't explicitly require a V-style lifecycle. It does state that the lifecycle processes should be defined and

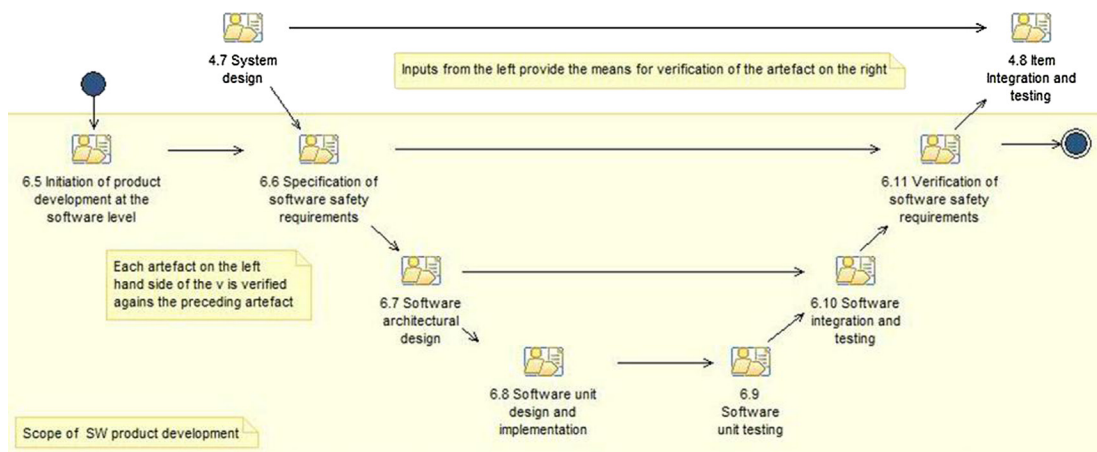


Figure 3.2

IEC 61508 notional software development lifecycle.

process assurance measures should verify that they are being followed. This means that any repeatable, verifiable approach that adheres to the objectives of the standard is acceptable. The standard *does*, however, give objectives and requirements for each of these activities that must be met by an acceptable process.

Interestingly the standard highly recommends the adoption of “structured processes” although it declines to say what those might be.⁴ Annex B identifies some specific modeling techniques that are either R (recommended) or HR (highly recommended). Table 3.2 shows the recommended semiformal methods from IEC 61508-3 table B.7.

As an aside, you will note the repeated use of UML, and especially class,⁵ sequence, and state diagrams in this book.

There are a number of industry-specific standards derived from the IEC 61508 standard. The 2011 ISO 26262 standard is adapted specifically for the development and production of passenger automobiles. It comes as 10 parts (nine normative parts and one guideline). This derived standard references ASILs (Automotive Safety Integrity Levels) ranging from A to D, equivalent to the IEC 61508 SIL levels 1 to 4. Rail safety is specified in EN 50128, IEC 62061 adapts the standard for various kinds of machinery, and IEC 61513 specializes the 61508 standard for nuclear power plant control and instrumentation systems.

⁴ In the 1997 edition of the standard, Part 3 Annex A gave some examples of “highly recommended” structured methods including JSD, MASCOT, SADT and Yourdon. The 2010 version removed the examples, presumably to be more open to equivalently rigorous or even more rigorous methods.

⁵ Entity-relationship-attribute diagrams are the basis on which class diagrams were derived. Now they are used not only for data models but for classes, which bind data elements to the functions that operate upon them.

Table 3.2 Semiformal Methods.

Technique/Measure		SIL 1	SIL 2	SIL 3	SIL 4
1	Logic/function block diagrams	R	R	HR	HR
2	Sequence diagrams	R	R	HR	HR
3	Data flow diagrams	R	R	R	R
4a	Finite state machines/state transition diagrams	R	R	HR	HR
4b	Time Petri Nets	R	R	HR	HR
5	Entity-relationship-attribute data models	R	R	R	R
6	Message sequence charts	R	R	R	R
7	Decision/truth tables	R	R	HR	HR
8	UML	R	R	R	R

(Adapted from IEC 61508 Part 3 Annex B).

3.5 DO-178B/C

DO-178B is a 1992 RTCA standard used to govern the certification of avionics software (including complete aircraft). The standard is solely focused on software development and is therefore similar in scope to IEC 61508 Part 3. It defines five levels of safety concern from E (no concern) to A (high concern). Table 3.3 elucidates the differences among the different safety levels defined in the DO-178B standard.⁶

The DO-178B standard has 66 objectives that must be met for Level A, with decreasing numbers of objectives for levels B on down. Some of these objectives – such as independence of verification and design and level of testing coverage – account for much of the cost and effort of compliance to these standards. Table 3.4 shows some of the important objective goals and how they differ by safety certification level.

3.5.1 Verification

The DO-178B standard is not the least concerned with validation (meeting customer needs) because that is not a deterministic goal. Instead, verification (compliance to requirements) under DO-178 is emphasized. The major points of concern for verification under the DO-178B standard include

- Verification of source code
- Verification of low-level requirements (design)
- Verification of software architecture

⁶ DO-178B does not have any guidance on how to do safety assessment nor the assignment of safety level. This is actually in a different standard, SAE ARP-4761 “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.”

Table 3.3 DO-178B Safety Levels.

Level	Consequences of Failure	Description	Example Systems	Max Probability per Flight Hour
A	Catastrophic: Major fatalities, including both passengers and crew	Software whose anomalous behavior would cause or contribute to a failure of system functions and therefore would prevent continued safe flight and landing of the aircraft	Flight controls, engine controls, primary displays	10^{-9}
B	Hazardous: Serious injuries or potential fatalities to a small number of occupants	Software whose failure condition would reduce the capability of the aircraft or ability of the crew to cope with adverse operational conditions, resulting in a large reduction of safety margin	Flight management system, navigation system, radios	10^{-7}
C	Major: Increased crew workload with possible discomfort and injury to occupants	Software whose failure conditions would reduce the ability of the crew to cope with adverse operations to a significant extent	Backup displays, backup radios	10^{-5}
D	Minor: Possible discomfort to crew or passengers	Software whose failure would not significantly reduce aircraft safety and would involve crew actions well within their capabilities	Engine vibration monitors, maintenance systems	Not applicable
E	No effect	Software whose failure has no impact on aircraft safety or increase of crew workload	Entertainment systems, galley services	Not applicable

Table 3.4 Some DO-178 Objectives.

DO-178 Objective Goal	Level A	Level B	Level C	Level D
Independence level	High	Medium	Low	Very low
Needs low-level requirements	Yes	Yes	Yes	No
Statement coverage (SC)	Yes	Yes	Yes	No
Decision coverage (DC)	Yes	Yes	No	No
Modified condition/decision coverage (MC/DC)	Yes	No	No	No
Configuration management	Strong	Strong	Medium	Low
Source to binary correlation*	Yes	No	No	No
Requirements correlate to target processor	Yes	Yes	No	No
Architecture and algorithmic verification	Yes	Yes	Yes	No
Code reviews	Yes	Yes	Yes	No
QA transition criteria	Yes	Yes	Yes	No

*"Source to binary conversion" means that for every source-level language statement, there is a mapping to binary operation codes running on the target processor.

- Degree of test coverage
- Control of the verification procedure
- Robustness testing
- Verification activities with indirect effect on error prevention or detection, such as conformance to the software development plan (SDP) and design and coding standards

Testing coverage is an interesting topic for DO-178B. On one hand, development organizations get *no certification credit* for performing unit testing, never mind that it's a great idea and teams should do it anyway. They only get certification credit for requirements-based testing. That is, all testing for which you get certification credit must verify some high-level software requirement. At the same time, the coverage requirements for testing are fairly severe. This is because you are not allowed to have dead code (code that cannot be reached) or code that isn't there to meet a requirement. That means that you must demonstrate that each line of code traces back to at least one high-level requirement as well as that each high-level requirement is represented in the low-level requirements (design) and the code. This often means that as (necessary) code is developed, new requirements must be added so that they have a "reason to exist."

Three levels of coverage are specified in the DO-178B standard. *Statement coverage* (SC) means that with high-level requirements-based testing, each source code statement is executed at least once. The next level up, *decision coverage* (DC), means that each decision point must be executed in both TRUE and FALSE conditions. The most onerous, *modified condition/decision coverage* (MC/DC), means that each Boolean subexpression in a decision must be independently varied in the test cases.

Consider the code sample below:

```
if (x < 0 && y > 10 && z == 15)
    foo(x,y,z);
```

Structural coverage requires a single test (the TRUE condition, such as when $x == -10$, $y == 17$ and $z == 15$). That will execute all the lines of code. Decision coverage requires one TRUE case and one FALSE case (such as if $x == 0$). MC/DC coverage, though, requires at least four tests to ensure all three Boolean clauses are independently varied. For example,

Test Case	x	y	z	Outcome
1	-10	17	15	TRUE
2	0	17	15	FALSE
3	-10	8	15	FALSE
4	-10	17	14	FALSE

In general, n Boolean clauses will require $n + 1$ test cases for MC/DC coverage.

This standard (properly) considers safety assessment outside the scope of software development but within the scope of system engineering. Software development is driven by requirements allocated from systems to software, and those requirements should take into account the safety measures to be implemented in software.

The standard focuses on the software lifecycle, both its definition (planning) and providing evidence that the plans were followed (enactment). The standard requires five key plans and a host of supporting lifecycle data ([Table 3.5](#)).

The DO-178 standards cast the discussion of process in terms of a traditional waterfall process, but do not specify the lifecycle. Instead, the standards specify the set of objectives that acceptable processes must follow and the evidence required to demonstrate that compliance. Annex A of DO-178B summarizes the objectives to be met in a set of tables with reference back to document sections that provide more detail.

3.5.2 DO-178C

In 2012, the standard was updated to DO-178C. The standard remained almost the same, although a small number of objectives (five) were added, bringing the total number of objectives in the standard to 71. The primary change was the addition of four different supplements providing guidance on different technology topics. The DO-178C supplements will be described later in this section.

These supplements include

- DO-330 Software Tool Qualification Considerations
- DO-331 Model-Based Development and Verification: Supplement to DO-178C and DO-278A
- DO-332 Object-Oriented Technology and Related Techniques: Supplement to DO-178C and DO-278A
- DO-333 Formal Methods: Supplement to DO-178C and DO-278A

Each of these supplements provides guidance for the application of the technology and identifies additional objectives to be met. Of particular interest to developers is DO-331 and DO-332, while DO-330 is of special interest to tool vendors. These technologies were being used to develop avionics systems back with the DO-178B standard, but because there was no guidance, getting certification depended on the personal viewpoint of the Designated Engineering Representative (DER) or representative of the certification agency. Now with more concrete guidance the ways to use these technologies are clearer.

I should note that the contents of DO-331 differ little from the guidance in this book. DO-331 does identify two different kinds of models. Specification models represent models of high-level requirements and may not be used for the purpose of code generation. Design models

Table 3.5 DO-178B/C Lifecycle Data.

Abbreviation	Data Item	Description
PSAC	Plan for Software Aspects of Certification	The PSAC serves as the primary means of communication between the development organization and the certification agency.
SDP	Software Development Plan	The SDP defines the software lifecycle and how the software process will meet the development objectives.
SVP	Software Verification Plan	The SVP defines how the software verification objectives will be met.
SCMP	Software Configuration Management Plan	The SCMP defines how configuration and change management will be performed and how CM objectives will be met.
SQAP	Software Quality Assurance Plan	The SQAP identifies how compliance to quality standards and guidelines and other QA objectives will be met.
SRS	Software Requirements Standards	The SRS defines methods, rules, and tools used to develop high-level requirements, including notations, formatting, and constraints on the use of tools.
SDS	Software Design Standards	The SDS defines methods, rules, and tools used to develop architecture and design (which DO-178B refers to as “low-level requirements”), including notations, formatting, constraints on the use of tools, and constraints on the use of design patterns and idioms (e.g., use of dynamic memory).
SCS	Software Coding Standards	The SCS defines the programming languages, permitted subsets, methods, rules (e.g., naming conventions and formatting rules), and tools.
SRD	Software Requirements Data	This is the set of software “high-level requirements” including functional, safety, performance, and quality of service requirements; constraints; hardware-software interface definitions; fault detection and safety monitoring; and partitioning requirements.
SDD	Software Design Description	The SDD provides a detailed description of how the software structure (both architecture and more detailed design) satisfies the high-level requirements, including data and control flow, data structures, and allocation to hardware processors and concurrency units.
SC	Source Code	The SC is composed of detailed machine instructions written in a programming language.
EOC	Executable Object Code	The EOC is the set of machine instructions directly useable by the target computer.
SVCP	Software Verification Cases and Procedures	The SVCP defines the requirements-based test cases and how they will be applied to the EOC to verify their correctness.
SVR	Software Verification Results	The SVR are produced by executing the verification test cases against the EOC.
SECI	Software Lifecycle Environment Configuration Index	The SECI identifies the set of elements in the software engineering environment, including tools, processes, and plans, and their configuration revision IDs.

Table 3.5 DO-178B/C Lifecycle Data.—Cont'd

Abbreviation	Data Item	Description
SCI	Software Configuration Index	The SCI identifies the revision numbers for all of the relevant work products and lifecycle data.
PR	Problem Reports	The PRs are the means by which anomalous software behavior is reported and corrections are managed.
SCMR	Software Configuration Management Records	The SCMR are the records of the activities of the CM process, including configuration identification lists, baselines, archive records, and release records.
SQAR	Software Quality Assurance Records	The SQAR record the outputs of the QA activities, such as audit reports, meeting minutes, and review outcomes.
SAS	Software Accomplishment Summary	The SAS is the key document for certification in that it shows compliance of the evidence to the PSAC.

represent low-level requirements (design) and architecture and may be used as the basis from which code is generated. This means that high-level requirements may be represented in models or as text or as both with traceability links between them. This book recommends the last representation as the most viable approach. Model coverage analysis for design models demonstrates that design elements exist and trace back to high-level requirements and that each requirement in the specification model is represented by one or more model elements in the design model. In addition, non-normative model elements (model elements not there to represent or meet a requirement and for which there are no conformance requirements) must be clearly identified. The standard gives no recommendation as to how to accomplish this, but it could be done by metaclass (“All comments are non-normative”), stereotype (“All non-normative elements are stereotyped «non_normative»”), by adding a tagged value to the elements, or by putting them into a package that contains all non-normative elements.

Object-orientation brings a number of design and conceptual benefits to designers and programmers but does so at the cost of a loss of control over detailed implementation, especially around the use of dynamic memory, type consistency, and verification needs. DO-332 mentions a number of specific concerns that must be addressed to use object-oriented (OO) programming, which are summarized in DO-332 Annex D – Vulnerability Analysis.

For example, because inheritance is all about type substitutability, you must ensure that any subtype can be correctly substituted for the base type at runtime, or if it isn't, you must ensure that there is no opportunity for an inappropriate substitution to occur. Dynamic polymorphism isn't so much of a concern of development as it is of testing; because of the potential set of functions that could be executed in the case of an inherited subtype, the coverage requirements of testing become more difficult to ensure, resulting in a much larger number of test cases.

An additional concern of OO methods is their heavy dependence on dynamic memory. Dynamic memory allocation has problems with fragmentation, memory loss and recovery,

unpredictability of timing, and so on. Most OO designs use dynamic memory more heavily than structured designs, so ensuring freedom from memory concerns is a Big Deal with respect to DO-178B/C certification. For this reason, the most common memory management strategy employed for Level A certified software is “none” – the programs allocate memory at start up and never deallocate it. This addresses the most common memory-related problems:

- Fragmentation starvation: Failure of an allocation request because although there is enough total memory available to satisfy a request, there is no single block large enough.
- Deallocation starvation: Failure to satisfy an allocation request because unreferenced memory has not been reclaimed.
- Heap exhaustion: A dynamic memory request can fail because not enough memory is available
- Premature deallocation: Memory is deallocated even though there is an active reference to it.
- Stale reference: If a garbage collector is used, memory can be moved without updating an active reference.
- Time-bound exceeded: A program can fail because an allocation or deallocation has an unexpected delay due to dynamic memory management.

My book *Real-Time Design Patterns*⁷ provides a number of design patterns that deal with memory allocation and management.

3.5.3 Tool Qualification

Applications are *certified* for use, but tools used in their development or verification may be *qualified*. The limited discussion of tool qualification provided in DO-178B was elaborated in the DO-178C supplement, DO-330. This standard identifies three criteria to determine whether a tool must be qualified:

1. Tool whose output is a part of the resulting software and could thus insert an error
2. Tool that automates verification processes and could therefore fail to detect an error and whose output is used to justify the elimination or reduction of other verification or development processes
3. A tool that, within the scope of its intended use, could fail to detect an error

The standard identifies different levels of qualification depending on the safety level of the systems on which it is used, from Tool Qualification Level 1 (TQL-1) to Level 5 (TWL-5). The most rigorous of these levels is on a par with the objectives for the certification of airborne software at level A with a total of 76 objectives that must be met.

⁷ Douglass, Bruce Powel. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2003).

However, a critical point is that you do not need to qualify your tools *if you verify their output*. This means that modeling tools, compilers, and so on, which could, potentially, insert defects in the avionic product, must either be qualified (we trust their output so we don't need to verify it) or we must independently verify their output. This is really no different than Joe (a hypothetical programmer on the avionic software project). We don't trust Joe to make zero mistakes, so we review and test the code he produces. Similarly, we typically don't trust the modeling tools and compilers to insert zero defects in the generated design and code, so we test their output. Verification (testing) tools are usually not verified independently, so while development tools (criteria 1) are generally not qualified, verification tools (criteria 2 and 3) generally are.

3.5.4 Complying with DO-178B/C Using the Harmony Process

The Harmony process has detailed mappings to the DO-178B, DO-178C, DO-331, and DO-332 standards. For each objective in these standards, there is a list (with links) to the process elements – tasks, roles, work products, standards, checklists, templates, and so on – that realize it. Put another way, the Harmony process consists of a set of best practices that achieve most of the goals of this (and other standards) and the process content contains explicit mappings from the standard to the process elements to show how compliance can be achieved.

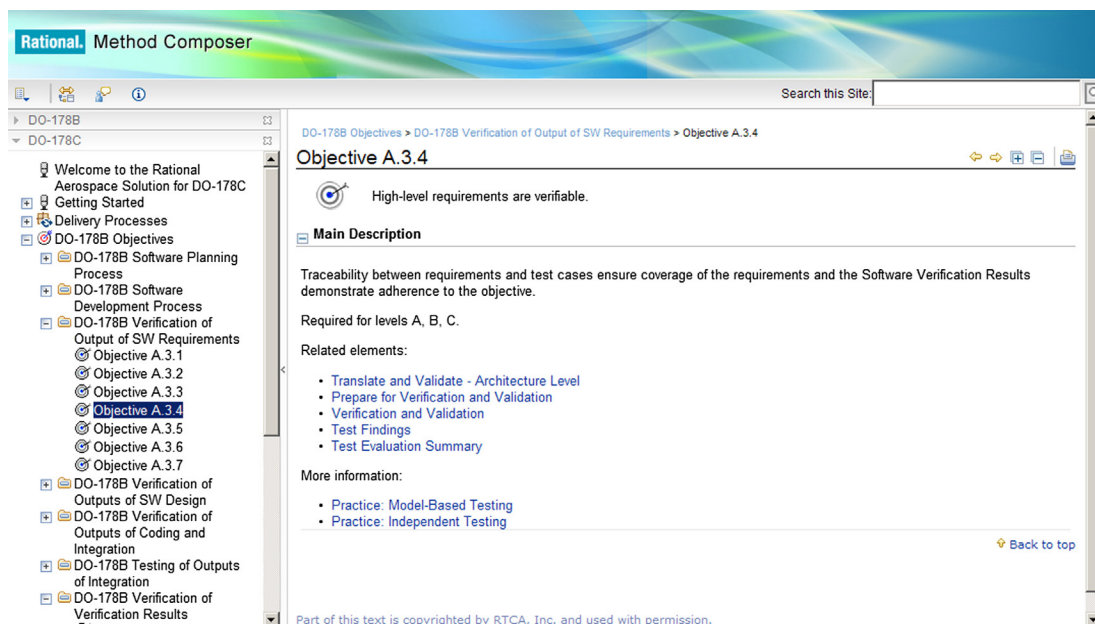


Figure 3.3

Mapping between the Harmony process and DO-178B/C.

Figure 3.3 shows an example of the mapping between DO-178B/C and the Harmony process. The referenced model elements are hyperlinks to the relevant pages in the process content.

3.6 IEC 62304

The IEC 62304 standard, entitled “Medical Device Software – Life Cycle Processes,” is used for medical devices. It is similar in scope to DO-178B or IEC 61508 Part 3 in that it focuses on the software lifecycle. This standard identifies three levels of certification (called *safety classes*), from level A (no safety risk) to level B (nonserious injury is possible) to level C (death or serious injury is possible).

The major areas of concern for the IEC 62304 standard are

- General requirements – general requirements for a high-quality software development and management environment
- Software development process – requirements for planning the activities and work products in software development
- Software maintenance process – how software is maintained, updated, and upgraded during its lifecycle
- Software risk management process – the process by which safety risks are identified, characterized, and mitigated in the development activity
- Software configuration management process – the process by which developed software (including SOUP – Software of Unknown Pedigree) is stored, versioned, and controlled.
- Software problem resolution process – the process by which defects are identified and resolved.

You can see that a greater emphasis is placed on maintenance of the software than in the DO-178B standard, but less is placed on quality assurance and traceability. Similar to the other standards discussed here, IEC 62304 does not dictate a particular lifecycle or process but provides objectives that a process must meet. This standard is less demanding than IEC 61508 or DO-178B/C; it suggests nothing that a good process shouldn’t already do with the exception of having a risk management process.

The Harmony process has a detailed mapping to the IEC 62304 standard. For each objective in this standard, there is a list (with links) to the process elements – tasks, roles, work products, standards, checklists, templates, and so on – that realize it.

3.7 CMMI-DEV

The Capability Maturity Model Integration for Development (CMMI-DEV) standard provides a framework for examining the maturity of an organizational process. It identifies 22 process areas that are present in varying degrees in five levels of maturity, each building upon the process areas of the levels in [Table 3.6](#).

Table 3.6 CMMI Maturity Levels.

Maturity Level	Description	Process Area
1 – “Initial”	Processes are usually ad hoc and chaotic. Unstable environment. Depends on competence and heroics. Tendency to overcommit, abandonment of processes and inability to repeat successes.	None (ad hoc)
2 – “Managed”	Requirements are managed and processes are planned, performed, measured, and controlled. Seven process areas.	Requirements Management Project Monitoring and Control Project Planning Supplier Agreement Management Configuration Management Measurement and Analysis Process and Product Quality Assurance
3 – “Defined”	Processes are well characterized and understood and are described in standards, procedures, tools, and methods. Processes are managed more proactively.	Product Integration Requirements Development Technical Solution Validation Verification Organizational Process Definition Organizational Process Focus Organizational Training Integrated Project Management (Project) Risk Management Decision Analysis and Resolution
4 – “Quantitatively Managed”	Predicable process performance. Processes are controlled and measured.	Organization Process Performance Quantitative Project Management
5 – “Optimizing”	Continual process improvement. Addresses common causes of process variation.	Organization Innovation and Deployment Causal Analysis and Resolution

There are two ways to consider compliance with CMMI. The one exemplified in [Table 3.6](#) is the “staged representation.” It assumes a particular set of practices must be in place for the level of maturity; miss one of those areas and you are automatically downgraded to the next lower level. The “continuous representation” skips the categories but focuses just on the process areas.

CMMI is very much a “say what you’re going to do, then do what you say” standard. Little consideration is given to ensuring that what you say makes sense (as long as the process area specific goals are met).

Each process area has a set of specific goals, and each goal contains a set of specific practices. For example, consider one of the fundamental (Level 2) process areas – Requirements Management. It has one specific goal (SG) and five specific practices (SP):

- SG-1: Manage Requirements

- SP 1.1: Obtain an Understanding of Requirements
- SP 1.2: Obtain Commitment to Requirements
- SP 1.3: Manage Requirements Changes
- SP 1.4: Maintain Bidirectional Traceability of Requirements
- SP 1.5: Identify Inconsistencies between Project Work and Requirements

The Requirements Management process area is basic, in the sense that it identifies fundamental objectives that a Level 2 organization must have. It is not basic in that relatively few organizations actually achieve all of these specific practices. Note that there are more requirements practices that could be specified here, such as eliciting customer needs, determining requirements feasibility, analyzing requirements, and allocation of requirements to components. These are not part of this process area but show up in more mature ones, specifically Requirements Development and Technical Solution, both added in Maturity Level 3.

The Harmony process has a detailed mapping to the CMMI-DEV standard (Figure 3.4). For each objective in these standards, there is a list (with links) to the process elements – tasks, roles, work products, standards, checklists, templates, and so on – that realize it. Similar to the mapping to the DO-178 standards shown in Figure 3.3, the references are live hyperlinks that navigate to the referenced process element.

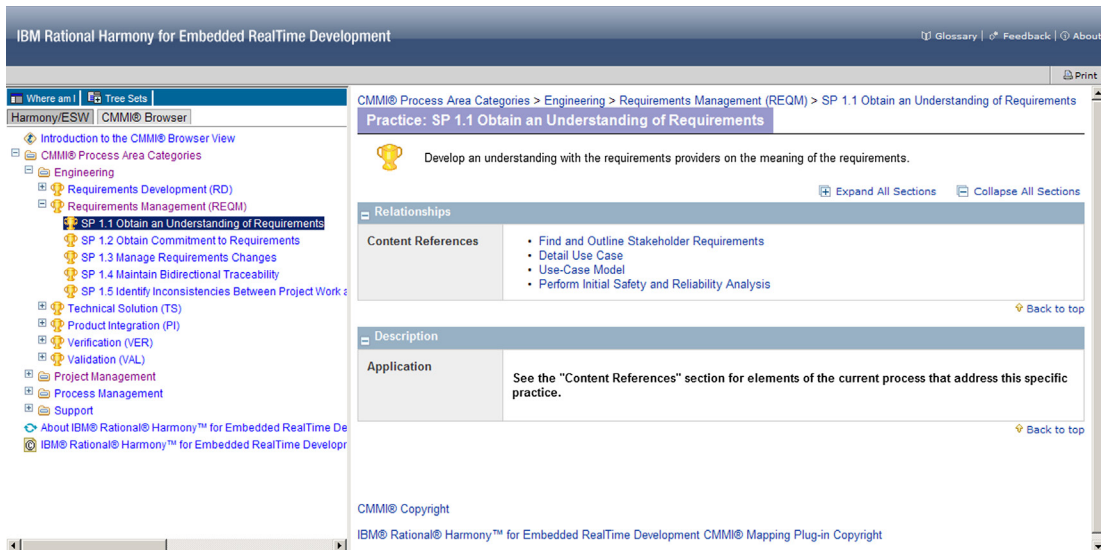


Figure 3.4
Harmony process mapping to CMMI-DEV.

3.8 Summary

I've often said that one of the nice things about standards is that there are so many from which to choose. In this chapter, we've discussed just a few of them – from DoDAF and UPDM to DO-178 to CMMI-DEV. Standards are useful in that they provide guidance to developers and auditors as to what should be represented, how it should be shown, the tasks that should be performed, and industry-specific objectives that should be met. The standards are necessarily vague. Good standards should specify invariants only, that is, things that must always be present, and allow the users to decide the best way to meet them. This provides the flexibility for innovation and improvement while still ensuring that fundamental goals are met. The downside of such flexibility is that users need to figure out how to realize these standards.

DoDAF and its ilk are framework standards and identify a set of work products that together constitute an enterprise architecture – in this case, for defense-related systems. DoDAF contains a number of viewpoints and each view contains a number of views (diagrammatic, tabular, or textual) bound together with an integrated dictionary (one of the views). DoDAF itself does not define a representational schema for the views, so users are left to their own devices. The UPDM profile provides a standard way to do that with UML/SysML, enabling users to deploy the wide array of UML and SysML tools to the specification of DoDAF- and MODAF-compliant architectures. DoDAF contains no process guidance, nor does it identify which of the views are necessary (certainly not all of them in all cases). The Harmony process provides some guidance for this; Mitre Activity-Based Modeling (ABM) provides somewhat different guidance.⁸

IEC 61508 is different in that it is clearly a safety standard. Its focus is to provide criteria for determining safety level (specified as a Safety Integrity Level, or SIL), a set of objectives that must be met for each level of safety, and a list of possible technical safety measures. It is a fairly comprehensive standard, consisting of seven parts covering systems, electronic, and software concerns. Unlike DoDAF, IEC 61508 does provide an implicit acceptance of the V-cycle and provides process objectives that must be met. However, it doesn't define a full lifecycle, since it limits its focus to safety-related concerns for the most part.

DO-178B is a 1992 standard used for the development and certification of avionics software and qualification of tools used in that industry. The DO-178B standard defines five levels of safety concern, from E (no concern) all the way up to A (catastrophic concern) with up to 66 objectives to be met. While the discussion of process is cast in terms of a traditional waterfall, the standard doesn't assume or require any particular development lifecycle. DO-178C and four supplements were released in early 2012. DO-178C itself is a very minor update to

⁸ See http://www.mitre.org/work/tech_papers/tech_papers_04/ring_dodarch/ring_dodarch.pdf.

DO-178B, while the supplements provide guidance for the use of some modern software engineering technologies, notably model-based development and verification (DO-331), object-oriented methods (DO-332), and formal methods (DO-333). The requirements for tool qualification were elaborated and expanded in DO-330. In general terms, development tools need not be qualified as long as their output is verified, but verification tools generally do require qualification for their use.

IEC 62304 is in a similar vein to DO-178B in that it identifies objectives that a software lifecycle must meet but doesn't go so far as to dictate that lifecycle. This standard focuses on the development of software for medical devices and identifies three levels of concern from A (no concern) up to C (potential fatality).

CMMI-DEV is a general systems and software development lifecycle standard. It doesn't define a particular lifecycle, but does identify five maturity levels that meet, to various degrees, 22 process areas. Each process area has one or more specific goals and each specific goal has one or more specific practices. Each level of maturity includes the process areas met by the levels below it.

A key point I want to emphasize is that *none* of the process standards (IEC 61508, DO-178B and DO178C, IEC 62304, or CMMI-DEV) requires a particular lifecycle, even though they might seem to give an implicit recommendation. These standards identify objectives that must be met by the process used. This means that agile methods *can and have* been used to develop avionic and medical systems and have been applied by organizations assessed at high levels of CMMI maturity. The point to remember is that the process you use must meet all of the relevant objectives of those standards.

References

CMMI-DEV. <http://cmmiinstitute.com/cmmi-solutions/cmmi-for-development/>.

DO-178. <http://www.rtca.org/default.asp>.

DoDAF. <http://dodcio.defense.gov/dodaf20.aspx>.

Hilderman, V. and T. Baghi. "Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)" (Avionics Communications, 2007).

IEC 61508. <http://www.iec.ch/functionalsafety/explained/>

IEC 62304. http://webstore.iec.ch/preview/info_iec62304%7Bed1.0%7Den_d.pdf

ISO 26262. <http://www.iso.org/iso/home/search.htm?qt=26262&sort=rel&type=simple&published=on>.

UPDM. <http://www.omg.org> document: formal/2012-01-04.

Specifying Requirements

What You Will Learn

- How to identify types of requirements
- How to group requirements together
- How to identify use cases
- How to capture and represent quality of service requirements
- How to identify operational scenarios
- How to use state machines for representing requirements
- How to capture complex requirements
- How to capture algorithmic requirements with activity diagrams
- How to validate requirements models

4.1 Overview

One of the most important things that engineers must do is specify requirements on systems. This is a crucial step in any high-reliability process but, unfortunately, often is poorly executed. Few engineers have been trained in the capture and management of requirements, and this lack of training results in huge “Victorian novel” style requirements documents being constructed that are incomplete, inconsistent, and ambiguous. Because such documents are captured in text, they inherently lack the precision necessary for their validation prior to initiating design. This means that the quality of the requirements cannot be assessed in any reliable way. Requirements defects are one of the most important kinds of strategic flaws in systems and are certainly the most expensive to repair because they are introduced early, (typically) identified late, and have far-reaching implications and scope. UML and the Harmony process offer a better way.

UML only implicitly represents requirements, but does explicitly represent coherent requirements sets via the concept of use cases. A use case can be thought of an operation view of the system clustered around some generic kind of system usage. As such, it implicitly “contains” a coherent set of requirements clustered around that system usage. More specific uses of the system, scenarios, are highly specific “paths” through the use case. From the DoDAF and MODAF point of view, use cases and scenarios can be part of the “operational architecture” of a set of interacting operational nodes. From the operational perspective, the use cases and

scenario views are vital to understanding how the operational nodes (that will be met by actual systems)¹ collaborate with other elements in the context of the system environment.

We can also model the requirements from the specification, rather than the operational, perspective. These are statements of static and dynamic properties of the system – the “system architecture” in DoDAF/MODAF terms. When we look at requirements from this perspective, precise, unambiguous statements of system properties and aspects must be made. Formal languages within UML, such as statecharts and activity diagrams, are tailor-made for such precise specifications.

The question arises as to how best represent requirements from the operational and specification perspectives. If you didn’t know how to do this in a language as commonplace as English, it must be much more daunting to do it with a less common language, such the UML!

This chapter provides a set of exercises that will give you practice and experience representing requirements, from both the operational and specification points of view. The answers to the exercises are given in Chapter 11. Be aware that there is no one “right” answer – there are many good ways to model requirements so that they are clear, complete, consistent, and unambiguous. It turns out, though, that there are many more *bad* ways to model requirements than good ones, but nevertheless, there are still many good ways. So, if your answers differ from those in the answers section of the book, it doesn’t mean that your answers are necessarily wrong. In my experience, however, use cases are the most misused element in UML. I believe this is not because use cases are inherently difficult to understand but because few engineers are trained in capturing requirements. Most engineers end up trying to do what they *are* trained in – that is, design – with use cases, rather than capturing requirements in a design-free way.

Note: Remember – all the problems in this book are based on two example problem statements, which are given in appendices of this book. Appendix A details the requirements for the Roadrunner Traffic Light Control System while Appendix B details the requirements for the Coyote Unmanned Air Vehicle (CUAV).

4.2 Representing Requirements in UML and SysML

By far, the most common way to represent requirements is with text. Even though fraught with problems, the ease with which requirements can be churned out and the inherent understandability makes text an obvious choice. Word processing and spreadsheet tools are

¹ An operational node can be thought of as a role in the operational architecture, such as a “Recon Platform” which, when the mission is actually run, is realized by a physical system such as a “Satellite” or “AWACS.”

common, low-cost ways to create and organize requirements, but they lack the ability to manage traceability links in an automated way and to verify that the requirements are consistent, complete, accurate, and correct. Requirements management tools, such as DOORS, provide strong traceability and management facilities for requirements. When coupled with UML's ability to create executable requirements models, a complete requirements solution is readily available.

It is common to create textual requirements in DOORS (or other requirements management tool) and import them into Rhapsody (or another modeling tool). Once in UML, we can create verifiable requirements models. This means we can construct models that have far higher quality than otherwise. This is a common practice in development organizations where the cost of requirements failure is high, such as military, aerospace, transportation, and medical. This chapter will give you practice in building such requirements models.

First, though, let's review the basics of putting requirements into a UML model. Figure 4.1 shows a use case diagram with dependency links to the its requirements.

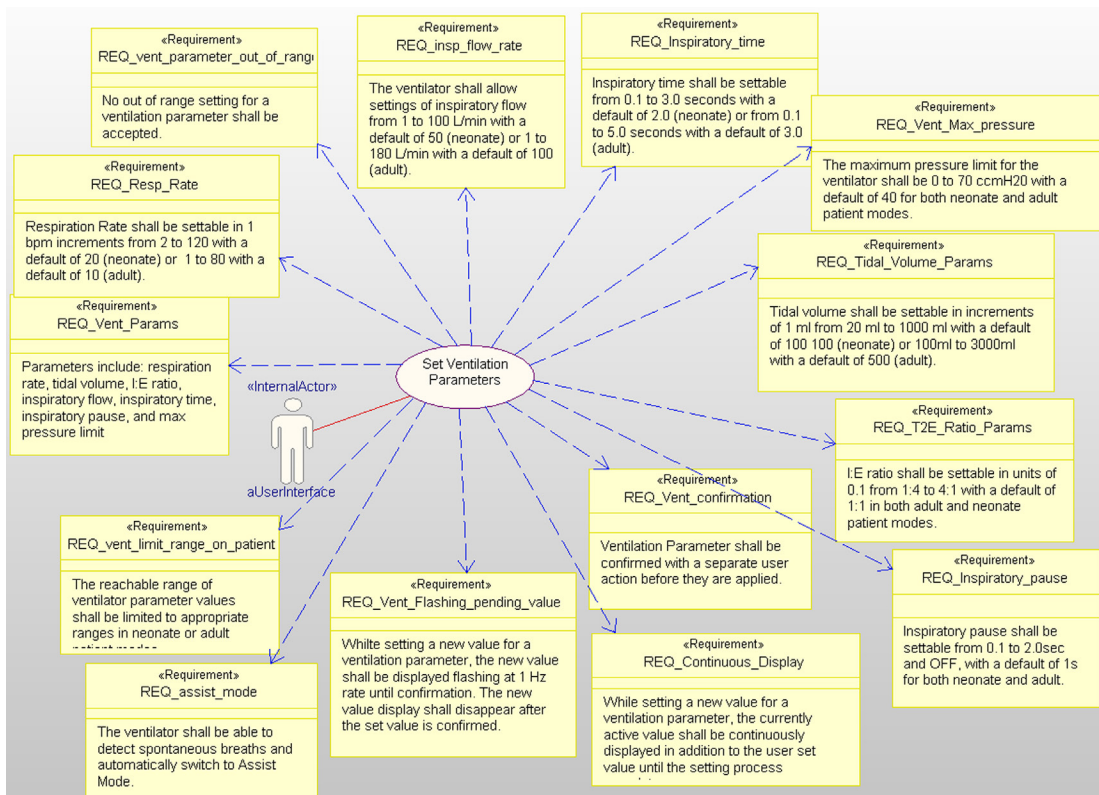


Figure 4.1
Requirements linked to use case.

From: UseCase	Scope: RequirementsAnalysis	Monitor Bre...	Monitor O2...	Set Ventilation Parameters	Display Machine Status
REQ_Vent_Params			REQ_Vent_Params		
REQ_Tidal_Volume_Params			REQ_Vent_Params_0		
REQ_T2E_Ratio_Params			REQ_Resp_Rate		
REQ_Vent_Display_Always			REQ_Tidal_Volume_Params		
REQ_Vent_confirmation			REQ_T2E_Ratio_Params		REQ_Vent_Display_Always
REQ_vent_parameter_out_of_range_setting			REQ_Vent_confirmation		REQ_Vent_confirmation
REQ_Continuous_Display			REQ_vent_parameter_out_of_ra...		
REQ_Vent_Flashing_pending_value			REQ_Continuous_Display		REQ_Continuous_Display
REQ_vent_limit_range_on_patient_mode			REQ_Vent_Flashing_pending_v...		REQ_Vent_Flashing_pending_valu
REQ_insp_flow_rate			REQ_vent_limit_range_on_patient...		REQ_vent_limit_range_on_patient...
REQ_Inspiratory_time			REQ_insp_flow_rate		
REQ_Vent_Max_pressure			REQ_Inspiratory_time		
REQ_Display_Inspiratory_O2			REQ_Vent_Max_pressure		
REQ_Display_CO2					REQ_Display_Inspiratory_O2
REQ_Display_Measured_Drugs					REQ_Display_CO2
REQ_Display_Set_Drugs					REQ_Display_Measured_Drugs
REQ_Display_Drug_Uptake					REQ_Display_Set_Drugs
REQ_Display_Gas_Flow					REQ_Display_Drug_Uptake
REQ_Display_Pressures					REQ_Display_Gas_Flow
REQ_Display_Status_constantly					REQ_Display_Pressures
REQ_informational_alarms					REQ_Display_Status_constantly
REQ_warning_alarms					
REQ_Critical_alarms					
REQ_Alarm_categories					
REQ_Informational_alarms					
REQ_Warning_sounds					
REQ_Critical_reannouncement					
REQ_Critical_alarm_sounds					
REQ_Dismiss_alarms					
REQ_Alarm_retention					
REQ_Chart_recorder_alarms					
REQ_Alarm_condition					
REQ_patient_alarms					
REQ_assist_mode				REQ_assist_mode	
REQ_Inspiratory_pause				REQ_Inspiratory_pause	
REQ_BCM_01		REQ_BCM_...			
REQ_BCM_02		REQ_BCM_...			
REQ_BCM_03		REQ_BCM_...			

Figure 4.2
Requirements to use case trace links.

When this is done for a number of use cases, it can be summarized in tables, as shown in [Figure 4.2](#). This figure shows the requirements as rows and the related use cases as columns. The view shown in this figure is Rhapsody; not all tools can generate this kind of tabular summary of related model elements.

These tables allow for different kinds of checks. For example, are all requirements represented in the use case model or design? You can also do *impact analysis* – if a requirement changes, what model, code, or verification elements require modification as a consequence?

Problem 4.1 Identifying Requirement Types for the Roadrunner Traffic Light Control System

UML does not have an explicit concept called a “requirement.” Even though use cases, sequence diagrams, state machines, activity diagrams, and constraints of various kinds are used to model the properties of requirements, the requirement concept itself is lacking. For this reason, the Systems Modeling Language has added one, and Rhapsody lets us add requirements into the models.

A useful way to think about requirements in UML is that they are a model element that specifies some aspect of a system from an external, rather than internal, view of that system. Note that although a requirement is technically a stereotype of a class in the SysML metamodel, it doesn't instantiate in the same sense as classes. A requirement is a kind of "correctness" constraint that applies to a system. The SysML specifications state that

A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition a system must achieve.

SysML also goes on to define a requirements diagram that can only display requirements, test cases, and other such elements. However - more usefully - requirements may be added to any diagram and attached to model elements via various kinds of dependencies.

SysML defines a number of different kinds of relations with requirements, all stereotypes of dependency, such as

- «trace» – used in UML as the basis for traceability among elements
- «copy» – used to indicate a requirement copied from one context to another
- «deriveReq» – used to indicate that one requirement is derived from another
- «verify» – used to trace between a test case and a requirement
- «satisfy» – used to trace between an analysis or design element and a requirement

I believe that there are many different kinds of requirements, as shown in [Figure 4.3](#). An *operational requirement* is one that specifies how the system is to collaborate with other elements (actors) in its environment. For example, if an unmanned aerial vehicle (UAV) must interact with a set of GPS satellites to navigate its terrain, how that interaction occurs is an operational aspect of that system. *Functional requirements* have to do with the behavior of the system. If a UAV must be able to point a gimbaled optical sensor at a point on the ground, then it is a function the UAV must be able to do. If functional requirement is a verb, then a *quality of service requirement* is an adverb since it specifies "how much." A quality of service requirement might be that our gimbaled optical sensor will be able to adjust in 0.1-degree increments with an accuracy of 0.01 degrees. A *parametric requirement* is both nonoperational and nonfunctional. Suppose that the UAV must be constructed so that unfueled it weighs less than 2100 pounds. The UAV's weight is a property of the system, which might be considered either static or dynamic, as the situation warrants. It is clearly neither a behavior nor an operational aspect, yet it is important to represent. Finally, a *design requirement* has to do with the characteristic of the design *per se* but not of the system as it exists in the field. An example design requirement might be that the UAV is designed so that the next UAV can reuse at least 70% of the model, or that the recurring cost of the system is less than \$1.2M (not unreasonable at all for our Coyote UAV).

We model these requirements in somewhat different ways. For example, modeling reliable communications is very straightforward with a statechart, but how would you use a statechart to represent the fact that the nose of the UAV should be dark grey in color?

For this exercise, create a new model with high-level packages, *Requirements* and *Functional Analysis*. Use the former package to hold the textual requirements elements and the latter to hold the use cases and their analysis.

In the Functional Analysis package, add two diagrams (either a use case diagram or class diagram will do). Label one "Overview Requirements" and the other "Emergency and Priority Mode Requirements." Take the text in the Roadrunner Traffic Light Controller problem statement in Appendix A

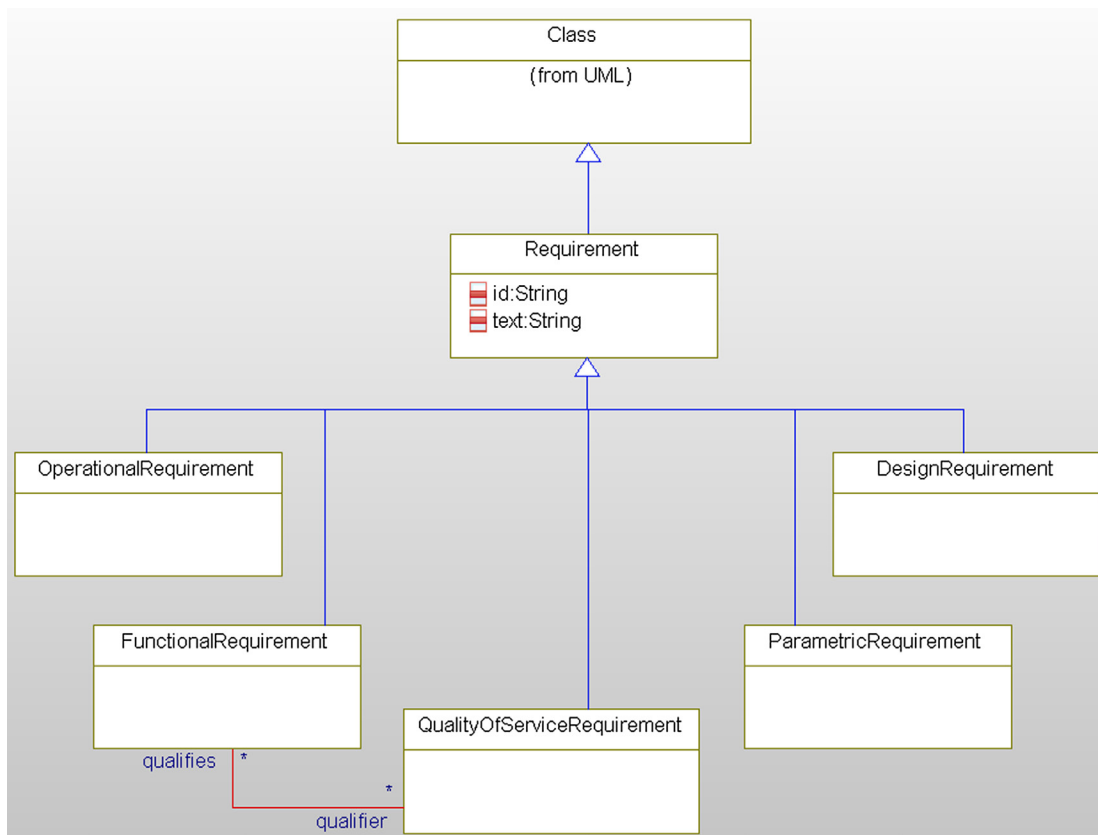


Figure 4.3
Types of requirements.

up to the “Configuration Parameters” section and put each requirements statement in a separate requirements element. Link the requirements elements together with dependencies as appropriate, using the following stereotype dependencies: refine or refineReq (one requirement is a refined version of another), derive or deriveReq (one requirement is derived from another), satisfy (a model element fulfills a requirement), or trace (general traceable relation among model elements).²

² Don’t get caught up in worrying about the exact subtype of requirement or dependency. Ask yourself, “Does making sure this relation is exactly the right subtype add clarity to my requirements model?” Usually, the answer is “not really.” On the other hand, you can add your own, such as «include», «extend», «refine», and so on, as long as they improve the understandability of your model.

Problem 4.2 Identifying Use Cases for the Roadrunner Traffic Light Control System

Generally speaking, there is little value in just pasting some requirements on a diagram. A typical system will have hundreds to many thousands of requirements. It is crucial to organize the requirements into chunks to permit functional analysis and other forms of reasoning about requirements. Use cases are the key organizational unit for requirements reasoning.

Use cases are functional uses of a system from an external operational standpoint. Remember that for a good use case, all the following conditions must be true:

- It returns a value to at least one actor.
- It contains at least three scenarios, each scenario consisting of multiple actor-system messages.
- It logically contains at least a few, but up to very many operational requirements.
- It does not reveal or imply anything about the internal structure of the system implementation.
- It is independent of other use cases and may (or may not) be concurrent with them.
- There should be no fewer than three and no more than three dozen use cases at the “high level.”
 - “Large” use cases may be decomposed with «includes» and «extends» relations into smaller “part” use cases.
 - Use cases may be specialized with the generalization (“is-a-specialized-kind-of”) relation.

Use cases may be identified in many different ways. Some analysts prefer to identify operational scenarios and then group them together into sets related by common system usage. Other analysts prefer to identify separate system uses and then detail them with many scenarios. Still others prefer to identify the actors that interact with the system and concentrate on how each actor collaborates with the system.

Select one of these strategies and identify the use cases for the Roadrunner Traffic Light control system. Also identify the actors in the system context and draw the use case diagram for the Roadrunner system. Once completed, apply the criteria provided above to make sure you’ve made a good selection of use cases.

Problem 4.3 Mapping Requirements to Use Cases

In addition to the stereotypes of dependencies between requirements, I also use the «satisfy» dependency to show that a use case is partially specified by a requirements element. I draw this dependency from the use case to the requirements element.

Take one of the use cases identified in the previous problem and create a new use case diagram for it. On this diagram use the dependency relations to relate the operational requirements to that use case as a way of defining that use case. You may also add additional relations among the requirements themselves, as appropriate.

In addition, create a table showing the traceability relations between the use case and the requirements.

Problem 4.4 Identifying Use Cases for the Coyote UAV System

I’ve seen customer models that have a flat organization of 500 or more use cases. These models are practically impossible to use. Remember one of the previous rules of good use cases: *There*

should be no fewer than three and no more than three dozen use cases at the “high level.” When the system is highly complex, it becomes necessary to create taxonomies of use cases at different levels of abstraction. The bottom, or leaf, level use cases contain most of the real meat of the requirements but the taxonomy allows us to view the requirements organization hierarchically as well as horizontally. In practice, I have found this a crucial aspect of useful use case models for complex systems.

For this exercise, first create a model that has the following high-level packages:

- *Architecture* (empty for now, but we’ll fill in stuff later)
- *Requirements* (to hold requirements elements imported from the Coyote UAV specification)
- *Functional Analysis* (for use case analysis)

In the Functional Analysis package, identify the use cases for the Coyote UAV (CUAV) system (see Appendix B). Because of the complexity of the system, it will be necessary to identify “high-level” use cases that are decomposed into more detailed use cases. Put these high-level use cases on one diagram.

For use cases that require decomposition into smaller use cases, copy the relevant high-level use case to a new diagram and add the aggregated and specialized use cases here. Indicate aggregated use cases with the «include» dependency stereotype to create a whole-part ontology of use cases, and specialization with the generalization relation. Do this for all high-level use cases that break down into appropriate smaller uses cases.

Note that the same model element (use case, in this case) can appear on multiple diagrams. It is still the *same* use case element, but we are showing it in different contexts. For a use case, it might appear on an overview use case diagram, showing all the high-level use cases, but it might also appear on another use case diagram to show how it decomposes to smaller use cases, and on a third diagram showing the mapping of the use case to the requirements that it satisfies.

Inside the Requirements package, add a nested package to hold nonoperational requirements such as system weight and payload capacity. Stereotype this package with the «parametrics» stereotype as a way to indicate that this will be a placeholder for system parametric requirements.

Problem 4.5 Creating a Requirements Table

If your tool supports construction of a requirements table, create such a table. If not, you might use a spreadsheet tool or, if you’re lucky enough to have a full requirements management tool such as Rational DOORS, you can do it there. Place the table in the *Requirements* package.

Which view – a requirements diagram or a requirements table – do you think is a better visualization of the textual requirements? What are requirements tables best for? What are requirements diagrams best for?

Problem 4.6 Capturing Quality of Service Requirements in Use Cases

Use cases are coherent sets of functional requirements. Good use cases are named with strong verbs.³ If use cases and their contained requirements are verbs, then quality of service (QoS) requirements are adverbs; that is, they modify the use case or the specific requirements within those use cases. They specify “how much,” “how fast,” and “to what degree.” Such requirements are historically known as nonfunctional requirements, but these days they are more commonly known as QoS requirements. How are QoS requirements represented?

QoS requirements are themselves requirements and can be related to use cases or other requirements (see [Figure 4.3](#)). Standard SysML uses the «deriveReq» stereotype for these relations between requirements and «satisfy» between use cases and requirements. I use the «qualifies» stereotype on dependencies to represent the relation between a QoS requirement and another type of requirement. In this problem, take the *Fly UAV* use case and, in a separate use case diagram, add the QoS requirements that modify this use case and attach them with dependencies.⁴

Note that while Rhapsody provides its own requirements element (from SysML), it is quite common to use constraints to represent requirements elements. Such constraints (or requirements elements) can be shown on use case diagrams or in the more detailed view of sequence, timing, and state diagrams, topics to be discussed later in this chapter.

³ As a naming convention, use cases should be named with strong verbs, although sometimes for highly reactive (stateful) systems, you’ll see use cases named with the word “... mode” in the title.

⁴ A common question I hear is whether a separate package or use case should be created for all QoS requirements. In my mind the answer is, “Clearly not.” If a QoS requirement applies to a use case or a requirement within a use case, then that adverb should be positioned next to the verb it is modifying – that is, it should be placed into the same package as the use case. If it is a parametric requirement, then it should be placed in the package with the parametric requirement it qualifies.

Problem 4.7 Operational View: Identifying Traffic Light Scenarios

So far, we’ve been focused on requirements specification and identifying the use cases to which they belong. This is little more than a modernization of the traditional requirements approaches. This *can* be useful, but large numbers of requirements are still problematic – they are difficult to understand, ambiguous, and almost impossible to validate. Is it possible to use the modeling power of UML to help us?

The answer is a resounding YES (although I’ll bet you could guess I’d say that ;-). We call this “detailing the use case,” although it is known in more formal circles as *functional analysis*. First, we’ll discuss the notion of scenarios as a means of looking at the operational requirements in a use case. In later problems we’ll examine the use of formal languages (state machines and activity diagrams) for specification.

A scenario can be thought of as a specific sequence of inputs and outputs that represents a single path through a use case. A use case normally has many different scenarios – several dozen is not uncommon – that show combinations of different inputs and different sequences.

Scenarios are useful because they relate the system capabilities with the expectations of how the system will interact with other elements in its environment (i.e., actors). Further, while domain experts will understand the flow of scenarios and be able to discern reasonable from unreasonable scenarios, they are normally at a loss when confronted with more formal languages, such as state machines. Lastly, scenarios are valuable at every level of abstraction. At the system level, there is a single lifeline for the system use case (or the system itself, if you prefer) – all other lifelines are actors. At the subsystem level, the system use case lifeline is “opened up” and the internal parts (subsystem use cases or subsystems) can be used as lifelines as well. At the collaboration level, the subsystem lifelines are decomposed into their internal object roles, which collaborate to fulfill the detailed functionality required.

The most common representation for scenarios, by far, is sequence diagrams. However, timing diagrams and communication diagrams (formerly known as collaboration diagrams) are sometimes used, although usually for design-level scenarios only.

A common problem with the application of scenarios is that there are *so many possible scenarios* that people get bogged down, particularly with the so-called “rainy day” scenarios. These are scenarios where faults occur and the system must handle them in some fashion. My recommendation is to define the sunny day “everything goes right” scenarios *first*, and then create the rainy day scenarios.

How many scenarios? A *minimal spanning scenario set* for a use case has all the operational and QoS requirements contained within that use case represented at least once. For faults, it is usually necessary to classify them into *fault equivalence classes*. A fault equivalence class represents a set of faults that are identified and handled using the same means and every fault equivalence class should be represented in at least one scenario.

In an anesthesia machine, for example, a fault created when the endotracheal tube dislodges from the patient’s throat is in the same equivalence class as if the oxygen supply line disengages from the wall supply or if the oxygen supply fails. In all these cases, the fault is detected in same way (e.g., lack of end-tidal pressure wave) and the action is the same (sound the same alarm to the attending physician). Thus, it is only necessary to have a single scenario for “failed to inflate patient’s lungs.” I recommend that all of the specific faults that can give rise to the generic fault are listed in a constraint, but I don’t believe it necessary or useful to replicate the same scenario many times, once with each possible specific fault.

Workflow

I am going to recommend a rather specific workflow to do scenario analysis. Feel free to deviate from it if you have a good reason to do so, however understand that this workflow has emerged as a best practice from literally hundreds of design sessions I have attended with engineers developing real-time embedded systems.

- Organize the model.

In the model organization we created for the CUAV model before, create a set of packages at the highest level:

- Requirements package – contains the requirements
- Functional Analysis package – contains the use case analyses
- Architecture package – contains architectural specifications
- Put the textual requirements in the Requirements package.
- Create use cases and use case diagrams in the Functional Analysis package. You will create use case diagrams to show the different system capabilities, to show the relations among the use cases, and to show the relation between sets of requirements and specific use cases.

- For each use case,⁵
 1. Create a nested package within the Functional Analysis package named something like `<usecasename>Pkg`, such as `FixedCycleModePkg`, to hold the detailed analysis of that use case.
 2. In that package, create a class diagram to serve as the context for the use case
 3. On that diagram (and within that package) add a class (UML) or block (SysML) to represent the use case. I use the naming convention `<usecasename>_uc` or `uc<usecasename>` to show that this element is used to represent the use case and not an internal design element.
 4. Add similar classes (or blocks) for all the actors with which the use case has associations. I use the naming convention `<actorname>_a` or `a<actorname>` to indicate its purpose.
 5. Make the classes into objects (instances), add ports (optional, but this aids in defining the interface specifications later), and add the links between the actor blocks and the use case block.

At this point, we have an operational context in which we can analyze the use case. The less-than-obvious part is using classes (blocks) to represent the use case and actors. Why?

The reason is that we will use this operational context to perform requirements simulation, execution, and verification later in the functional analysis (a topic we'll get to in this very chapter). Using classes makes that simulation easier. Further, classes can have *ports* (named connection points), which are typed by their *interfaces*, so it is a very straightforward way to capture the interfaces required between the system and the actor when it executes the use case.

The use case and actor blocks will be used for the lifelines in the sequence diagrams and will be (later in this chapter) specified with state machines.

If you don't plan to verify the use case with simulation or verification activities, then you can just create scenarios with the original use case and actors. It's a little less work but doesn't support deeper analysis as well.

The Exercise

For this problem take a use case identified for the traffic light system and create three different scenarios. Take care when identifying the actors – they should be objects outside the scope of the system that have significant interactions with the system during the execution of the use case. Also, the preconditions for each scenario must be explicitly specified. I like to put these in a note or constraint at the top of the scenario.

Use a Functional Analysis package to hold your work. Create a new package for each use case you want to analyze (name it and in that package create the context for that use case's analysis). Remember, create the operational context class diagram with the classes (blocks) representing the use case and the actors, wire them together with the ports, and use those elements as lifelines on the sequence diagram.

⁵ If you're doing an incremental development lifecycle, you'll only do this now for the use cases under development in the current iteration. Other use cases will be added in subsequent iterations. If you're doing a more V- or waterfall lifecycle, you can add them now.

Problem 4.8 Operational View: Coyote UAV Optical Surveillance Scenarios

The Coyote UAV system is much larger in size and scope than the Roadrunner TLCS. Consequently, there are more use cases as well as a deeper layering of them. However, the same technique of expressing the operational aspects of the system can be used. It is, however, even more important to organize and target the scenarios better because there will be so many. Each use case will have many scenarios.

In this problem, first set up a `FunctionalAnalysis::PerformOpticalSurveillance` package and create the model context as we did for the Roadrunner system. Then, in this package take the use case *Perform Optical Surveillance* and create three scenarios. In the first scenario, the payload operator moves the gimbaled video camera around with a joystick to examine the area over which the Coyote is flying. In the second scenario, the payload operator should locate a target visually, select the target in the visual field, and then command the camera to follow that target autonomously as the UAV flies in the area. In the third scenario, the payload operator manually scans the area, and upon finding a potential target zooms in 3x on the target and then manually scans the surrounding area.

Problem 4.9 Specification View: Use Case Description

Text is a very useful medium for capturing requirements – it is both expressive and readable. However it lacks precision and is ambiguous and so is problematic if it is the only representation for requirements. In previous examples, we’ve used requirements elements to organize requirements into taxonomies. It is useful, however, to also provide a “structured text” description of the use case itself. This is normally done in the use case description field in tools such as Rhapsody. The format I like to use for this is as follows:

Name: Name of the use case.

Owner: Person or team in charge of developing the use case.

Purpose: The user purpose of the use case (what value it brings, what purpose it serves).

Requirements (optional): List of requirements (optional if requirements diagrams or other traceability means are used).

Data (optional): Data values inherently consumed, manipulated, or generated by this use case – need not be shown especially if information flows are specified. Can reference data types and subranges defined elsewhere in the model or specifications.

Preconditions: What must be true before the use case runs.

Postconditions: What the system guarantees to be true when the use case completes.

Reference documents: Hyperlinks to relevant standards or specifications.

Different authors suggest different formats for this standard use case header. Some, for example, will include a list of actors or constraints. My recommendation is to not replicate information already present (or to-be-captured) graphically, so I neither describe the actors nor list the primary scenarios here.

This problem is to use this (or whatever standard format you prefer) to describe one use case for the Roadrunner traffic light control system (*Detect Vehicle*) and one use case for the Coyote UAV.

4.3 Specification View: State Machines for Requirements Capture

Note: I consider this subject to be the most important one in the entire chapter because most people can solve the other issues fairly easily. This problem, though, is probably the most challenging for students and gives the most value to the use case analysis activity.

Almost all embedded systems exhibit *reactive behavior*; that is, the system waits for events of interest and then reacts to them by executing some set of actions, changing state, and then waiting for the next event. This kind of behavior is precisely what state machines excel at specifying. We will use such state machines to specify the requirements of reactive systems. We may (and most likely will) use state machines later in the design and implementation of the very same systems, but in this section, we are entirely concerned with the appropriate specification of requirements and not the design or implementation.

Some engineers are surprised at this use of state machines in requirements specifications, but state machines have been used to help specify requirements of complex reactive systems for more than 30 years. Statemate™ from I-Logix (now owned and produced by IBM Rational⁶), is one example tool that was originally developed for military aerospace system requirements specification in the mid-1980s. It is still in common use in such environments. In Statemate, functional decomposition is applied to the system, and these functions are defined using a combination of statecharts, truth tables, and action statements. It is more common today to use UML tools, such as Rational Rhapsody, to develop these high precision models of use cases.

State machines have plenty of advantages over the use of text for requirements specifications. While text is very flexible and allows for subtle nuance, state machines are precise and unambiguous. It is because of this precision and lack of ambiguity that state machines are so powerful in requirements specifications. In addition, state machines are *executable* – a characteristic that is invaluable in determining the consistency, correctness, and accuracy of complex sets of requirements. State machines are *testable*, which allows requirements to be verified before the system is actually designed. Furthermore, tests can be automatically generated from such formal requirements and be applied to the completed system later. Text remains useful – there is no convenient or obvious way to explain “Why?” with state machines – but state machines are even more useful than text for the specification of requirements.

Because use cases group requirements into coherent clumps, state machines will be applied a use case at a time. Each use case performs a set of control and data transformations triggered by events of interest – an ideal application of state machines. For use cases that simply run until they are done – that is, they execute algorithms – activity diagrams may be more

⁶ <http://www-01.ibm.com/software/awdtools/statemate/>.

appropriate. In the next section, we will see how activity diagrams can also be used to specify behavior.

When modeling a use case state machine, there are some simple guidelines to follow:

- Incoming messages, commands, or data from actors become event transitions (possibly with data attached as parameters) in the state machine.
 - When data only is incoming, you can use SysML *flow or proxy ports* to represent the data influx and *change events* on the state machine to invoke state behavior when the data changes.
 - For commands, asynchronous messages represent the simplest means to model their impact on state behavior.
 - For autonomously generated action based on time, use timeout events on the state machine using the `tm(interval)` or `after(interval)` event.
- Outgoing messages, commands, or data become *actions* in the state machine.
- Underlying technology and design should not be expressed in this state machine because the purpose of a use case state machine is to specify the required behavior, not the internal design.
- Incrementally develop and verify the state machine, rather than creating a complex state machine and only *then* trying to debug it. A much better approach is the incremental nanocycle, which will be discussed shortly.
- Scenarios for the use case are nothing more (or less) than various transition “paths” through the state machine.
 - Every scenario for the use case should represent transition paths of the use case state machine.
 - For minimal coverage, every state machine transition and action should be represented in at least one scenario.
 - As mentioned before, use case scenarios form the basis for the verification test suite of the final system product.

Scenarios are a very useful, *operational* view of the system behavior. However, scenarios are only “partially constructive,” meaning that they only tell part of the story. A (usually large) set of scenarios is required to show the complete behavior of the use case. A state machine, on the other hand, is “fully constructive” and specifies all of the behavior for the use case in one place. If necessary, these state machines can be decomposed in the standard ways, but whether a nested state is shown on one diagram or decomposed on another, it is still logically a singular view of the behavior.

4.3.1 Developing the State Machine in the Harmony Nanocycle

Developing state machines doesn’t come naturally for many people; it requires a different way of looking at the problem than writing code. However, even people comfortable with

creating state machines can find that they quickly become too complex to easily debug. My solution to this difficulty is to *avoid putting defects in the state machine as you create it*. Easy to say, and – it turns out – not that hard to do in practice if you follow the guidelines I’ll set out here.

The key to defect avoidance is continuous verification of the element you’re creating. In writing code, this premise is the basis of the test-driven development (TDD) practice. In the TDD practice, unit-level tests are developed at the same time as the code, rather than months or weeks in the future, which is more common. These tests are applied as the code is developed; if the code has a flaw, that flaw is immediately repaired. Through this continuous verification of the code, the coding lifecycle produces code that is of far higher quality than more conventional approaches. The Harmony nanocycle is simply a model-based version of TDD:

- Repeat
 - Select *one or a few requirements* (or messages on a sequence diagram) for the use case of interest.
 - Add state elements (states, transitions, events, actions) to support them.
 - Verify through execution that the outcome of the state machine control and data transformations is correct.
- Until all requirements allocated to the state machine are accounted for and all this use case’s scenarios can be replicated via the execution of this state machine.

This nanocycle usually ranges from as little as 10 to as many as 60 minutes. We *do not* insert dozens to hundreds of state machine elements and then pray⁷ that we got it right. We continuously verify that we’re doing the right thing *as we do it*.

Of course, the goal isn’t to create an executable specification; that is merely the means. The goals are to clearly understand the requirements, and identify and correct any missing, incomplete, incorrect, and inconsistent requirements. My experience with hundreds of projects is that creation of executable state machine specifications is an excellent way to achieve that goal.

Problem 4.10 Simple Use Case State Machine Specification

Statecharts are a great way to formally capture a behavioral specification for a system executing a use case. In this problem, you must create a state machine for a simple use case. The *Evening Low Volume Mode* use case is very straightforward, but be sure to capture all the behavioral requirements – the primary road should flash yellow, and the secondary road should flash red. Flashing should occur at 0.5 Hz, with an ON duty cycle of 75%. Also add an initial delay

⁷ It is a personal mission of mine to remove prayer from engineering.

when entering the mode so that any traffic in the intersection has time to clear before engaging in the flashing control.

Note: If you need a more detailed refresher for state machines syntax and semantics, see the companion volume *Real-Time UML 3rd Edition* (Addison-Wesley, 2004).

Problem 4.11 Specification View: Capturing Complex Requirements

The second state machine is more complex. Your goal is to create a statechart for Fixed Cycle Time mode. I recommend you do this in three steps.⁸ First, capture the behavior ignoring turn lanes and pedestrians. Once you get that part correct, then add turn lanes. Remember that there are two different turn lane modes – one in which the turn lanes complete before any straight traffic can go (SIM, or “simultaneous” mode) and another where the turn lane is green along with the straight traffic going in the same direction (SEQ, or “sequential” mode). For this problem, focus solely on SIM mode. A complicating factor is that you must be able to detect when a turn lane detection event has occurred and remember that event until it is time to deal with it – then you can forget it. Use the Latch State Pattern from [2]: create an and-state for each turn lane that has two states: `NoWait` and `Waiting`. When a turn vehicle request occurs for a given turn lane, transition to the `Waiting` state. Once that request has been satisfied (the turn light turns to Red, for example), transition back to the `NoWait` state.

Once turn lanes have been properly handled – and their correctness has been verified through simulation – add pedestrian lanes.

Hint: Initially, don’t try to create an optimal statechart with a minimum number of states. Once you have the behavior modeled correctly, you can then attempt optimization, if desired. Your goal here is to aim for correctness and simplicity, even if there is redundancy in the state machine. This will not impact the design of the system at all because at this point we are just specifying the requirements.

There are two advantages of the recommended approach here. First, the fact that you are specifying the requirements in a precise language will quickly identify many missing or ambiguous requirements. Going on to simulate the specified behavior will identify the remaining requirements issues.

⁸ Breaking down a problem into small chunks and validating each chunk through execution is part of the “nanocycle” of the Harmony process. This approach *greatly* simplifies the task of getting the end product working correctly.

Problem 4.12 Operational to Specification View: Capturing Operational Contracts

In this last requirements analysis problem, we will take one of the Coyote UAV use cases, *Perform Optical Surveillance*, and walk through the way the Harmony process captures requirements. This approach has proven quite useful with large-scale development projects that include a separate systems engineering team (or at least, a distinct systems engineering effort), followed by an architectural specification and finally by a decomposition into the various engineering disciplines of electronic, software, mechanical, and even chemical design.

In software-only projects, particularly ones in which a high degree of agility is important, a simpler approach may be used. However, in this example, we are going to assume that the systems engineering team will be specifying the detailed requirements and systems-level architecture first, and then, at the subsystem level, decomposing the subsystems into electronics, software, mechanical, and chemical parts, assigning specific requirements to each.

The Harmony Requirements Capture Process

The overall Harmony process is shown in Figure 4.4. It is a kind of “V-Process” in which requirements analysis and overall system architecture is specified up-front, followed by a set of analysis-design-implementation-validation increments (the latter will be discussed in more detail in later chapters).

In this chapter on requirements analysis, we will naturally be focusing on the requirements capture and specification aspects of the systems engineering part of the overall process. The workflow for the systems engineering part is shown in Figure 4.5.

For the CUAV, we have already defined a set of system use cases. In this exercise we will perform the black box and white box use case analysis of one of these. In the “black box” step of system functional analysis, we will determine the operational contracts into which the system enters when performing the use case. The operational contracts are defined to be the set of services – provided or required by the system – that are used in the operational execution of

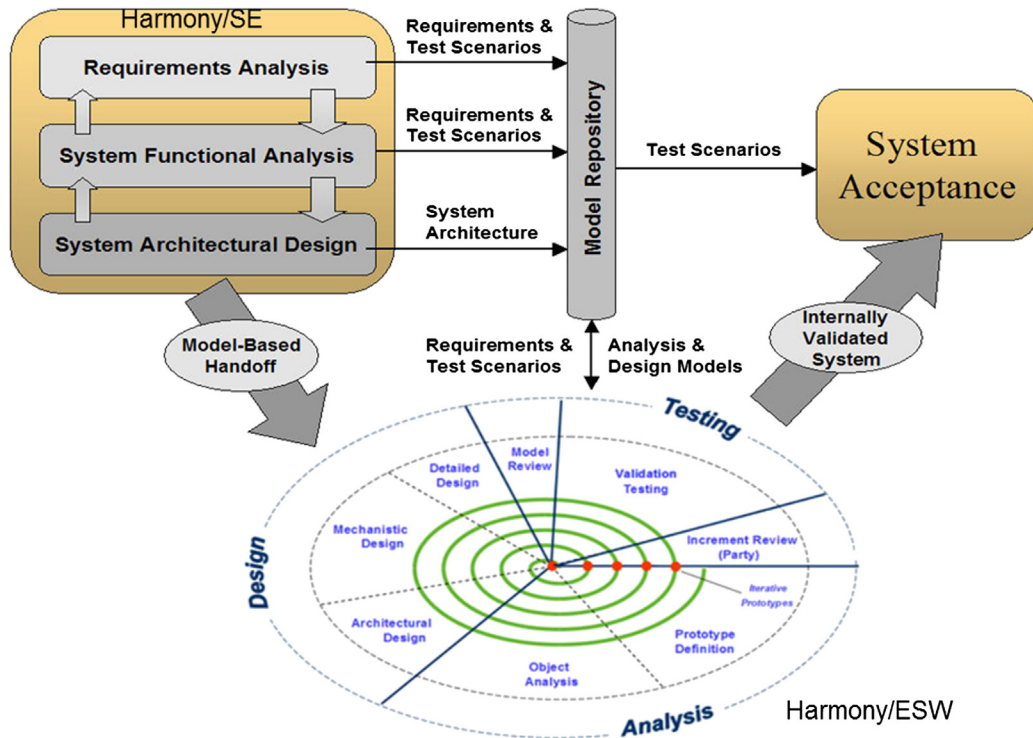


Figure 4.4
Harmony process overview.

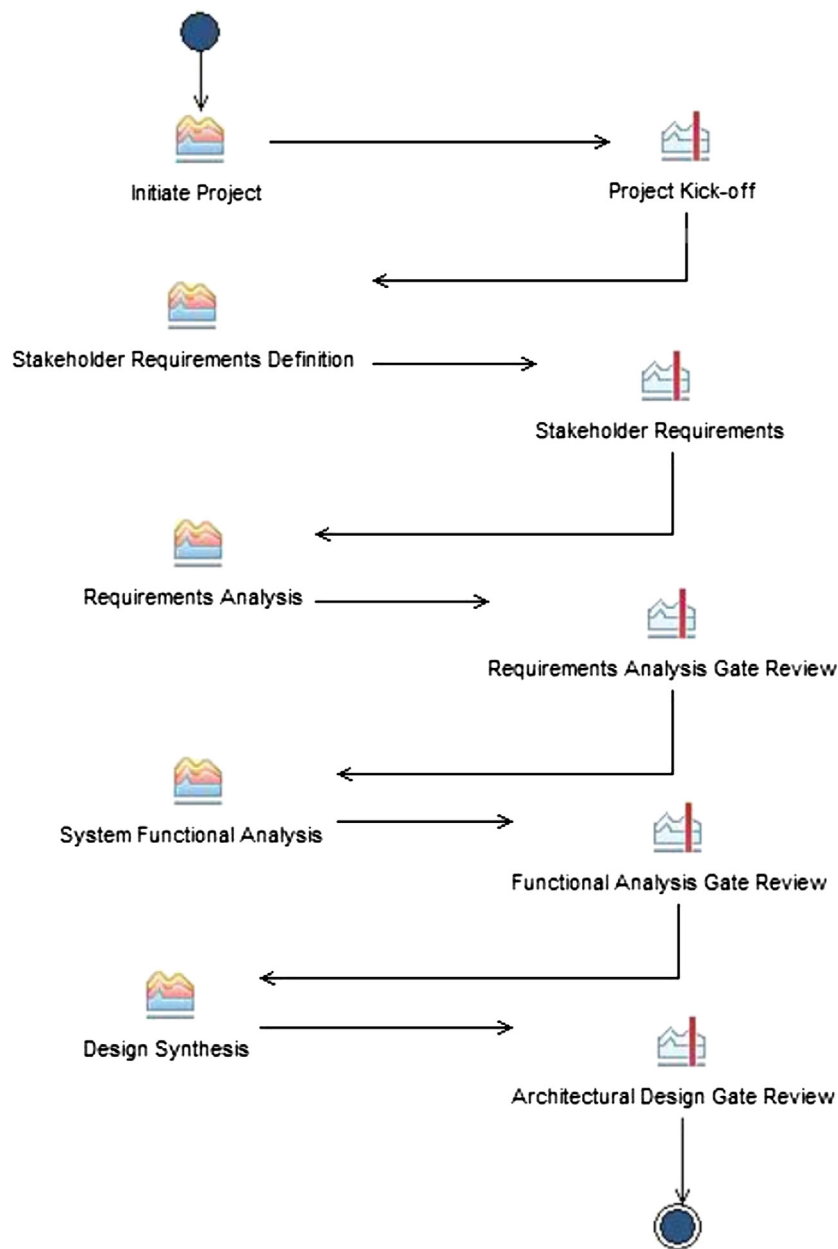


Figure 4.5
Harmony systems engineering workflow.

the system in its environment. This will include the messages sent, the data associated with those messages, and the pre- and postconditions of the messages. In the next step, we will define subsystems for the CUAV, and finally, in the “white box” analysis, we will map these operational contacts to the identified subsystems.

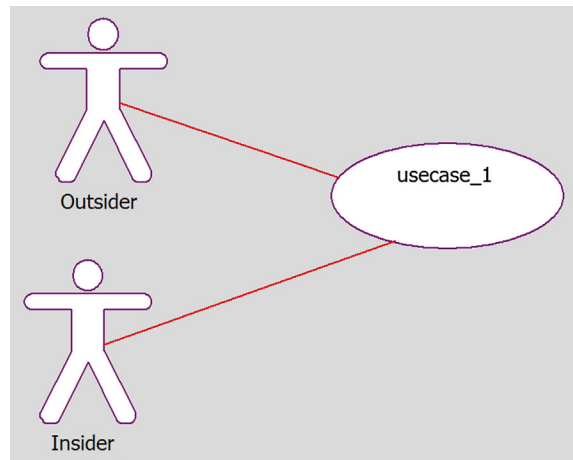


Figure 4.6
Use case diagram (standard).

A Note on Notation

It is perfectly permissible to use a state machine to specify a use case, and that is what we have done here. However, it is also useful to begin to specify interfaces, and that is something that cannot be shown directly on a use case diagram. For this reason, we will represent the use case with a class (or block, if you're using SysML). As a class, I can add ports to the use case and specify with precision the interfaces (including the services and data) that define the contracts that the ports support. This will also be useful later when we add internal structure to the collaboration of elements realizing that use case – we can simply nest the object roles within the use case class as parts. To show clearly that the class is representing a use case, I will preface the name of the class with “uc.” I will also represent actors as classes, because UML lacks a notation to show these as instances or to show the interfaces that they might require or provide – a crucial consideration for systems engineering. Actor class names will be prefaced with “a.” For example, [Figure 4.6](#) is a standard use case diagram that is represented using the classes in [Figure 4.7](#). The latter figure is more explicit than the former with respect to connection points and interfaces.

In SysML, a class is represented (exactly) by a block. Objects can be made by instantiating the classes (or blocks). This is necessary to link together the ports on the elements of the diagram, as shown in [Figure 4.7](#). An object can also be drawn directly; in this case, the object has an implicit class type.

Converting Coyote Flight Requirements to Operational Contracts

The basic approach for specifying operational contracts is shown in [Table 4.1](#):

Note: In systems engineering, the defined interfaces start life as “logical” interfaces; they define the service and logical data passed to and from, but not the actual physical interfaces that will be invoked and not the actual physical (bit-defined) data formats. For example, a logical interface to a radar might specify an `evRadarTrack(track: RadarTrack)` event; this would be modeled as an asynchronous event sent from the radar actor to the use case. The data type *RadarTrack* would define the necessary data and their characteristics, such as range (from 10m to 20,000m),

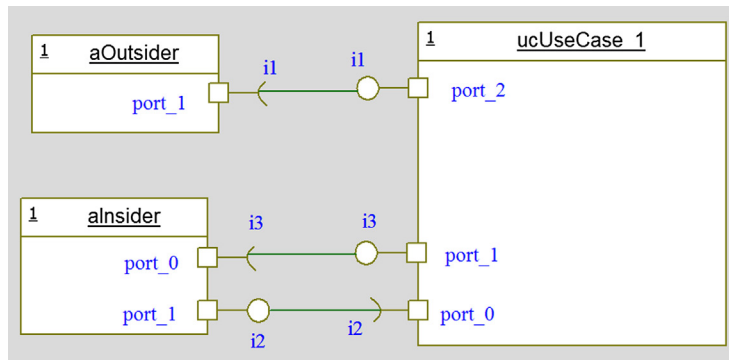


Figure 4.7
Use case diagram (class notation)

Table 4.1 Harmony Process for Defining Operational Contracts.

Step	Task	Work Product	Comment
1	Define use case model context	Use case diagram; one structure diagram per use case	Ports are defined but “empty” at this stage.
2	Identify system-level operational contracts in the black-box use case scenarios	“Black box” sequence diagrams; system-level operational contracts	Lifelines can be actor or use case blocks.
3	Define use case functional flow	Use case black box activity diagram	This task may run in parallel to step 2.
4	Realize messages as operations	Populated black box structure diagram	It is easiest to not populate the model with the messages until the sequence diagram is “stable,” and then populate it using the “auto realize” feature.
5	Define interfaces; allocate messages to interfaces		
6	Define system-level state behavior	Use case statechart diagram (per use case)	Derived from black box use case scenarios.
7	Verify and validate black box use case model through model execution	Validated system-level use case model	

accuracy (within 1m) and time fidelity (within 0.5s), but not the actual bit format. When the logical interface is translated into a physical interface, it might be defined as a 1553 bus message with defined fields and the data type must be defined down to the placement and meaning of individual bits. In the Harmony process, that is done during the hand-off to software engineering, as described in Chapter 2.

For the CUAV use case *Perform Optical Surveillance* we already have a few scenarios. Note that messages from the actor will become operational contracts defined on the ports while “messages to self” – called “reflexive messages” – will be used to specify the functional decomposition of the required processing done in response to the invocation of the operational contract. These reflexive messages are strictly optional in the black box use case analysis, but will become important when the systems architecture is defined, as they will be realized through a collaboration of messages sent among the subsystems.

Your job, in this exercise, is to

1. Draw the block diagram representing the use case, the actor(s), and the interfaces between them.
2. The messages that come from the actor(s) must be collected into *provided* interfaces on the appropriate port of the use case block and *required* interfaces on the actor port. Messages sent to the actor(s) must be collected into *required* interfaces on the use case block and *provided* interfaces on the actor port. Messages to the use case block should have public visibility while reflexive messages (if any) should be protected or private.
3. For each service (operation) the system or its actors provides, specify pre- and postconditions of the service and the logical data types of parameters, if necessary. Model the services as asynchronous event receptions that pass data as necessary.
4. For each logical data type that defines data passed as a part of the services, specify the important data attributes and properties.
5. Construct a use-case activity diagram representing *all* of the scenarios previously specified for the use case.
6. Define a state machine for the use case block that is consistent with the set of scenarios.

Arguably, the last item in the list will be the most challenging.

References

- Douglass, Powel, B., 2004. *Real-Time UML Third Edition: Advances in the UML for Real-Time Systems*. Addison-Wesley.
- Douglass, Powel, B., 1999. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley.

Systems Architecture: Deployment and Subsystems Architecture

What You Will Learn

- Essential properties of systems architecture
- Organizing systems models
- Identifying systems architecture
- Specifying subsystem interfaces
- Mapping requirements to subsystems

5.1 Overview

This chapter is specifically for projects that contain a systems aspect – that is, systems that will ultimately be realized using more than a single engineering discipline. It is common, for example, for a system to contain software, electronic, mechanical, and even chemical parts. The systems architecture identifies requirements of the system as a whole as well as the architecture of the system, constructed of subsystems that are themselves to be implemented with some combination of engineering disciplines. In the previous chapter we discussed the capturing of systems requirements. In this chapter we will focus on the specification of a systems level architecture.

Many projects are primarily software-oriented and needn't spend effort defining the systems architecture, although significant effort should be spent in specifying the software architecture (the subject of Chapters 6, 8, and 9). However, even for software-only projects, this chapter may be highly useful as it introduces the important aspects of architecture and focuses on one of these – the identification of subsystems, the decomposition of system use cases into subsystem use cases, and the specification of interfaces and interactions among the subsystems.

Let us review what we mean by the term *architecture*. Architecture encompasses the large-scale design decisions that affect most or all of the system. Note that architecture is a part of design. In the Harmony process, analysis is all about capturing the essential aspects of the system – the properties and characteristics of the system that are essential for correctness. Design, on the other hand, focuses on optimization of the analysis model against what are

collectively termed the *design criteria* – the set of aspects of the system against which different designs and technological choices are measured, evaluated, and, ultimately, selected. Design criteria may refer to system performance (such as worst-case execution time, bandwidth, or throughput), runtime usage of system resources (such as memory), design-time “goodness” metrics (such as complexity or encapsulation), design properties (such as maintainability, reusability, or portability) or even project properties (such as work effort required).

The Harmony process identifies three levels of design. *Architectural design* attempts to optimize the entire system scope at a gross level with a coherent set of architectural choices. *Mechanistic* (also known as *collaboration*) *design* seeks to optimize a collaboration of objects working together to realize a system-level capability (e.g., a single use case). *Detailed design* optimizes individual classes, objects, functions, and data structures. We’ll focus on just architecture here since that is the perspective of system engineering.

Architecture is not a monolithic thing, but has different aspects. The Harmony process identifies five key aspects of architecture, highlighted in [Figure 5.1](#).

The *concurrency and resource view* of the architecture identifies the concurrency aspects of the system and how resources are shared among those concurrent units. The *distribution*

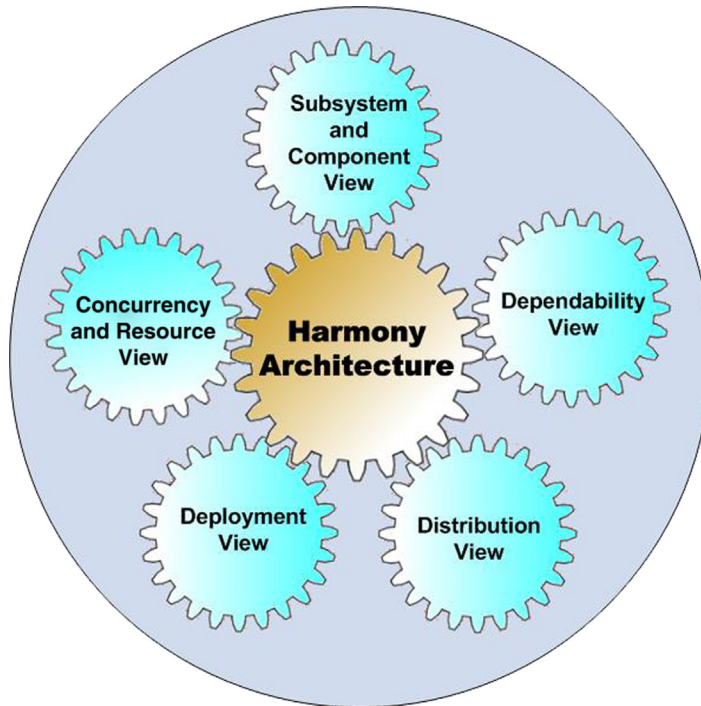


Figure 5.1
Aspects of architecture.

view specifies how the objects will be distributed across multiple address spaces (e.g., CPUs) and how they will communicate and collaborate efficiently and reliably. The *dependability view* of the architecture concerns itself with architectural mechanisms that manage our ability to depend upon the system, and is composed of three primary aspects – safety, reliability, and security. The *deployment view* identifies which aspects of the system shall be implemented in the various engineering disciplines and how they will interact. Finally, the *subsystem and component view* concerns itself with the specification of the largest scale pieces of the overall system, how the system functionality maps into those structures, and how those elements interact in the large scale. It is this last aspect of architecture that is the primary focus of systems engineering, although the other aspects may be considered as well.

What we mean by *systems architecture* is the identification of the strategic design decisions that affect most or all of the system from the *systems* point of view. The systems point of view is “above” software, electronic, or mechanical engineering. Thus, the systems architecture will focus on the specification of the set of subsystems into which the system will be decomposed, the allocation of requirements and functionality to those subsystems, and the interfaces between those subsystems. At the end of the systems architecture activities, portions of the systems model are handed off to the subsystems team where the subsystem is then decomposed into the engineering disciplines and more detailed analysis and design work begins.

To this end, the systems architectural definition will focus almost exclusively on the subsystem and component view of the architecture, although it may have to “drill down” into the other views from time to time.

Because architecture is a big topic, this book will discuss it in three chapters. This chapter will focus on the subsystem and component view as well as the deployment view. Chapter 6 will deal with dependability. These two chapters are presented now because of their impact on system design, including its realization in different engineering disciplines such as mechanical, electronic, and software. Chapter 9 will delve into concurrency and resource architecture, as this is almost solely a software concern. Chapter 8 focuses on distribution architecture, which is a concern of software and to a lesser degree, systems.

Problem 5.1 Organizing the Systems Model

Although the Harmony architectural views are focused around the properties of the physical system, we must also concern ourselves with the architecture of the design per se – and the primary concern with a UML- or SysML-based design is that of *model organization*.

Organizing the model is something not usually considered until the team runs into a problem managing the burgeoning complexity of a system under construction. Reorganizing a model at that time requires a nontrivial and (almost always) unscheduled effort. Projects that have a systems team performing systems analysis are usually complex enough to warrant some early consideration of how you would like to manage the model.

There are many ways to organize models and while we will focus on only one here, there are viable alternatives. As with most things, though, there are many more *bad* ways to organize models than good ones. Some points to consider include:

- Large project or small?
- One model or many?
- Single product or family of products?
- How will common parts be shared among team members?
- How will the architectural aspects be made available to the team?
- How will the architectural decisions be enforced among the team?
- Will the team be co-located or distributed?
- How will we minimize model management overhead?
- What principles will be used to group elements together?
- How will configuration management be used?

One of the first considerations is whether to have one large model shared among the team or separate models. A single model has the advantage of simplicity of management – you only have one entity to manage (even if it has subparts). On the other hand, a single model takes longer to load – a real concern if you have 50 or more members on a team – and finding things to facilitate sharing and collaboration in a large monolithic model can be difficult. Multiple models have the advantage that system complexity can be divided across many different models and each model is smaller, more manageable, and takes less time to load than a larger single model. On the other hand, significant thought should be put into what the submodels should be, what criteria should be used to locate model elements in the various models, and how the models will be shared across multiple stakeholders.

As a recommendation, for teams of 15 or fewer members, a single model may be a good choice; for teams of 20 or more, multiple collaborative models are probably a better choice. Of course, the properties of the system under development impact the decision. If the model is linearly separable (i.e., able to be broken into a number of more-or-less independent pieces), management of multiple models is easy. If the system is not linearly separable, it may be much more difficult to create multiple models. If, in order to work on one model one must import all the other models as well, breaking the system up into multiple models won't help you.

The primary reasons to break up large models are to (1) decrease load/save times and (2) provide smaller but sufficient models to teams who have narrow focus. It is this latter concern that we will address here. The typical stakeholders for the system model are shown in [Table 5.1](#)

[Figure 5.2](#) shows a typical recommended model organization for a medium to large project. The icon for a model is a package that contains a nested triangle. This figure shows a systems engineering model (`SE_Model`), a model holding shared elements (`Shared_Model`), and a set of subsystem models. Each of these models has internal packages, only some of which are shown. In fact, any of these models can be split into multiple models; for example, there might be multiple SE models focusing on different use cases.

The models are related in three different ways. The subsystems «import» a copy of their specification package while they reference (via the «usage» relation) elements that are used by at least two subsystems. The «refine» relation indicates that the logical elements (types and interfaces) must be refined into physical elements; this is done during the hand-off from systems to downstream engineering.

Table 5.1 Model Stakeholders.

Group	Purpose	Scope of Concern
Systems engineers	Construct a discipline-independent model of requirements and architecture Map requirements and functionality to subsystems Specify interfaces between subsystems	System requirements System architecture
Subsystem team	Create a deployment architecture for the specific subsystem Create software and hardware specifications (submodels)	Single subsystem Mapping of subsystem requirements to engineering disciplines
Engineers (software, mechanical, electronic, chemical)	Perform analysis and design of a specific subsystem within their discipline	Model elements within their discipline for a single subsystem
Design architect	Oversee architectural design for the set of subsystems	Common architectural model for entire system Discipline-related architecture for each subsystem
Testers	Subsystem-level testing Integration testing Validation testing	Subsystem – normally done within the subsystem team Integration – normally done with system-level scope Validation – normally done at system-level scope

In UML, the unit of model organization is the *package*. A package is a model element that contains and provides a namespace for other model elements, some of which may be (nested) packages themselves. Model organization can be shown in either a package diagram (a class diagram whose purpose is to show the organization of packages in a model or set of models)¹ or in a browser (tree) view.

The Harmony process recommends a canonical systems engineering model organization. This is shown in [Figure 5.3](#). The organization can be seen both in the browser at the left or as a package diagram on the right.

The highest level packages are Requirements, Functional Analysis, and Architecture. Functional Analysis holds the use cases and use case diagrams as well as one package per use case for functional analysis, as was done in the previous chapter. The Architecture package is divided into two packages, one for architectural trade-off analysis and one for elaborating the selected architecture (ArchDesignPkg in [Figure 5.3](#)). This package holds the architectural specifications that will ultimately be handed off to downstream engineering; one package per subsystem to hold allocated requirements and subsystem level use cases and their specifications, and packages to hold the subsystem interfaces and the user-defined logical types used by those interfaces.

It should be noted that more packages can be added as necessary. If you have 10 use cases, then a single package to hold them (FunctionalAnalysisPkg) is fine. If you've got 100, then you will want to organize them into subpackages within the FunctionalAnalysisPkg. I don't care *how* smart you are, at some point, your brain just gets too full and you need to create a nested organization to manage all the pieces.

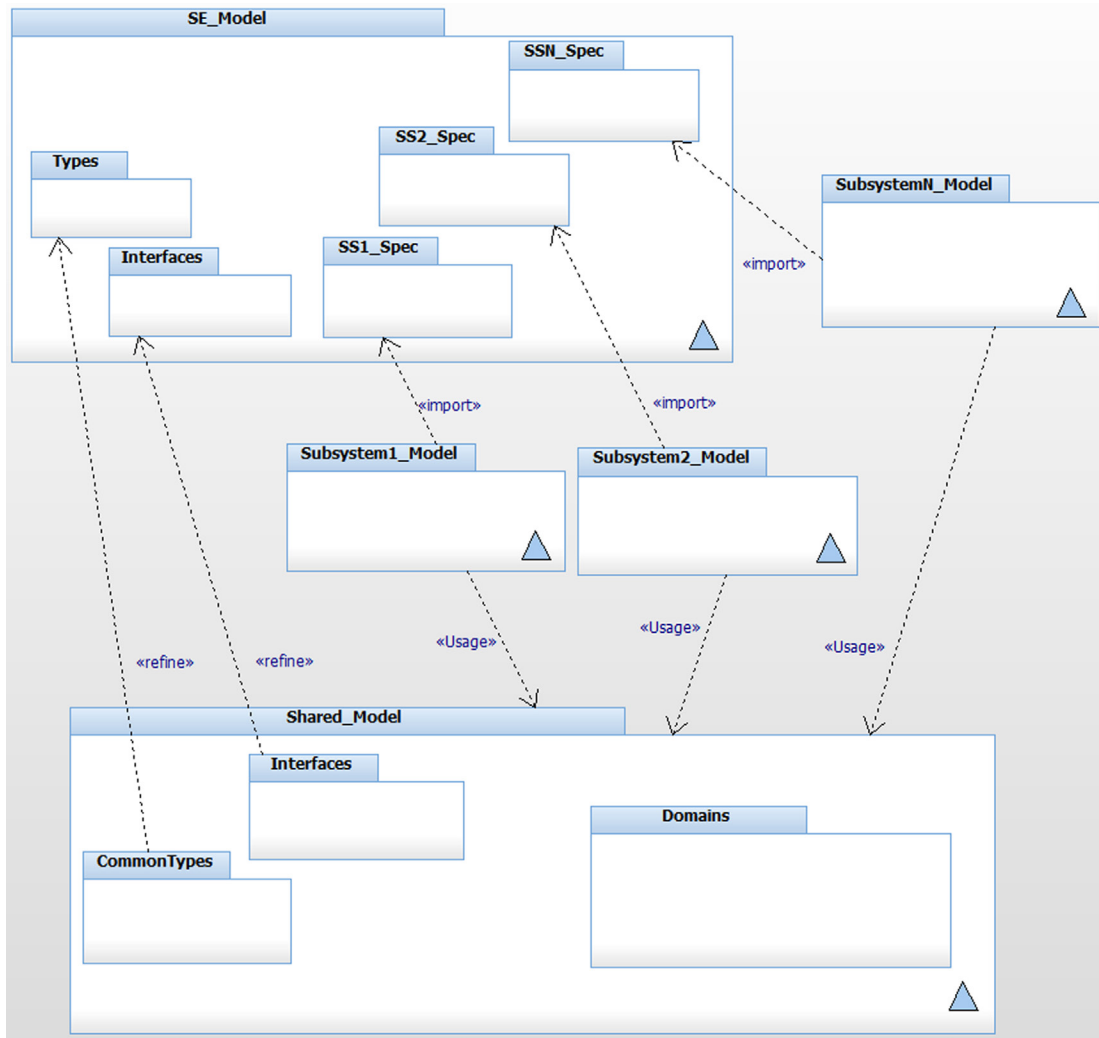


Figure 5.2
Recommended large project organization.

The subsystem models from Figure 5.2 also have a recommended canonical structure, which is shown in Figure 5.4. Although this model is used primarily for downstream software engineering of the subsystems, it provides a common specification for all involved engineering disciplines, including mechanical and electronic. The primary usage difference is that the latter disciplines will use the model only for specification and will (typically) use other tools for their design, while the software design will continue in the model.

The systems engineering specification of that subsystem is imported from the systems engineering model – this is the package named `SubsystemSpecPkg`. This specification is multidisciplinary and is allocated within this model to different engineering disciplines, such as mechanical,

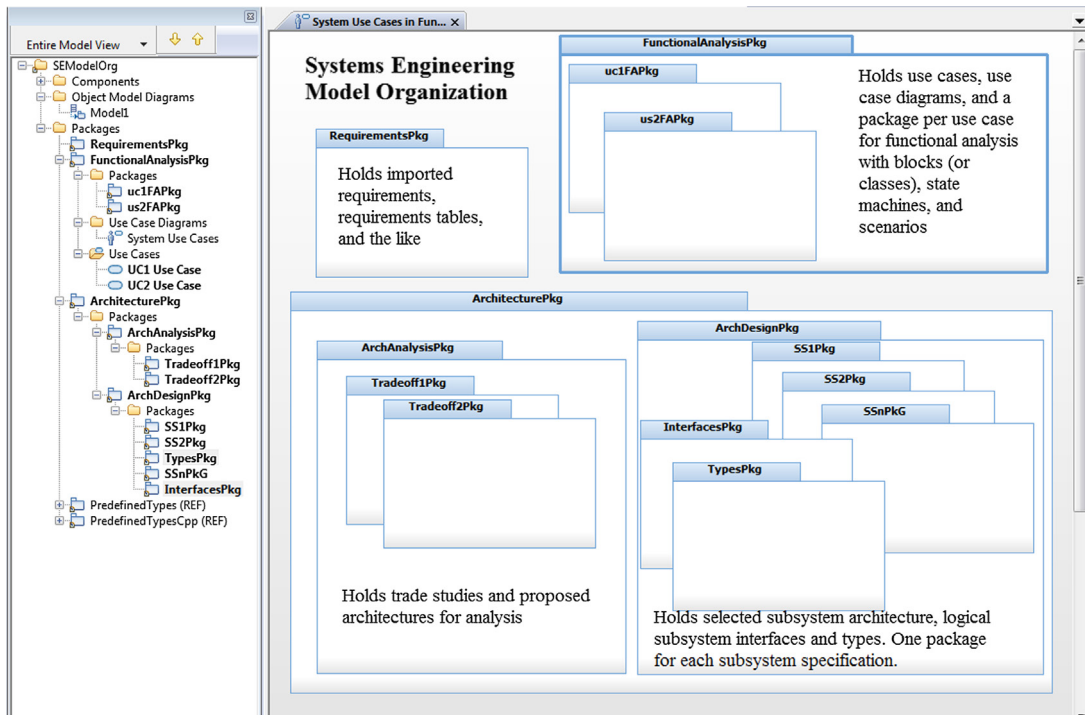


Figure 5.3
Canonical systems engineering model organization.

electronic, and software (sw). This allocation – and the cross-disciplinary interfaces (e.g., sw-electronic, electronic-mechanical) – are held within the `DeploymentPkg`. The `SWReqPkg`, `SWFAPkg`, and `SWArchPkg` serve the same function for software as the `RequirementsPkg`, `FunctionalAnalysisPkg`, and `ArchitecturePkg`, respectively, did for systems in the systems engineering model. The software analysis and design are held within the `SWDesignPkg`, which – outside of a `TypesPkg` package for widely used types – is organized by internal packages holding the collaborations for each of the use cases. Lastly, there are three packages referenced from the shared model. The `InterfacesPkg` and `CommonTypesPkg` packages reference system and subsystem interfaces and types used by those interfaces. The `Domains` package holds software-specific elements that are used by multiple subsystems so they don't have to be defined everywhere they are used.

For the first problem in this chapter, you must

1. Create a single model for the Roadrunner Traffic Control System organized as above; that is, include both the systems engineering model and the software model together in the same model.
2. Create a set of models for the Coyote UAV (CUAV) using the suggested large-scale organization discussed above. Since this is a large-scale model, use separate packages for the system-level use cases, nested within the primary packages.

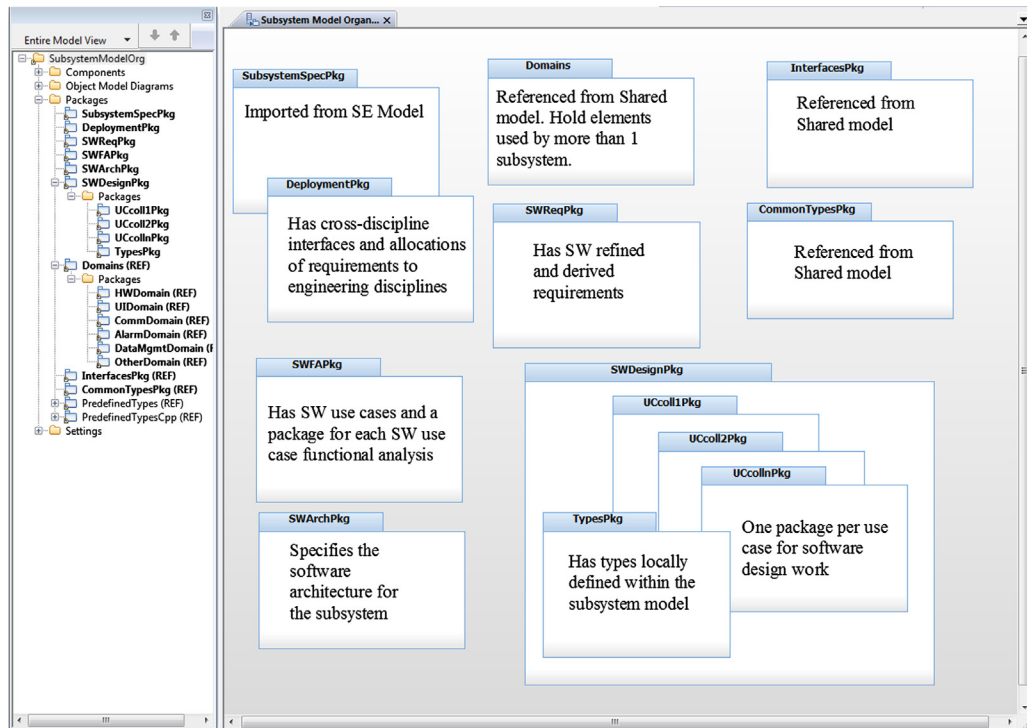


Figure 5.4
Canonical subsystem model organization.

Note that we haven't yet identified the subsystems or domains, so you can leave the subsystem packages of your model empty. We will elaborate this model organization as we identify subsystems and domains later in this chapter.

¹ A model is a kind of package in the UML metamodel.

Problem 5.2 Subsystem Identification

At this point, we have selected an overall model organization (the “logical architecture”) for the system. It may be in one model – as is the case with the Roadrunner Traffic Light System – or it may be a set of interrelated models – as is the case for the CUAV system. This means that we've identified the set of organizational units that exist at design time to help us manage the complex problem and solution data we have set about to create. We need to make a similar set of decisions about how we organize the set of runtime elements as well. Packages, the elements used to organize the models, only exist at design time. Packages are not instantiable elements but serve only to organize the elements of the design.

To organize the elements that exist in the running system (e.g., the objects) we must use instantiable elements – objects of some kind. In UML, this means instances of classes. UML

defines a couple of kinds of classes that are used for large-scale runtime organization. Specifically, subsystems and components are used for this purpose, but these are basically just structured classes. A *structured class* is nothing more (or less) than a class that contains internal *parts* (that are themselves specified by classes). A subsystem or component is basically a large-scale structured class. There are a few technical differences, such as a component associated with an artifact, but these differences are minor. An artifact, for our purposes here, is the embodiment of the runtime entity, such as a .dll, .exe, or .lib file that manifests the object. A subsystem is just a kind of component. But really, it's all just about classes and their instances.

While UML defines the elements Class, System, Component, and Subsystem, it really doesn't mandate how they are used. The Harmony process recommends a particular set of guidelines that we have found effective:

- A System element is a class that encompasses the largest scale part of your project.
- A Subsystem element is a class that is the first-level compositional unit of a System object (that is, Subsystems are parts of Systems).
- A Component is a software class that is the first-level compositional unit of a Subsystem.
- A Task is a structured class, contained within a Component, that is active (i.e., owns the root of a thread) and is the fundamental unit of concurrency in the model.

In practice, runtime systems are organized by large-scale objects that define these organizational units, and these instances are typed by classes. In very large-scale systems, your project might contain multiple instances of each of these levels. In small systems, you might skip some levels of organizational abstraction altogether.

So what makes up a good set of subsystems? How is functionality decomposed into subsystems? How do subsystems interact? Over the years, I have developed a set of guidelines that I have found useful for subsystem identification

- A subsystem should be loosely coupled with other subsystems, but relatively tightly coupled internally.
- Subsystems should enforce encapsulation through the use of ports and contract-driven subsystem design.
- A subsystem should contain elements that contribute to a small number of coherent capabilities.
- A subsystem should have a well-specified set of interfaces, consisting of
 - A set of services provided by the subsystem
 - A set of services required from other subsystems
 - Common type definitions for data passed across the subsystem interfaces
 - Pre- and postconditions for each service
 - Constraints and invariants for each service
- A subsystem should use a common set of hardware to support its functionality.
- In a multiprocessor system, a subsystem normally operates entirely on a single processor or names a coherent set of functionality operating on a tightly coupled set of processors.
- If multiple design teams are to be employed, a subsystem should be designed by a single team.

Of course, these are guidelines and not strict rules – good design is still as much an art as it is an engineering discipline.

Ports are often used in subsystem architecture. Ports are a design pattern that

- Allows the explicit delegation of services defined on the subsystem to internal parts
- Enforces encapsulation by not allowing outside clients to have direct knowledge of the internal structure of the structured class
- Explicitly specifies interface contracts
- Introduces some level of extra complexity and has some small performance and memory overhead that cannot always be optimized away

Because of the overhead of port usage, ports are not recommended for use everywhere, but for subsystems and architectural objects, ports provide benefits that usually outweigh their costs.

For this problem, using the guidelines above, model the subsystem architecture for both the Roadrunner Traffic Light System and the CUAV. This means that you should identify the subsystems and draw the subsystem architecture using a structure diagram with the composite class representing the entire system, and the internal parts representing instance roles of the subsystems. Add ports and connect the subsystems together with links as you think appropriate. Understand that these ports and links may change as we do further analysis and design.

The Roadrunner model contains a `SEModel::SystemArchitecture` package. Place the subsystem classes, objects, and diagrams in this package. Note that the CUAV may have multiple levels of subsystem (such as the air vehicle, the ground station, and subsystems within those primary systems). Since the CUAV is spread over multiple models, identify the subsystem architecture model in the systems engineering model in the `Architecture::ArchDesignPkg` package.

Problem 5.3 Mapping Operational Contracts into Subsystems Architecture

As we have seen, requirements are captured in a number of forms, even in model-based development projects. One of the key forms is the *scenario* – a sequenced set of service invocations and responses that occur during the execution of a system. In this context, a scenario is a path through a use case state machine, in which a particular set of services are executed in a particular sequence to get a specific result. At the highest level (“black box”) of abstraction, the entire system becomes a single lifeline on the sequence diagram. However, system-level things do very little work in and of themselves; primarily they organize and orchestrate the behavior of their internal parts to achieve the system’s operational goals. This means that a measure of the “goodness” of the subsystem architecture is whether they can collaborate together to achieve the operational goals of the entire system. If they can do so in an efficient way, then the subsystem architecture is “good”; if they cannot, then it “needs improvement.” The set of services in a set of scenarios invoked between two elements at the same level of abstraction constitutes those elements’ “operational contract” and will be used to specify the interfaces on the ports between those elements.

In this exercise, you will be given a use case for each of the two examples and one or more scenarios. Your job will be to take the scenario, and the operational contracts that it implies, map it down to the subsystem level of abstraction, and demonstrate how the subsystems collaborate together to realize the required use case behavior.

Figure 5.5 shows the use cases for the Roadrunner Traffic Light Control System. In this exercise we will select one use case and one or two scenarios and do the subsystem elaboration (“white box”) view, identifying the roles that the subsystems play in the execution of the scenarios.

We will consider the Responsive Cycle Mode use case. Remember, in this mode, the system senses and responds to vehicles and pedestrians by setting the lights appropriately. In the next figure, Figure 5.6, we see how this use case plays out in one particular case (Scenario 1, not shown, depicts the execution of the system in this mode with no arriving traffic).

The next figure shows a different scenario for the same use case. Just to illustrate a slightly different way to model, in the previous scenario (Figure 5.6) the system use case is shown on the lifeline; in the next figure (Figure 5.7) the system object is used. The difference is immaterial and which you use is a matter of personal choice. If you use the use case as a lifeline, then it represents the system executing that functionality; if you use the system object, then it still represents the system, but in this scenario you’re still focused on the functionality in that one use case. Whichever you prefer, these two scenarios are illustrating different ways in which the Responsive Cycle Mode use case can play out. In the former case, a vehicle arrives on the secondary road, causing the traffic light controller to cycle through the lights. In the latter case, a pedestrian signals that they would like to cross the street, causing the traffic light controller to cycle through the lights to allow the pedestrian to traverse the street.

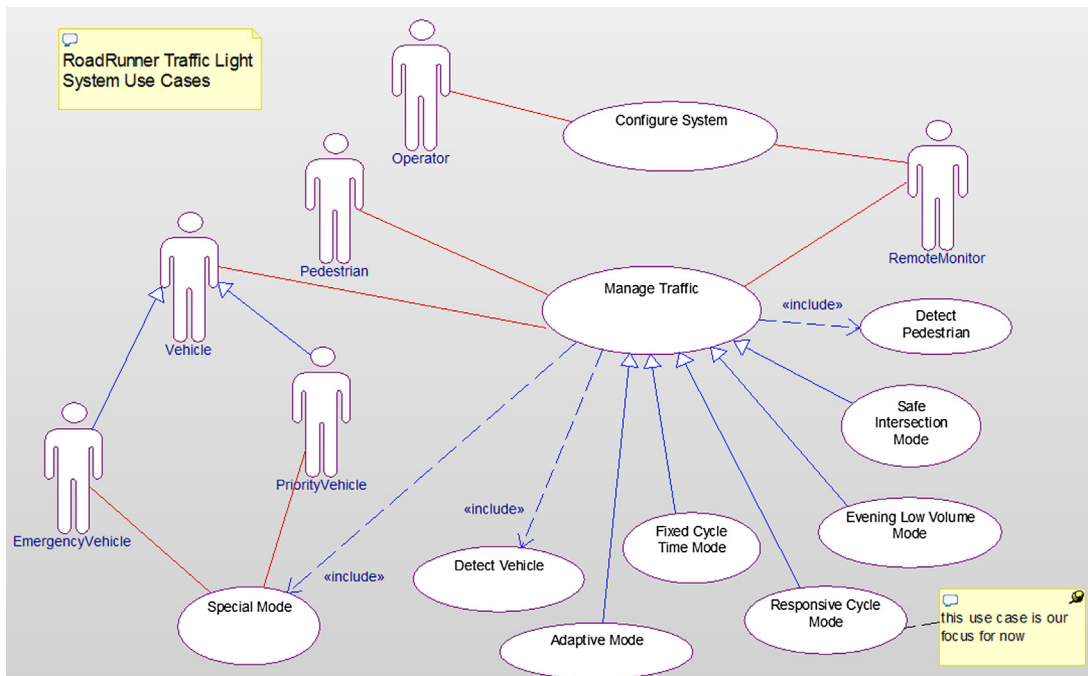


Figure 5.5
Roadrunner use cases.

Take these two scenarios and elaborate them to include the subsystems identified for the Roadrunner Traffic Light Control System, showing how the subsystems collaborate to realize the required system-level behavior.

The second part of this exercise is to do the same thing for the CUAV. We want to map the operational contracts identified in the Perform Area Search use case. The use case, and its

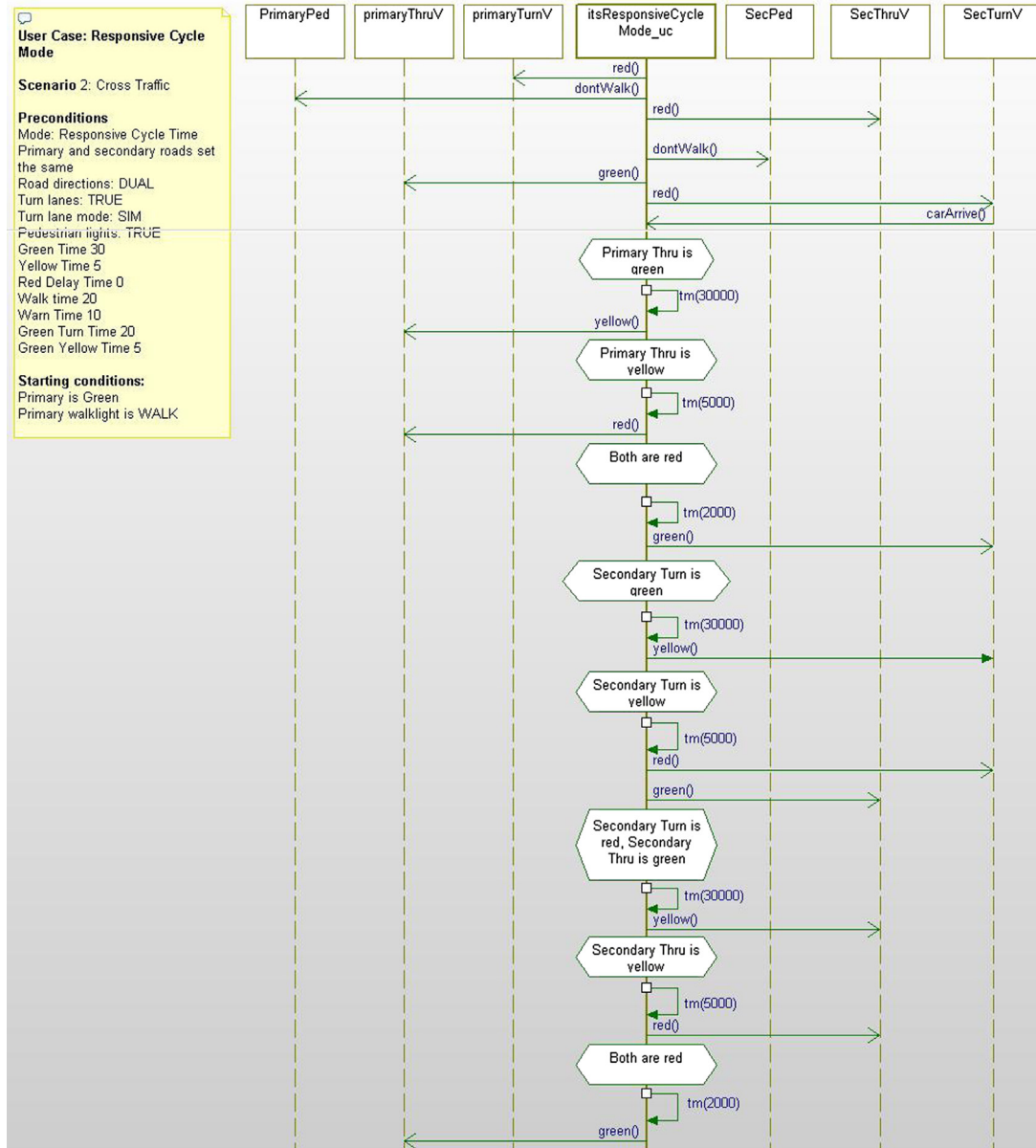


Figure 5.6
 Roadrunner Responsive Cycle Mode Scenario 2.

description, is shown in [Figure 5.8](#). The CUAV is, of course, a large-scale complex system. The Perform Area Search use case includes other use cases as summarized in [Figure 5.9](#); since Perform Area Search is a kind of Execute Mission use case, it includes (indirectly) navigating and flying the UAV, managing the datalink, acquiring and processing surveillance data, and so on. So when we model the operational contracts of the complete Perform Area Search, a great deal of the CUAV behavior is ultimately included. We will simplify it somewhat here to make the exercise more tractable for a homework exercise, but we will include enough to give you a flavor of what's involved. The beginning student shouldn't be too discouraged at the complexity of the system – after all, aircraft requirements and systems analysis usually takes place over several years by a set of highly specialized engineers.

The first thing to do is to create the context diagram for the Perform Area Search use case, shown in [Figure 5.10](#). The object instances on the diagram will be used for the lifelines on the sequence diagrams to follow. Note the PAS (short for Perform Area Search) prefix on the name of the actor blocks and the interfaces. This allows this analysis to be identifiably independent from the analysis of other use cases that might use the same actors. Notice also that in [Figure 5.9](#), no actor is directly associated with the Perform Area Search use case, but there are actor blocks in [Figure 5.10](#). This is because the use case inherits associations with actors via its generalization/specialization relationship – it is, after all, ultimately a kind of Execute

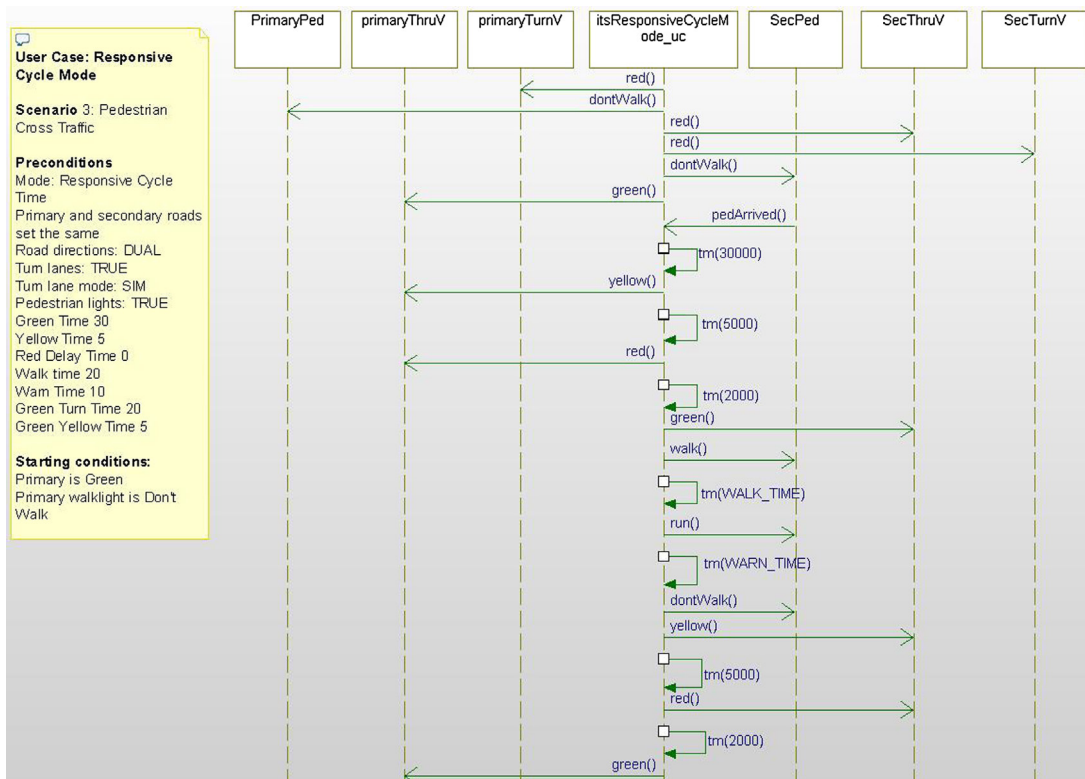


Figure 5.7
Roadrunner Scenario 3.

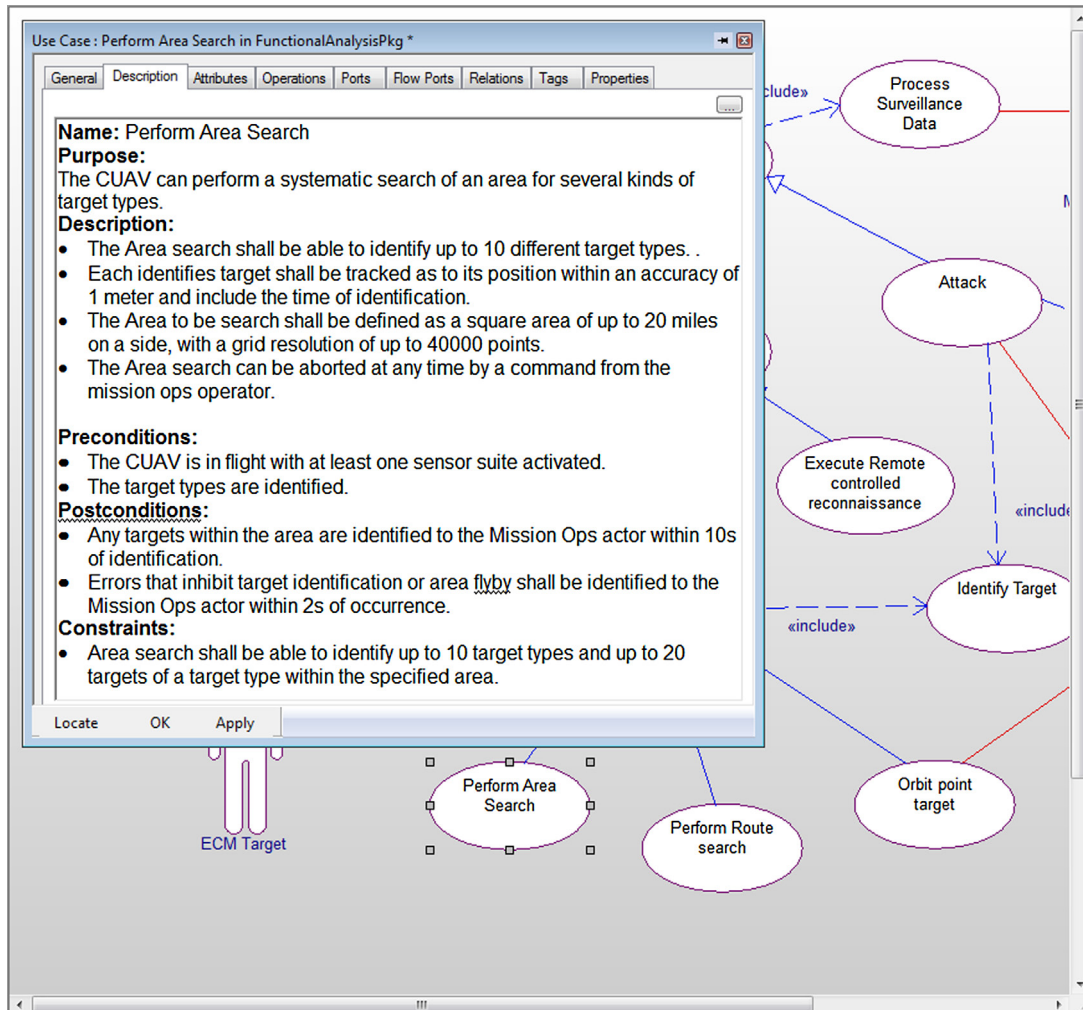


Figure 5.8
CUAV Perform Area Search use case.

Mission. Its parent use cases either directly associate with the actors or they include other use cases that do.

The task here is to create scenarios to identify the content of the interfaces between the subsystems. In the next several figures, we will see the black box system-actor interfaces for the Perform Area Search use case. Your task is to identify the internal subsystems and allocate services or decomposed services to those subsystems. The content of the interfaces defines the contract between the system (executing the use case) and the actor. This can be elaborated to define the ICD (Interface Control Document) between the system and its actors.

Because of the length of the scenario (and to illustrate the technique), the scenario is broken across multiple sequence diagrams. The first is, in some sense, the “master,” or high-level view (Figure 5.11).

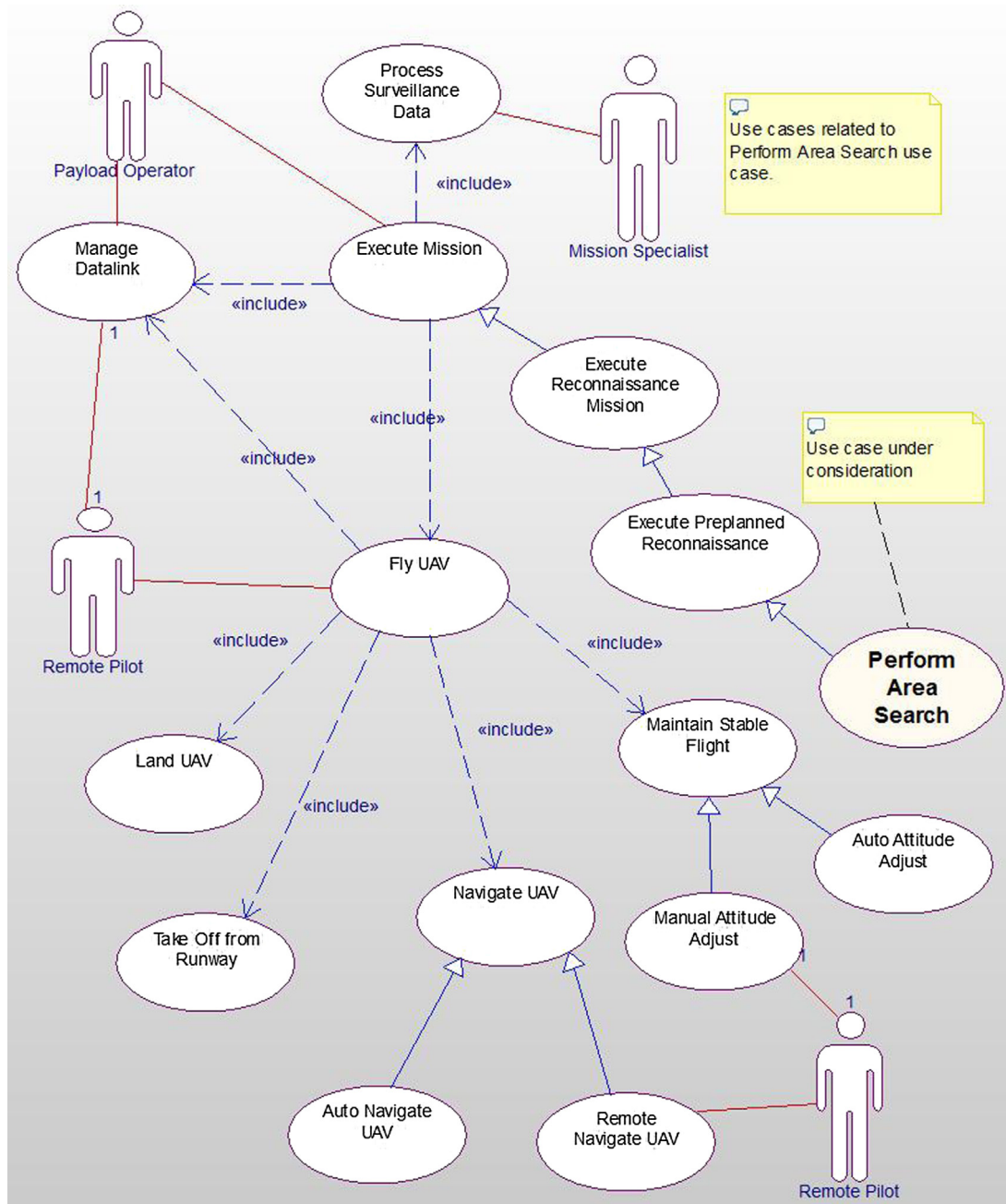


Figure 5.9
Use cases related to Perform Area Search.



Figure 5.10
Perform Area Search use case context.

It is broken up into three subdiagrams, indicated with the interaction references. Each of these refers to a separate diagram focusing on some part of the overall scenario. In fact, this high-level sequence diagram is decomposed along the same lines as the use cases – and this is not accidental. As an aside, I’ve renamed the lifelines from the block names (such as `itsAPAS_RemotePilot`) to something a little bit more readable (such as `Remote Pilot`), but the lifelines nevertheless semantically represent the blocks.

Figure 5.12 shows the elaboration of the aircraft taking off. Note that this is simplified to show the *set* of commands used by the pilot to control the aircraft and the set of the responses and status messages from the aircraft to the pilot, but not the precise *order*. This shows a useful way in which this kind of system, which is continuously controlled by the pilot (with that control mediated by a set of discrete commands and messages), can be

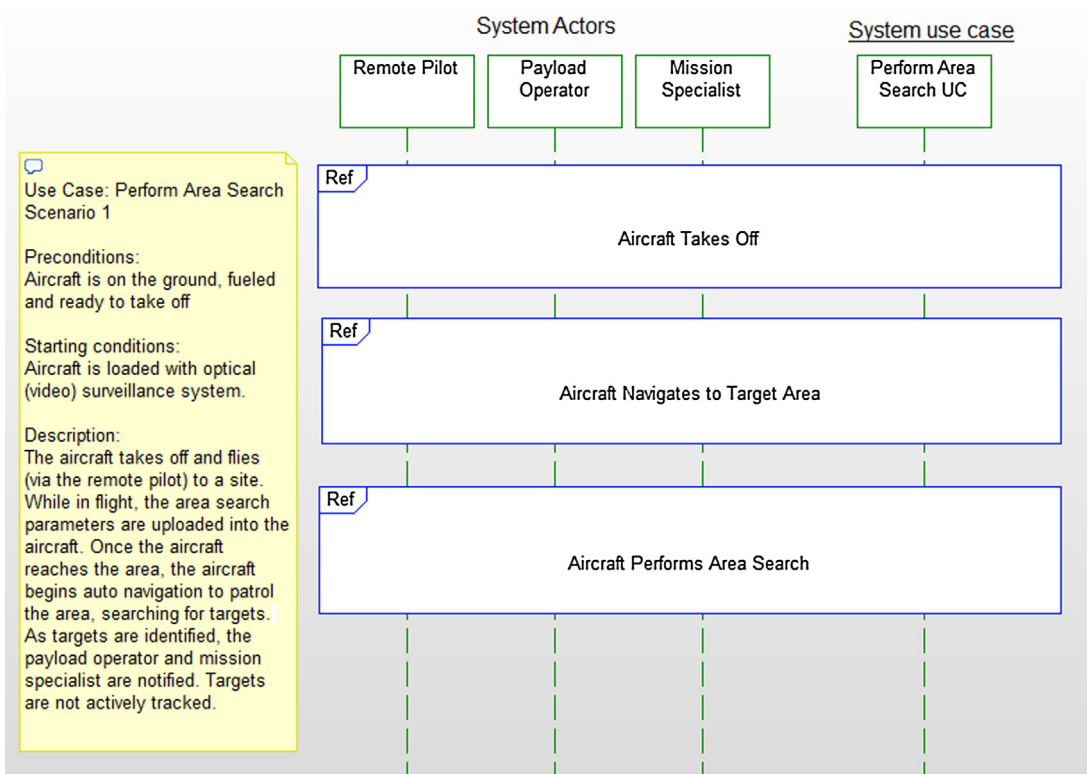


Figure 5.11
Perform Area Search Scenario 1: High-level view.

modeled. Note the use of the parallel operator – in this case, it means that these two sets of interactions are independent and can be interleaved in any fashion. The upper section depicts commands from the pilot and the lower set depicts status messages from the aircraft.

Once the aircraft takes off, it must navigate to the desired location. Of course, the remote pilot is, in this case, flying the aircraft remotely. It would be onerous to capture the hundreds or possibly thousands of individual messages and commands issued as a result of flight adjustments to the remote controls. As in the previous sequence diagram, we have satisfied ourselves with the set of commands that *can* be issued (see [Figure 5.13](#)). We’ve added a constraint to show that the commands from the pilot are order independent – that is, he can adjust any control at any time and in any order, but the commands are sent to the aircraft no more frequently than 10 commands/second.

In the next figure, [Figure 5.14](#), the aircraft is commanded to search a specific area. As a part of this, the aircraft makes a grid of the rectangular area to be searched, and then flies over the area in by grid “columns” search for targets. When a target is found, its location is noted and transmitted to the ground station. The aircraft also maintains a list of identified targets. While doing an area search, the aircraft navigates via its internal autopilot.

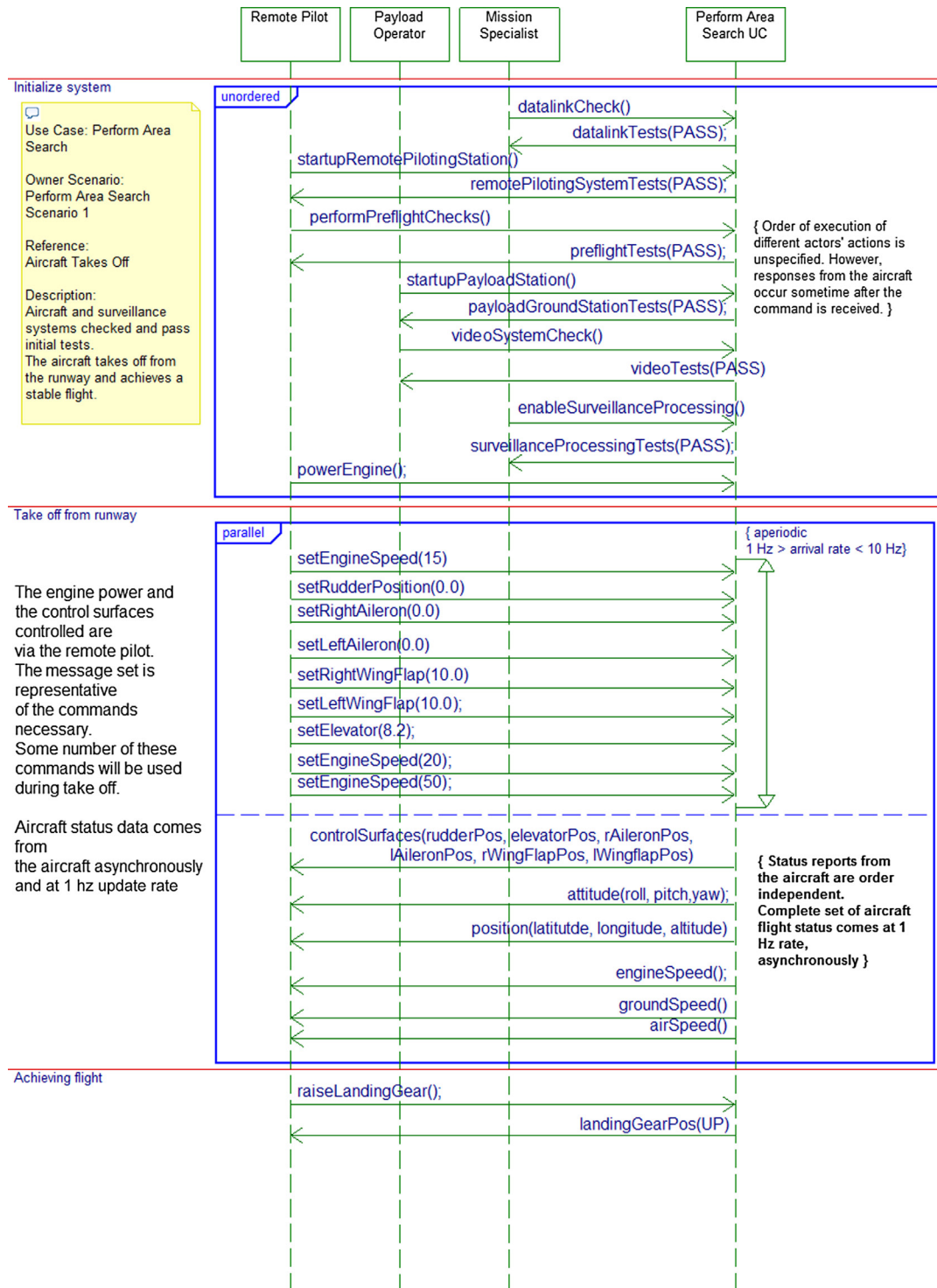
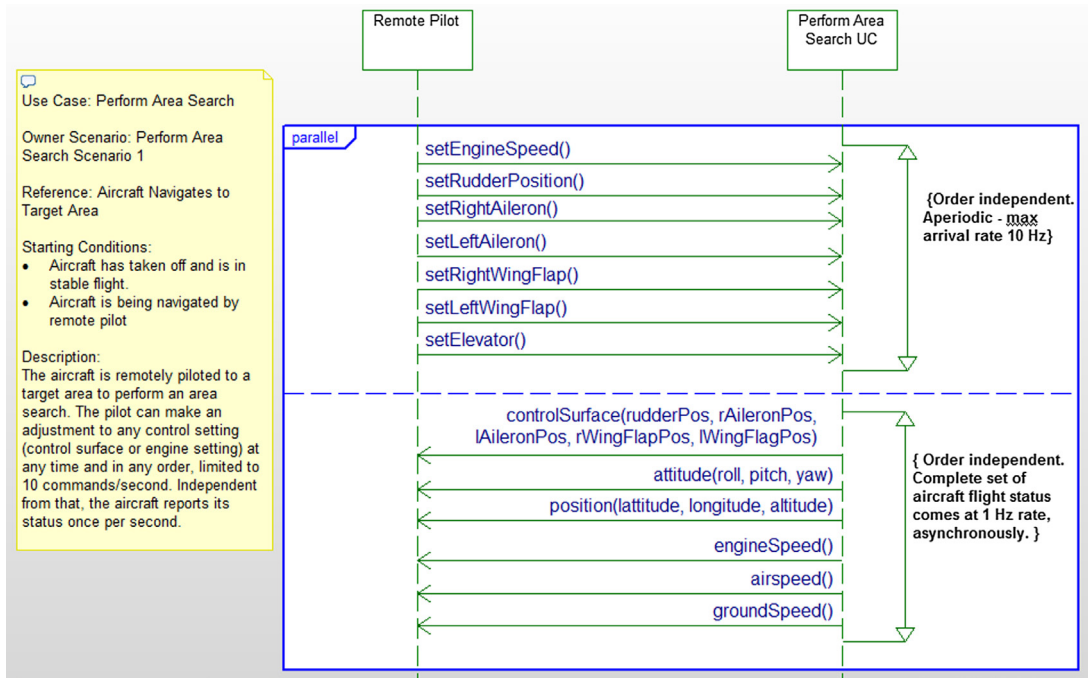


Figure 5.12
Perform Area Search Scenario 1: Aircraft takes off

**Figure 5.13**

Perform Area Search Scenario 1: Remote navigation.

The nested scenario in [Figure 5.14](#) itself has a nested scenario for the setup of the area search. This is shown in [Figure 5.15](#).

For the second part of this exercise, map the operational contracts identified in the Perform Area Search use case into the Coyote system and subsystem architecture. Remember that the Coyote UAV is a large-scale system. We are working with only a single scenario of a single use case. In real system development, there would be an average of a dozen or more scenarios for each use case and this process would be repeated for all scenarios and all use cases. Once several primary scenarios for a use case are completed, additional “sunny day” scenarios may add only a few, if any, operational contracts. The “rainy day” scenarios that model what happens when things go wrong add even more services and behavior. In any case, it is an important exercise to go through to ensure that all required services are provided by some element in the system, that the system’s overall behavior is coherent, and that the system architecture meets all the requirements.

Remember – your task in this exercise is to identify interfaces and their details between the subsystems. Your two primary inputs for this task are the black box system-actor interfaces and the subsystem architecture, identified in the previous exercise.

A Note about ICDs in UML/SysML

An interface consists of a set of metadata defining how a set of at least two elements interact. An interface may be offered (aka “provided”) or required; in the former case, it means that element provides those services for other elements to invoke. In the latter case, it means that the element

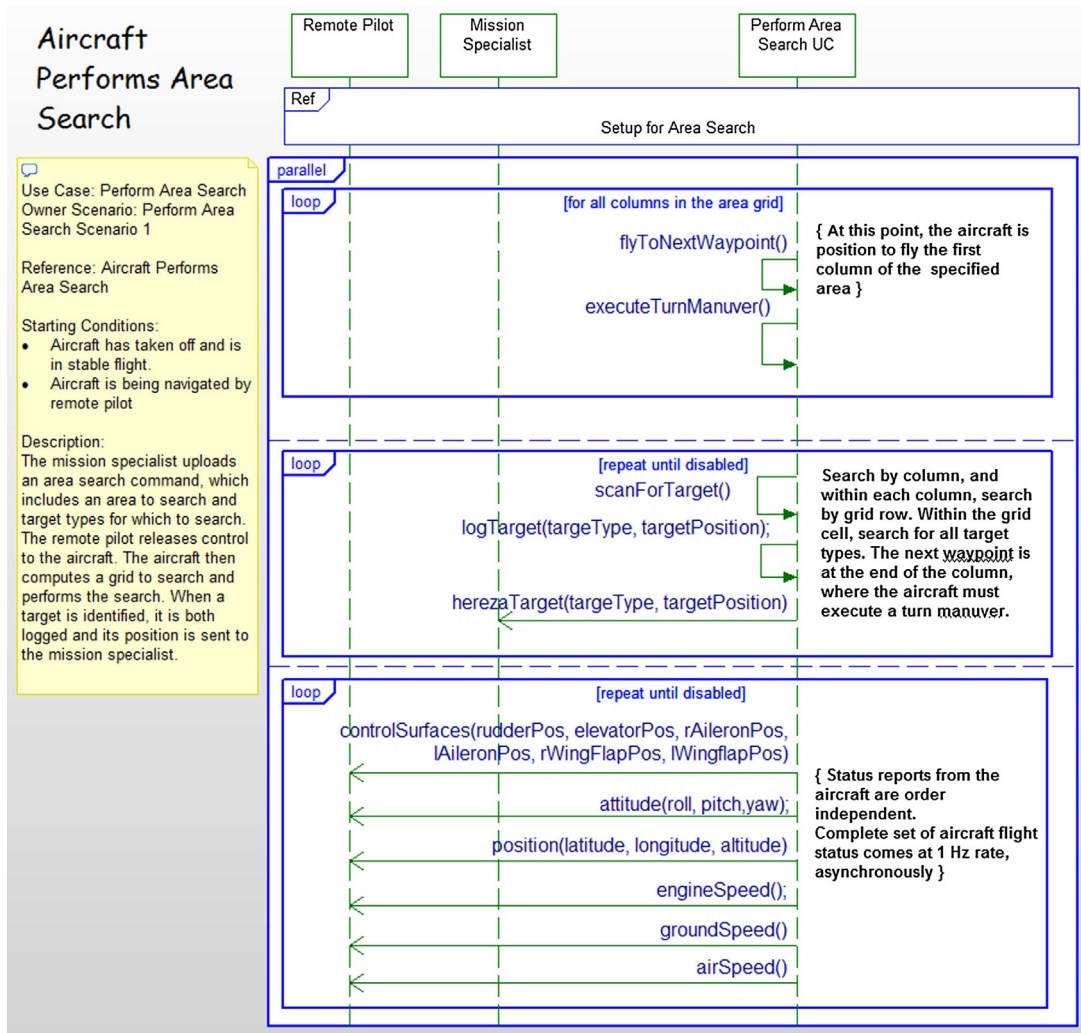


Figure 5.14
 Perform Area Search Scenario 1: Aircraft performs area search.

requires some other element to provide the services of the interface. Interfaces may be used specify the services invocable from classes (blocks) or may be used to define the messages sent to or from a port in a class (block). A service within an interface may be a function call, an asynchronous event, or a synchronous event. A *flow* can also be used, but they are used mostly for continuous values (like pressure or electrical current) and connected via *flow ports* in SysML; such flow ports are bound to resources (such as variables) within the connecting elements.

An interface is more than just a collection of services. It contains

- For each service:
 - Description of the functionality, including error detection and correction
 - Preconditions

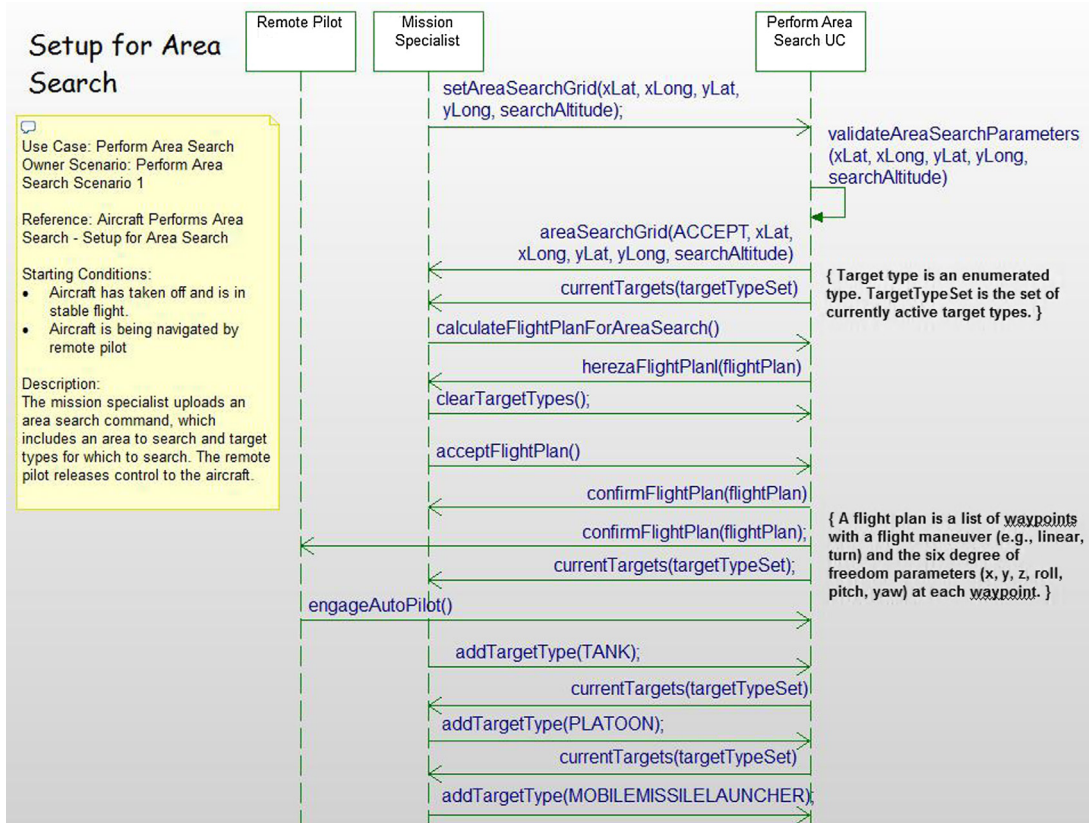


Figure 5.15
Setup for area search.

- Postconditions
- Synchronization mode
- Minimum response time
- Maximum response time
- Return value type and subrange, including error codes
- Interface type (message, memory mapped, interrupt, shared memory, ...)
- For each parameter:
 - Direction (in, out, in/out)
 - Type
 - Allowable subrange
 - Default value
 - Units
 - Physical realization within the service (depends on interface type)

These can be modeled easily in UML or SysML by creating stereotypes that define this information as *tags*. Tags in stereotypes provide a way to extend the metadata about the elements modified by the stereotype. Once this data is in the model, traditional ICDs can be produced as a report containing the relevant information in the model.

Two such stereotypes that I like to use are «ICDService» and «ICDType». Figure 5.16 shows how they are used to define the `iPASpilot` interface. This figure shows the contents of the interface and the specifics of the `remotePilotingSystemTests` event and `SpeedType` type as an example.

I want to point out that at this juncture, the interface and types defined here are *logical* and not *physical*. Logically, the interface might contain a service pair such as

```
getRadarTrack(sectorID: SectorType);
herezaRadarTrack(track: RadarTrack);
```

These logical services (`getRadarTrack` and `herezaRadarTrack`) with logical data types (`SectorType` and `RadarTrack`) specify the need to get a radar track, identifying the sector. The services would be modeled at this point as asynchronous events, but they might actually be implemented as message types sent over a 1553 serial data bus, requiring that the command and response be set up as part of the bus schedule with the services actually being represented as a message with a well-defined format, initiated with a 16-bit TRANSMIT

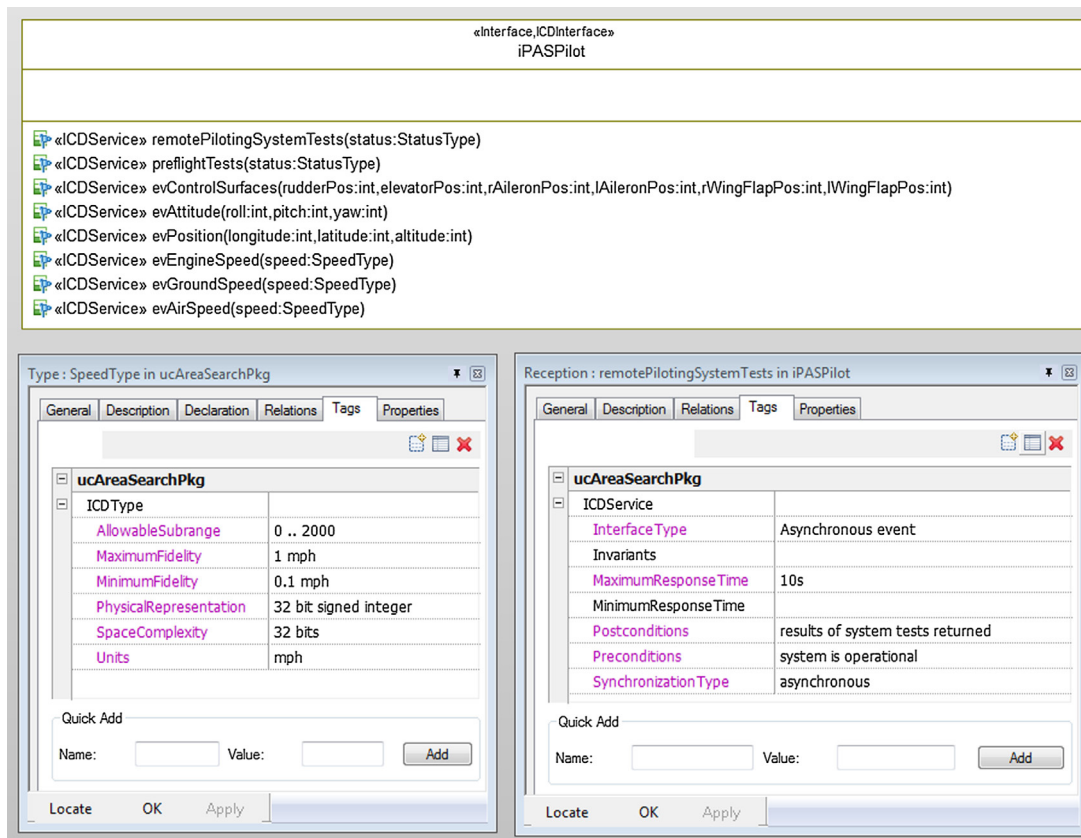


Figure 5.16
ICD stereotypes and tags.

COMMAND word, followed by the command code or data, and terminating with a 16-bit STATUS word. That is a very different implementation from the logical representation captured in the interface above.

Similarly, the `RadarType` here might be a struct identifying the necessary information: the track ID, combat ID, and range to target and their metadata constraints. It doesn't contain the bit-level representation of that information required by the actual message sent over the 1553 bus. Here, the data is also "logical." During the hand-off, this logical interface information will be transformed into a physical interface, but that is after systems engineering per se has finished.

Problem 5.4 Identifying Subsystem Use Cases

Large projects are almost always constructed by multiple teams. The most common way to organize such teams is by allocating each subsystem to a different team. A *subsystem* is a large-scale architectural piece of the physical system. It may be composed of software alone or be a combination of multiple engineering disciplines, typically software, electronics, mechanical, and chemical. These teams may or may not be co-located, but they need two distinct kinds of information to proceed effectively.

First, they need to know the requirements for their particular subsystem. If I'm on a team constructing a power subsystem for a spacecraft, I need to understand what capabilities are expected of my subsystem by its clients – the system, the system actors, and peer subsystems. We organize the requirements of systems into use cases, and it seems reasonable to do the same for subsystems.

Secondly, we need to know the operational context into which the subsystem must fit – what are the clients of my subsystem's services and what servers are available for my subsystem to use? I need to know the interfaces that my subsystem provides to those clients – including the pre- and postconditions, data elements and types, quality of service constraints, and the allowable set of sequences of those services. I need the same information for interfaces that my subsystem requires from its peer subsystems.

These two kinds of information are provided from two sources – the systems requirements model and the systems architecture. These two kinds of information are clearly codependent. A different system architectural structuring will result in a different set of use cases for the subsystems.

So the question is, "How is the identification of the subsystem use cases actually done?" As mentioned above, it requires two sources of information – the system use case model and the systems architecture. The job then can be done in two ways. First, it can be done "top-down" by *decomposing* the system use cases into a set of subsystem use cases that are then wholly met by individual subsystems. This is semantically accomplished with the «include» dependency. The other approach is "bottom-up" in which the detailed requirements (or the use case model events and messages) are allocated to subsystems and then use cases are used to organize the set of allocated requirements. Both approaches can be effective, but in this book we'll follow the top-down approach.

The basic algorithm for the generation of the subsystem use cases is

- For each system use case:
 - Identify clusters of requirements that are met by a single subsystem.
 - Create a subsystem-level use case to organize that cluster.
 - Add the «include» dependency to link the subsystem use case to its parent system-level use case.
 - Allocate the use case to the subsystem.
 - Move the allocated subsystem use case to the requirements area for its specific subsystem.

At the end of this process, each subsystem has a set of use cases that capture its requirements. Then the system architectural model is updated with the interfaces to support the required interactions.

Every operational and operational quality of service requirement must be represented in at least one subsystem use case; some requirements will be represented in all or most of the subsystems. Other requirements must be decomposed into derived requirements, each of which maps into a single use case on a single subsystem. The idea is that the subsystem requirements model represents a complete set of requirements for that particular subsystem. The subsystem team can then go construct their system more-or-less independently with assurance that their subsystem will fit into the system architecture and function appropriately. Traceability is achieved because the links between the system and subsystem use cases are traceable dependencies. Explicit traceability links are commonly added via requirements traceability tools, such as DOORS, as well.

This effort results in new use case diagrams focused at a more detailed level of abstraction. In general, you want to create a new use case diagram for each decomposed system-level use case. In addition, you want to create one or more new use case diagrams for each subsystem. The missions of these two kinds of use case diagrams are different. The former shows the relationships of the parent system use cases to their decomposed subsystem use cases. [Figure 5.17](#) is an example of such a diagram. These diagrams will be kept in the system requirements area of the model. The latter shows the set of use cases for the subsystem, and will be located in the subsystem area of the model. The subsystem level use cases themselves will be specified in their associated subsystem package in the model. Eventually, the subsystem area of the model will be passed off to the subsystem teams and they will elaborate this package in their own design models. This hand-off will be discussed at the end of this chapter.

For the first part of this exercise, use the subsystem architecture for the Roadrunner Traffic Light Control system. Focus on the Configure System and Detect Vehicle use cases, which we specified in some detail earlier. [Figure 5.18](#) shows these use cases in the use case model.

The Roadrunner architecture is shown in [Figure 5.19](#). The functionality (and requirements) represented by the Configure System use case must be mapped into a set of use cases met by the set of subsystems. This system-level use case may not impose any requirements on some subsystems and may result in multiple use cases on other subsystems. It is exceedingly rare for a system-level use case to map entirely onto a single subsystem, however.

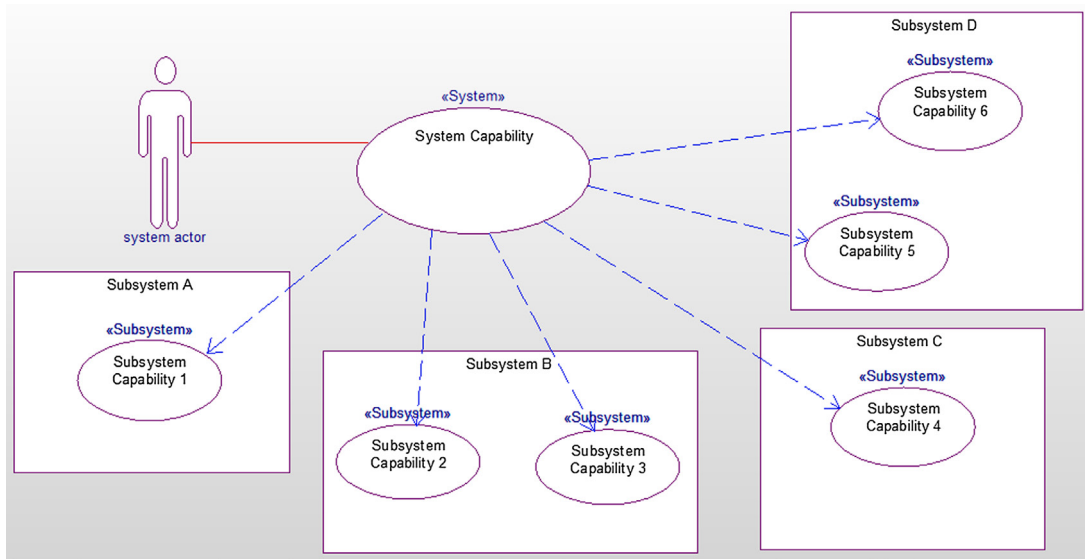


Figure 5.17
Subsystem use cases.

To complete this portion of the exercise, create new use case diagrams for the Configure System and the Detect Vehicle use cases; in these new diagrams, decompose the system use case into a set of subsystem-level use cases. Once this is complete for each subsystem-level use case, create a use case diagram for the `IntersectionController` subsystem with all of its identified use cases.

The second part of this exercise is to do the same thing for two of the use cases for the CUAV: the Manual Attitude Adjust and Perform Area Search use cases. These two use cases are highlighted in [Figure 5.20](#) and [5.21](#).

The first of these is the simpler of the two. The capability described by the Manual Attitude Adjust use case means that a remote pilot in the ground station can control the attitude (roll, pitch, and yaw) of the aircraft. To do this, the pilot will need to monitor that information and have it displayed in some meaningful way and then be able to control these aspects of the aircraft via the communications link. This will have impacts on many subsystems, both in the ground station and in the aircraft. The overall system architecture is shown in [Figure 5.22](#). The two primary systems are shown in the next two figures. The aircraft itself is detailed in [Figure 5.23](#) – this shows the air vehicle subsystems and their interconnections. The ground station is shown in [Figure 5.24](#) – this figure is less complex, but shows that there is a set of manned control stations, each of which contains a UAV control station and a UAV monitoring station.

The first part of the problem involving identifying subsystem use cases is to map the Manual Attitude Adjust use case to the set of subsystems identified here. The second part of the problem is to map the Perform Area Search use case to the same subsystem architecture.

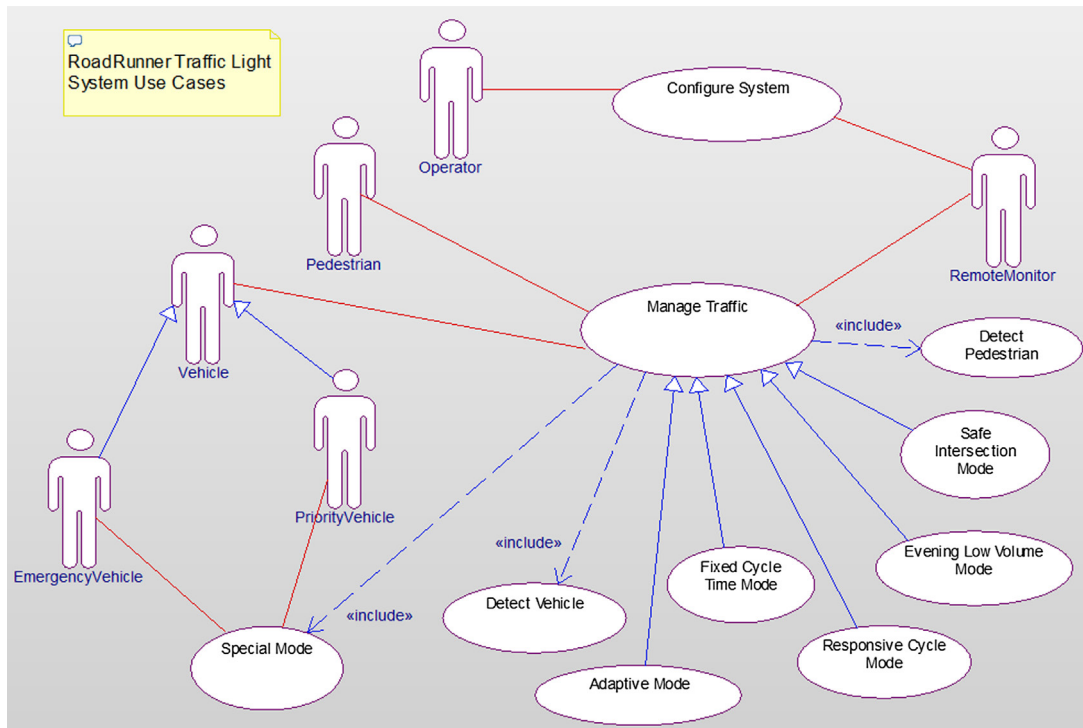


Figure 5.18
Roadrunner use cases.

5.2 The Hand-off from Systems to Downstream Engineering

Thus far, this chapter has focused on the specification of the architecture of the system. This includes the identification of subsystem themselves, their responsibilities (as specified by their subsystem-level use cases), and the interfaces between them. Following systems engineering specification is the design and implementation of the system, a set of workflows we refer to as the hand-off to *downstream engineering*. This involves the evolution from logical interfaces to physical interfaces, the allocation of subsystem responsibilities to different engineering disciplines (such as mechanical, electronic, and software engineering), and the identification of the cross-disciplinary interfaces. The goal of the hand-off to downstream engineering is to allocate the design and implementation of the system specification to the different subsystem teams with enough information and direction so that their work can proceed with an appropriate expectation that it will fit together later.

If we're going to all this trouble to really understand and specify the requirements and architecture from a systems perspective, it makes little sense to lose precision by handing off text-based specifications to downstream engineering. Figure 2.10 shows the workflow for the hand-off to downstream engineering.

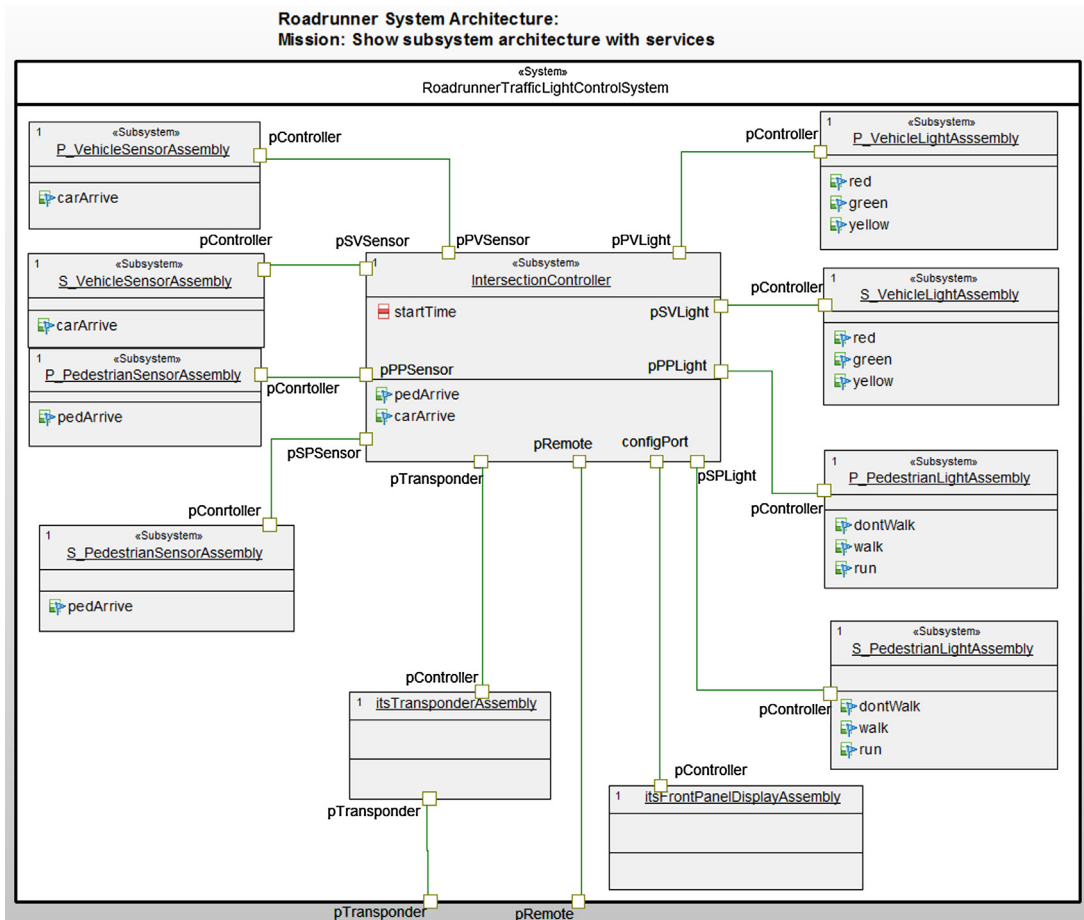


Figure 5.19
Roadrunner system architecture.

Problem 5.5 Creating the Shared Model

At the gross level, creating the shared model is easy. It is generally a separate model with three high-level packages: `CommonTypesPkg`, `InterfacesPkg`, and `DomainsPkg`. The latter package will be used later to share software types used in multiple subsystem designs and is elaborated during the software design and implementation phases (and so may be ignored here). The first two packages are to hold specification data that applies to more than one subsystem, that is, common data types passed between subsystems and the physical interfaces that carry data of those types. The difficulty arises in that to this point the interface specifications have been logical and not physical, but we want to specify these interfaces and types to support integration of the actual subsystems. This requires interfaces and types that are physical and not logical.

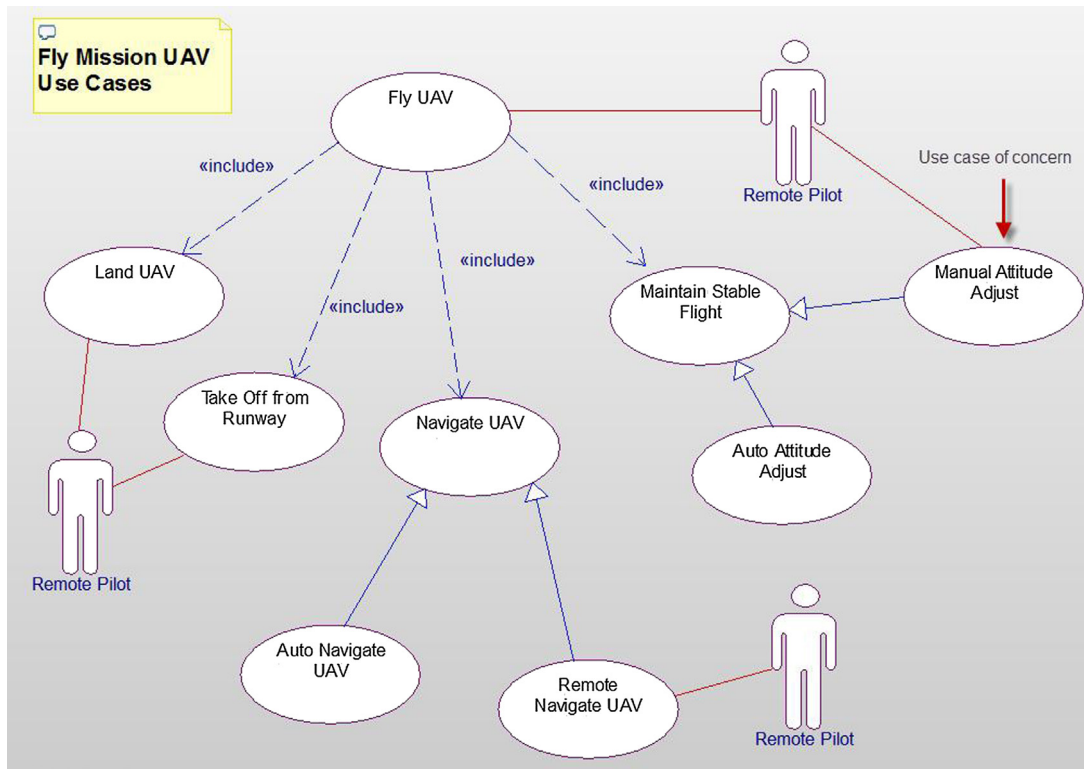


Figure 5.20
Manual Attitude Adjust use case.

To understand the problem, imagine a portion of a sequence diagram that refers to the `TargetingSubsystem` requesting a radar track from the `RadarSubsystem`. In the systems engineering model, this would most likely be shown as an event pair, and each event would carry data, as shown in [Figure 5.25](#). In the actual implementation, it would almost certainly *not* be implemented in this way, even though it would achieve the same purpose. The actual implementation might be over a 1553 avionics bus that connects to both subsystems. The bus message would have a specific data format with bits allocated to the message header, message contents, cyclic redundancy check (CRC), and other important message fields. The data carried by the logical event would have to be represented not as some abstract data type but as a physical type with endianness and a specified bit organization. A corresponding physical interface implementation is shown in [Figure 5.26](#). Of course, the internal messages on the `RadarSubsystem` are not part of the interface but help to explain the usage of the messages that are.

It is crucial that the physical interface be specified and shared with the subsystem teams using the interface so that they build the correct system. Storing the interface in a shared location means that all teams using the interface must rely on that definition and not make up one of their own. Should a defect in the interface be discovered during development, the relevant parties involved in the use of the interface should be represented during the negotiation of the correction.

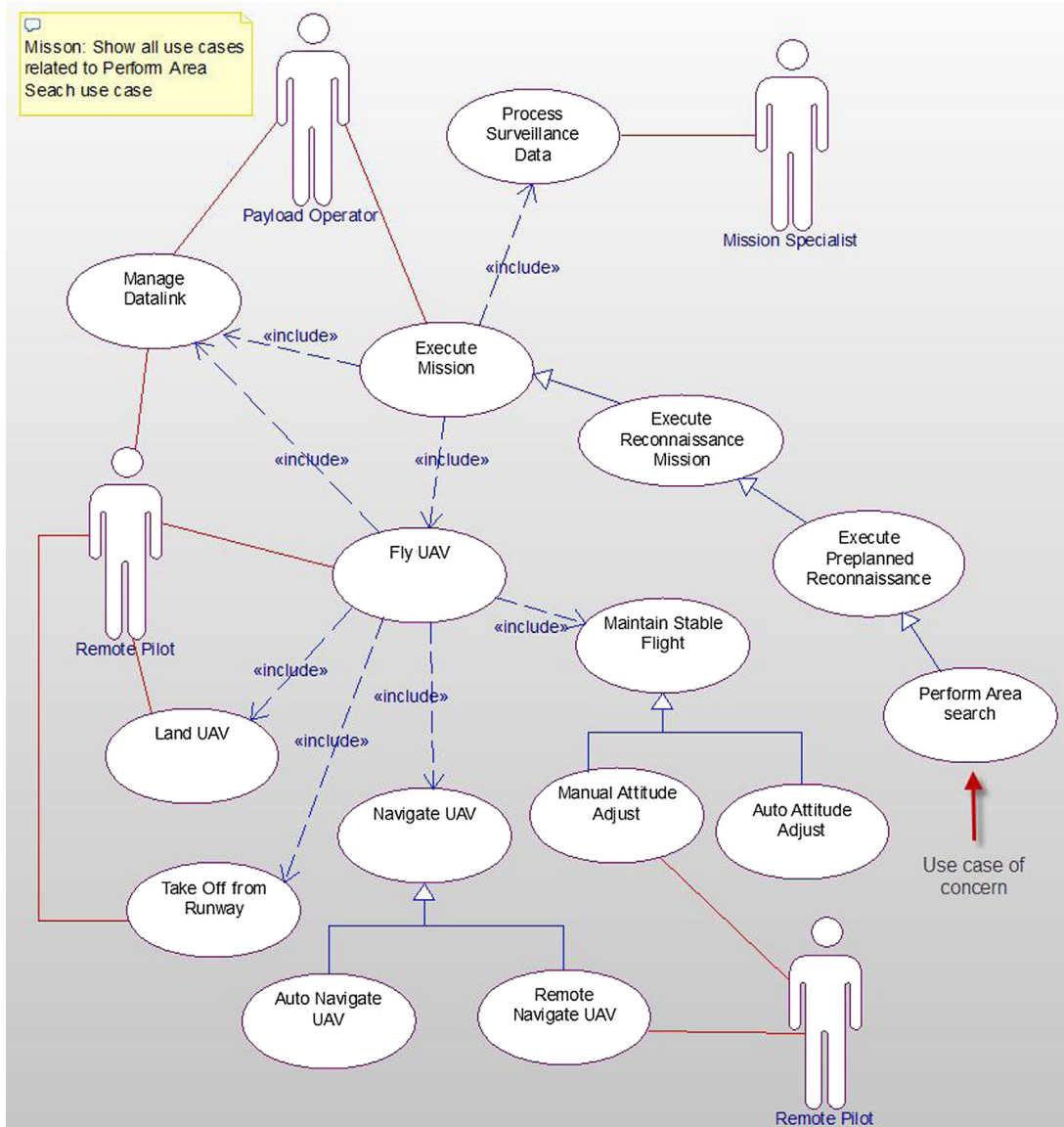


Figure 5.21
Perform Area Search use case.

For bigger systems, each of the primary packages will require an internal suborganization – a single package containing many dozens of interfaces and types is hard to use. There are many ways these packages may be internally organized. My preference is usually as follows:

- For interfaces provided by a single subsystem, create a nested package in the `InterfacesPkg` corresponding to that subsystem. Every other subsystem that wishes to require that interface can find it there.

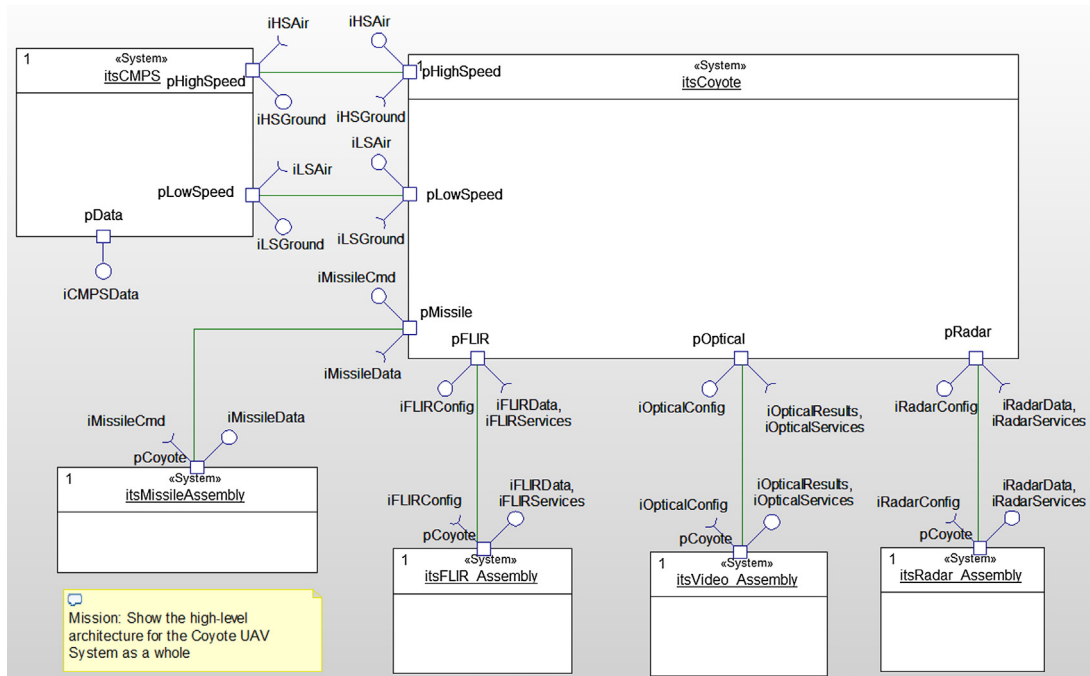


Figure 5.22
CUAV system architecture.

- For interfaces provided by multiple subsystems (if any), create a `CommonInterfacesPkg` and place them there.

For larger systems, I tend to place the common types in the interfaces packages in the same way. If a type is used only within a single interface, then I place it in the same package as that interface, otherwise it goes into the generic `CommonInterfacesPkg`. As I mentioned, there are many good ways to organize this information and any method that allows all clients of an interface or type to easily find and manipulate it will probably be fine.

For this problem, create a package within the Roadrunner model to hold the interface information outside of the systems engineering package. Then take an interface identified between a pair of subsystems from each of our sample projects, decide on a design approach, and clarify the physical interface between them.

For the UAV system, create a separate shared model and structure that model to hold the shared hand-off information.

Problem 5.6 Initiating the Subsystem Model

The last problem in this chapter is to perform the right-side workflow for the hand-off, shown in Figure 2.10. The tasks are

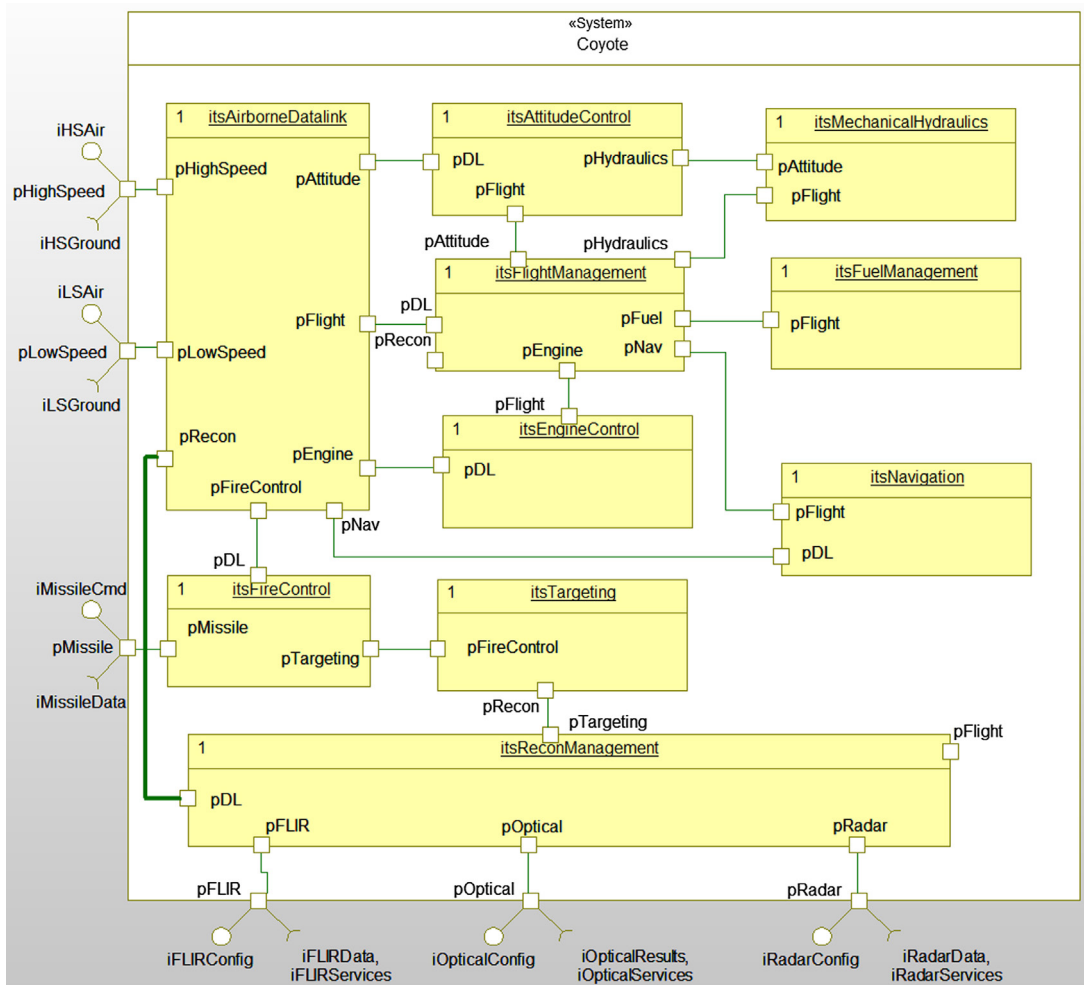


Figure 5.23
CUAV aircraft architecture.

- Create the subsystem model
- Define interdisciplinary interfaces
- Allocate requirements to engineering discipline

The first part of this workflow creates a separate model (or, in small systems, package) for each subsystem, each subdivided into internal nested packages. Larger systems will use separated models for each subsystem. In this case, you must import the subsystem specification from the systems engineering model for the subsystem in question. For small models, it is more appropriate to merely reference the corresponding specification package from within each subsystem package.

In terms of organizing the subsystem model (or package), I recommend a subsystem organization that has packages for

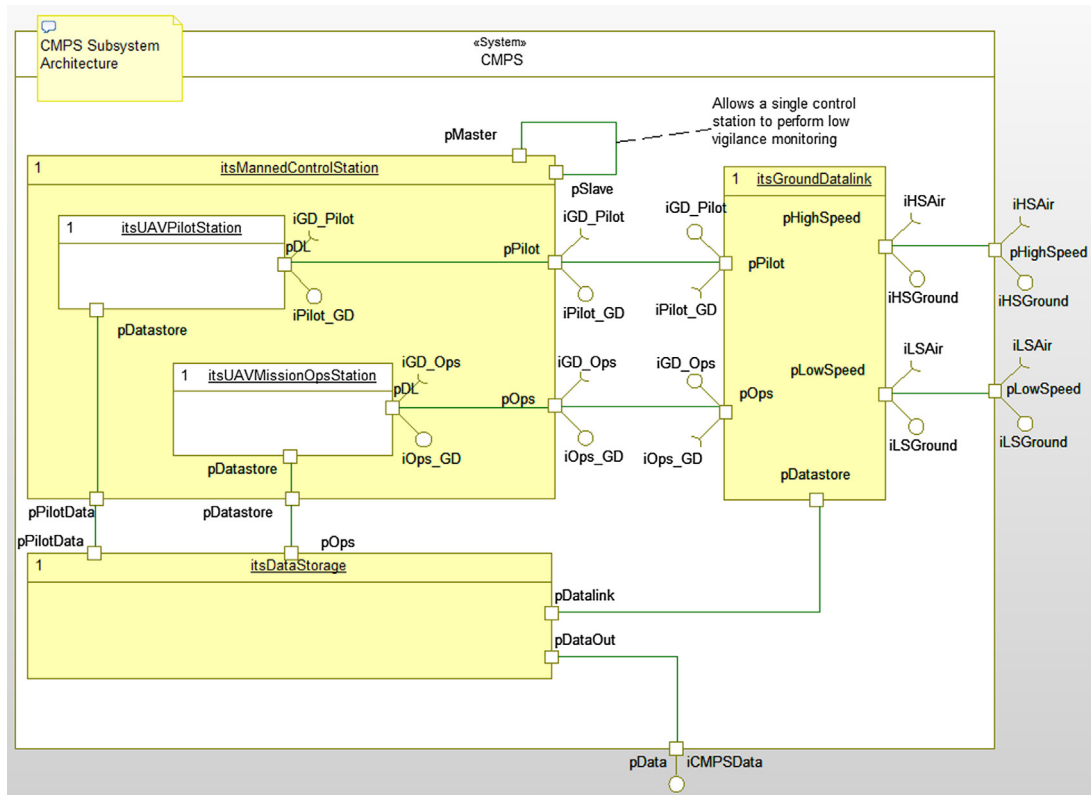


Figure 5.24
CUAV ground station architecture.

- Subsystem requirements and use cases
- Subsystem functional analysis (if required, based on complexity)
- Deployment architecture and interfaces
- Software development
 - Software architecture
 - Software design

These nested packages will most likely be further subdivided as work progresses.

The latter two tasks create what is called the *deployment architecture* for the subsystem. This identifies the presence and roles that different engineering disciplines (e.g., mechanical, electronic, and software) will play in the subsystem, allocate requirements to those disciplines, and define interfaces among the disciplines to support the mission of the subsystem. It is expected that software development will continue within the model but that other disciplines will use their portion of the model solely as specifications and perform their engineering work with other means and tools.

The deployment model created to support the subsystem development will not be greatly detailed – in fact, it is common to just have the collective work products of each represented as

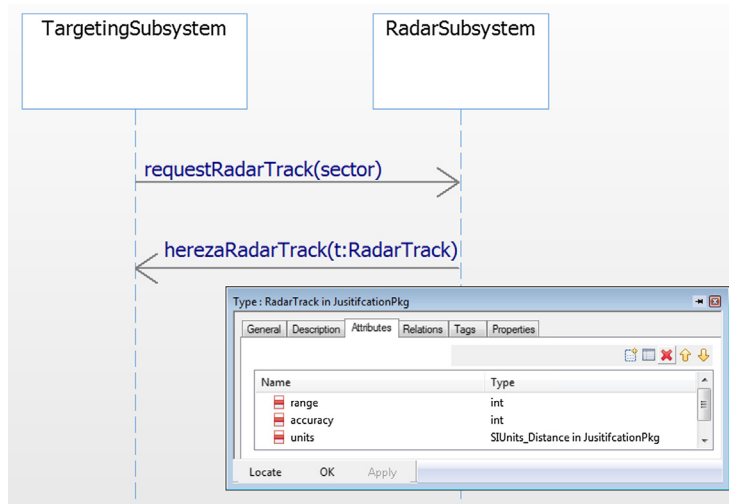


Figure 5.25
Logical interface example.

a single block, although a more detailed deployment architecture identifying major elements within the discipline is also common. The important aspects here are to specify the requirements allocated to each discipline and how the elements of that discipline will interface with the others, particularly how the software will interface to the electronics.

UML does provide a diagram called a deployment diagram, but it is insufficiently expressive to be used for embedded systems. In fact, it was so underpowered that when we created the SysML profile, it wasn't even included. I recommend that you create a class (or block, if you prefer) diagram with stereotyped classes representing the disciplines or their major elements to show the deployment architecture.

For example, [Figure 5.27](#) shows a typical deployment diagram for a patient ventilator. Various elements from different disciplines are shown but little detail is shown, meaning that the necessary detail must be captured elsewhere.

An alternative is to use a class (or block) diagram to represent the same information. [Figure 5.28](#) shows the high-level view. This view simply shows classes (blocks) representing the different disciplines (identified by their suffixes – SW for software, EE for electronics, and ME for mechanics) with the interfaces. In this view, discrete interfaces are shown using so-called “standard” ports, typed by standard interfaces, while continuous values are depicted using flow ports. In this example, all the SW-EE interfaces are discrete and all the EE-ME interfaces are continuous. Also notice that port multiplicity is used to show the fact that there are multiple valves (one per gas supply) in the gas manifold as well as multiple knobs and buttons on the front panel display.

[Figure 5.29](#) shows the same deployment while identifying the large scale pieces of the included disciplines.

Allocations of requirements may be done either by moving them to a separate package for each discipline or by adding dependencies between each requirement and the discipline-specific

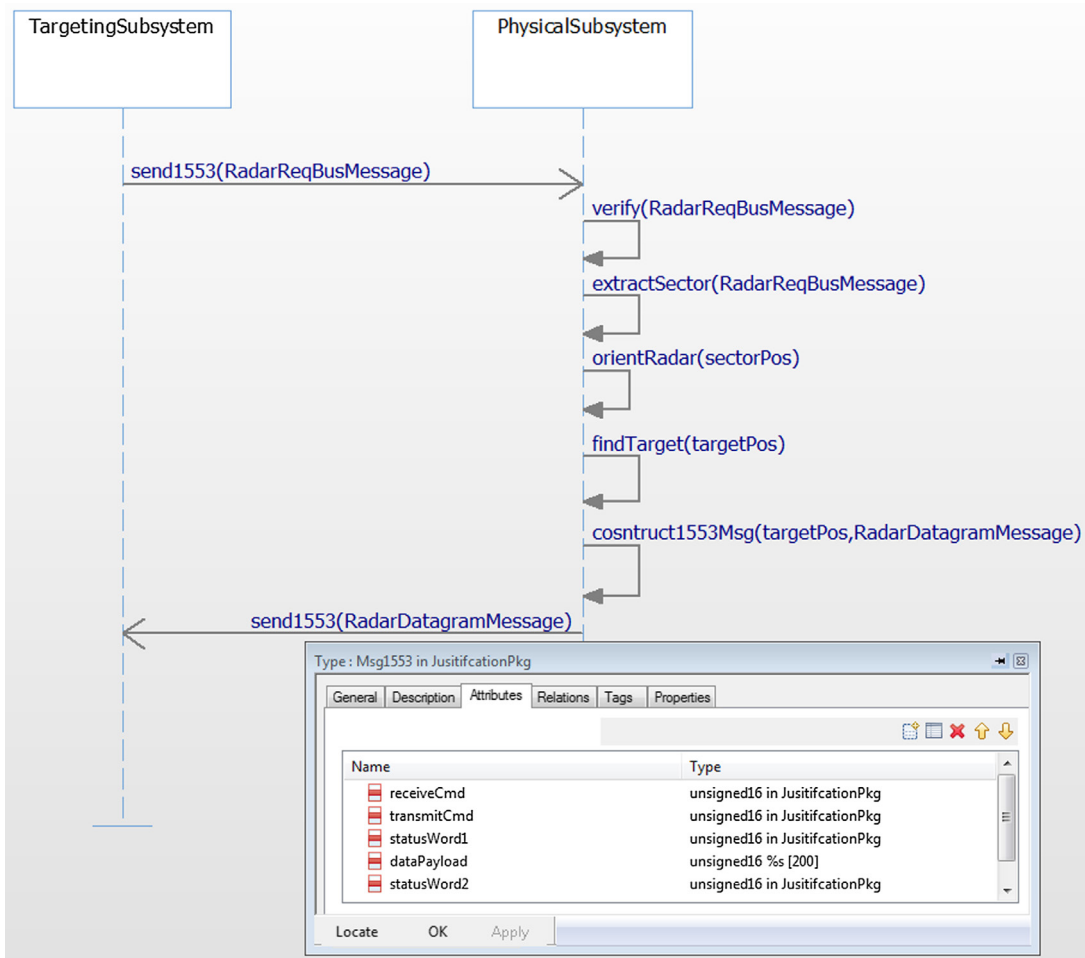


Figure 5.26
Physical interface example.

element that will realize it. In SysML, this dependency is stereotyped «satisfy» while in UML «trace» is more common.

In many cases, a requirement may be directly allocated to a single discipline. However, in many other cases, the requirement must be decomposed into *derived requirements*, which may then be allocated. For example, consider a requirement such as

The volume of each breath delivered by the ventilator shall be settable in 1 ml increments from 100 ml to 1200ml, delivered with an accuracy of +/- 0.5 ml.

This requirement decomposes into requirements for multiple disciplines. For example

The volume of each breath delivered by the ventilator shall be settable in 1 ml increments from 100 ml to 1200ml, delivered with an accuracy of +/- 0.5 ml.

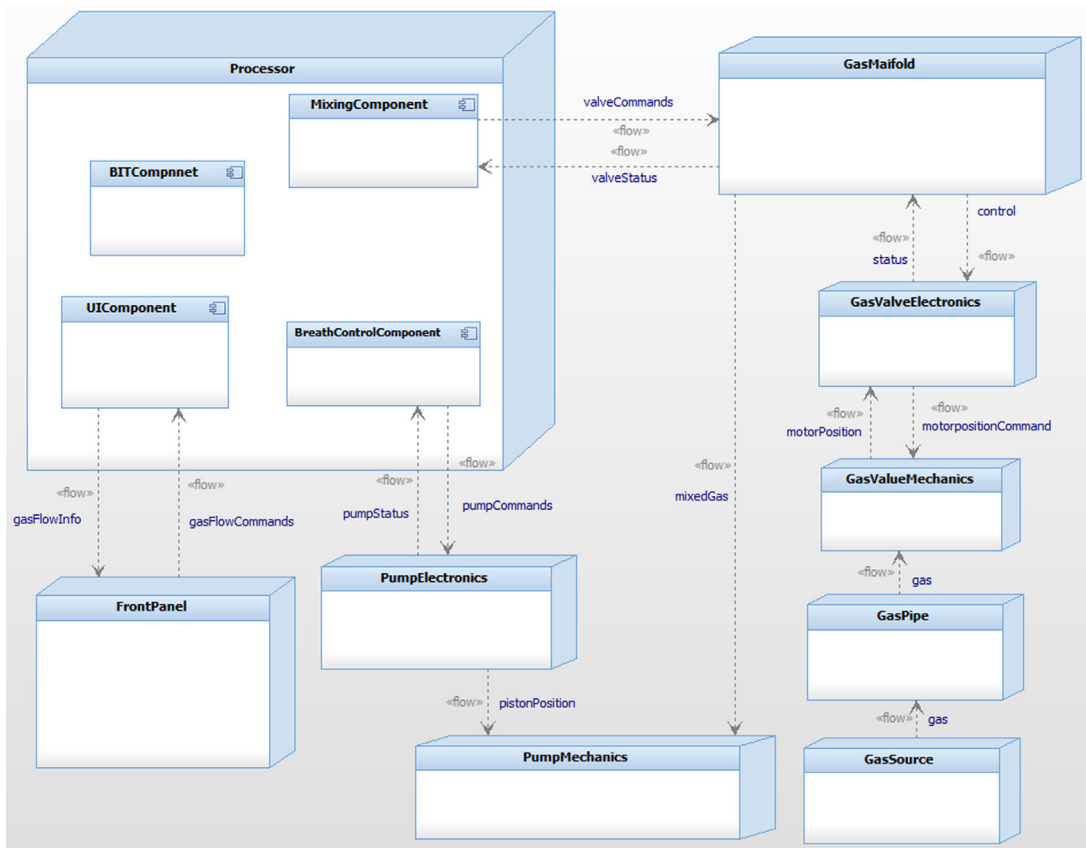


Figure 5.27
Deployment diagram for a patient ventilator.

- (Software) SW shall command the breath control pump in a range of 100 to 1200 ml, in increments of 1 ml.
- (Software) SW shall command the pump to follow the standard volume curve updating the pump speed at least every 100ms.
- (Electronics) The electronic pump control shall command the pump piston to a new commanded position within 1ms.
- (Mechanical) The gas manifold shall deliver mixed gas continuously to the pump.
- (Mechanical) The pump piston shall complete its response to an electronic signal to change position within 10ms with a positional accuracy of at least 5%.

These requirements may then be directly allocated to their respective disciplines.

The interfaces among the engineering disciplines are characterized with appropriate metadata describing the cross-disciplinary interface. The flow ports are typed by the electrical or mechanical properties of the interface. The discrete interfaces similarly characterize the properties of the interface. For SW-EE interfaces, I add tags to the interface for each service within the interface, identifying the kind of connection (interrupt-mapped, memory-mapped, or port-mapped), the address, the size of the connection, and the bit allocation. Figure 5.30 shows

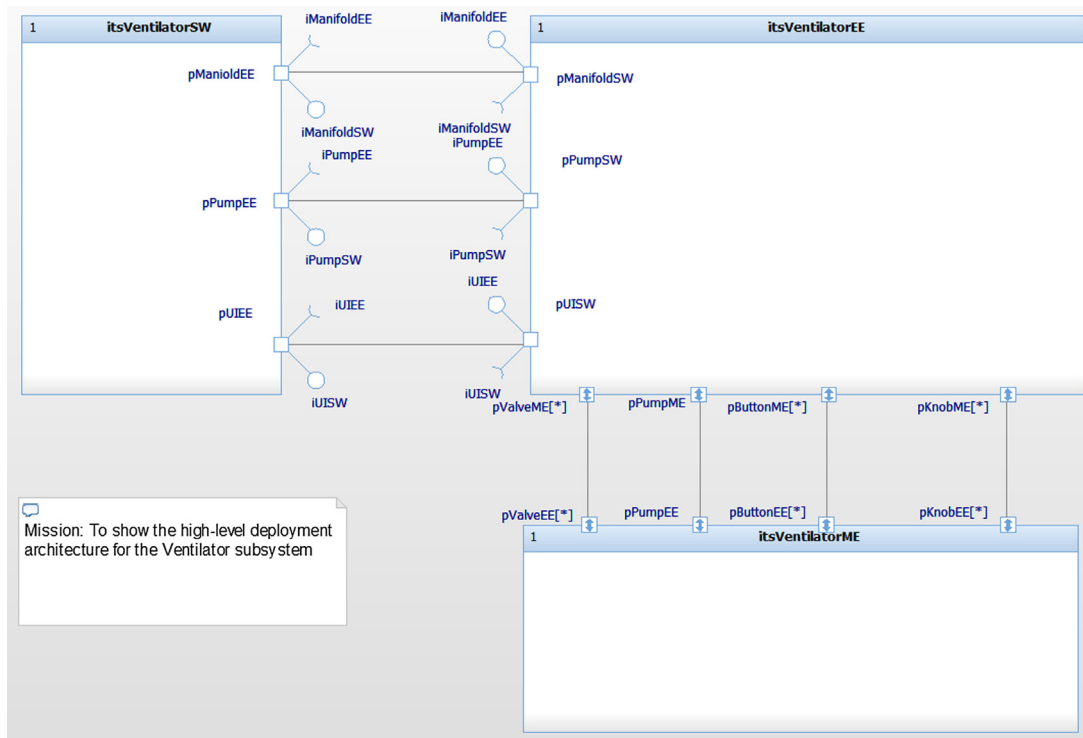


Figure 5.28
Ventilator high-level deployment architecture.

what this might look like for the `iManifoldEE` interface, which has a single operation, `setValvePosition(valveNum, position)`. This metadata ensures that the software and electronic engineers have enough information to design their portions of the system properly.

For this problem, do the following:

- If you have not yet done so, create a subsystem package for each subsystem in the Roadrunner model and a separate model for at least one subsystem in the UAV vehicle. Each should have a nested package organization as described in this chapter.
- For one subsystem each in the Roadrunner and UAV models, create a deployment architecture and allocate requirements to the engineering disciplines.

5.3 Looking Ahead

In this chapter we discussed exercises to create the systems architecture, including the specification of the subsystems and their requirements. We focused on the identification of the subsystems, the allocation of requirements to the subsystems, the hand-off to the subsystem

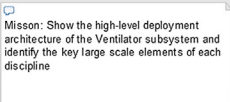


Figure 5.29
Ventilator deployment architecture.

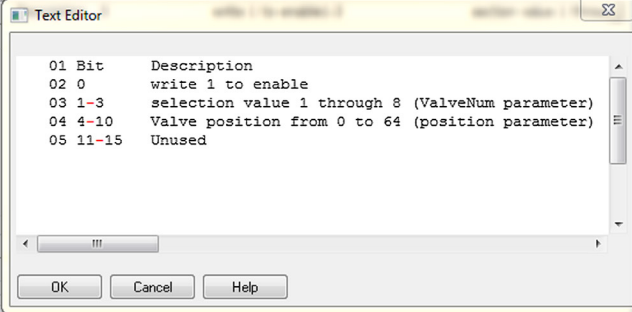


Figure 5.30
Metadata for SW-EE interface.

teams, the creation of the subsystem deployment architecture, and the allocation of requirements to the engineering disciplines.

The next chapter will focus on the architectural aspects of dependability, including safety, reliability, and security. This involves some “systems” activities for analysis, including the allocation to engineering disciplines. Aspects of architecture specific to embedded software will be discussed in Chapters 8 and 9.

Dependability Architecture

What You Will Learn

- The meaning of “dependability”
- Model-based safety analysis with fault tree analysis (FTA)
- Reliability with FTAs and failure mode and effects analysis (FMEA)
- Security and information assurance
- Architectural design patterns related to dependability

6.1 Overview

Smart systems are automating many processes that used to be reserved for the monitoring and intervention of highly trained personnel. This adds tremendous benefits in terms of cost and capability, but can we depend upon these systems? Dependability refers to the confidence with which we can entrust our lives to automated systems. Dependability has three primary aspects: safety, reliability, and security.

Reliability is a measure of the “uptime” or “availability” of a system – specifically, it is the probability that a computation will successfully complete before the system fails. It is normally estimated with mean time between failure (MTBF) or a related measure known as *availability*. MTBF is a statistical estimate of the probability of failure, and applies to stochastic failure modes.

Reducing the system downtime increases reliability by improving the MTBF. Redundancy is one design approach that increases availability because if one component fails, another takes its place. Of course, redundancy only improves reliability when the failures of the redundant components are independent.¹ The situation in which a single failure can bring down multiple components is called a *common mode failure*. One example of a common-mode failure is running software for both the primary and secondary processing on the same CPU – should the processor fail, then both components will fail. In reliability analysis, great care must be taken to avoid common mode failures or to provide additional redundancy in the event one element common to all redundant components fails.

¹ Strict independence isn’t required to have a beneficial effect. Weakly correlated failure modes still offer improved tolerance to faults over tightly correlated failure modes.

The reliability of a component does not depend upon what happens after the system fails. That is, regardless of the impact of the failure, the reliability of the system remains the same. Clearly the primary concern relative to the reliability of a system is the availability of its functions to the user.

Safety is very different from reliability, but a great deal of analysis affects both safety and reliability. A safe system is one that does not incur too much risk of loss, either to persons or equipment. A *hazard* is an undesirable event or condition that can occur during system operation. *Risk* is a quantitative measure of how dangerous a system is and is usually specified as

$$Risk = Hazard_{severity} * Hazard_{likelihood}$$

The failure of a jet engine is unlikely, but the consequences can be very severe. Overall, the risk of flying in a plane is tolerable because even though it is unlikely that you would survive a crash from 30,000 feet, such an incident is extremely unlikely. At the other end of the spectrum, there are events that are common, but are of lesser concern. A battery-operated radio has a hazard of electric shock, but the risk is acceptable because even though the likelihood of the hazard manifesting is relatively high, its severity is low.²

Faults come in two flavors. *Errors* are systematic faults introduced in analysis, design, implementation, or deployment. By “systematic,” we mean that the error is *always* present, even though it may not be always be manifest. In contrast, *failures* are random errors that occur when something breaks. Hardware exhibits both errors and failures but software exhibits only errors. The distinction between error and failure is important because different design patterns optimize the system against these concerns differently.

The key to managing both safety and reliability is *redundancy*. Redundancy improves reliability because it allows the system to continue to work in the presence of faults. Simply, the redundant system elements can take over the functionality of faulty ones and continue to provide system functionality. For improving safety, additional elements are needed to monitor the system to ensure that it is operating properly and possibly other elements are needed to either shut down the system in a safe way or take over the required functionality. The goal of redundancy used for safety is different – the concern is not about continuing to provide functionality, but instead to ensure that there is no loss (to either persons or equipment).

The example I like to use to demonstrate the difference is the handgun versus my ancient Plymouth station wagon. The handgun is highly reliable piece of equipment – most of them

² For more on safety, see *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999) by Bruce Powel Douglass or *Safety Critical Computer Systems* by Neil Storey (Addison-Wesley, 1996). For a general overview of the issues of safety in software-intensive systems, see *Safeware: System Safety and Computers* by Nancy Leveson (Addison-Wesley, 1995).

fire when dirty or even under water. It is, however, patently not very safe since even in the absence of a fault, you can (and people do) shoot yourself in the foot. On the other hand, my enormous 1972-vintage station wagon (affectionately referred to as “The Hulk”) is the safest automobile on the planet. It has a fail-safe state³ (“OFF”) and it spends *all of its time* in that state. So while the vehicle is very safe, it is not at all reliable.

As with the other architectural dimensions, safety and reliability are achieved through the application of architectural design patterns.⁴ All design patterns have costs and benefits, and selecting good safety patterns requires balancing the design concerns, such as

- Development cost
- Recurring (manufacturing) cost
- Level of safety needed
- Level of reliability needed
- Coverage of systematic faults (errors)
- Coverage of random faults (failures)
- Complexity
- Resource demand
- Ease of certification against relevant standards

In general, safety and reliability patterns can be categorized into either homogenous or heterogeneous patterns. The former creates exact replicas of the architectural elements to provide redundant processing, and adds glue logic to determine when and under what circumstances the replicas run. The latter patterns use different implementations, designs, or approaches to provide redundant processing. These systems can be further subdivided into *lightweight* or *heavyweight* patterns. Lightweight patterns use fewer resources but may not be able to provide the full functionality or fidelity of the primary system elements. Heavyweight redundancy replicates the full functionality but at a greater cost.

Security is a bit different from reliability or safety but intersects with both. Security of information is called information assurance, but security is a broader issue with embedded devices. Certainly, security in the IT sense – managing the wired and wireless connections to prevent intrusions – is important, but it is not the only concern. In cyberphysical systems, we must concern ourselves with more mundane threats (such as walking off with the device) and we must concern ourselves with the severity of the potential outcomes as security of a nuclear power plant may compromise its safety. The solutions are likely to include a mixture of standard IT approaches and physical system security measures.

We will discuss details of these different aspects in more detail in the upcoming problems.

³ A fail-safe state is a condition (state) of a system known to be always safe, i.e., free from harm.

⁴ For details of safety-related patterns, see my book *Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* (Addison-Wesley, 2002), Chapter 9.

6.2 A (Not-So) Quick Note about Design Patterns⁵

In the Harmony process, analysis is about the essential things – aspects of the system without which the result would be incorrect – and design is about optimization – selecting optimal solutions. Since architecture is, in a very real sense, “high-level design,” it makes sense to talk a little bit about design and design patterns before we go much further.

Figure 6.1 shows the conceptual design workflow in the Harmony process. We generally start with some analysis model that is demonstrably correct, clarify how we want to optimize it, select specific optimizations, apply them, and finally verify the result. It is common, however,

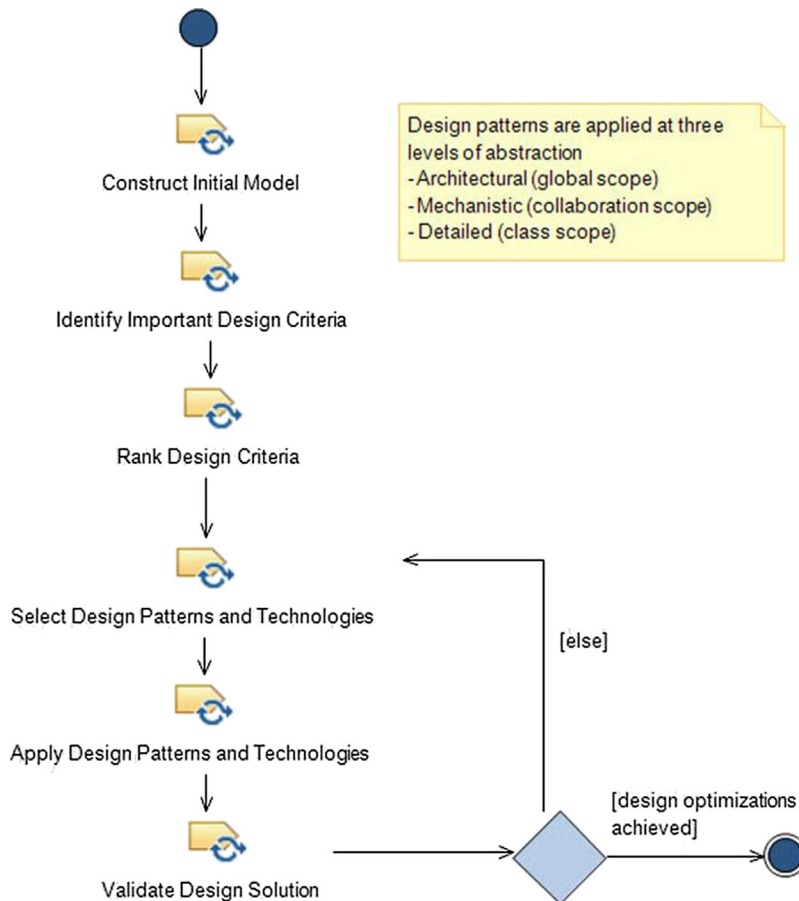


Figure 6.1
Basic Harmony design workflow.

⁵ This section is adapted from Chapter 2 of my book *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit* (Elsevier, 2011).

for some aspects of architecture to be decided early before there is very much model to be demonstrated. Because of the systemic nature of dependability, for example, a number of design decisions that optimize safety, reliability, or security may be made early. Nevertheless, the conceptual workflow provides a good framework for understanding how to apply a design pattern-oriented approach to systems development.

Let us discuss the steps involved in this conceptual workflow.

6.2.1 Construct the Initial Model

This step is crucial but often omitted in the developer's haste to optimize. The step means that before optimization is considered, a working model and code base is created. By "working" I mean not only that it compiles and runs, but also that it is robust and functionally correct. I also mean that it is debugged and unit tested for functional correctness. It may be tested for performance to determine whether optimization is required but that is optional at this point.

6.2.2 Identify Important Design Criteria

This is another commonly omitted step. It is important to remember that whenever some system aspects are optimized, it is almost always true that some other aspects are deoptimized. In order to select appropriate design patterns and technologies to achieve optimization, we need to first understand what the important aspects to optimize are so we can trade them off appropriately. Some common design optimization criteria are:

- Performance
 - Worst case
 - Average case
- Predictability
- Schedulability
- Throughput
 - Average
 - Sustained
 - Burst
- Dependability
 - Reliability
 - With respect to errors
 - With respect to failures
 - Safety
 - Security
- Reusability
- Distributability

- Portability
- Maintainability
- Scalability
- Complexity
- Resource usage, e.g., memory
- Energy consumption
- Recurring cost, i.e., hardware
- Development effort and cost

6.2.3 Rank Design Criteria

The ranking of the criteria is necessary for us to make appropriate trade-offs when we select design patterns. Some patterns will optimize worst-case performance at the expense of using more memory; others will optimize robustness at the expense of average-case performance. Others may optimize energy economy at the expense of responsiveness. The ranking of the design criteria makes their relative importance explicit, so that we can reason about which design solutions are best.

6.2.3 Select Design Patterns and Technologies

There is a wealth of effective design solutions and technologies available. A *trade study* is a comparison of technologic solutions based on their effectiveness at achieving a balanced set of optimization goals. Although we may choose to be less formal in our approach, the Federal Aviation Administration (FAA) recommends a formalized decision analysis matrix.⁶ Basically, this is a spreadsheet that summarizes the “goodness” of a design solution based on the degree to which it matches our weighted design criteria. For example, Table 6.1 shows the kind of analysis that we might do formally in the spreadsheet or informally in our heads.

Table 6.1 Design Trade-off Spreadsheet.

Design Solution	Design Criteria					Total Weighted Score
	Criteria 1 Weight = 7 Score	Criteria 2 Weight = 5 Score	Criteria 3 Weight = 3 Score	Criteria 4 Weight = 2 Score	Criteria 5 Weight = 1.5 Score	
Alternative 1	7	3	6	9	4	106
Alternative 2	4	8	5	3	4	95
Alternative 3	10	2	4	8	8	120
Alternative 4	2	4	9	7	6	84

⁶ http://www.atcvantage.com/docs/FAA_System_Safety_Handbook_2001.pdf

The weight for each criteria is the relative importance of the criteria on a scale of 0 to 10. Each alternative – a combination of design patterns and technologies – is assigned a score (degree to which the criteria is optimized) in the range 0 to 10. Each alternative is then assigned a computed total weighted score equal to the sum of the criteria weights times their relative scores. The alternative with the largest total weighted score provides the best-balanced design solution for the set of design criteria.

6.2.4 Apply Design Patterns and Technologies

This step refers to the application of the selected design alternative. In design pattern terms, this is known as pattern instantiation. As we will see, this is primarily a matter of replacing the formal pattern parameters with the relevant elements from the working analysis model.

6.2.5 Validate Design Solution

The validation step ensures that the solution works and checks two different facets of the solution. First, it ensures that the functionality of the system continues to be correct; that is, adding in the design solution didn't break the running model and code. Secondly, since we selected the design patterns to optimize the system, we must ensure that we've achieved our optimization goals. This means that we will rerun the previous functional and unit tests and also add quality of service tests to measure performance, memory usage, or whatever measures are appropriate to verify our design criteria are met.

6.3 What is a Design Pattern?

A design pattern is “a generalized solution to a commonly occurring problem.” To be a pattern, the problem must recur often enough to be usefully generalizable and the solution must be general enough to be applied in a wide set of application domains. If it only applies to a single application domain, then it is probably an *analysis pattern*.⁷ Analysis patterns define ways to organize problem-specific models and code for a particular application domain.

Remember that analysis is driven by *what* the system must do while design is driven by *how well* the system must achieve its requirements. A design pattern is a way of organizing a design that improves the optimality of a design with respect to one or a small set of design criteria, such as quality of service (QoS).

A design pattern always has a focused purpose – which is to take one or a small set of these QoS properties and optimize them at the expense of the others. Each pattern optimizes some aspect of a system at the cost of deoptimizing some other aspects. These constitute a pattern's pros and cons.

⁷ A coherent set of analysis patterns is also known as a *reference model*.

A good design is one composed of a set of design patterns applied to a piece of a functional system that achieves a balanced optimization of the design criteria and incurs an acceptable cost in doing so. That is, a good design optimizes the important design criteria at a cost that we are willing to pay. Note that a design always encompasses several different design patterns, so each alternative in our trade study must also examine how well the patterns work together in addition to how each works in isolation.

As mentioned previously, patterns may be applied at the different levels of design abstraction. Architectural patterns have systemic scope and apply mostly to only one of the five key views of architecture.⁸ At the next level down in design abstraction, collaboration design patterns apply to individual collaborations, optimizing the same criteria, but in a more narrow focus. Detailed design patterns – often referred to as design idioms – optimize the primitive elements of the software.

6.3.1 Basic Structure of Design Patterns

According to Gamma, et al.,⁹ a pattern has four important aspects:

- Name
 - The name provides a “handle” or means to reference the pattern.
- Purpose
 - The purpose provides the problem context and the QoS aspects the pattern seeks to optimize. The purpose identifies the kinds of problem contexts where the pattern might be particularly appropriate.
- Solution
 - The solution is the pattern itself.
- Consequences
 - The consequences are the set of pros and cons of the use of the pattern.

The pattern *name* brings us two things. First, it allows us to reference the pattern in a clear, unambiguous way, with the details present, but unstated. Secondly, it gives us a more abstract vocabulary to speak about our designs. The statement “The system uses a Homogeneous Channel redundant architecture with Cyclic Executive Concurrency distributed across a set of processors with a Proxy Pattern” has a lot of information about the overall structure of the system because we can discuss the architecture in terms of these patterns.

The *purpose* of the pattern identifies the essential problem contexts required for the pattern to be applicable and what design criteria the pattern is attempting to optimize. This section

⁸ See Figure 5.1 earlier in this book.

⁹ Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading MA: Addison-Wesley, 1995).

specifies under which situations the pattern is appropriate and under which situations it should be avoided.

The *solution*, of course, specifies the structure and behavior of the pattern itself. It identifies the elements of the pattern and their roles in the context of the pattern. This pattern solution is integrated into your design in a process known as *pattern instantiation*.

The *consequences* are important because we always make trade-offs when we select one pattern over another. We must understand the pros and cons of the pattern to apply it effectively. The pros and cons are usually couched in terms of improvement or degradation of some design properties as well as a possible elaboration of problem contexts in which these consequences apply.

6.3.2 Using Design Patterns in Development

You're facing a design problem. How do you find the patterns that can be applied to solve your particular design issues? We recommend a multistep approach as shown in [Figure 6.2](#).

1. Familiarize Yourself with the Patterns

First, before starting your design familiarize yourself with the patterns literature. There are a number of books, papers, and websites devoted to patterns in many application domains. Some of those patterns are given here and others are given in the references. Once you have increased your vocabulary to include patterns likely to be relevant to your application domain, you have more intellectual ammunition to face your design challenges.

2. Identify Design Criteria

In parallel with reading about potential design alternatives, you must identify design criteria. There are many (often conflicting) optimizations you can make to your system. This step identifies the design optimizations that are important for this particular system in its operational and development contexts.

3. Rank Design Criteria

Not all optimizations are equally important. For this reason, the design criteria must be ranked to enable good trade-off decisions to be made.

4. Identify Potential Design Alternatives

This step finds patterns that optimize the design criteria of importance at a cost that seems reasonable. While it is possible to add a single pattern at a time in this way and repeat the workflow, it is also common to consider a set of compatible patterns together as a single design alternative.

5. Evaluate the Design Alternatives

Using the trade study approach suggested earlier, this step compares the design alternatives (consisting of one or more design patterns) against each other on the basis of their effectiveness against the design criteria, weighted with the relative importance of each.

6. Select and Apply Design Alternatives

Once a design alternative is selected, the patterns are usually added to the collaboration one at a time, including verification of the efficacy of the solution.

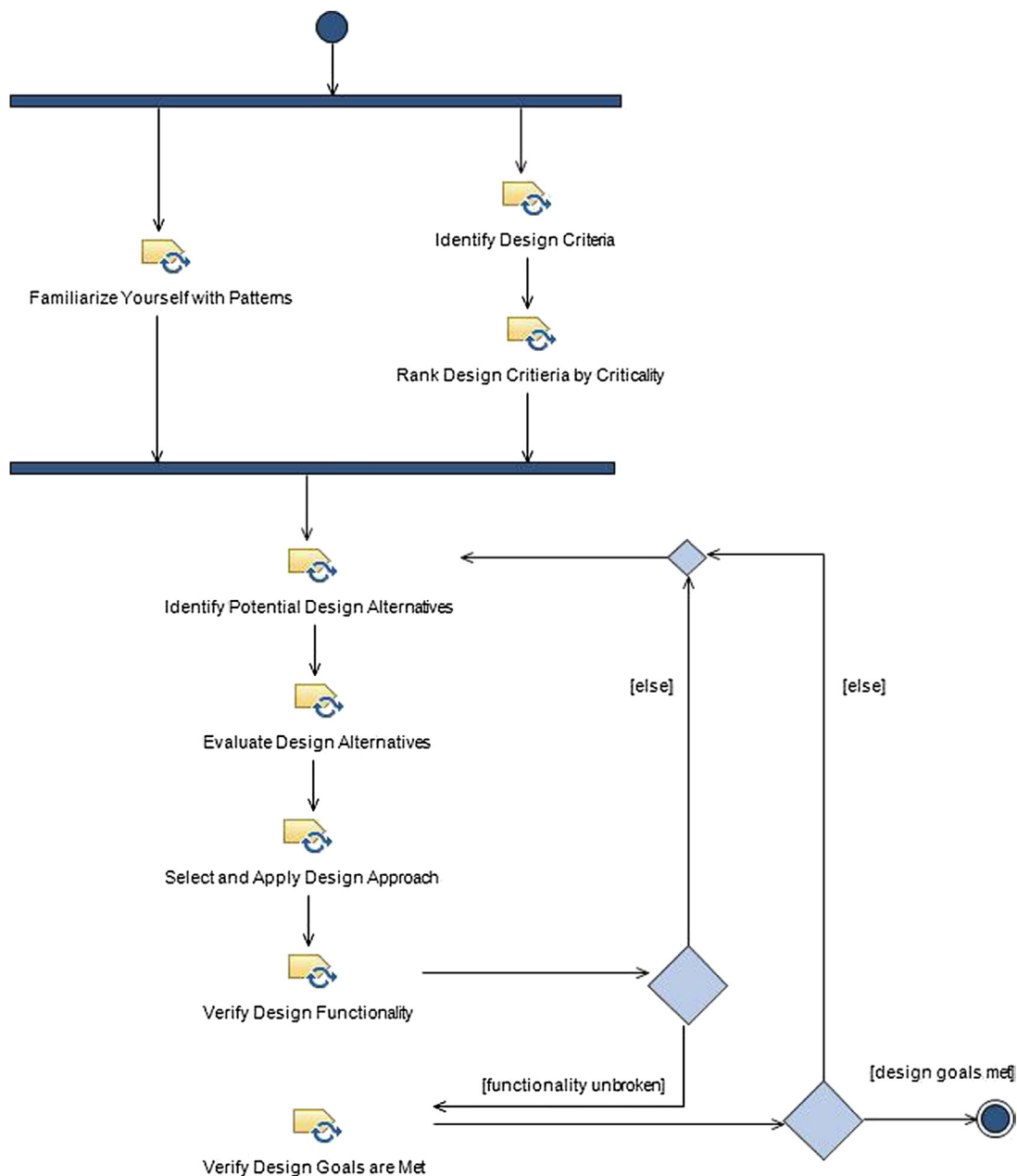


Figure 6.2
Pattern hatching.

7. Verify Design Functionality

The collaboration of software elements worked before applying the design patterns. This task ensures that it still does. Because the application of design patterns usually requires some (possibly small) amount of refactoring, some unit tests may require updates. If the pattern application has broken the functionality, the software must be repaired so that it passes its functional unit tests.

8. Verify Design Goals Met

Once the functionality of the design is verified, we must also ensure that the design achieves its optimization goals. If we've optimized the code for memory usage, the usage should be measured as the software runs. If we've optimized worst-case execution time, then that time should be measured.

Problem 6.1 Safety Architecture

The safety architecture specifies how the architecture identifies, isolates, and corrects faults at runtime with the goal of preventing harm. At the architectural level, this boils down to ways in which redundancy is defined and used to achieve the system safety goals.

Assessing the adequacy of the design solution is done with safety analysis (such as fault tree analysis (FTA)¹⁰) and/or reliability analysis (such as failure mode and effects analysis (FMEA)¹¹). FTA supports reasoning about what combinations of events and conditions result in hazards. The notational elements of FTA are shown in Figure 6.3.

The purpose is to identify the conditions and events that must occur for hazardous conditions to arise. The logical operators are AND, NAND (Not AND), OR, NOR (Not OR), XOR, and NOT. The circle is a basic fault and the rectangle is an "intermediate fault," that is, a result of a logic operator working on more primitive faults. A typical, if simplified, FTA is shown in Figure 6.4. The system depicted is a cardiac pacemaker, and the hazardous condition depicted is *Pacing Too Slowly*. Once such an analysis has been performed, safety or control measures are

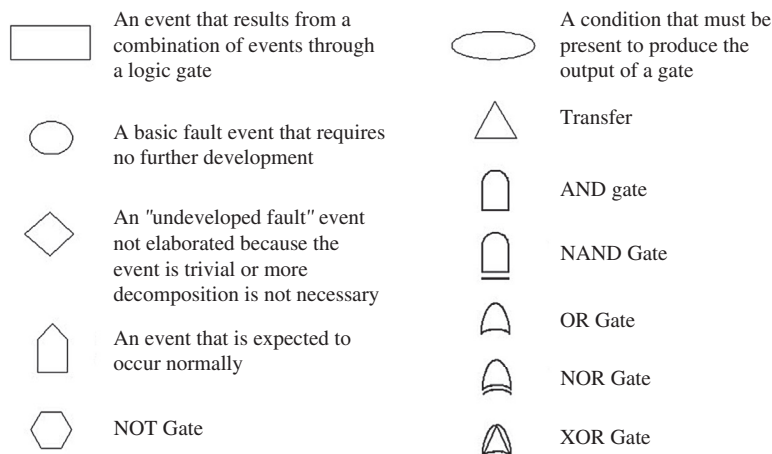


Figure 6.3
FTA notational elements.

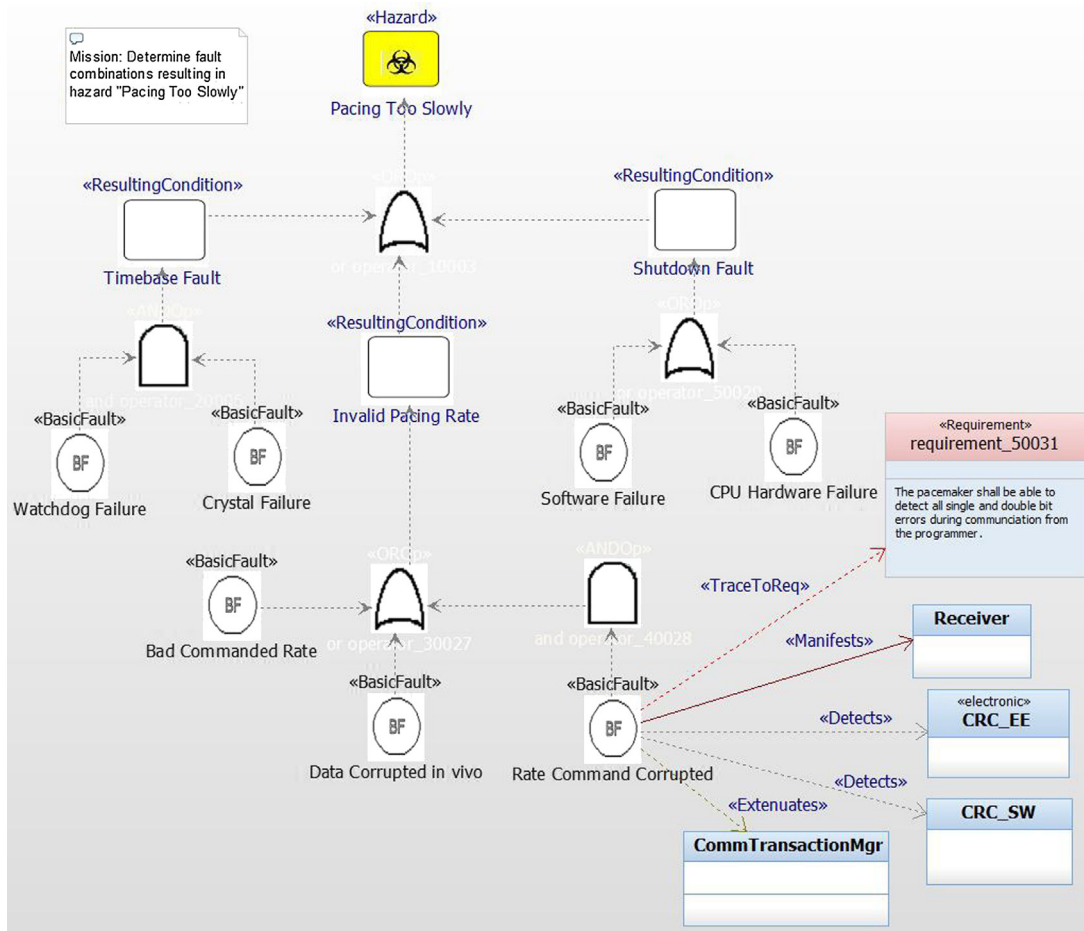


Figure 6.4
Example FTA.

added that must be ANDed with the other conditions to allow the hazardous condition to arise. In the example FTA shown, for a Timebase Fault to arise, both the CPU crystal must fail AND the added watchdog safety measure must fail. I typically do a separate FTA for each identified hazardous condition when doing a system safety analysis.

As a quick aside, Figure 6.4 is a screenshot from the Rational Rhapsody tool using the fault tree analysis profile that I developed. Elements in the FTA model include safety relevant metadata (such as severity, likelihood, and risk). The profile extends the content of the standard FTA to permit explicit links to requirements and design elements, including links to

- Requirements dealing with the fault, condition, or event
- Design elements that could cause the fault (manifestors)
- Design elements responsible for detecting the fault (detectors)
- Design elements responsible for dealing with the fault when it occurs (extenuators)

Hazard Analysis

How to use this spreadsheet

The spreadsheet computes risk = severity * likelihood, where severity is a ranking of 1 (very low) to 10 (very high).

Note that various safety standards may use a different range for severity. Likelihood is the probability of occurrence of the hazard in the life expectancy of the product (0 (impossible) to 10 (certain)). Risk is computed from these values.

Exposure time is computed as the sum of the Detection Time + Action Time. For a safe system this value must be less than the Tolerance Time.

"Is Safe" is computed as = (Exposure Time <= Tolerance Time).

Note that the spreadsheet assumes that the time units are the same for an entire row.

Hazard	Fault	Severity (1 (low) -10 (high))	Likelihood(0 (impossible) – 10 (certain))	Computed Risk	Time units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE
				0.00						0	TRUE

Figure 6.5
Hazard analysis.

The profile also defines tables and matrices for summarizing fault and hazard information within the model including the fault and hazard metadata and the relations among the fault analysis elements, the requirements, and the design.

The result of this analysis usually culminates with a hazard analysis that specifies each fault with a description, severity, likelihood, risk, fault tolerance time, control measure (design approach to handle the fault), fault detection time, and fault exposure time. I represent this information as a spreadsheet, such as that shown in [Figure 6.5](#), which shows some hazards for a patient ventilator. The actual performance of safety analysis is beyond the scope of this book, but interested readers are referred to the cited references for more information.

All safety measures do one or more of the following:

- Obviation – eliminate the safety concern by making the preconditions impossible
- Reduce likelihood – decrease the likelihood that the hazard will manifest
- Reduce severity – decrease the damage of the outcome

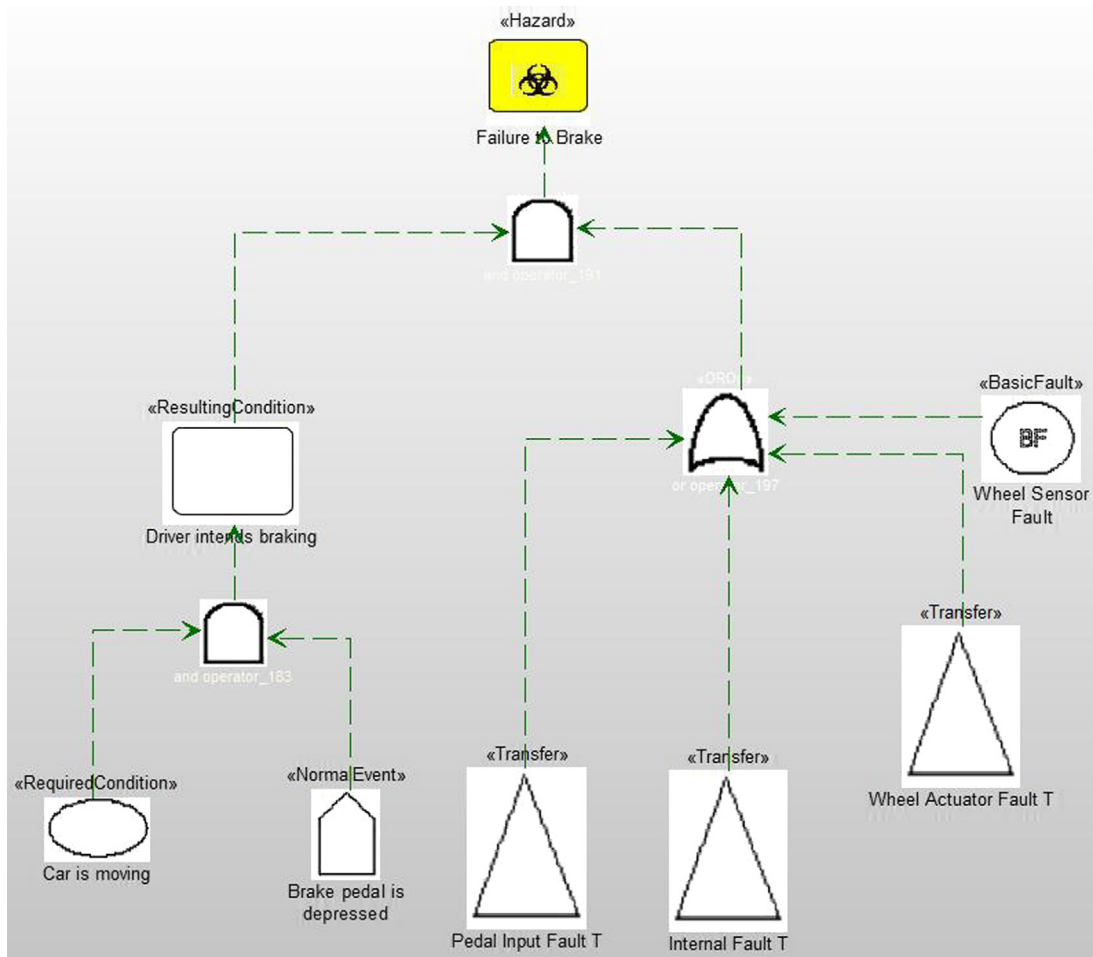


Figure 6.6
Failure to Brake FTA.

For a quick example: consider an automotive braking system that contains a pedal assembly; the hazard we want to address is Failure to Brake. Figure 6.6 shows the high level FTA for this hazard. It is broken down into subdiagrams (indicated with the triangular “transfer” operator). The details of the Pedal Input Fault are shown in Figure 6.7.

So the Pedal Input Fault, which could lead to a Failure to Brake hazard, can happen if any of the following occur: a pedal restriction fault, a pedal stuck on fault, or a pedal input sensor fault.

We can make the hazard less likely to occur by adding multiple sensors; only if all of the associated sensors fail does the depression of the brake pedal fail to be detected. In conjunction, we can make the hazard less severe by adding seat belts and an air bag. Both address the hazard risk but do so in different ways. The advantage of the FTA is that it allows us to reason about where to add safety measures and their effectiveness.

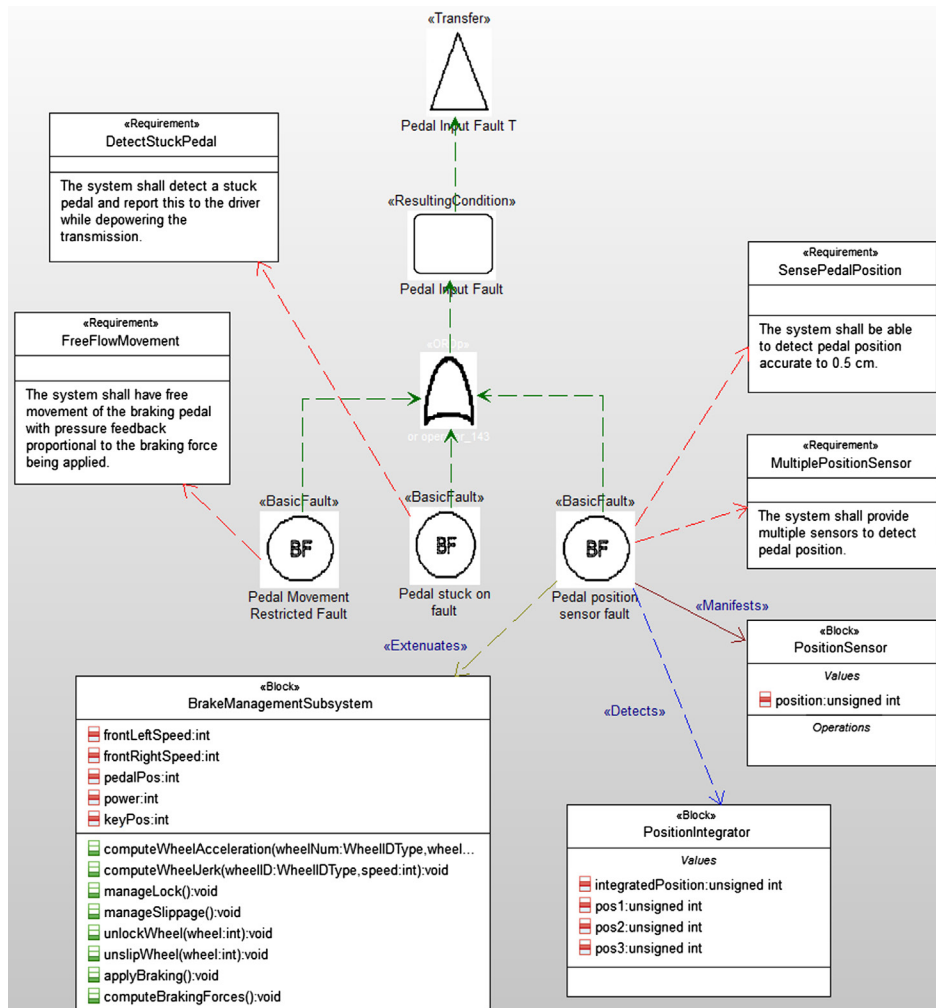


Figure 6.7
Pedal Input Fault detailed FTA.

Safety and reliability analysis is not directly supported by the UML but the UML can be extended to support such analyses (such as with the Fault Tree Analysis profile used here).

Most of the commonly used safety and reliability patterns are based on the *channel pattern*. This pattern is shown in Figure 6.8. A *channel* is a kind of subsystem that performs “end-to-end” functionality; that is, it takes data from physical sensors and performs a series of data transformational steps, culminating with output actuator control. The advantage of this organization is that it is easy (although not always inexpensive!) to replicate the entire channel so that if any point in the channel fails, another channel can take over the required functionality.

Figure 6.9 shows an example simplified from a medical delivery system. The channel shown delivers oxygen. The actors (shown as objects with the «actorInstance» stereotype) connect to

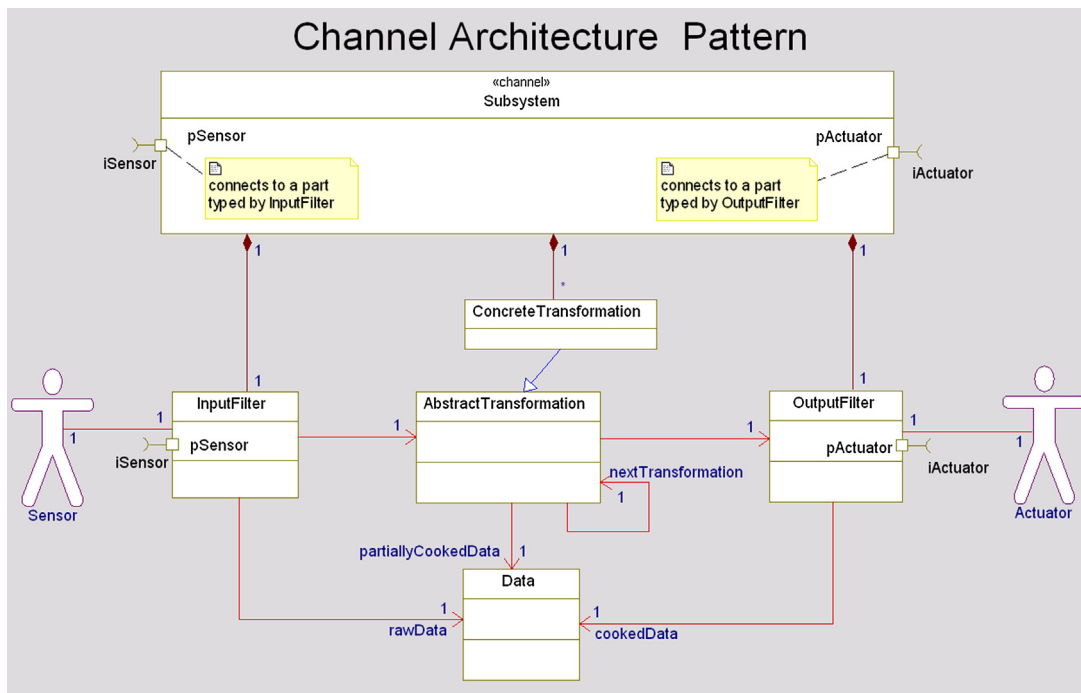


Figure 6.8
Channel architecture pattern.

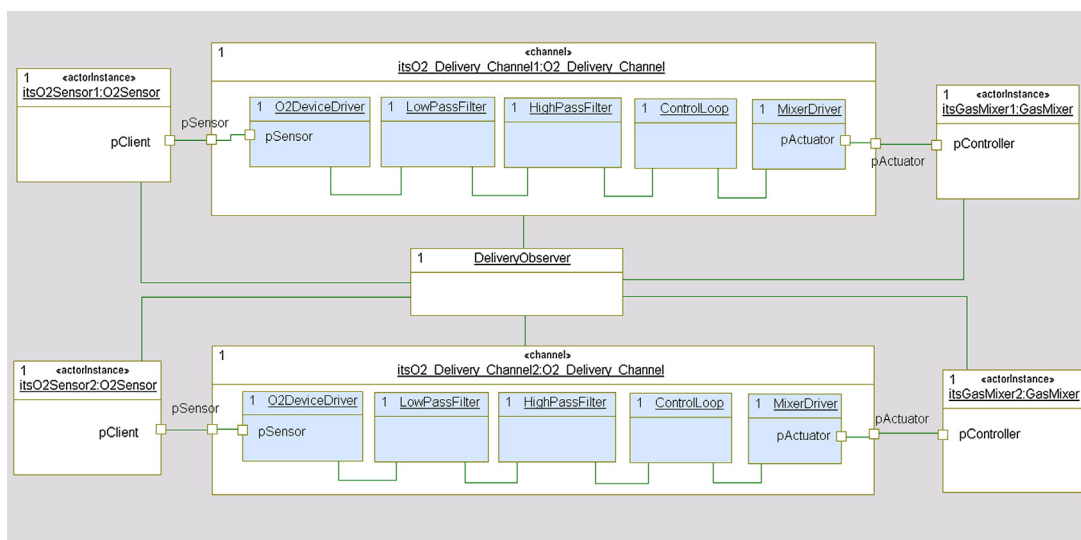


Figure 6.9
Example of channel architecture pattern.

the channels. Note that different O₂ sensors and gas mixers connect to the different channels as required to avoid common mode failures. Inside the channels, the acquired data goes through a series of transformations, including a control loop, until the appropriate settings for the gas mixer can be computed and then sent to the Gas Mixer actor. Additionally, there is a monitor object that compares the output of the channel with the input (and, presumably, the desired control settings) and can switch over to the other channel if the first fails.

This pattern is single point failure safe. If any element in the first channel fails (including the sensor or gas mixer), the monitor detects it and switches over to the other channel. If the monitor fails and switches to the other channel inappropriately, then the other channel still works properly (since in single-point failure safety analysis, we assume only a single error), and the proper amount of oxygen will still be delivered.

This pattern using homogeneous redundancy has the benefits of single point failure safety and low design cost, and it can continue to provide services safely in the presence of a failure. However, it cannot handle errors (since they will appear in both channels) and it has a high recurring cost. The first problem (handling errors) can be mitigated by using heterogeneous (aka “diverse”) channels – using a different design or design team – but this comes with a higher design cost.

Another common pattern is the *monitor-actuator pattern*. This is a variant of the channel pattern that has relatively low design and recurring costs but cannot continue in the presence of faults; that is, it requires that the system have a fail-safe state. The pattern structure is shown in [Figure 6.10](#).

The monitor-actuator pattern works by having a separate monitor that monitors the physical environment. If the actuation channel isn’t having the desired effect or the results are not within nominal operating conditions, then the monitor commands the channel into its fail-safe state (usually off or manual control). If the monitor has a fault, then either it allows the actuation channel to continue (which by single-point failure safety rules is still operating properly) or it mistakenly reverts to the fail-safe state. In all cases, if there are no common-mode faults, the system remains safe. Note that to avoid common mode faults, the monitor *must* use a different sensor than the actuation channel.

It should be noted that the monitor’s computation of “correct” may be complex. There is normally some range variance for a variety of reasons, and often flow or diffusion dynamics must be part of the computation. For example, in an oxygen flow system, the O₂ flow at the point of measurement won’t change the same instant the set point is changed. Partial differential equations can be used to estimate the expected rate of change as a function of the volume of the breathing circuit, the flow rates, predicted nonlaminar flow, diffusion, and O₂ update (extraction) by the patient.

[Figure 6.11](#) shows a straightforward example of the monitor-actuator pattern applied to the same system as the previous pattern. In this case, a separate sensor, perhaps connected at the terminal (patient) end of the inspiratory breathing tube, monitors delivered oxygen flow. The monitor compares it to the desired flow from the control panel (ultimately set by the physician). If it is not correct, then it shuts down the delivery of O₂ and alarms the attending physician, who (as part of the “safety loop”) then fixes the problem and restores the proper oxygen flow to the patient.

The last safety and reliability pattern we’ll mention in this chapter is the *single channel protected pattern*. This pattern offers both low design and recurring costs but provides less coverage than

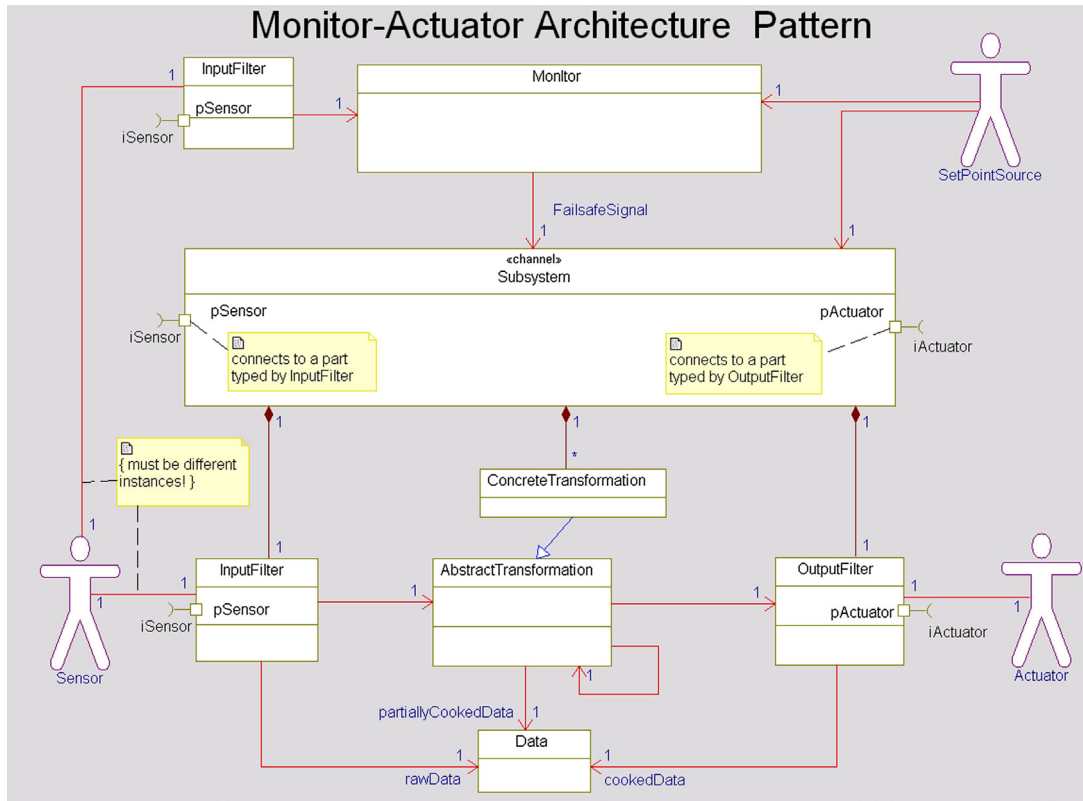


Figure 6.10
Monitor-actuator pattern.

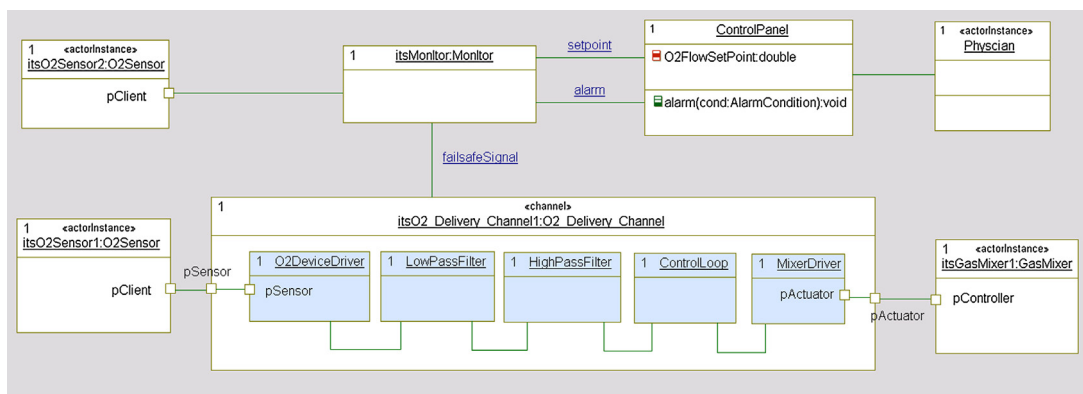


Figure 6.11
Example monitor-actuator pattern.

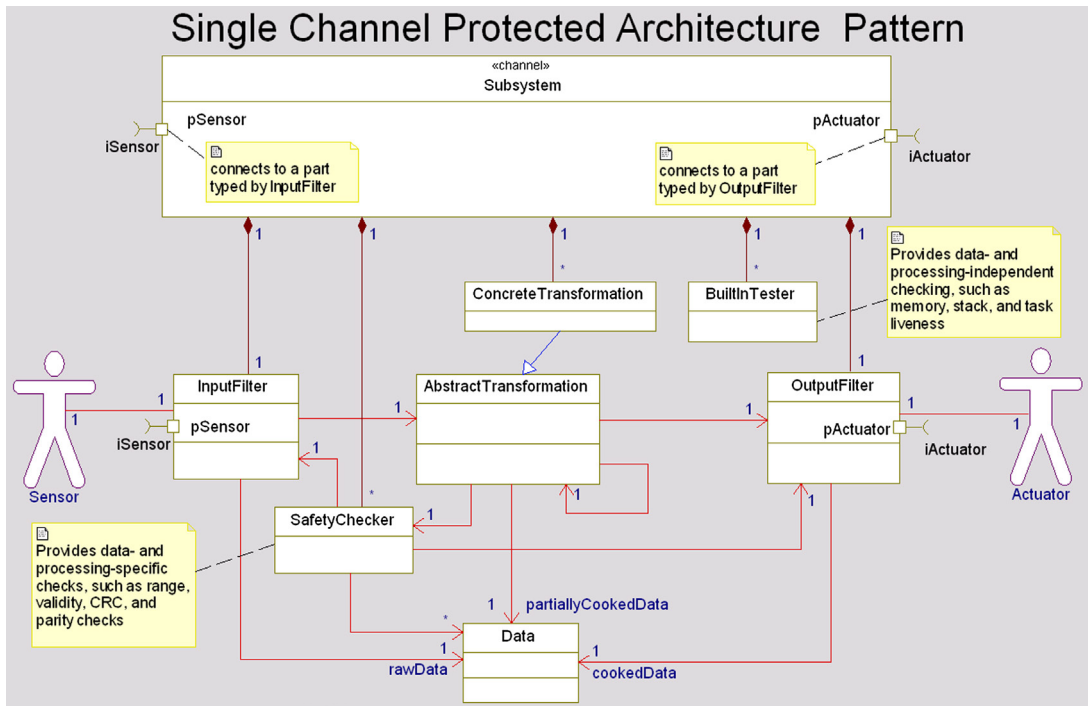


Figure 6.12
Single channel protected pattern.

some of the other patterns. It is similar to the monitor-actuator pattern in that it may only be used in safety-critical systems if there is a fail-safe state.

The structure of the pattern is shown in [Figure 6.12](#). You can see clearly that it is a variant of the channel pattern. It specializes that pattern by providing lightweight in-line data validity checks. It can detect data corruption and some computational error, but unlike the monitor-actuator pattern, it does not compare the physical output with the expected result. Internal data- and processing-specific safety checks are typically things like:

- Data corruption (feed-forward) checks
 - Cyclic redundancy check (CRC)
 - Parity check (weak)
 - Hamming codes
 - Redundant data check
 - Inverted redundant data check (e.g., store the date in a one's complement (bit inversion) form and when reading the data, invert the copy and compare to the stored value)
- Reasonableness checks
 - Range checks (e.g., patient weight should be in the range of 0 – 400 kg)

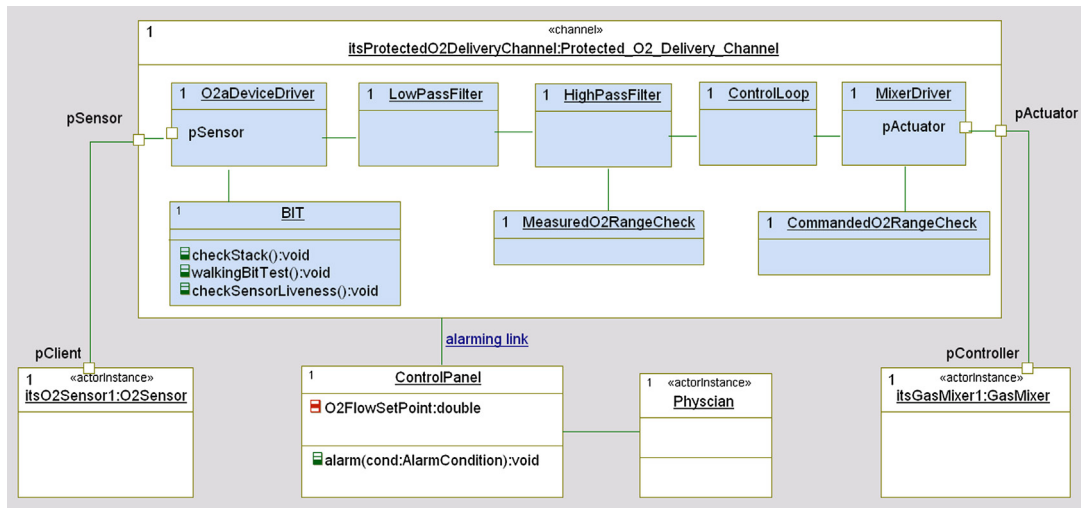


Figure 6.13
Example single channel protected pattern.

- *A priori* rule checks, e.g.,
 - Height should be a function of range of weight with a range around a mean.
 - If the patient type is neonate, then the weight range is 0 – 15 kg.
 - If the patient type is pediatric, then the weight range is 10 kg – 100 kg.
- Unit checks (e.g., kilograms, grams, pounds, when data may come from sources with different units)
- Backward derivation (feedback) checks
 - Run the computation backwards to compute the original value and compare

It is also common to run background tasks to detect stack overflow, memory faults, and liveness (e.g., lack of deadlock). Such checks are commonly known as “built-in tests.”

The channel provides a very low-cost but less complete check. An example is shown in [Figure 6.13](#).

Both the example systems in this book are safety-critical. The Roadrunner’s reason for existence is to provide a safe way for traffic to flow at points that are inherently dangerous (intersections). This traffic flow includes both vehicles and pedestrians. The system works by basically making everyone take turns so that they proceed through the intersection only when it is safe to do so. There are failures that could affect the safe fulfillment of this functionality, such as

- Both the primary and secondary through traffic have GREEN lights.
- Pedestrian traffic has a WALK light at the same time that the orthogonal vehicle traffic has a GREEN light.
- The turn lanes in SIM (simultaneous mode) are GREEN at the same time the through traffic in the same orientation has GREEN.
- The turn lanes in SEQ (sequential mode) are GREEN when the opposing traffic has a GREEN light.

- All lights are off.

The same rules may be applied to YELLOW lights and flashing WAIT as well, since traffic normally continues to flow under these conditions, at least for a while.

For the first part of this exercise:

- Take the collaboration done previously (refer to the solution in Chapter 13 if you need to) and do a separate FTA on each of the fault situations below:
 - Both primary and secondary through traffic have GREEN lights.
 - Pedestrian traffic has a WALK light at the same time that the orthogonal vehicle traffic has a GREEN light.
 - All lights are off.
- From the result of this analysis, create a hazard analysis in which hazards and faults are identified and control measures are added, including estimated fault tolerance times.
- Determine how architectural redundancy should be added to make the system single-point fault safe for the analyzed fault situations.
- Draw the safety architecture class or object diagram showing how the elements collaborate to achieve the desired safety.

For the second part of this problem, let's consider the Coyote UAV (CUAV) safety requirements. The major safety risks are:

- Loss of the aircraft
- Loss of life (friendly or noncombatant) due to crashing the aircraft
- Loss of life (friendly or noncombatant) because of a misdirected missile (from the CUAV itself), remote fire based on data from this system (e.g., a tank receiving information from the CUAV via the ground station), or not firing on hostiles when appropriate. These hazards could arise from
 - Data faults
 - Target misidentification (i.e., “false positive”)
 - Lack of (valid) target identification (i.e., “false negative”)
 - Loss or corruption of target information either in storage or transmission, e.g.,
Incorrect target classification¹² or Combat ID (CID)¹³
Incorrect location or coordinates
Loss of information about target
Incorrect target kinematics (vector, flight path, etc.)
 - Incorrect target management (e.g., failure in sensor fusion)
 - Missile faults
 - Missile flight control failure
 - Misfire of the missile
 - Firing a missile without proper authorization
 - Lost or corrupted commands or data during message transfer

The last of these is mitigated somewhat by including a mission ops person (human) in the loop and not leaving the use of lethal force entirely up to automated decision processes. However, even the smartest mission ops person can be misled by getting an incorrect Combat ID in the “fog of war.”

Since we've focused previously on developing the reconnaissance management subsystem – involved in target identification, tracking, and targeting – we'll continue in that vein.

- Create an FTA for the reconnaissance management subsystem that identifies the safety hazards and risks associated with this subsystem, for at least one hazard.
- From the result of this analysis, create a hazard analysis in which hazards and faults are identified and control measures are added, including estimated fault tolerance times.
- Determine how architectural redundancy should be added to make the system single-point fault safe for the analyzed fault situations.
- Draw the safety architecture class or object diagram for the subsystem showing how the elements collaborate to achieve the desired safety.

¹⁰ An excellent resource is *The Fault Tree Handbook*, NUREG 0492 (Nuclear Regulatory Commission, 1981), available at <http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/>.

¹¹ See <http://www.fmeainfocentre.com/> for some FMEA resources.

¹² Target classification is in terms of the type of platform (e.g., tank, truck, aircraft, platoon, etc.), class, unit, and/or nationality.

¹³ CID is the identification of the target friendliness, such as Friend, Assumed Friend, Hostile, Suspect, Neutral, or Unknown.

Problem 6.2 Reliability Architecture

As mentioned earlier, reliability refers to the availability of system services. Reliability analysis – most often captured in an FMEA – identifies the requirements for availability of system services and then examines how the availability of services from internal components propagate through the system to achieve system-level reliability. This results in reliability specifications at the component level. In general, reliability is improved through a combination of two design approaches. First, individual components may be engineered to be individually reliable. This is sometimes accomplished by choosing more robust materials, algorithms, and methods, even if they are more expensive. The other primary approach to improve reliability is by introducing component redundancy so that if a component fails there are other components ready to deliver the required component services.

The most common way to represent reliability metadata is in an FMEA.¹⁴ Key to the concept of reliability is the probability of failure, the availability of services, and the time between failures (which are all correlated). This information can be represented as quantitative (for example, a component might have a failure probability of 0.05 percent in the first two years) or qualitative (“relatively likely”). While quantitative values give the appearance of higher fidelity, they are only as good as their source information. The metadata is often shown with and without the mitigation so the effects of the design change are apparent.

FMEAs are most common represented as tables.

Steps to Creating an FMEA

First, identify the key services under consideration of the reliability analysis. These will form the sections of the FMEA. Then, for each service, identify the failure mode (externally visible or detectable condition) that could prevent the successful delivery of the service.

1. For each failure mode, identify the visible effects of the failure.
2. Identify the internal faults that could result in the failure mode. These form the rows in the table.

Failure Mode and Effects Analysis														
				Pre-action								Post-action		
Service / Function	Failure Mode	Faults	Failure Effects	Likelihood (1=impossible, 10=certain)	Severity (1=no effect, 10=catastrophe)	Detectability (1=certain, 10=no detection)	RPN (= sev * likely * detect)	Existing Control Measures	Recommendations	Responsible	Actions	Likelihood	Severity	RPN
Braking	Pedal nonresponsive	Pedal stuck	Pedal doesn't move; no braking action occurs	3	9	10	270	none	Make pedal assembly self-lubricating	Joe	Added sealed piston with lubrication	2	9	180
		Pedal position sensor fails	Pedal depresses; no braking occurs	4	9	8	288	Start up comm check with sensor	Use three pedal position sensors	Susan	Added two more sensors with voting	2	9	144
	Comm fault	CAN bus failure	Braking message not conveyed	3	9	2	54	Continuous monitoring of CAN bus	None	n/a				
		Loose bus connector	Braking message not conveyed	4	9	9	324	Continuous monitoring of CAN bus	Update monitoring to send lifeticks to every node on bus	Samuel	Updated lifetick protocol	2	9	162
	Processing fault	Braking ECU Failure	No braking occurs	2	9	2	36	Lifeticks every 1.0 seconds	none	n/a				
Wheel actuator fault		Brake actuator processing failure	No movement of brake master cylinder	2	9	10	180	None	Include in lifetick poll	Samuel	Updated lifetick protocol	1	9	90
		Brake hydraulic pressure low; O-ring leak; gasket leak	Reduced or no braking force	7	9	10	630	None	Add hydraulic pressure sensor; alert driver	Joe	Added sensor; updated periodic driver messages	3	9	270
		Push rod fracture	Hydraulic pressure not increased on braking	1	9	10	90	None	None	n/a				
		Vacuum boost leak	Reduced braking force	6	4	6	144	Sensor on pressure differential valve	None	n/a				
		Brake pads worn	Reduced braking force	9	4	10	360	None	Add periodic driver reminder for service every 6 months	Samuel	Updated periodic driver messages	4	6	240

Figure 6.14
Example FMEA for an automotive braking system.

3. Fill out the information related to each internal fault.
 - a. Identify the mission phase, if relevant.
 - b. Identify the likelihood of this failure. This is often a normalized value between 1 (impossible) and 10 (certain).
 - c. (FMECA) Identify the worst-case severity of the failure. This is often a normalized value from 1 (no effect) to 10 (catastrophe).
 - d. Determine the detectability of the fault. This is often a normalized value between 1 (certain to be detected with current measures in place) and 10 (undetectable).
 - e. Compute the risk priority number ($RPN = \text{likelihood} \times \text{severity} \times \text{detectability}$). This value is used to prioritize reliability engineering work.
 - f. Identify existing (if any) control measures to be used to reduce the likelihood (or severity).
 - g. Identify recommendations (if any) to improve reliability with respect to this fault.
 - h. Identify the responsible party or engineering role.
 - i. Record actual actions performed with regard to mitigation of this fault.
 - j. Identify resulting severity, likelihood, and risk *after* the control actions are implemented.

The actions taken include (but are not limited to) design actions, such as adding redundancy to make the system failure less likely or less severe, or making the individual components more robust. For example, the FMEA in [Figure 6.14](#) adds lubrication in a sealed piston to the braking pedal to make the pedal less likely to jam, but also adds additional pedal position sensors to provide redundancy in the event of a sensor failure.

For this problem, look at the service of the vehicle detection service of the Roadrunner intersection control system and think about the ways in which component failures could reduce its reliability. If the actions call for more robust components, specify the robustness as requirements associated with those subsystems and/or components; if the actions call for redundancy, add that into the architecture and/or as requirements.

Similarly, for the Coyote UAV, examine the vehicle service to manage the positions of the flight control surfaces – the ailerons (moveable surfaces on the wing trailing edges), elevators

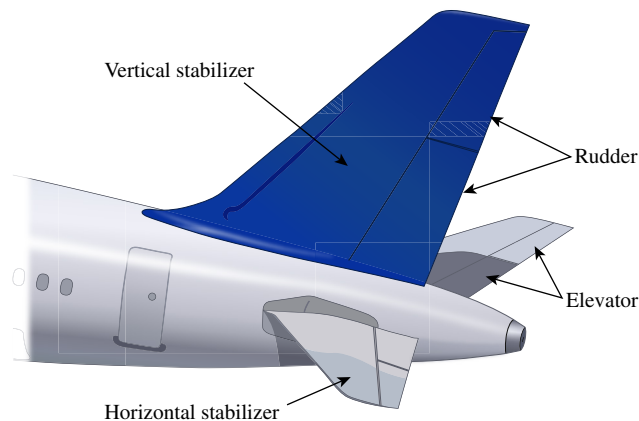


Figure 6.15
Elevator and rudder.

(moveable part of the horizontal stabilizer (rear wing flaps), and rudder (moveable surface on the trailing edge of the vertical stabilizer). Analyze the faults that might interfere with the delivery of this service, and capture the actions as requirements and/or architectural elements. See [Figure 6.15](#) for a visual explanation.

¹⁴ An FMEA, per se, does not have any indication as to the severity of a failure and so cannot be used in safety analysis. However, a failure mode, effects, and criticality analysis (FMECA) adds this information and so is relevant. Modern usage of the term FMEA often is intended to refer to FMECA.

Problem 6.3 Security and Information Assurance Architecture

The third tier of dependability is *security*. Security in the context of systems is about the prevention, detection, and response to unintended behavior caused by outside influences. Such unintended behavior includes theft (of information or of physical things), causation, prevention, or modification of system behavior, or interruption in the delivery of system services.

As more systems are connected, the risk of compromise increases exponentially because of the ability to remotely penetrate the system. In times past, hacking into a nuclear power plant required the perpetrator to be on the premises with a crowbar. However, now it is possible to hack into and damage, take control, or steal information from civil power networks, automobiles, and medical devices. We must all be concerned about security as complex devices evolve.

Although in complex systems a security solution may be as simple as adding a physical lock, most focus is on cybersecurity as we construct the Internet of (billions of) Things. The analysis of the means by which security can be penetrated is known as threat analysis. Similar to FTA or FMEA, threat analysis leads to control actions to maintain and improve the security of devices.

Readers may have surmised a predilection for modeling in this book and security is not immune from this focus. I've created a security analysis profile for UML that allows the analysis of assets, threats, and countermeasures. The profile is based on the metamodel shown in [Figure 6.16](#).

Key metaclasses in this model are outlined in [Figure 6.17](#).

Workflow for threat modeling is straightforward.

1. Identify assets that you want to protect.
2. Assign values to the asset metaattributes. This is important because it makes sense to spend more effort and cost protecting more valuable assets.
3. For each asset, identify means by which the asset can be compromised; these are the vulnerabilities.
4. For each asset above some threshold, for each of its vulnerabilities, identify the attack vectors (threats), that is, the means by which the vulnerability can be exploited.
5. For each threat, identify a countermeasure to prevent the attack.

The profile is meant to facilitate this reasoning so that engineers can identify what countermeasures are appropriate and where they are needed. This is similar to safety and reliability analysis but is expressed towards security concerns.

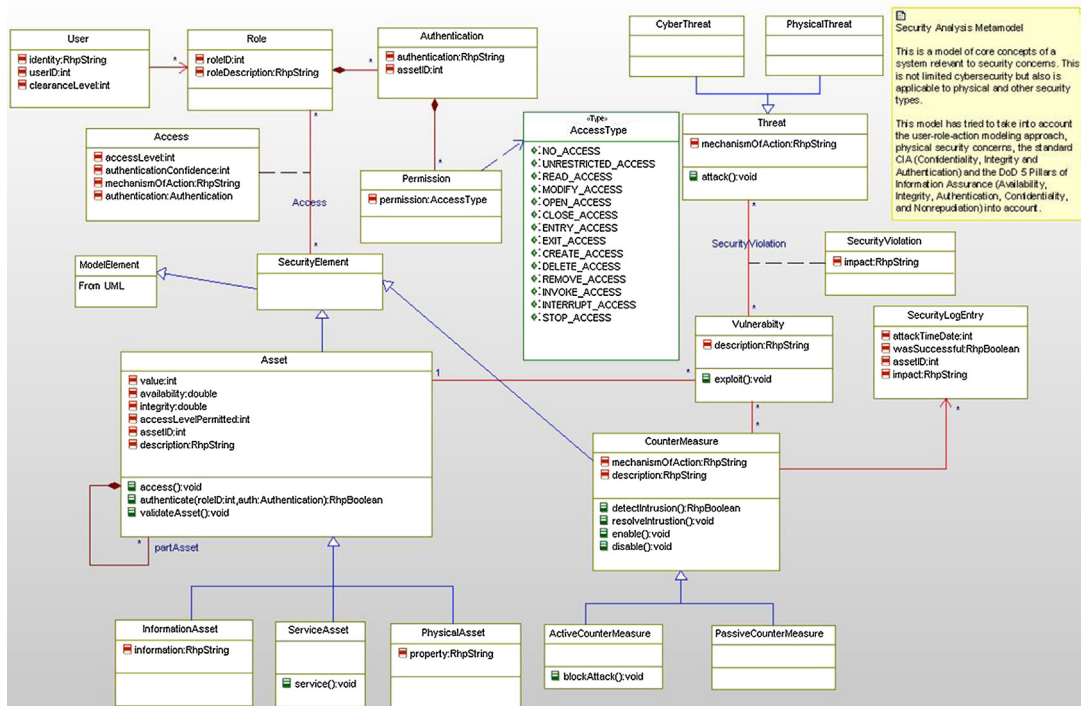


Figure 6.16
UML security profile metamodel.

Consider a medical ventilator. Such a device has a number of assets worthy of protection: patient demographic data, patient medical data, and ventilator configuration settings (information assets); gas mixing, delivery of ventilation, and device mode (service assets); and O₂, He, and N₂ gas supplies (resource assets). Figure 6.18 shows a model-based threat analysis for patient data.¹⁵

Just like the hazard analysis and FMEA, summary tables can be easily constructed for the security relevant metadata to support ranking the vulnerabilities and threats. Figure 6.19 shows such a table format.

Countermeasures are the design protections built into the system. As with other design measures, reusable solutions can be characterized as design patterns. There are a number of common design patterns used to enhance security such as the use of firewalls, trust regions, message and data encryption, passwords, policy enforcement protocols, and so on.¹⁶

For the last exercise in this chapter, perform a security analysis of the Roadrunner and Coyote UAV systems. Identify at least one asset (service or information is recommended), reason about how it might be compromised, identify threat vectors, and specify countermeasures. Optionally, summarize your analysis in tabular form such as in Figure 6.19.

¹⁵ If you don't appreciate the icons, they can be replaced with boxes as with normal class diagrams. The metatypes can be identified via stereotypes on the boxes.

Metaclass	Description
Asset	Assets are security relevant features of a system, including the value, availability, required authentication for access, and type of the asset, and may be ▶ Information assets – refers to information that should be kept secure, such as user credit card information, classified data ▶ Service assets, for example life support service (e.g., cardiac pacemaker or patient ventilation), power delivery service (e.g., power grid), credit authorization service ▶ Physical assets – meant to refer to physical elements and infrastructure, such as cash, paintings, automobiles ▶ Currency assets ▶ Resource assets ▶ Security assets – countermeasures may themselves have vulnerabilities and be attacked Assets may be “nested,” that is, assets may contain other assets with different levels of required authentication and security.
Authentication	The means by which an asset may be accessed. For example, a physical key, password, retinal scan, or encryption key. An authentication has a set of permissions.
Countermeasure	Features added to a system to ensure its security; they specifically address vulnerabilities. ▶ An active countermeasure looks for, detects, and acts on attempts to penetrate vulnerabilities. For example, a motion detector alarm or an authentication check. ▶ A passive countermeasure prevents access via a vulnerability; for example, a door, lock, enclosure, data encryption, or removing an access port.
Role	Represents the usage of a user within the security context that is granted permission to access some set of assets through a set of authentications.
Threat	The means by which a vulnerability is exploited.
User	Represents an actual human or system instance; may have a permission or security level that grants level of access to all its roles.
Vulnerability	The means by which an asset may be inappropriately accessed. It may be a physical, system, or software vulnerability.

Figure 6.17
Key metaclasses for security metamodel.

¹⁶ *Secure Design Patterns* by C. Dougherty, K. Sayre, R. Seacord, D. Svoboda, and K. Togashi (<http://www.sei.cmu.edu/library/abstracts/reports/09tr010.cfm>) is one good source. See also *Security Patterns: Integrating Security and Systems Engineering* by M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad (Wiley Press, 2006).

6.4 Summary

Dependability refers to our ability to depend upon our systems. It has three primary aspects: safety (freedom from harm), reliability (availability of services), and security (resilience against attack). These are broad cross-cutting concerns that will result in design and implementation that crosses disciplinary boundaries. Furthermore, these concerns remain ongoing throughout the design and implementation process, since technical design choices will impact

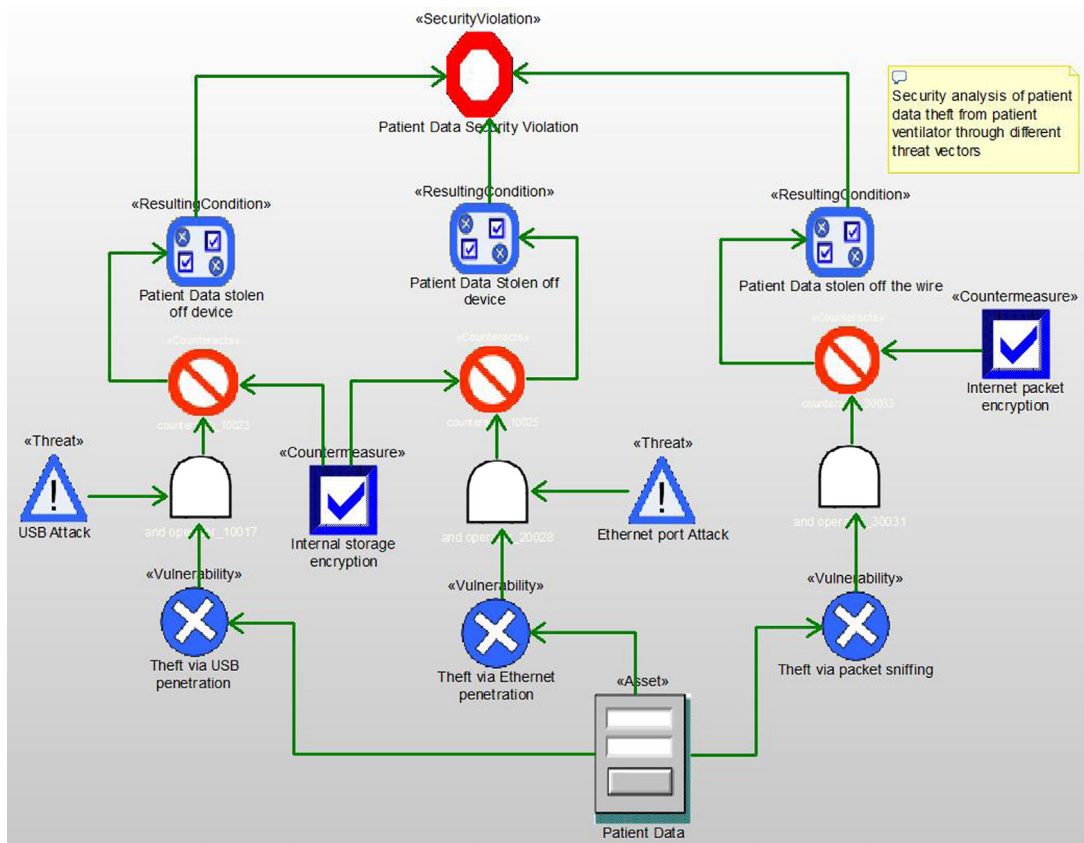


Figure 6.18
Security analysis of patient data theft.

all these of these areas. Hopefully this chapter introduces you to the tools necessary to develop dependable systems.

The next chapter will focus on software development, and specifically, *high-fidelity modeling* of the software.¹⁷ In this chapter, we will begin to model the software design and generate code. This implementation will be – demonstrably – correct but generally suboptimal against various design criteria. That optimization will be dealt with in the subsequent chapters on design.

¹⁷ Formerly known as “Obejct Analysis.” The name was changed to clarify the intent and actions of the activity.

Threat Analysis Table										
Asset value is the value of the asset to be protected (1=very low, 10=very high).										
Likelihood is the probability of the attack (1=very low, 10=certain).										
Reproducibility refers to how easy it is to reproduce the attack (for example, does it depend on timing or other circumstances?) (1=hard, 10 = very easy).										
Exploitability refers to how easy it is to launch the attack (1=very easy, 10=very hard).										
Breadth is the a measure of the extent of the attack. How widespread is it or how many systems are affected? (1=few, 10=very many).										
Discoverability is how easy is it for outsiders to find out about and exploit the vulnerability (1=very easy, 10=very hard).										
Threat Priority is the product of the above values and is used to prioritize the threats for countermeasures.										
			These are in the range of 1 -10							
Asset	Vulnerability	Threat Vector	Asset Value	Likelihood of attack	Reproducibility	Exploitability	Breadth	Discoverability	Threat Priority	Countermeasure
Patient Demographic Data	Access via Ethernet	Input validation weak	4	7	9	4	1	9	9072	Internal encryption
	Access via USB	Auto-execution of USB SW	4	7	9	3	1	9	6804	Internal encryption
	Access via packet snooping	Messages sent in plain text	4	7	9	5	1	8	10080	Message encryption

Figure 6.19
Threat analysis table.

High-Fidelity Modeling¹

What You Will Learn

- How to do high-fidelity modeling
- Object identification strategies
- Test-driven development
- Nanocycle iterations

7.1 Overview

The purpose of the *high-fidelity modeling* phase is to

- Identify the software elements essential to realizing the use cases
- Identify relations among the essential software elements
- With an object-oriented design
 - Identify attributes and allocate them to classes
 - Identify operations and allocate them to classes
 - Specify the behavior of reactive classes with state machines
 - Specify operation behavior with flow charts or activity diagrams
- With a structured design
 - Identify variables and allocate them to files
 - Identify functions and allocate them to files
 - Identify and specify reactive behavior with state machines
 - Specify function behavior with flow charts or activity diagrams
- Verify the correctness of the analysis model

You can see that the workflow is the same regardless of whether an object-oriented or a structured design approach is used.

¹ This was previously known as “Object Analysis” in the previous revision of the Harmony process. I changed to name to clarify the purpose of the workflow and to remove the connotation that it only applied to object-oriented design strategies.

7.2 A Quick Note about Structured Design with UML

UML *can* be used to represent structured design, so if you want to design systems to be implemented in standard C with no special idioms, it is pretty straightforward. The key concepts for structured programming are

- Functions
- Data types
- Variables
- Files (two types: header and implementation)

A *file* is the design-time organizational concept. Files contain the other three kinds of elements. The most obvious way to use UML to represent C is to stereotype a class to represent a file. What would normally be thought of as attributes are variables within the file, and operations are functions within the file. Including header files is a «usage» dependency. If you

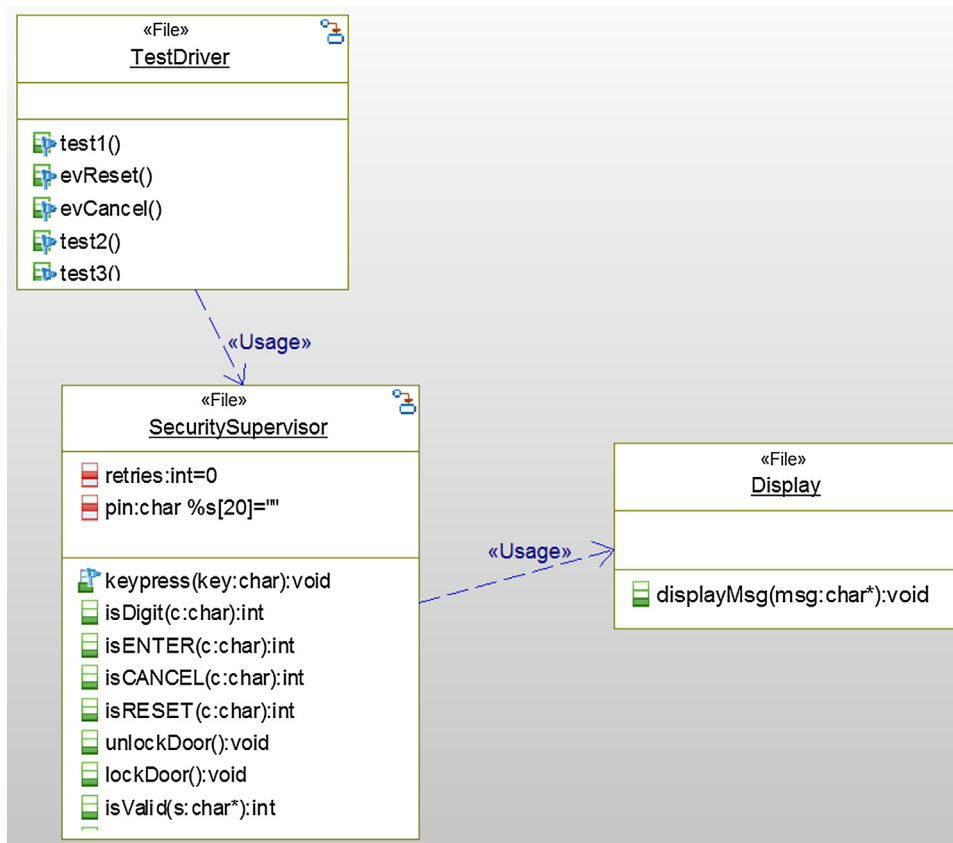


Figure 7.1
File-based C UML model.

look at Figure 7.1, you see what looks like classes but actually are C files containing types, variables, and functions.

The boxes in Figure 7.1 are a direct representation of the actual C code. This allows the structured designer to gain (almost all of) the benefits of UML without having to change his or her design or programming style.² Incremental enhancements to the design style allow the adoption of object-based (instance-oriented) and object-oriented (with specialization) approaches if desired.³

7.3 High-Fidelity Modeling Workflow

For the most part, we will proceed with an object-oriented design. However, if you prefer a structured design approach, be aware that virtually all of the techniques and methods we will describe fully apply to structured design as well. In this case, wherever you see “objects and classes,” think “functions and data structures.”

Since the requirements are organized into coherent use cases, high-fidelity modeling is normally applied a use case at a time, resulting in an collaboration of software elements that realizes all of the functional requirements of the use case. This collaboration is unlikely to be optimally efficient, but that is where object design comes in – the optimization of the high-fidelity model. But first, we must identify the essential objects and classes – the ones that *absolutely must be there* for the system to be correct.

High-fidelity modeling is a crucial phase of the Harmony spiral model.⁴ In each iteration through the spiral process, a number of use cases are elaborated into an executable, validated prototype. Figure 2.13 shows the workflow for high-fidelity modeling.

For each use case in the iteration build, we identify objects and classes with object identification strategies, and refine the models by adding details such as operations, attributes, and various relations. The key to effectively constructing the collaboration to realize the use case is to continuously verify the software all the way through its construction. Rather than create a collaboration of 50 objects and only then begin to execute, it is far more effective to begin execution as soon as a single object is added and to repeat the execution with each small incremental change. In this way, we can identify defects immediately and remove them just as quickly. Rhapsody provides a very powerful environment with its model execution and debug facilities that can greatly enhance this agile development approach.

² Rational Rhapsody supports this development style for C developers with a Functional C profile that ships with the product.

³ Object-based and object-oriented design approaches for C are discussed in more detail in my book *Design Patterns for Embedded Systems in C* (Elsevier Press, 2011).

⁴ Chapter 2 discusses the Harmony process workflows in more detail.

In high-fidelity modeling, we discover (more than invent) the essential elements of the system and represent them as objects, classes, attributes, operations, states, transitions, and relations. By “essential,” I mean that the collaboration is *wrong* if it doesn’t include that element. For example, if I am modeling a `Make Deposit` use case for a banking system, I would expect to see objects representing the customer, the bank account, the value held within the bank account, and the transaction itself. If those things are not represented, then the collaboration doesn’t properly represent the requirements in the use case. If we have a use case called `Heat Food in Microwave Oven`, then I would expect to see a door, door sensor, microwave emitter, a mechanism to vary the microwave intensity, timer (actual or implied) – or it’s not a microwave oven cooking food, but something else. In the high-fidelity modeling activity of the Harmony process, we identify these essential elements and how they are related to each other. And we do this in an *executable* fashion so that we can, at any time, verify the correctness of the model constructed thus far.

7.4 Key Strategies for Object Identification

There are many ways to identify the objects within a collaboration. [Table 7.1](#) outlines what I have found to be the most effective of these object-identification strategies. These strategies are a key technique in the development of software with the Harmony process.

We will briefly discuss all these strategies, but note that the analyst need not use them all on any specific project. These approaches are not orthogonal and the objects they will find will overlap to some degree. The best way to use these strategies is to find the two or three that work best for you and the problem at hand and apply this selected set of strategies to find all of the objects and classes in the analysis collaboration.

7.4.1 Underline the Nouns

The first strategy works directly with the written problem or mission statement. Underline each noun or noun phrase in the statement and treat it as a potential object. Objects identified in this way can be put into different categories:

- Objects of interest
- Classes of interest
- Actors
- Uninteresting objects
- Attributes of objects
- Events
- Synonyms for any of the above

Table 7.1 Object Discovery Strategies.

Strategy	Description
Underline the Nouns	Used to gain a first-cut object list, the analyst underlines each noun or noun phrase in the problem statement and evaluates it as a potential object, class, or attribute.
Identify Causal Agents	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify Services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify Messages and Information Flow	Messages must have an object that sends them and an object that receives them as well as, possibly, other objects that process the information contained in the messages.
Identify Real-World Items	Real-world items are entities that exist in the real world, but are not necessarily electronic devices. Examples include objects such as respiratory gases, air pressures, forces, anatomical organs, patient information, chemicals, vats, etc.
Identify Physical Devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. In the internal architecture, they are processors or ancillary electronic “widgets.” Note: This is a special category of the Identify Real World Items strategy.
Identify Key Concepts	Key concepts may be modeled as objects. Bank accounts exist only conceptually, but are important objects in a banking domain. Frequency bins for an online autocorrelator may also be objects. This strategy is an antipode to the Identify Real-World Items strategy.
Identify Transactions	Transactions are finite instances of interactions between objects that persist for some significant period of time. Examples include bus messages and queued data. This may be done with ACID (Atomic, Complete, Isolated and Durable) transactions or any kind of transactions – state-based or otherwise.
Identify Persistent Information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify Visual Elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify Control Elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply Scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects and relations.

7.4.2 Identify the Causal Agents

For every effect, there must be an element that is the cause. This strategy looks to identify the objects that cause things to happen, i.e., the ones that are behaviorally active ones. These are objects which

- Produce or control actions
- Produce or analyze data
- Provide interfaces to people or devices
- Store information
- Provide services to people or devices
- Contain more fundamental objects as parts

A causal object is an object that autonomously performs actions, coordinates the activities of component parts, or generates events. Whenever there is some initiating action, an object somewhere in the system must provide that service.

7.4.3 Identify Services (*Passive Contributors or Server Objects*)

In an object-oriented analysis, all (or at least the great majority of) services will be provided by objects. They may provide passive control (that is, they do what is requested of them when it is requested), data storage, or both. A simple switch is a passive control object. It provides a service to the causal objects (it turns the light on or off upon request), but does not initiate actions by itself. Passive objects are also known as *servers* because they provide services to client objects. Incidentally, servers almost always are at the receiving end of unidirectional associations from their clients; that is, clients know about the servers, but the servers do not know their clients.

7.4.4 Identify Messages and Information Flows

For each message, there is an object that sends it, an object that receives it, and potentially, other objects that process it. Messages correspond to information and control flows and are realized by either operation calls or event receptions. In either case, for every message, there is at least a sender object and a receiver object. In addition, the message itself is often an object. Certainly that is true for remote communications with network packets and datagrams. It is also true when the message must be remembered or processed in some way (see the Identify Transactions strategy, below).

7.4.5 Identify Real-World Items

Embedded systems need to model the information or behavior of real-world objects even though they are not part of the system *per se*. An anesthesia system must model the relevant properties of patients (name, weight, condition, billing information, etc.), even though customers are clearly outside the anesthesia system. A tracking system would typically model

relevant aspects of the things that it tracks, such as combat ID, target type, location, velocity and acceleration. These internal representations of real-world elements are normally not simulations of those systems but representations of information it is important for the system to represent. However, in some cases, such as simulation systems, it is crucial to model the physics of such systems as well, such as airflow in aerodynamic modeling.

7.4.6 Identify Physical Devices

Real-time systems interact with their environment using sensors and actuators in a hardware domain. The system controls and monitors such physical devices inside and outside the system and these objects are modeled as objects. Devices must also be configured, calibrated, enabled, and controlled so that they can provide services to the system. When device information and state must be maintained or services invoked, the devices are modeled as objects. Objects representing physical devices almost always represent the *interfaces* to those devices and serve as a means of encapsulating the invocation of services from them.

7.4.7 Identify Key Concepts

Key concepts are important abstractions within the domain that have interesting attributes and behaviors. These abstractions often do not have physical realizations, but must nevertheless be modeled by the system. Within the user interface domain, a *window* is a key concept. In the banking domain, an *account* is a key concept. In an autonomous manufacturing robot, the *task plan*, a set of steps required to implement the desired manufacturing process, is a key concept. In the design of a C compiler, *functions*, *data types*, and *pointers* are key concepts. Each of these objects has no physical manifestation. They exist only as abstractions modeled within the appropriate domains as objects or classes.

7.4.8 Identify Transactions

Transactions are objects that represent the interactions of other objects and must persist for a nontrivial period of time. Examples of transactions include bank account deposits and withdrawals, elevator requests, target designations for fire control systems, and objects that manage reliable message delivery. In many cases, the transactional objects disappear once the interaction has concluded (so-called *volatile transactions*) while in other cases, the transactional objects must be retained for long periods of time (also known as *persistent transactions*).

7.4.9 Identify Persistent Information

Persistent information is typically held within passive container objects such as stacks, queues, trees, tables, or databases. Configuration data for various devices is one kind of persistent data. Many systems also acquire information from their environments and store this information. Of course, when information must be stored, objects hold that information in

their attributes. Examples include a system that remembers patient data or usernames and passwords, calibration tables for sensors, and flight data for a black box recorder.

7.4.10 Identify Visual Elements

Many real-time systems interact directly or indirectly with human users. Real-time system displays may be as simple as a single blinking LED to indicate power status, or as elaborate as a full Windows-like GUI with buttons, windows, scroll bars, icons, and text. Visual elements used to convey information to the user are objects within the user interface domain. These visual elements are themselves objects, and have both attributes and behavior.

7.4.11 Identify Control Elements

Control elements are entities that control other objects. These are specific types of causal objects. Some objects, called *composites*, often orchestrate the behaviors of their part objects. These may be simple objects or may be elaborate control systems, such as

- PID control loops
- Fuzzy logic inference engines
- Expert system inference engines
- Neural network simulators
- State-based systems
- Algorithmic systems

7.4.12 Apply Scenarios

The application of use case scenarios is another strategy to identify missing objects as well as to test that an object collaboration adequately realizes a use case. Using only known objects, step through the messages to implement the scenario. The object collaboration structure must support the elaborated use case scenarios – including providing the information to be manipulated and the services realizing the messages in the scenarios. This is one of the most useful strategies because it allows you to identify elements and behavior missing from your collaboration.

In the course of the exercises in this chapter, we will apply some of these strategies to identify the objects within one or more collaborations. You will need to refer to the written problem specification in the appendices as well as the previous requirements models. In an actual project, usually three or four of these strategies is sufficient for any given use case. However – as will discuss in more detail – the best approach is not to apply the strategies exhaustively until you have the software model and then begin to get it working. It is much better to *incrementally* apply the strategies to construct working debugged collaborations that are associated with tested source code. We'll get to that a bit later in this chapter. But first ...

Problem 7.1 Apply Nouns and Causal Agents Strategies

In this problem we will limit ourselves to applying the Underline the Nouns and Identify Causal Agents strategies. The first of these is a very common strategy although in my experience, the least helpful. The application of this strategy is to simply read the parts of the problem statement that apply to the use case in question, and then underline the nouns and noun phrases. These noun phrases are our candidate objects and classes.⁵ The difficulty with the strategy, besides the fact that it only finds the explicitly stated objects, is that it also identifies attributes, actors, and objects we don't care about, and synonyms for already identified elements. Thus the list requires careful pruning and won't be complete in any case. Also be aware that in this book, we have very short problem statements – in many systems projects, the problem statements will be much longer and complete, possibly running to hundreds of pages for large systems. However the strategy can be used even in those cases, because we apply the strategy a use case at a time.

The causal agent strategy looks for autonomous and triggered behaviors in the system and tries to identify the sources and targets of those behaviors.

The first part of this problem is to apply these strategies to – you guessed it – the Roadrunner Intersection Controller. Because the system is relatively simple, we'll apply the strategy to two system-level use cases – Detect Vehicle and Fixed Cycle Time Mode (a subclass of the Manage Traffic use case). The problem statement relative to the Detect Vehicle use case is shown below:

The Vehicle Detector

Three types of Vehicular Detectors shall be supported: subsurface passive loop inductors (SPLIs), above-surface infrared sensors (ASIs), and above-surface radars (ASRs).

Subsurface detectors shall use a wired interface to communicate with the controller, while ASIs and ASRs shall support both wired and secure wireless communication. All vehicle detectors shall be able to perform vehicle counting.

In addition, ASIs and ASRs shall be able to receive directional transmissions from priority vehicle and emergency vehicle transmitters. The maximum range of such reception shall be no less than 250 feet and no more than 1000 feet.

Figure 7.2 shows the relevant measures for both ASI and ASR detectors. When a vehicle enters the detection area (shown as the shaded area in the figure), the detector shall report the presence of a vehicle. Separate detectors are used for each lane in each direction.

The part of the problem statement that applies to the fixed cycle mode is shown below:

Mode 2: Fixed Cycle Time

Mode 2 is the most common operational mode. In this mode, the lanes cycle GREEN-YELLOW-RED in opposite sequences with fixed intervals. The system shall ensure that if any traffic light is non-RED, then all the lights for cross traffic shall be RED and pedestrian lights (if any) shall be set to DON'T WALK. Note that the turn lane times and/or pedestrian times are only valid in this mode if (1) the turn lane and/or pedestrian parameter is set TRUE in the RIC system parameters and (2) if a signal from the appropriate detector determines the existence of waiting traffic for the turn or pedestrian light

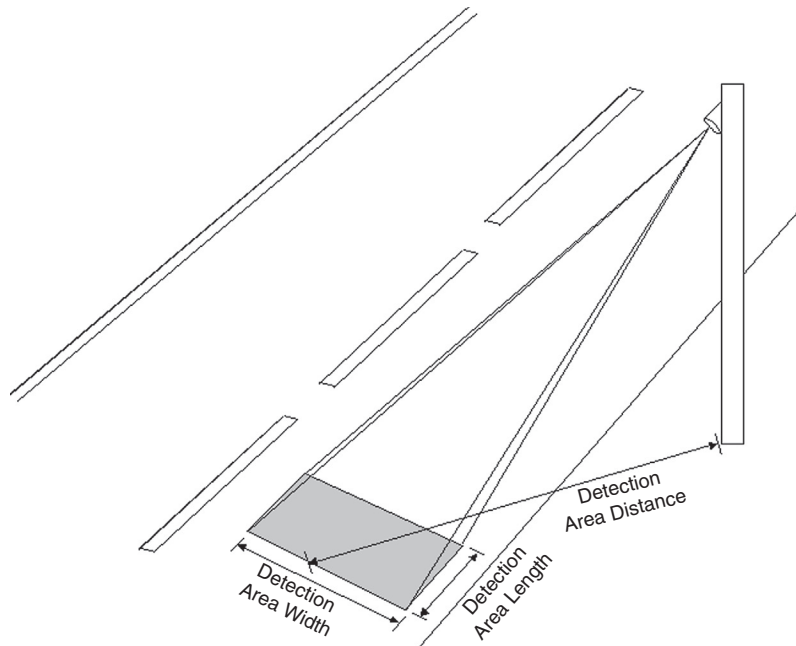


Figure 7.2
Above-surface vehicle detector.

The durations of the light times shall be independently adjustable by setting the appropriate parameters (see [Table 7.2](#)). Note that in the table, the values in parentheses are defaults.

The default values depend on the system configuration.

The values in [Table 7.3](#) are true for each direction, independently. Thus, if the primary road has a car waiting in its turn lane and a pedestrian walking, but the secondary road has neither, then the timing diagram in [Figure 7.3](#) represents the cycle times for simultaneous turn lane mode (i.e., the turn lanes in both directions for a road turn together and the straight traffic doesn't begin until the turn lanes have cycled to Red).

The first part of the problem is, then, to apply the Underline the Nouns and Identify Causal Agents strategies to this part of the Roadrunner specification and construct the resulting analysis object model.

The second part of this problem applies the strategies to the Coyote UAV (CUAV). The CUAV is a *much* larger system than the traffic light control system – probably two or three orders of magnitude larger. In such cases, it is common to apply the object identification strategies at the subsystem level rather than the system level. This fits well into the normal work allocation for systems as well. It is common to hand off subsystems to different (interdisciplinary) subsystem teams with a specification of what that subsystem must do. We recommend that the subsystem specification include the subsystem use cases, the system architecture into which the subsystem must fit, and a complete specification of the interfaces the subsystem must provide or may require from other elements (subsystems or actors). These subsystem level use cases are much more tractable in scope.

Table 7.2 Mode 2 Parameters.

Parameter	Value Range	Description
Reset Parameters	FALSE, TRUE	(FALSE) Sets all the parameters for Mode 2 to defaults
Primary Green Time (PG2)	10 to 180 seconds	(30) Length of time the primary green light is on
Primary Yellow Time (PY2)	2 to 10 seconds	(5) Length of time the primary yellow light is on
Primary Red Delay Time (PR2)	0 to 5 seconds	(0) Length of time between when primary red light is turned on and the secondary green light is activated
Primary Walk Time (PW2)	0 to 60 seconds	(20) Length of time the primary WALK light is on when the primary GREEN light is activated
Primary Warn Time (PA2)	0 to 30 seconds	(10) Length of time the primary FLASHING DON'T WALK light is on after the WALK light has been on
Primary Turn Green Time (PT2)	0 to 90 seconds	(20) Length of time the primary turn light is GREEN. Note: Only valid when the Primary Turn Light parameter is TRUE.
Primary Turn Yellow Time (PZ2)	0 to 10 seconds	(5) Length of time the primary turn light is YELLOW. Note: Only valid when the Primary Turn Light parameter is TRUE.

Table 7.3 Default Cycle Times for Mode 2.

Turn Lane	Ped Signal	Green	Yellow	Red	Walk	Don't Walk	Turn Green	Turn Yellow
F	F	30	5	0	0	0	0	0
T	F	50	5	0	0	0	15	5
F	T	50	5	0	15	5	0	0
T	T	50	5	0	15	5	15	5

We will apply the strategies to a single subsystem level – Reconnaissance Management. The use cases for this subsystem are shown in [Figure 7.4](#). For this exercise we will use the Acquire Image use case. As an aside, note that the peer subsystems in the figure are prefaced with a lowercase “a” – this is used to indicate that we are using the subsystem as an actor from this point of view. In reality, it is a different metaclass with a very similar name. This allows us to provide behavior for the actors (called “instrumenting the actors”) for simulation and test without modifying the actual subsystems they represent.

It should be noted that one of the difficulties of applying the Underline the Nouns strategy in big systems is that the textual specification may not be organized in a fashion to facilitate a per-use-case application of the strategy. It may require reorganization of the textual

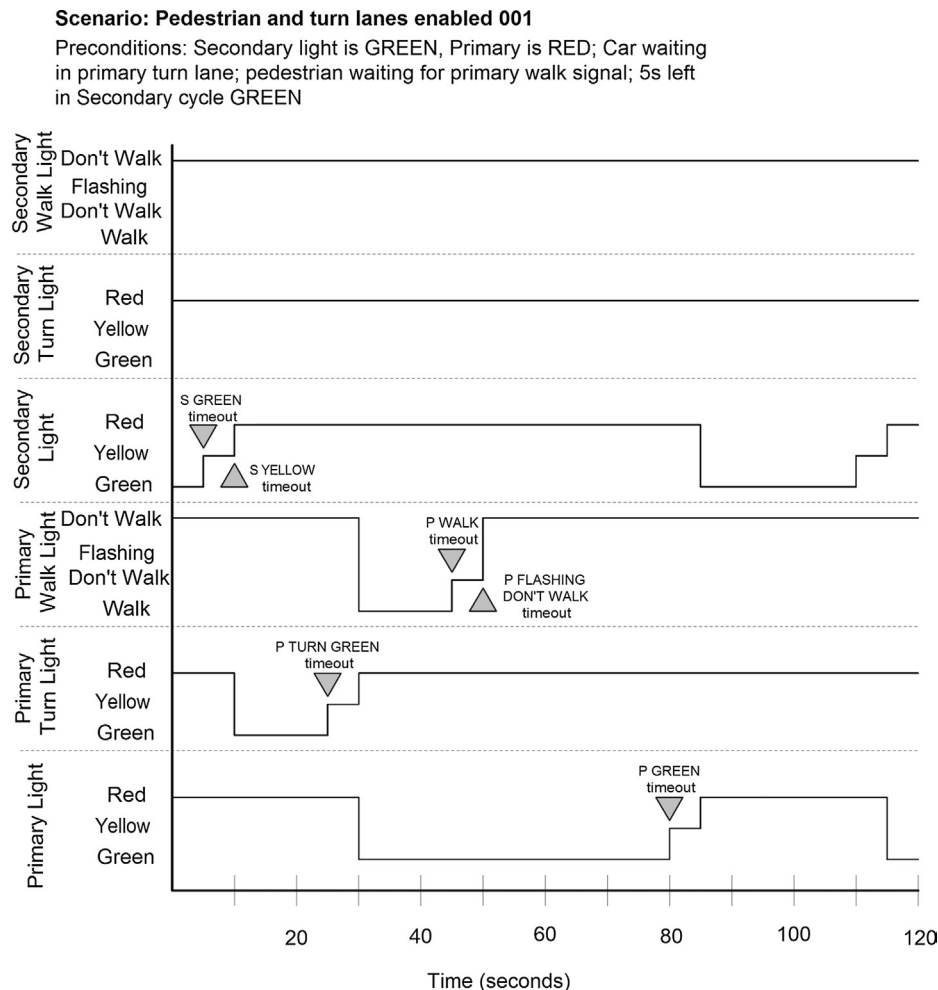


Figure 7.3
 Timing diagram for Mode 2 example.

specification or multiple passes through the document as a whole to identify all the sections related to the use case under consideration. If the document is organized by subsystems and then by use, this problem may disappear but in my experience, that is an unusual organization.

The Unmanned Aerial Vehicle (UAV)

The Coyote UAV is meant to be a multipurpose reusable UAV with multimission capability. It is meant to operate at altitude of up to 30,000 feet with ground speeds of up to 100 knots (cruise) and 150 knots (dash) and carry a payload of up to 450 lbs for durations in excess of 24 hours. The Coyote is meant to fly unimpeded in low-visibility environments while carrying either reconnaissance or attack payloads. While controllable from the ground station CMPCS, it is also capable of flying complex flight plans with specific operational goals of systematic area search, ground route (road-based) search, and orbit surveillance of point targets. Coupled with manned control from the ground, the Coyote provides

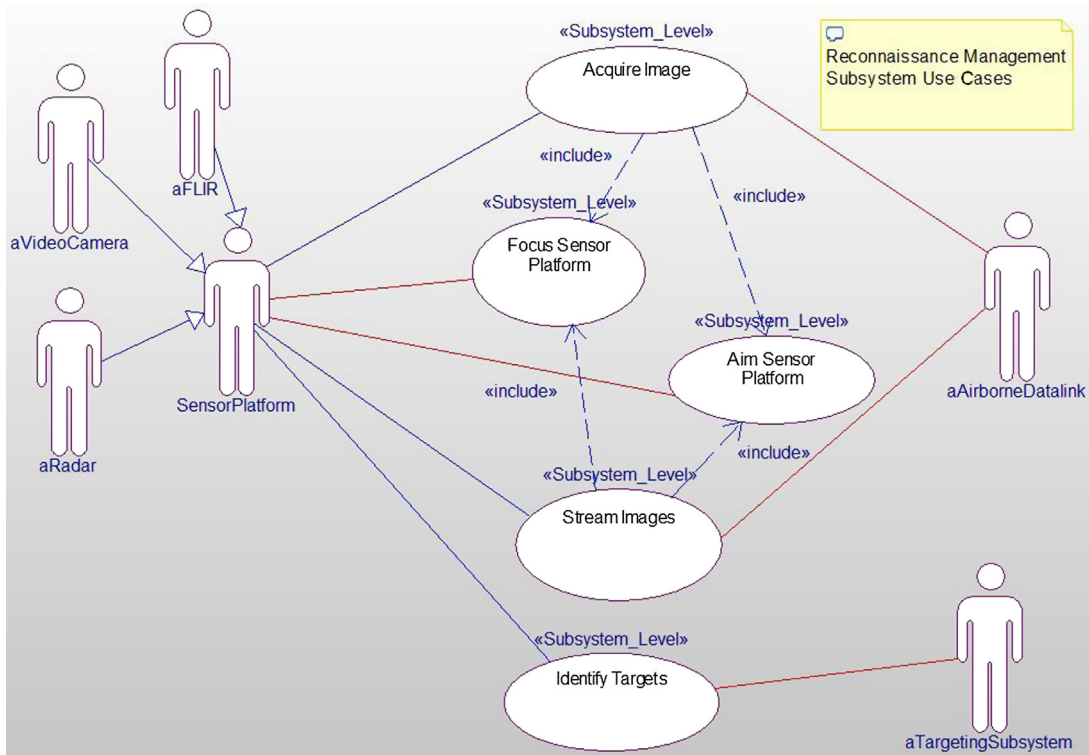


Figure 7.4
Reconnaissance Management use cases.

sustained 2- hour flight with real-time visual, infrared, or radar telemetry, with target recognition preprocessing. Communications are jam-resistant, although they need not be anti-jamming in a high ECM environment. Control commands shall be encrypted while telemetry data can be compressed but unprotected. Telemetry rates for visual telemetry support 30 frames per second (fps) at 640x400 resolution. Range of flight is meant to be fully supported within line of sight (LOS) range but since the Coyote also has the ability to be passed among different CMPCSs, its range is considerably greater than LOS. For navigation, the Coyote has on-board Global Positioning System (GPS) based navigation as well as being directly controllable from the ground station.

Unlike many smaller UAVs, the Coyote does not require specialized launch and recovery vehicles. It can use a short runway for either automated or remote-controlled takeoff and landing.

The Coyote Mission Planning and Control System (CMPCS)

Mobile CMPCS with capability to control up to four UAVs with a manned control station per UAV that fits into a smaller towable trailer. Each control station consists of two manned substations – one for controlling the CUAV and one for monitoring and controlling payloads. If desired, both functions can be slaved together into a single control substation. Control of the aircraft shall consist of transferring navigational commands, which may be simple (set altitude, speed, direction), operational (fly to coordinate set, orbit point, execute search pattern, etc.), planned (upload multisegment flight plan), or remote controlled with a joystick interface. Stable flight mechanics shall be managed by the aircraft itself but this can be disabled for remotely controlled flight.

The CMPCS displays real-time reconnaissance data as well as maintaining continuous recording and replay capability for up to 96 hours of operation for four separate CUAVs. In addition, with attack payloads, the Coyote can carry up to four Hellfire missiles with fire-and-forget navigation systems.

The Unmanned Air Vehicle (UAV) Operational Requirements

...

Mission Modes

Beyond flight modes, the CUAV shall be designed for highly flexible mission parameters. Normal mission modes include:

- Preplanned reconnaissance
- Remote controlled reconnaissance
- Area search
- Route search
- Orbit point target
- Attack

A mission can consist of any number of sequential submissions each operating in a different mission mode, depending on the current payload.

The Coyote Mission Planning and Control System (CMPCS) Operational Requirements

The CMPCS is housed in a 30x8x8 triple-axis trailer that contains stations for pilot and payload operations, mission planning, data exploitation, communications, and SAR viewing. The CMPCS connects to multiple directional antennae for communication with the CUAVs. All mission data is recorded at the CMPCS since the CUAV has no onboard recording capability. The CMPCS has a UPS that can operate at full load for up to four hours in addition to use commercial power or power generators.

A single CMPCS can control up to four CUAVs in flight with one station per CUAV. Each CUAV control station provides both pilot and payload operations with separate control substations, although both functions can be slaved to a single substation for low vigilance use.

For the reconnaissance payloads, the CMPCS shall provide enhanced automated target recognition (ATR) capability for all surveillance types – optical, infrared, and radar. While the CUAV has a rudimentary capability, the CMPCS provides much more complete support for the quick identification of high value targets in the battlefield. This capability is specifically design to identify mobile and time-limited targets that may only be exposed for brief periods of time before they go back into hiding. The system is expected to provide high clutter rejection and a low false-positive error rate. The ATR shall able to identify and track up to 20 targets within the surveillance area, with likely identification and probability assessments for each. In addition to the ATR, the payload operator can add targets visually identified from reconnaissance data or gathered from other sources. The battlefield view can be transmitted over links to remote command staff for tactical and strategic assessment.

Image Acquisition and Processing

Images may be acquired from all three sensor platforms – optical, FLIR (forward looking infrared), and SAR (synthetic aperture radar). Optical and FLIR are passive systems using emitted energy from

terrain or targets to gather information. The SAR is an active sensor, painting relatively stationary targets with energy (in the microwave range) and using the reflection of these pulses to determine reflectivity and altitude. The optical and FLIR resolution for single images may be as high as 1900x1600 resolution while real-time video for all sensor platforms is limited to 640x480 resolution at a rate of 30 fps. Streaming imagery shall be sent with enough redundancy so that complete loss of random frames shall not affect the quality of other frames. The sensor platforms may be focused at any range from 10 meters to infinite and may be zoomed up to 100x actual. The sensor platforms shall be mounted on a gimbaled assembly so that the system can be aimed without affecting the attitude of the CUAV. The FLIR includes a laser range finder to determine target range so that it can be used in fire control applications. The SAR shall emit a series of pulses meant to emulate the behavior of much larger physical aperture antenna; the images from the SAR are a combination of timed radar surface reflections to be combined into a SAR image in the ground station using Fourier transforms. Thus, a single SAR image results from a set of images each resulting from a single radar pulse from the SAR platform, but combined in the ground station. Aiming the SAR is done through the use of Doppler sharpening, limiting the amount of information that must be transmitted to the ground station to construct the SAR image. The use of two pulse emitters in the SAR platforms allows the interference patterns to be constructed providing altitude determination as well as radar reflectivity data.

Images may be compressed using lossy or nonlossy methods to minimize communication bandwidth requirements. The JPEG 2000 compression standard shall be used; for streaming video the associated MJ2 standard shall be used. The compression may be set dynamically by the payload operator to be 0% to 80% with the default setting to be nonlossy 50% compression. The imaging system is required to achieve the desired compression only within 20% of the requested due to the variances in the image contents. The selection of lossy or nonlossy compression shall be determined automatically by the imaging system, switching to lossy compression only when the desired compression rate cannot be achieved using lossless compression.

For this part of the first problem, apply the Underline the Nouns and Identify Causal Agents strategies to the Acquire Image use case, and draw the resulting high-fidelity modeling model.

⁵ It is true that you can also identify verbs to find behavior and adverbs for quality of service constraints (especially ones with quantitative values), but our initial focus is to identify the objects and classes.

Problem 7.2 Apply Services and Messages Strategies

The Services strategy looks for the services or “operational contracts” that the system containing the collaboration provides. Each of these services must either be met by a single object in the collaboration or be decomposed into nested services (possibly multiple times), each of which must be provided by a single object. The Messages strategy applies the same principle to messages and information flows; every message or information flow must be either provided or received by an object in the collaboration. In some sense, these strategies are the same. However, in the former case, the strategy is applied when the system containing the collaboration is specified in terms of a set of operational contracts (services), and in the latter case, the strategy is applied when the system is specified by information flows.

For this problem, apply these two strategies to the same use cases as before for the Roadrunner and Coyote systems. Because this level of detail is missing in the work that precedes the

exercise, both operational contract and information flow views of the Roadrunner Traffic Light Control System and CUAV systems are given below.

The two use cases under consideration for the Roadrunner system are Detect Vehicle and Fixed Cycle Time Mode.

The services for the subsystem architecture of the Roadrunner Traffic Light Control System are shown in [Figure 5-19](#).

For the UAV, consider the Acquire Image use case and apply this strategy to the identification of objects in the Reconnaissance Management subsystem. There are a number of services associated with acquiring an image, and certainly data must be passed. Draw the class diagram derived from applying this strategy.

Problem 7.3 Apply the Strategies with a Test-Driven Development Approach

All of the strategies work, albeit they work best in slightly different circumstances (and I recommend that you try them all to see how they work for you). However, *none* of them should be applied to identify all of the classes, operations, attributes, relations, and behaviors before beginning to execute, debug, generate code, and produce running versions of the application.

I see far too many people new to modeling try to do exactly that – spend weeks or months constructing a complete and accurate model before even attempting to debug or verify the model or generate code. I believe this is a major mistake because it allows design and implementation defects to be inserted but they are not identified and removed until later. It is better to avoid putting defects in the system in the first place. Test-driven development (TDD) with the nanocycle is the key means in the Harmony process to avoid entering defects into the system design.

[Figure 2.13](#) shows the structure of the nanocycle. The basic workflow is very simple:

- Identify some software elements (and define unit tests for those elements)
- Add them into the collaboration
- Generate code for the collaboration
- Verify (apply unit tests)
- Repeat until done with the use case

The key part is that this loop is very short – usually 20-30 minutes but certainly under an hour. You can't get very much design done in 20 minutes, let alone defining and applying unit test cases, but this nanocycle iteration does allow you to proceed with very few – if any – defects in the system. It is far more effective than modeling for days or weeks and *then* trying to get all the bugs out. Note that any of the object identification strategies can be used for the first step (identity some software elements), although my favorite is to walk through scenarios. At each input-output message pair I ask: What elements, processing, and data are needed to perform *this particular step*? I then decide how I would verify that I got that step correct (define the test cases), add the new elements into the evolving collaboration, generate code, and execute the tests. If they work, then I move on to the next message pair in the sequence diagram. If not, then I fix it. In any case, I move forward with very few, if any, defects in the design. It might take a week – or two – to completely implement a use case but by the time I'm done, I've executed the model and code dozens of times and have high confidence in its correctness.

Let's consider developing a medical ventilator. The set of use cases for this (slightly simplified) example is given in Figure 7.5.

Of these use cases, let's focus on the Mix Gases use case. This use case has a number of interesting scenarios, such as

- Deliver a single gas (O_2 or Air I should hope!)
- Attempt to deliver a single gas that isn't O_2 or Air (should raise an error or fault)
- Deliver a mixture of gases by selecting a gas flow from a set of wall supplies (O_2 , N_2 , He, and Air)
- Attempt to deliver a hypoxic gas mixture by selecting gas flow
- Deliver a mixture by selecting an O_2 concentration with the remainder being supplied by a "balance gas" of either N_2 or He
- Attempt to deliver a hypoxic gas mixture by selecting a low O_2 concentration with the remainder being supplied by a balance gas of either N_2 or He
- Deliver gas flow with manual switch to tank supply (for patient movement)
- Deliver valid gas mixture but the wall supply becomes deleted (auto-switch to tank supply)

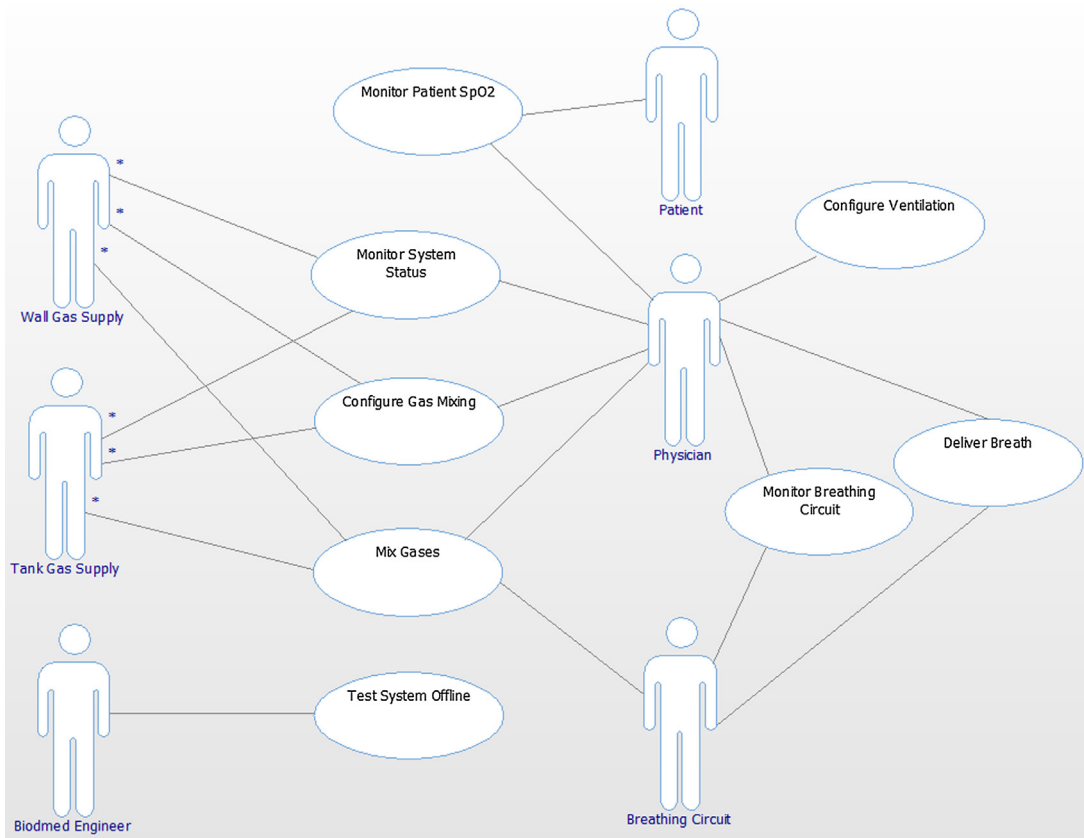


Figure 7.5
Ventilator use cases.

- Deliver valid gas mixture but a leak develops (detect, report, and correct)
- Deliver valid gas mixture but the gas mixing pump fails
- Commanded flow is out of range
- Flow parameter and O₂ concentration parameter are corrupted by EMI during delivery

And so on. With a little thought, I'm sure we can identify a few dozen scenarios to account for all of the ventilator gas mixing requirements.

Figure 7.6 shows a simple scenario for delivery of O₂ at 40% concentration using N₂ as the balance gas with a total flow of 11 L/min. The first lifeline (labeled `itsUcMixGases`) represents the collaboration of elements in the system that together perform the desired sequence. Using our TDD nanocycle approach, we can develop a functionally correct model and source code base to realize this scenario. The basic idea of is to execute the nanocycle work flow from Figure 2.13 with tiny iterations lasting less than an hour. We can use any (or all, for that matter) of the strategies we discussed earlier in this chapter. The first several messages are

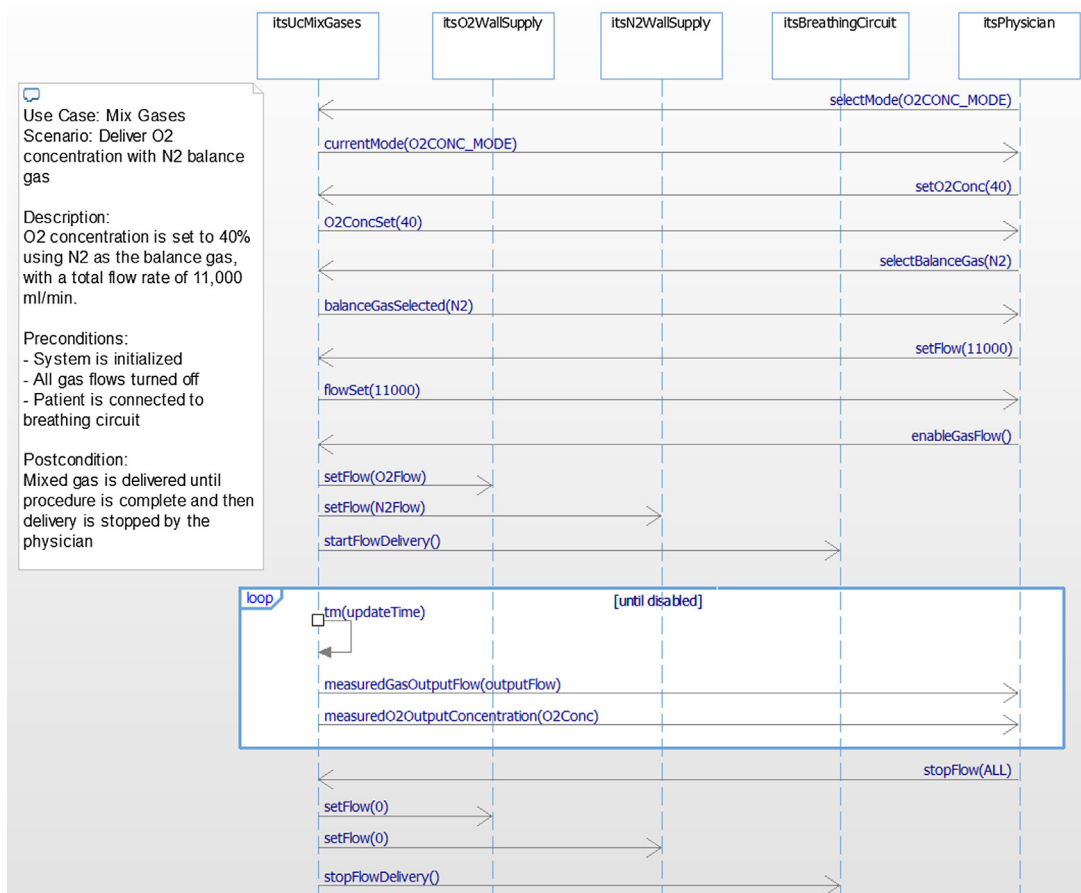


Figure 7.6
Deliver O₂ with N₂ balance gas scenario.

concerned with the interaction of the system and the physician. Let's start with the actual gas input and output flow control and build up the collaboration from there. As a proposal, here are our nanocycle iteration goals:

1. Control the input for one gas with a valve
2. Control multiple gases
3. Support balance gas setting and usage
4. Add the ability to set the gas flows to preserve an O_2 concentration with a specific total gas flow
5. Add the ability to measure and report flow delivery on a periodic basis
6. Add the ability to measure O_2 concentration at the output flow nozzle
7. Add the UI elements for the physician to set and visualize
 - a. Mode (`O2CONC_MODE` is the only mode of concern so far)
 - b. Enable/disable gas delivery
 - c. O_2 concentration (set)
 - d. Total gas flow (set)
 - e. O_2 concentration (measured)
 - f. Total gas flow (measured)

For the purpose of illustration, let's do a couple of the steps so that you can get a sense for the scope of the individual nanocycle iterations (and still keep the book light enough to carry).

Step 1: Control the Input for One Gas with a Valve

Our goals for the first nanocycle is to realize a very narrow functional purpose, but also set up the execution environment. In this case, we'll use a simulated environment compiling with the Cywin C++ compiler and debugging on the Windows platform. In an actual development environment, at some nanocycle, we would start downloading the code to the embedded target, but that is beyond the scope of this very simple example.

The class diagram for this very first step is shown in [Figure 7.7](#). The `Gas` class represents the system information about the gas status; right now that is the commanded concentration. Later, we are likely to add measured concentration as well. The `Valve` class is the device driver that commands the aperture position within the valve. The `Gas` class contains an operation that converts from the flow command to an aperture value. This value is set when the `Gas` instance enters the enabled state. It can be changed while in the enabled state by commanding a new value and then sending the `evEnable` event again.

The `Gas` class also contains a few helper functions, such as a constructor, destructor, and an operation to set the link to the `Valve` instance at runtime ([Figure 7.8](#)).

The `GasTester1` class holds the test cases for the software we're developing. In this case, it uses a state machine to drive the test cases. The two test cases (named `evTest1` and `evTest2`) are shown in the class state machine. You can see how easy it will be to add test cases by adding new events to the state machine shown in [Figure 7.9](#).

The `BuilderStep1` class is just to speed the creation of the built system. I could easily write code to create the objects in `main()` and link them together, but `BuilderStep1` does this for me automatically in Rhapsody. This is a popular idiom for quickly producing executable code with all the elements properly linked.

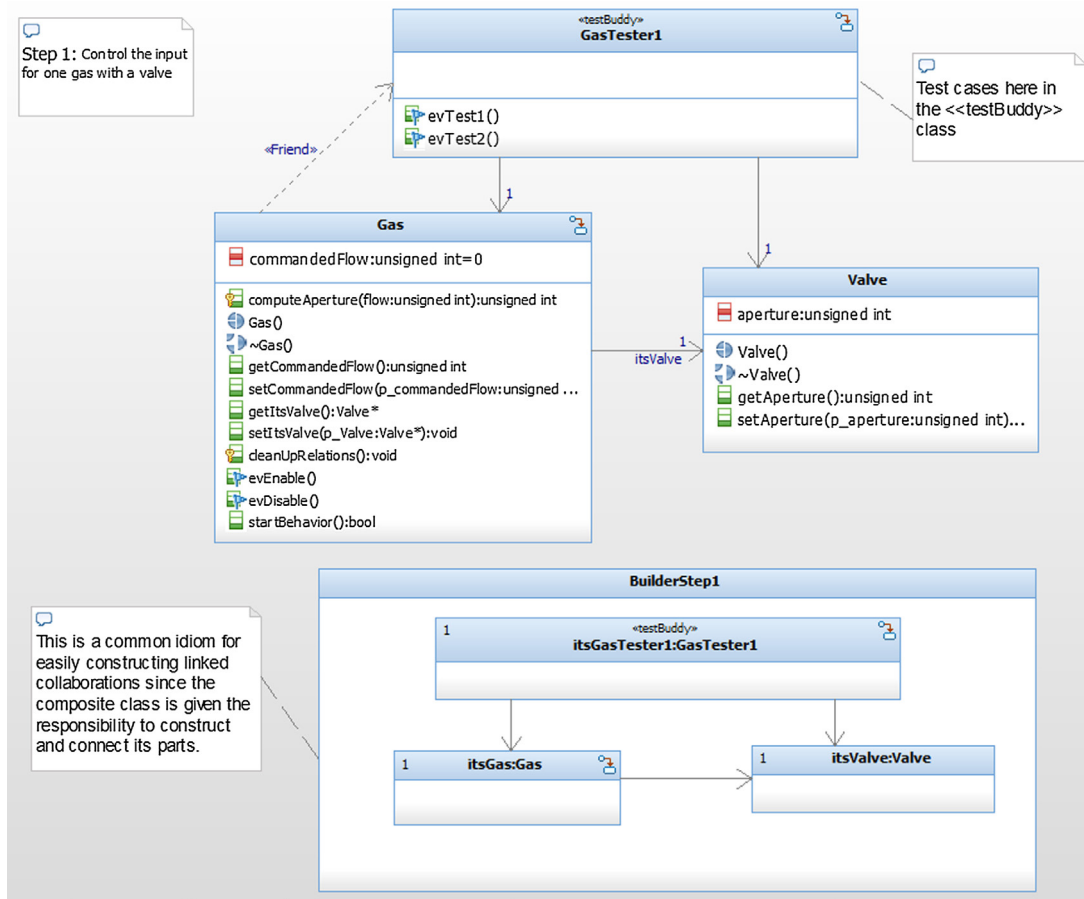


Figure 7.7
Gas mixing collaboration Step 1.

Rhapsody also allows me to generate code and use *animation* to visualize the execution of that code in UML terms. Figure 7.10 is a sequence diagram generated by running the code produced by Rhapsody.⁶

Finally, just so you can see what such auto-generated code looks like, I've included `gas.h` and `gas.cpp` in the Code Listings 7.1 and 7.2.

Code Listing 7.1 `gas.h`

```

/*****
Rhapsody : 8.0
Login : Bruce Douglass
Component : Step1
Configuration : Step1NoAnim
Model Element : Gas
//! Generated Date : Tue, 9, Jul 2013
File Path : Step1/Step1NoAnim/Gas.h
*****/

```



Figure 7.8
State machine for the `Gas` class.

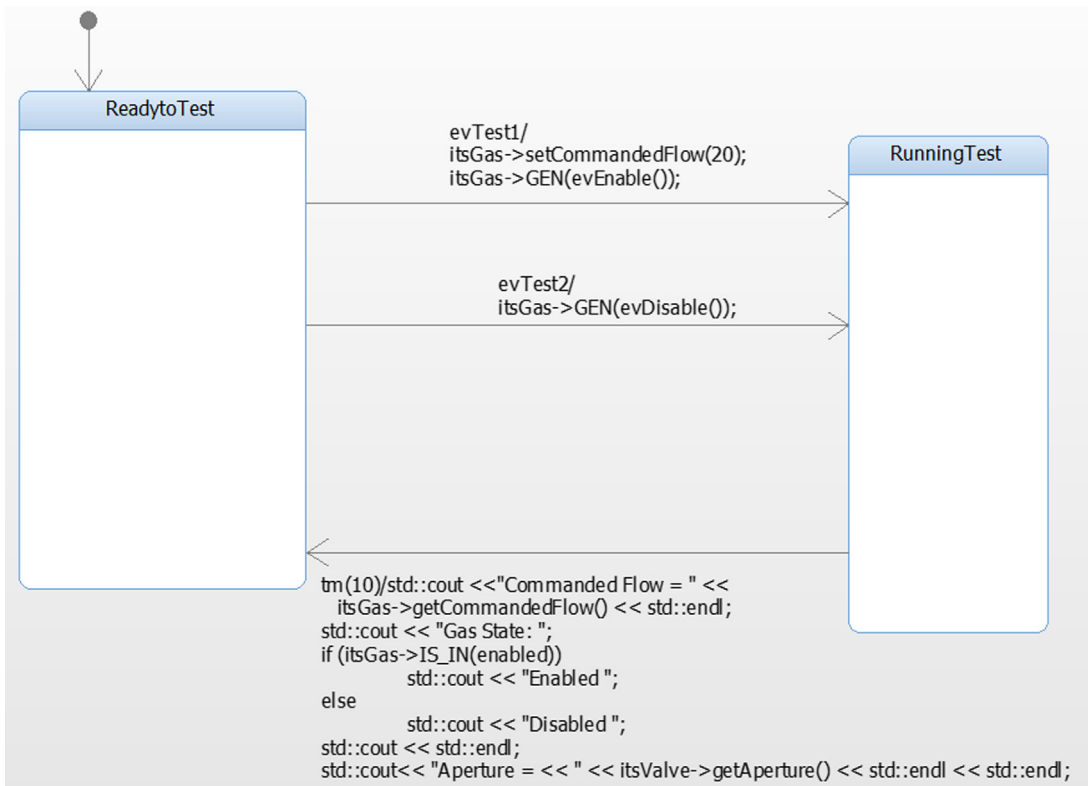


Figure 7.9
State machine for `GasTester1`.

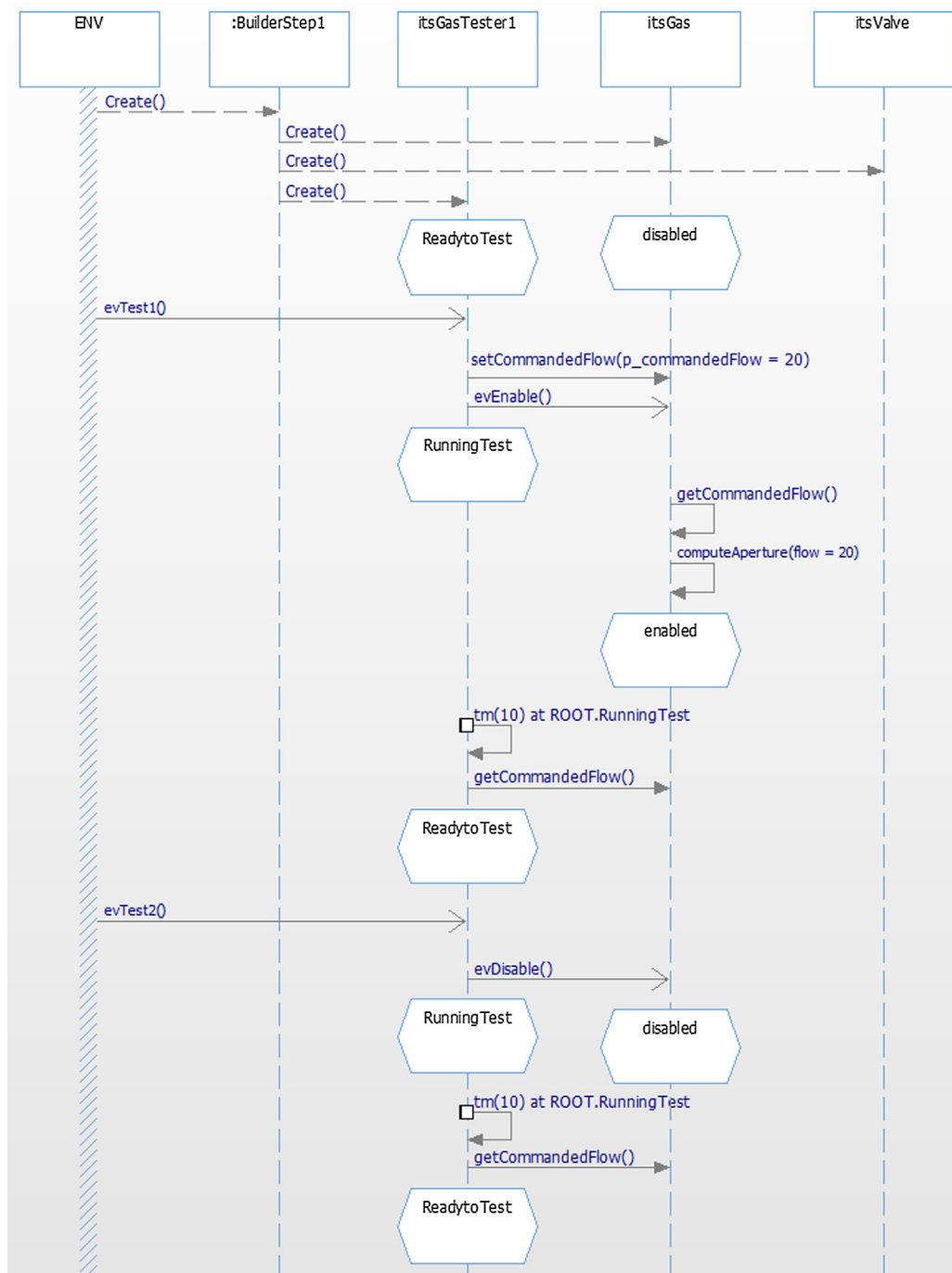


Figure 7.10
Execution of test cases for Step 1.

```

#ifndef Gas_H
#define Gas_H

///## auto_generated
#include <oxf/oxf.h>
///## auto_generated
#include "SWDesign.h"
///## auto_generated
#include <oxf/omreactive.h>
///## auto_generated
#include <oxf/state.h>
///## auto_generated
#include <oxf/event.h>
///## class GasTester1
class GasTester1;

///## link itsValve
class Valve;

///## package SWDesign

///## class Gas
class Gas : public OMReactive {
    ///      Friends      ///

public :

    friend class GasTester1;

    ///      Constructors and destructors      ///

    ///## auto_generated
    Gas(IOxfActive* theActiveContext = 0);

    ///## auto_generated
    ~Gas();

    ///      Operations      ///

    ///## operation getCommandedFlow() const
    unsigned int getCommandedFlow() const;

    ///## operation setCommandedFlow(unsigned int)
    void setCommandedFlow(unsigned int p_commandedFlow);

protected :

    ///## operation computeAperture(unsigned int)
    unsigned int computeAperture(unsigned int flow);

    ///      Additional operations      ///

```

```

public :

    ///# auto_generated
    Valve* getItsValve() const;

    ///# auto_generated
    void setItsValve(Valve* p_Valve);

    ///# auto_generated
    virtual bool startBehavior();

protected :

    ///# auto_generated
    void initStatechart();

    ///# auto_generated
    void cleanUpRelations();

    ///# Attributes      ///#

    unsigned int commandedFlow;    ///# attribute commandedFlow

    ///# Relations and components      ///#

    Valve* itsValve;    ///# link itsValve

    ///# Framework operations      ///#

public :

    // rootState:
    ///# statechart_method
    inline bool rootState_IN() const;

    ///# statechart_method
    virtual void rootState_entDef();

    ///# statechart_method
    virtual IOxfReactive::TakeEventStatus rootState_processEvent();

    // enabled:
    ///# statechart_method
    inline bool enabled_IN() const;

    // disabled:
    ///# statechart_method
    inline bool disabled_IN() const;

    ///# Framework      ///#

```

```

protected :

//#[ ignore
    enum Gas_Enum {
        OMNonState = 0,
        enabled = 1,
        disabled = 2
    };

    int rootState_subState;

    int rootState_active;
//#]
};

inline bool Gas::rootState_IN() const {
    return true;
}

inline bool Gas::enabled_IN() const {
    return rootState_subState == enabled;
}

inline bool Gas::disabled_IN() const {
    return rootState_subState == disabled;
}

#endif
/*****
File Path : Step1/Step1NoAnim/Gas.h
*****/

Code Listing 7.2 gas.cpp

/*****
Rhapsody : 8.0
Login : Bruce Douglass
Component : Step1
Configuration : Step1NoAnim
Model Element : Gas
//! Generated Date : Tue, 9, Jul 2013
File Path : Step1/Step1NoAnim/Gas.cpp
*****/

//## auto_generated
#include <oxf/omthread.h>
//## auto_generated
#include "Gas.h"
//## link itsValve
#include "Valve.h"
//## package SWDesign

```



```
///# class Gas
Gas::Gas(IOxfActive* theActiveContext) : commandedFlow(0) {
    setActiveContext(theActiveContext, false);
    itsValve = NULL;
    initStatechart();
}

Gas::~~Gas() {
    cleanUpRelations();
}

unsigned int Gas::getCommandedFlow() const {
    ///#[ operation getCommandedFlow() const
    return commandedFlow;
    ///#]
}

void Gas::setCommandedFlow(unsigned int p_commandedFlow) {
    ///#[ operation setCommandedFlow(unsigned int)
    commandedFlow = p_commandedFlow;
    ///#]
}

unsigned int Gas::computeAperture(unsigned int flow) {
    ///#[ operation computeAperture(unsigned int)
    int ap;
    ap = flow * 100;
    if (ap > 10000) ap = 10000; // clip at the high end
    return ap;
    ///#]
}

Valve* Gas::getItsValve() const {
    return itsValve;
}

void Gas::setItsValve(Valve* p_Valve) {
    itsValve = p_Valve;
}

bool Gas::startBehavior() {
    bool done = false;
    done = OMReactive::startBehavior();
    return done;
}

void Gas::initStatechart() {
    rootState_subState = OMNonState;
    rootState_active = OMNonState;
}
```

```

void Gas::cleanUpRelations() {
    if(itsValve != NULL)
    {
        itsValve = NULL;
    }
}

void Gas::rootState_entDef() {
    {
        rootState_subState = disabled;
        rootState_active = disabled;
    }
}

IOxfReactive::TakeEventStatus Gas::rootState_processEvent() {
    IOxfReactive::TakeEventStatus res = eventNotConsumed;
    switch (rootState_active) {
        // State disabled
        case disabled:
        {
            if(IS_EVENT_TYPE_OF(evEnable_SWDesign_id))
            {
                //#[ transition 1

                itsValve->setAperture(computeAperture(getCommanded
Flow()));
                //#[
                rootState_subState = enabled;
                rootState_active = enabled;
                res = eventConsumed;
            }

        }
        break;
        // State enabled
        case enabled:
        {
            if(IS_EVENT_TYPE_OF(evEnable_SWDesign_id))
            {
                //#[ transition 3

                itsValve->setAperture(computeAperture(getCommanded
Flow()));
                //#[
                rootState_subState = enabled;
                rootState_active = enabled;
                res = eventConsumed;
            }
        }
        else if(IS_EVENT_TYPE_OF(evDisable_SWDesign_id))
        {
            //#[ transition 2

```

```

        itsValve->setAperture(0);
        //#]
        rootState_subState = disabled;
        rootState_active = disabled;
        res = eventConsumed;
    }

}
break;
default:
    break;
}
return res;
}

/*****
File Path : Step1/Step1NoAnim/Gas.cpp
*****/

```

So our collaboration doesn't do very much (yet), but it didn't take very long for us to create it, produce code for it, and verify it.

Step 2: Add Multiple Gases

For this next step, we want to add the rest of the gases, including the ability to set the flow of each gas, to start and stop them together, and to ensure that we don't deliver a hypoxic gas mixture ($< 21\% \text{ O}_2$).

This is a straightforward elaboration of the collaboration of the previous model. We need four instances of the `Gas` and `Valve` classes, and a `GasManager` class to coordinate their execution. [Figure 7.11](#) shows the class diagram for the elaborated collaboration.

We explicitly show the instances as parts of the builder class in [Figure 7.12](#).

The `GasTester2` class embodies four test cases:

1. Just set O_2 to a flow of 30
2. Just set N_2 to a flow of 20 (note this should fail because it is a hypoxic mix)
3. Set He to 60 and O_2 to 40
4. Set He to 40, O_2 to 30, and N_2 to 30

This is shown in the state machine for `GasTester2`, shown in [Figure 7.13](#) (the `cout` statements have been moved to a `printStatus()` operation within the class to simplify the state machine. The `printExpectations()` operation just prints a string as to what is the expected outcome.)

The expected usage is that the `GasManager` receives commands to set flows for the various gases with invocations of `setGasFlow()` operation. Then it enables the flows with the `startFlowDelivery()` operation. The code for this operation is shown in Code Listing 7.3.

Code Listing 7.3 `startFlowDelivery()` implementation.

```

bool GasManager::startFlowDelivery() {
    computeCmd02();
}

```

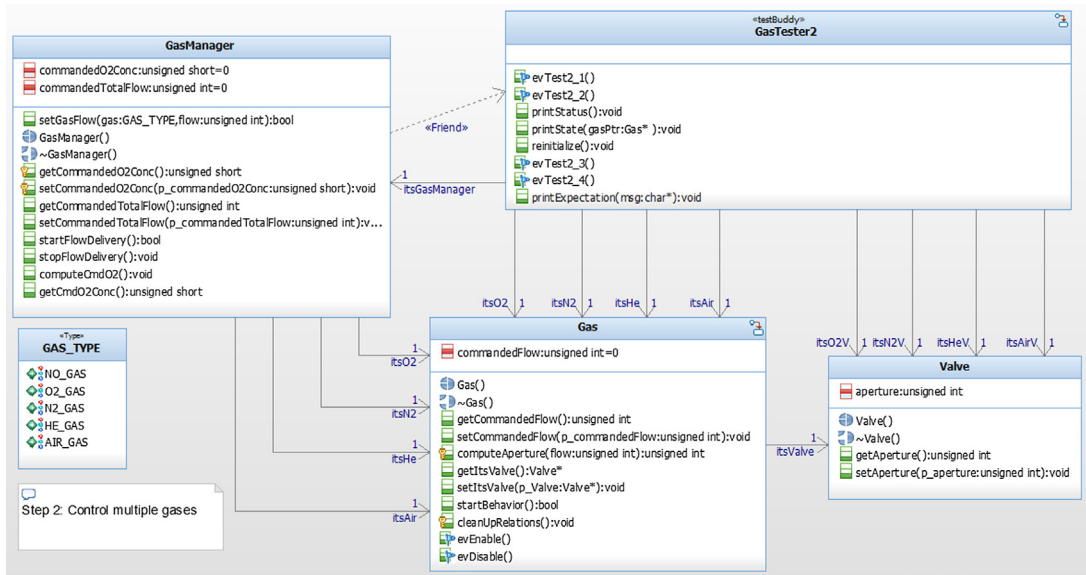


Figure 7.11
Gas mixing collaboration Step 2.

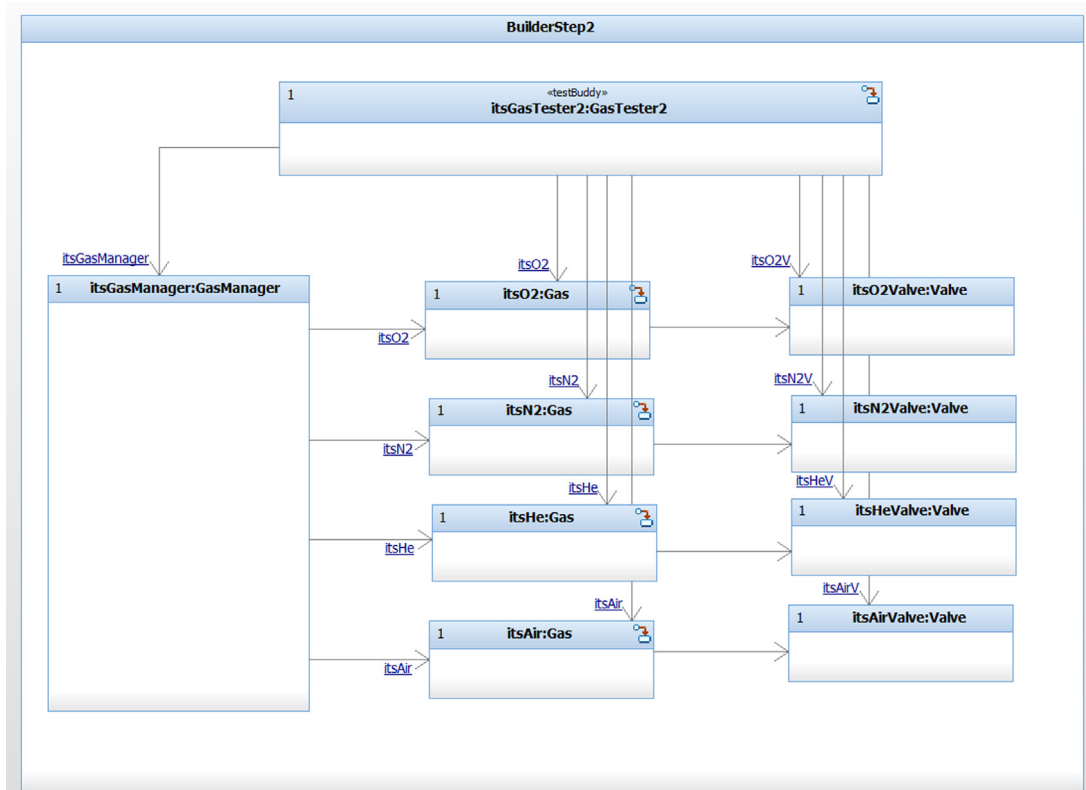


Figure 7.12
Step 2 builder class.

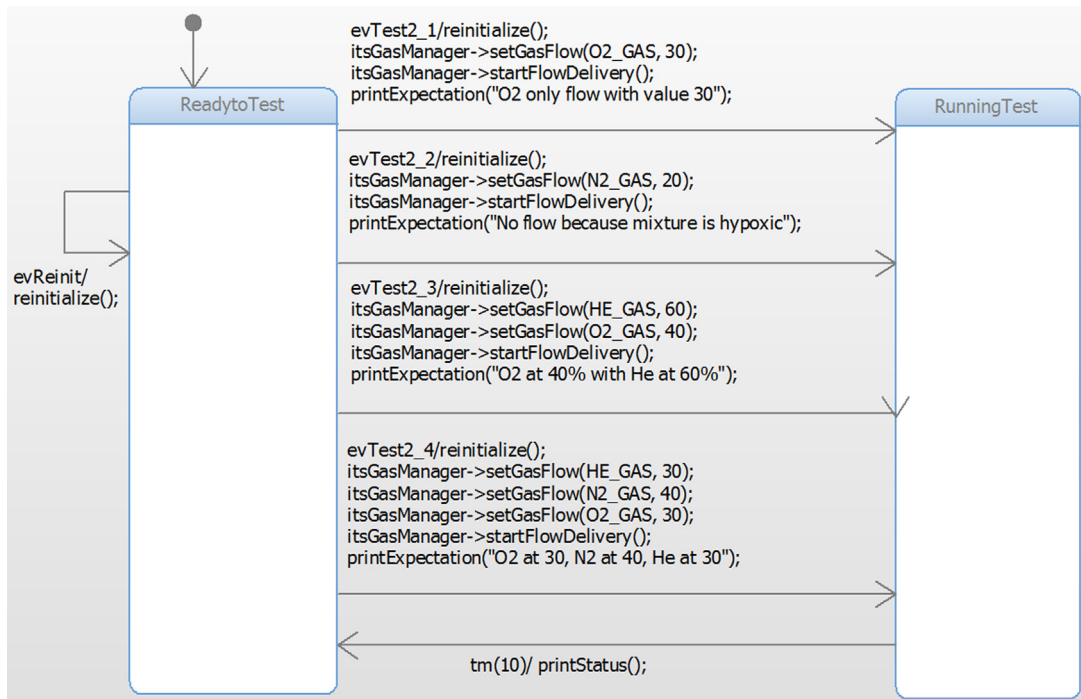


Figure 7.13
State machine for GasTester2.

```

if (getCommandedO2Conc() > 20) { // not hypoxic mixture
    if (getCommandedTotalFlow() > 0) { // nonzero flow
        itsO2->GEN(evEnable);
        itsN2->GEN(evEnable);
        itsHe->GEN(evEnable);
        itsAir->GEN(evEnable);
        return TRUE;
    }
    else {
        std::cout << "Failed at Total flow = " << getCommandedTotalFlow()
        << std::endl;
        return FALSE;
    }
}
else {
    std::cout << "Failed at Commanded O2 check = " << getCommandedO2Conc()
    << std::endl;
    return FALSE;
}
}

```

Key to this function is the execution of `computeCmdO2()`, which iterates over the `Gas` instances to get their data. The implementation of this function is shown in Code Listing 7.4.

```

C:\Users\Bruce Douglass\AppData\Roaming\Microsoft\Internet Explorer\Quick Launch\IBM\IBM R...
Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cygwinru...
n.bat" Step2.exe
Value = 100
Expect: O2 only flow with value 30
Commanded O2 Flow = 30 State: Gas State: Enabled Aperture = 3000
Commanded N2 Flow = 0 State: Gas State: Enabled Aperture = 0
Commanded He Flow = 0 State: Gas State: Enabled Aperture = 0
Commanded Air Flow = 0 State: Gas State: Enabled Aperture = 0

Value = 0
Failed at Commanded O2 check = 0
Expect: No flow because mixture is hypoxic
Commanded O2 Flow = 0 State: Gas State: Disabled Aperture = 0
Commanded N2 Flow = 20 State: Gas State: Disabled Aperture = 0
Commanded He Flow = 0 State: Gas State: Disabled Aperture = 0
Commanded Air Flow = 0 State: Gas State: Disabled Aperture = 0

Value = 40
Expect: O2 at 40% with He at 60%
Commanded O2 Flow = 40 State: Gas State: Enabled Aperture = 4000
Commanded N2 Flow = 0 State: Gas State: Enabled Aperture = 0
Commanded He Flow = 60 State: Gas State: Enabled Aperture = 6000
Commanded Air Flow = 0 State: Gas State: Enabled Aperture = 0

Value = 30
Expect: O2 at 30, N2 at 40, He at 30
Commanded O2 Flow = 30 State: Gas State: Enabled Aperture = 3000
Commanded N2 Flow = 40 State: Gas State: Enabled Aperture = 4000
Commanded He Flow = 30 State: Gas State: Enabled Aperture = 3000
Commanded Air Flow = 0 State: Gas State: Enabled Aperture = 0

```

Figure 7.14
Output from Step 2 collaboration.

Code Listing 7.4 `computeCmdO2()` implementation.

```

void GasManager::computeCmdO2() {
    // get gas flows and compute output concentration
    unsigned int o2, n2, he, air, totalflow, O2flow;
    o2 = itsO2->getCommandedFlow();
    n2 = itsN2->getCommandedFlow();
    he = itsHe->getCommandedFlow();
    air = itsAir->getCommandedFlow();
    totalflow = o2 + n2 + he + air;
    setCommandedTotalFlow(totalflow);
    O2flow = unsigned ((o2 + air*0.21));
    if (totalflow == 0)
        setCommandedO2Conc(0);
    else
        setCommandedO2Conc(unsigned (100.0 * O2flow / totalflow));
}

```

Finally, does it work? The screen capture from running the test set (Figure 7.14) suggests “Yes.”

Step 3: Mixing with a Balance Gas

Let’s take this example just one step further, and add the ability to work with a balance gas. We’ll need to identify the selected balance gas and use the total flow and desired O₂ concentration to compute the flows of O₂ and the balance gas.

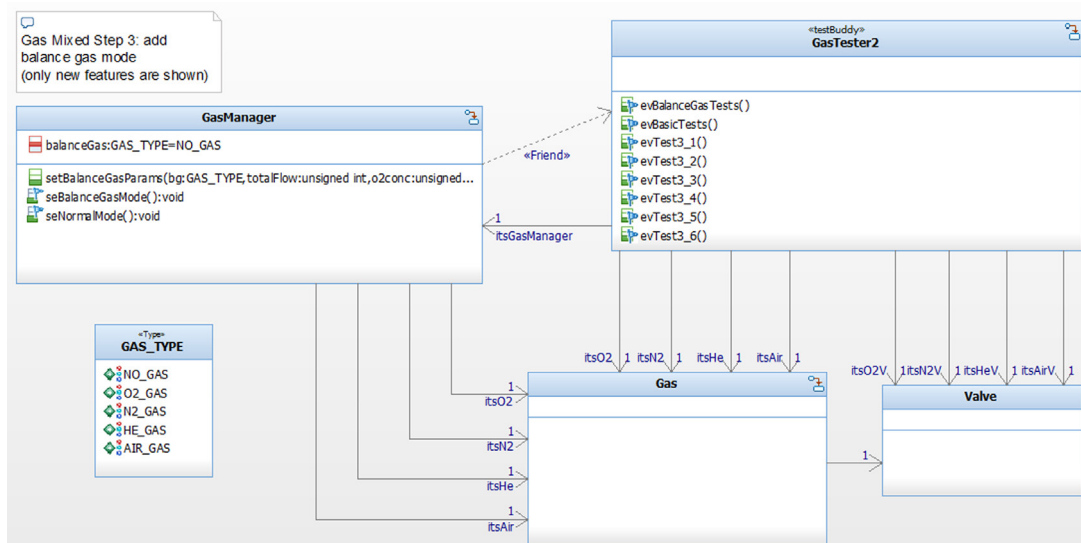


Figure 7.15
Gas mixing collaboration Step 3.

So far, the `GasManager` class allows setting individual gas flows. To set using a balance gas method, we want to

- Specify the desired balance gas (such as He or Air)
- Specify the desired O_2 concentration of the combined gas flow
- Enable the flow

To support this, the collaboration in [Figure 7.15](#) has been modified in a few minor ways. First, a state machine was added to the `GasManager` class to be in normal mixing mode or in balance gas mode. Also a new operation `setBalanceGas(g: GAS_TYPE, totalFlow: unsigned, O2Conc: unsigned)` and a new attribute `balanceGas` have been added to the `GasManager` class. And, of course, some new test cases have been added to `GasTester2`. The figure only shows the class features that are new in this step. The previous features of the classes are shown in [Figure 7.11](#).

The states are changed internally when either `setGasFlow()` is called (`GasMixer` goes to NormalMixing mode) or `setBalanceGas()` is called (`GasMixer` goes to BalanceGasMixing mode). To avoid any possibility of race conditions – which can arise when mixing synchronous and asynchronous communication among classes – the events `seBalanceGasMode`⁷ and `NormalMode` are synchronous events (“triggered operations” in Rational Rhapsody-speak).

The only tricky bit is calculating the gas flows for O_2 and the balance gas, especially when the balance gas is Air (which has 21% O_2). This is all in the `setBalanceGasParams()` operation, as shown in Code Listing 7.5.

Code Listing 7.5 Implementation of `setBalanceGasParams`.

```

bool GasManager::setBalanceGasParams(const GAS_TYPE bg, unsigned int totalFlow,
unsigned short o2conc) {
    unsigned int o2Flow, bgFlow;
    if (IS_IN(BalanceGasMixing)) {
        if (o2conc < 0 || o2conc >100)
            return FALSE; // o2conc must be in range 0..100
        else {
            initializeGases(); // reset all to zero flow
            o2Flow = unsigned( totalFlow * (double(o2conc)/100.0) );
            bgFlow = totalFlow-o2Flow;
            balanceGas = bg;
            // calculate total flows
            switch (bg) {
                case N2_GAS:itsN2->setCommandedFlow(bgFlow);
                    break;
                case HE_GAS:itsHe->setCommandedFlow(bgFlow);
                    break;
                case AIR_GAS:if (o2conc < 21)
                    return FALSE; // can't be less than 21 here
                    else { // account for the O2 that is in air
                        o2Flow = unsigned( totalFlow*(double (o2conc)-0.21)/0.79);
                        bgFlow = totalFlow-o2Flow;
                        itsAir->setCommandedFlow(bgFlow);
                    };
                    break;
                case O2_GAS:
                    if (o2conc != 100)
                        return FALSE;
                    break;
                default: return FALSE; // what gas is that???
            };
            itsO2->setCommandedFlow(o2Flow);
            return TRUE;
        } // end else
    }
    else
        return FALSE; // must be in balance gas mode
};

```

Other code changes include requiring explicit transition to either normal or balance gas modes, and only allowing `setGasFlow()` to be called in normal mode and `setBalanceGasParams()` to be called in balance gas mode, but these changes are minor.

The `GasTester2` class adds a couple of new states for testing in balance gas mixing mode and six new test cases. The additions are shown in [Figure 7.16](#). The execution of `Test3_1`, which sets up balance gas mode, and sets the delivered O_2 concentration at 30%, the balance gas as N_2 , and the total flow to 1200, is shown in [Figure 7.17](#).

The output of the same test case is shown in [Figure 7.18](#).

We'll stop here with these nanocycle iterations, but you should have the idea. Add small incremental changes to the collaboration to add a few new requirements or features, add test

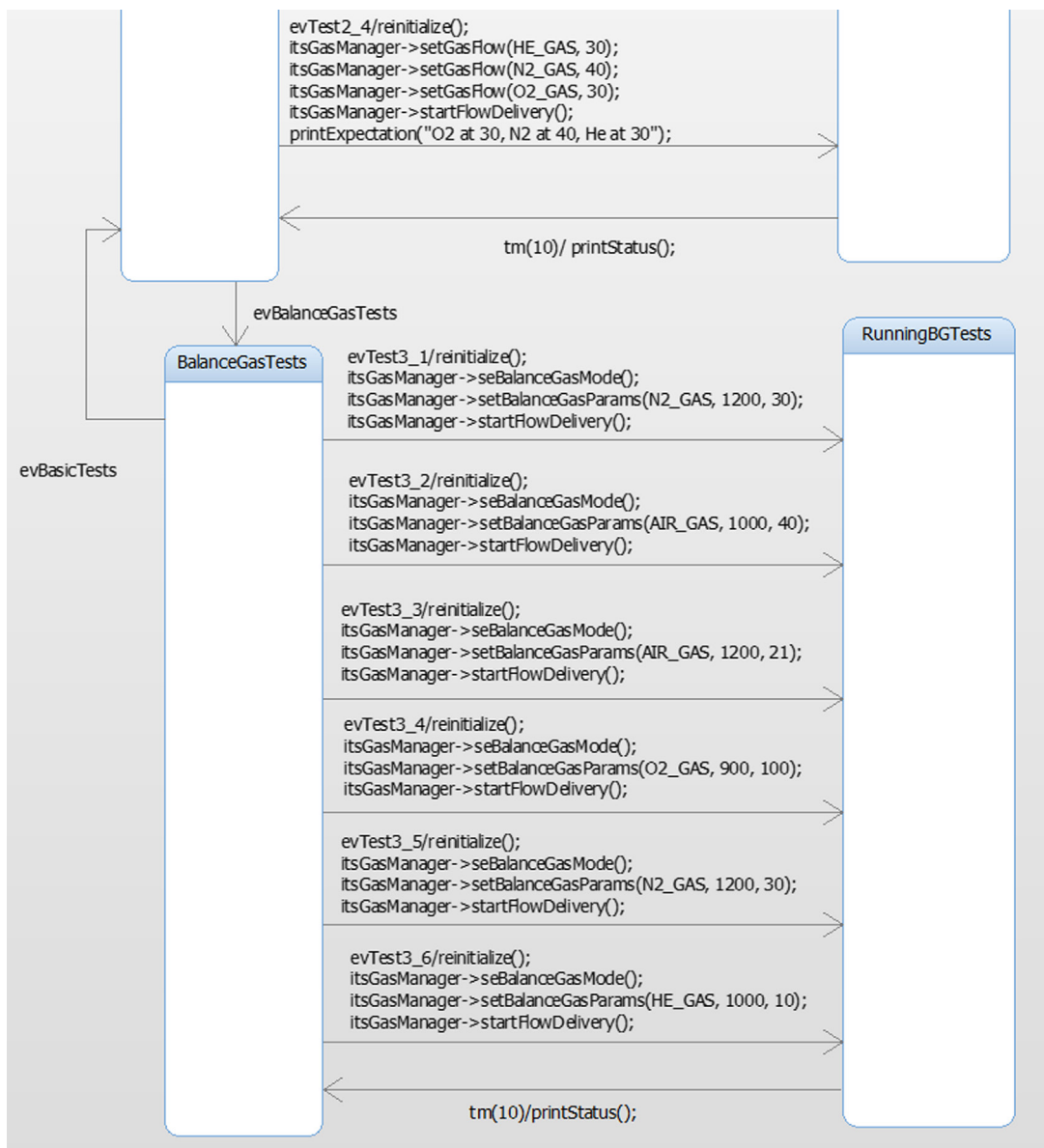


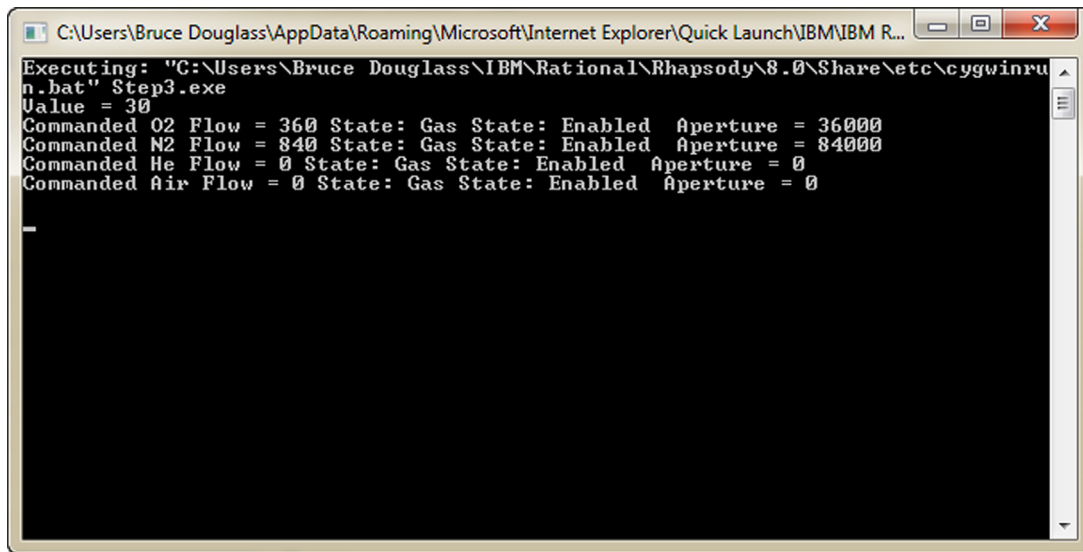
Figure 7.16
Additions to `GasTester2` state machine.

cases, generate code, verify the collaboration, and move on. Whether it takes you 20 minutes or even a couple of hours is of far less concern than building your implementation in small, verified increments.

For the final problem of the chapter, apply this nanocycle iteration technique to the Fixed Cycle Mode of the Roadrunner even though this was the subject of some earlier exercises in this



Figure 7.17
Execution of Test3_1.



```

C:\Users\Bruce Douglass\AppData\Roaming\Microsoft\Internet Explorer\Quick Launch\IBM\IBM R...
Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cygwinru
n.bat" Step3.exe
Value = 30
Commanded O2 Flow = 360 State: Gas State: Enabled Aperture = 36000
Commanded N2 Flow = 840 State: Gas State: Enabled Aperture = 84000
Commanded He Flow = 0 State: Gas State: Enabled Aperture = 0
Commanded Air Flow = 0 State: Gas State: Enabled Aperture = 0

```

Figure 7.18
Outcome of Test3_1.

chapter. Start with a clean slate and build up the model incrementally using the nanocycle approach we just went through. Try to ensure that no iteration lasts longer than 90 minutes but don't stress about the exact amount of time. This exercise might take several hours to complete and should include at minimum six or more nanocycles.

For extra credit (solution not provided), add priority and emergency vehicle management to the collaboration. Remember that these modes are activated by the detection of specific coded signals from a directional IR source; the feature is only available with above-surface vehicle detectors but it uses a different physical sensor to detect the signal than it does to detect normal vehicle arrival. Priority mode extends the Green time for the affected road by 10 seconds or (if the affected road is currently Red) it shortens the Green time of the cross road by 10 seconds. Emergency mode, however, cuts the cross traffic cycle as short as possible (turns cross traffic immediately to Yellow if it is currently to green) and when the system can transition to Green on the affected road, it remains green for as long as the emergency directional signal is active. Have fun.

For the Coyote UAV system, let's build a collaboration for the Auto Adjust Attitude use case. Here are some salient points, requirements, and concerns to consider while constructing an elaboration of this use case.

- Aircraft attitude has three independent dimensions; roll, pitch and yaw (see [Figure 7.19](#)).
- Assume, as a starting point, someone else has provided classes `KinematicModel`, `Attitude`, and `Position`, as shown in [Figure 7.20](#).
 - The `KinematicModel` class computes desired values for `Attitude` and `Position` based on current attitude and position and planned aircraft movement.
`KinematicModel::takeStep()` is called frequently by the FMS (flight management system) to compute low-level commands to the aircraft actuator control systems (such as those in the collaboration you're building)

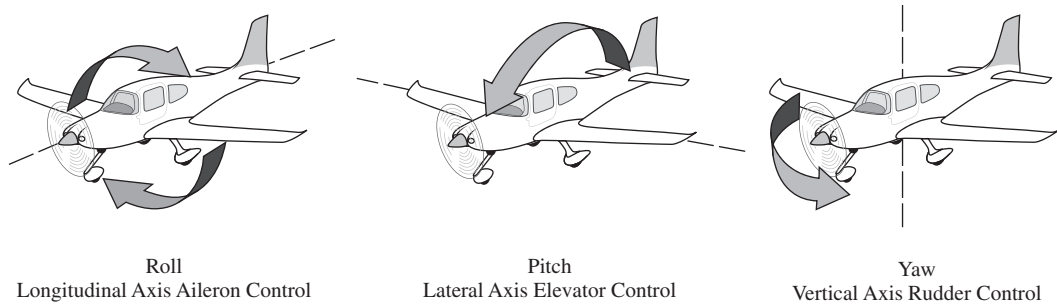


Figure 7.19
Aircraft attitude.

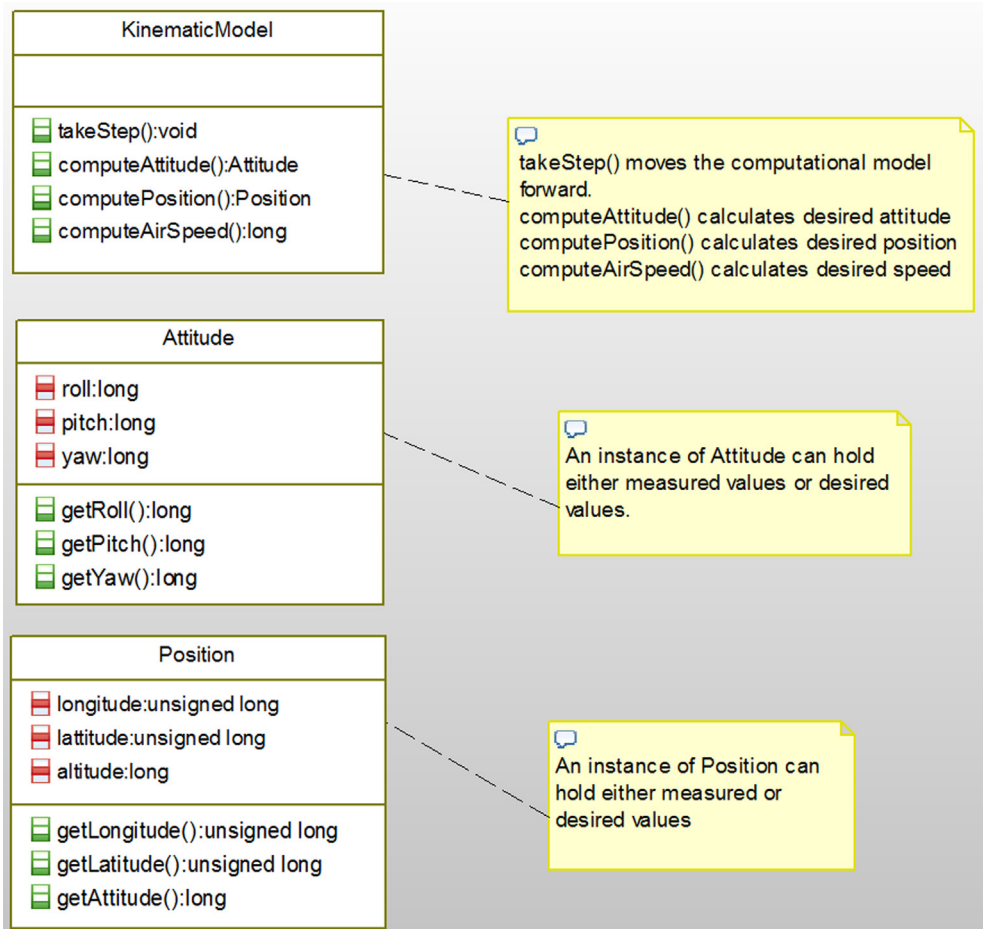


Figure 7.20
Starting point for Auto Attitude Adjust collaboration.

- Attitude is mostly data, but instances can be used to hold either desired or actual measured attitude
- Similarly, `Position` represents the position of the aircraft – either desired or measured.
- You will have to convert attitude from desired aircraft roll, pitch, and yaw into positions of the aileron, elevator, and rudder control surfaces, respectively.⁶
- Although there is a single rudder, there are independently controlled ailerons and elevators for each side of the aircraft to support maneuvers such as banking turns.
- Once the desired positions of the control surfaces are determined, you’ll have to use the hydraulic system to move those surfaces
- You’ll have to provide a means to measure roll, pitch, and yaw. In this case, we’ll use a three-axis rotating gyroscope assembly with automatic procession correction. All you need to know about it is that it will reliably give orientation in angular degrees from normal in all three axes (pitch angle, roll angle, and yaw angle).

Your task for this exercise is to create and implement a plan for several short iterations to construct the collaboration to realize the use case Auto Attitude Adjust. This should include iterations that last less than an hour in length, are verifiable via the test cases developed during that iteration, and result in executable code.

⁶ You can also just write the code by hand – the process works the same (if a little more slowly).

⁷ “se” for “synchronous event,” in case you were wondering.

⁸ It’s a bit more complicated than that, but this will do for the purposes of this book. See *Introduction to Avionics* by R. P. G. Collinson (Chapman & Hall, 1996) for more on the physics.

7.5 Summary

We have used a number of different strategies to identify the “essential” or “analysis” software elements. These strategies are only partially orthogonal; that is, they identify some, but not all, of the same model elements. In practice, the application of three or more strategies will identify all of the objects in the analysis model. These objects are “essential” in the sense that their presence is required by the nature of the problem being solved. A microwave oven had better have a microwave emitter, a track manager requires the notion of a track, and a navigation system needs the concepts of position, flight plan, and waypoint to do its job.

In practice, these identification strategies are not applied to construct a complete collaboration of the system for subsequent testing, but rather the development is broken up into small nanocycle iterations, each typically lasting less than an hour in duration. During the iteration, the collaboration is elaborated in some small way, test cases are added, code is generated, and the collaboration is verified. This workflow is repeated until all functional requirements of the use case have been met by the collaboration. This allows us to avoid most defects that commonly creep in during development.

While analysis is all about identifying elements required to meet the functional requirements, design is all about optimizing the analysis model to meet performance requirements and other

design optimization criteria. I believe that a fundamental mistake made by even experienced professional software developers is optimizing the system and code too soon – even before the software works at all. This leads to software that may be fast, but it is wrong, often in ways that would be obvious were the complexity of the “optimized” design not obscuring them.

Another advantage of this approach is independence of your analysis models from “technology churn” and the ability to adapt the analysis model to meet different optimization criteria, to use different technologies, and to execute on different platforms. In this way, the approach insulates you from technology churn and increases the interoperability of your system and its portability to new platforms. For systems that have a long life – such as military and aerospace systems – or systems that are part of product families and so have high reuse requirements, this model-driven architecture approach is both highly practical and effective. How to achieve this optimization through design is the subject of the remaining chapters.

Distribution Architecture

What You Will Learn

- What constitutes distribution architecture
- Some distribution architecture patterns

8.1 Overview

The distribution architecture focuses on how objects are allocated to different processors and the means by which they collaborate. These means include not only the underlying communications protocols, but also the policies and patterns that govern their use. As with all such design matters, different design patterns offer different technical solutions with different pros and cons.

Distribution patterns come in two basic flavors: asymmetric and symmetric. Asymmetric architectures are distinguished in that the allocation of objects to processing nodes is done at design-time. This enables a number of simplifying assumptions to be made for the collaboration, since client objects can reliably know where their servers are and how to talk with them. Symmetric architectures are characterized by runtime allocation of objects to processor nodes. This organization is more flexible than asymmetric but comes at a cost of increased complexity, required infrastructure support, and possibly runtime overhead. Another benefit of symmetric architectures can be improved reliability because objects can be relocated when a processor node fails.

Asymmetric architectures can be supported by simple infrastructures such as networks (e.g., Ethernet), buses (e.g., CAN or 1553 bus), serial links (e.g., USB or RS-232), or shared memory (e.g., dual-ported RAM). In all these cases, it is most common to have a general-purpose communication protocol (i.e., one that is not application-specific) and use this protocol to exchange application data and request services. A communication protocol is generally a logically multitiered structure in which low-level bit or byte exchange facilitates the logical exchange of application messages at a higher level of abstraction in the protocol. For example, Ethernet can provide the underlying data link and transport facilities for TCP datagrams, but a system can implement the higher-level exchange mechanism of Remote Procedure Calls (RPCs) using these lower-level facilities. The exchange of messages between remote applications takes place at the application layer in the protocol

stack, but this is ultimately realized by lower-level mechanisms in the transport, data link, or physical protocol layers.

A good system design isolates the communication details away from the application semantics as much as possible. This not only simplifies the application but also allows the application to more easily run over different communication protocols. Distribution patterns appear at or just above the application layer in the protocol stack. As with all patterns, distribution patterns optimize some aspects of the system design at the expense of others.

By far, asymmetric architectures are more usual in embedded systems. They are simpler and tend to be more time-efficient than symmetric architectures, which fits in well with the highly constrained environments in which many embedded systems must reside. However, the simple approaches often don't scale well to larger scale systems. The most common infrastructure for use in symmetric systems is to use an ORB (Object Request Broker) architecture pattern, and the most common ORB standard is CORBA (Common Object Request Broker Architecture).¹

Problem 8.1 Roadrunner Distribution Architecture

The Roadrunner Traffic Light Control System is a moderately distributed system. Sensors and actuators are scattered about the intersection and may have small processors that allow them to communicate with the intersection controller. Certainly a connection must exist between the front panel display and the intersection controller as well. The system must also connect to the remote service and management system via a wired Ethernet connection and also must support directional IR communication from high-priority and emergency vehicles.

For this problem, create class models (and diagrams) for the communication of the intersection controller subsystem to things with which it will communicate. Further, model the distribution architecture for the above-surface IR vehicle detector. Include necessary communications for the software elements, such as protocol stacks, message types, and distribution pattern-specific classes and add them to the collaboration.

For this problem, make the following assumptions:

- The distribution architecture is asymmetric.
- Each light assembly and sensor assembly will be connected to the intersection controller and each other via a serial multidrop RS-422 link. Simple messages shall be exchanged using synchronous RPCs (remote procedure calls) over the serial link.
- The multidrop RS-422 link is managed as a token ring, each node getting the master token for a brief period (not to exceed 20ms) to send messages. This ensures that the entire ring is cycled in no less than 1/5 second.
- The connection to the front panel display and remote service and management system shall be a wired Ethernet link. This is to allow reviewing and management of the configuration status and modes of the system.

¹ See *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects* by Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (John Wiley and Sons, 2000) for a detailed description of a number of different CORBA-related patterns.

- The above-surface vehicle detectors support both wired (multidrop RS-422 above) and wireless (802.11b) communications. The above-surface vehicle detectors additionally house the transceivers used to detect high-priority and emergency vehicles; these additional sensors will use the vehicle detector assembly communication apparatus to communicate with the intersection controller.
 - The transceiver shall support a simple, custom IR protocol that is focused and directional, allowing the reception (only) of messages to indicate the arrival and presence of high-priority and emergency vehicles.
 - There shall be four IR transceivers (one for each through direction) that transmit the emergency and priority vehicle messages to the intersection controller.
-

Problem 8.2 Coyote UAV Distribution Architecture

For the Coyote UAV, we'll focus on adding the distribution architecture to the system as a whole and the aircraft. At the system level, the ground system and the aircraft communicate via two telemetry links to the ground using different frequencies. The low-speed data link is encrypted and secure with a bandwidth of 100 bits per second (bps). The high-speed data link is unencrypted and is used for high-bandwidth video and image streaming. It must support 640x480 resolution images at 30 fps. If uncompressed, that means that it must support a bandwidth of $640 \times 480 \times 10 \times 30 = 92,160,000$ bps. However, lossless compression of at least 30% on average is possible, leading to a required minimum bandwidth of 64,512,000 bps. Since we need some margin for error, 100 Mbps should be adequate to meet our needs. That means that the internal network must be able to handle this bandwidth in addition to other data necessary to fly and maneuver the aircraft; a minimum of 200 Mbps of real bandwidth should be more than adequate. If we use a CDMA (Collision Detect Multiple Access) protocol, such as Ethernet, then such a network saturates at about 30% of available bandwidth. Thus we need a 1 Gbps Ethernet internal bus to ensure we have no significant delay in the delivery of the data due to collisions on the network if everything operates on the same network.²

How best to model this? As always, there are multiple approaches to solving the problem. The simplest, with Rhapsody, is to simply stereotype the interfaces that specify the contracts on the ports «CORBAInterface». Rhapsody then “CORBA-enables” these ports so that it generates the interface definition language (IDL) for the proxies that marshal and unmarshal the services invoked across those ports. But what if we don't want to use CORBA?

In some sense the most ideal solution would be to create a class called `TCPPortClass` that has the structure, relations (to the protocol stack, for example), and behavior to marshal and unmarshal service requests and responses and then send them via datagrams, with sessions and sockets, as appropriate. Then use instances of this class to replace the existing ports on the subsystem and other structured classes. This, however, is not easily supported in modeling tools since in most tools ports are not easily extensible, requiring you to write significant amounts of code to implement the solution.

Another solution is to refactor the architecture, deleting the old “semantic” ports that are specified in terms of application data and services, and replacing them with TCP/IP ports that are specified in terms of their ability to receive (or require) requests for sessions and sockets, and

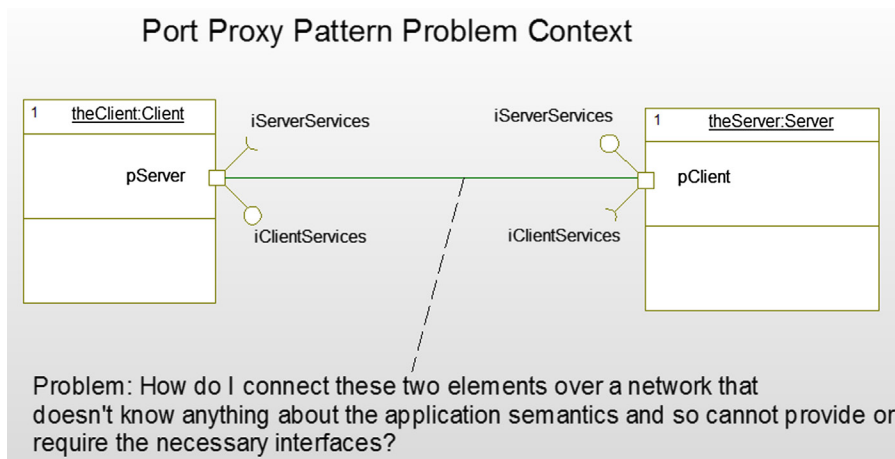


Figure 8.1
Port proxy pattern problem.

that can send and receive datagrams. This approach is less satisfactory from a model standpoint because it disrupts the continuity from the original architecture model – which is based on data and services in the system – and modifies it to be based on the design infrastructure rather than the application semantics.

In order to make the best choice, we need to ask ourselves “What do we want from our solution?” An ideal solution should

- Provide the connections to the other systems over the desired network
- Require minimal modification (in the best case, no modification) to the existing collaborations and to the existing architecture
- Be easy to reapply to different distribution architectures, networks, and data links

One more approach, and the one recommended here, is to create proxies using a variation of the proxy pattern called the *port proxy pattern*. A proxy is a “stand-in” for another element and in this case understands how to marshal and unmarshal the service requests and responses into TCP/IP datagrams, set up connections and sessions, and so on. The port proxy pattern is specifically designed to address the issue of connections across ports that are to be implemented using a networking infrastructure.

The port proxy pattern problem is shown in [Figure 8.1](#).

The solution – adding in proxies to manage the communication over the networking infrastructure – is shown in [Figure 8.2](#). It is a “simple” matter of creating proxies that manage the networking communications. These proxies support the required application semantic interfaces and then create the necessary network data structures (such as datagrams) and execute the necessary network behaviors (such as creating sessions and sending and receiving network packets) to achieve the communication goals. The primary advantage of this pattern is that the original objects (in this case, the subsystems) don’t need to change *at all* to support the network infrastructure. The change is isolated to the links among the objects – now they are connected via proxies instead of directly. The original architecture still executes

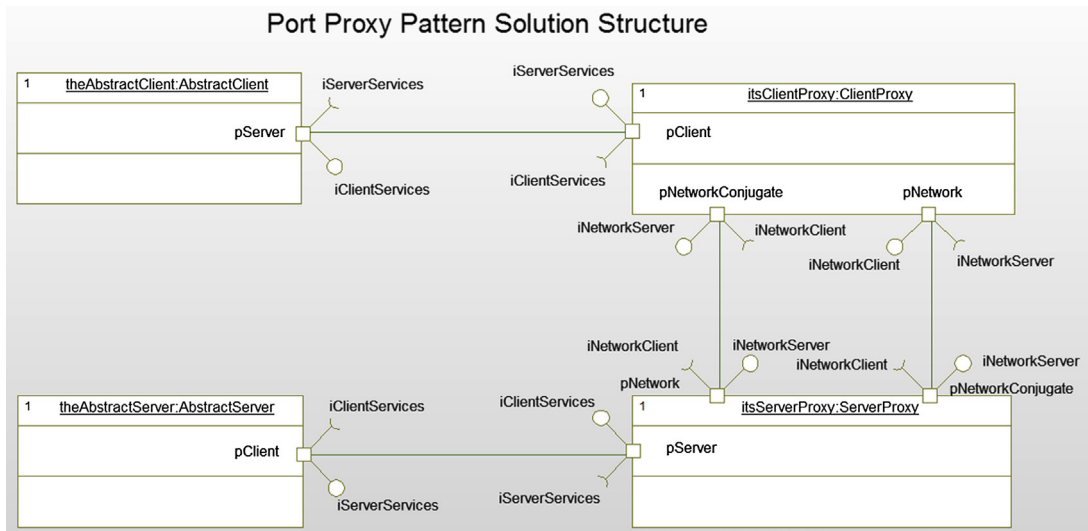


Figure 8.2
Port proxy pattern structure.

because the interfaces haven't changed and the new architecture with the proxies added works as well.

Note that the proxies are connected via a pair of ports, each of which is a conjugate³ of the other. This is required because you cannot both offer and require the same set of services with the same port. Using paired conjugates makes the connections symmetric, simplifying the design.

Your task for the CUAV system is two-fold. First, draw the distribution architecture that connects the ground system and aircraft. Second, apply this port proxy pattern to the internal aircraft architecture for subsystem-subsystem links. Draw the related subsystems and connect them via proxies.

² If the 1 Gbps network is unpalatable, then you might decide to either run multiple networks (one for command and control and another for high-speed data), or reduce the fidelity of the video.

³ In this context, by *conjugate*, I mean that what one port offers, the other requires, and vice versa.

8.2 Summary

This brief chapter focused on identifying and modeling the distribution architecture. For the simpler exercise, we defined the distribution architecture for the intersection control system. We defined the three networks (RS-422 multidrop token ring), wired (Ethernet) and wireless (802.11b) networks. Additionally, we detailed the serial connection to the various distributed sensors and lights.

The CUAV is a much larger problem. This system supports both high-speed and low-speed connections between the ground system and the aircraft. We defined that structure, although not the details. Within the aircraft, we applied the port proxy design pattern to connect the subsystems together in such a way that we needn't modify any software in the subsystems to support the communication, and so that it is easy to change and adapt.

The next chapter focuses on the concurrency and resource architecture. As with all architectural viewpoints, the solutions in this space are governed by design patterns. After that, we'll drill down in the optimization at a more detailed level to finish out the book.

Concurrency and Resource Architecture

What You Will Learn

- What constitutes the concurrency and resource architectural model
- Harmony concurrency and resource architecture workflow
- Strategies for task identification
- Concurrency modeling in UML

9.1 What is the Concurrency and Resource Architecture?

The concurrency and resource architecture is one of the key architectural views identified in Chapter 5. Unlike some of the architectural views, it is almost entirely a software architectural concern. Concurrency is a key aspect of almost any real-time and embedded system because it so directly influences its performance.

Concurrency refers to the simultaneous execution of action sequences. *Pseudo-concurrency*, more common in software, refers to the simulated concurrency achieved when you execute action sequences on the same single-threaded execution environment by switching between tasks when it is appropriate. The task that is currently executing is said to have *focus* and other tasks may be *idle* (not waiting to run), *ready* (waiting to run but not having focus), or *blocked* (waiting for a resource to be release before it can become ready).

A *concurrency unit* is a sequence of actions executed so that the sequence of actions *within* the concurrency unit is fully deterministic (although possibly complex due to branching) but the sequence of actions *between* concurrency units isn't known except at explicit synchronization points. Concurrency units are often referred to as *processes*, *tasks*, or *threads*. We will treat all concurrency units the same here because the only difference is really one of scope.

Each concurrency unit typically has its own stack and may have its own heap. The really important thing is that the sequence of actions within a concurrency unit is fully known but the sequence of actions between concurrency units is “don't know, don't care.” This is the power of tasks – the ability to divide computations or functionality up into sequences in which the relative order of action execution between these sequences is unimportant. It's like having minions (and who can't use a few more of those?) that contribute to the overall plan without having to interact with the other minions. Of course, the overall plan has to be designed so that this can happen efficiently and without undue blocking or deadlock.

Tasks have all kinds of properties (collectively referred to as *concurrency metadata*). Two of the most fundamental are *urgency* and *criticality*. *Urgency* refers to the “time stress” of the task. If you model a task as having a *deadline* (a point in time at which the completion of the task action sequence becomes irrelevant or incorrect), then the urgency is the nearness of the deadline. The closer the deadline, the more urgent the task. *Criticality* refers to the importance of the task completion.

Figure 9.1 shows what I mean. The abscissa shows the time axis while the ordinate shows the criticality. The curve represents the *utility function*; this is the value to the system of the completion of the task’s actions as a function of time. Note that the utility function in this case is a step function. In real life, a task’s utility is usually not a step function but a more general curve; people model the utility as a step function because this makes the math easier.

Since we typically have multiple tasks ready to run, some criterion must be applied to determine who actually gets to run, if you’re running the set of tasks on the same execution resource (CPU). Scheduling patterns include

- **Interrupt Driven (first-come, first-served)**
A set of tasks is driven entirely by incoming events without oversight. The Good: Simple and highly responsive to incoming events. The Bad: Doesn’t scale to many tasks, event handlers must be short, very easy to lose incoming events by not completing within the time window, difficult to share resources and data.
- **Cyclic Executive**
A set of tasks is executed cyclically, each task running to completion. The Good: Very simple and very predictable. The Bad: Suboptimal in terms of responsiveness to incoming events, difficult to coordinate tasks and share data, all tasks run at the same rate.

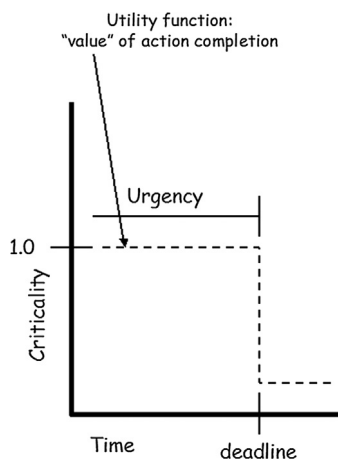


Figure 9.1
Urgency versus criticality.

- **Timed Round Robin**
A set of tasks is assigned an equal time slice and executes within that time slice. The Good: Simple, starvation-free, and everybody gets to play. The Bad: A single misbehaving task can prevent any other task from executing; not responsive to incoming events.
- **Priority-Based Preemption**
A set of tasks is scheduled on the basis of *priority*. The Good: Responsive to incoming events, handles high-priority processing at the expense of low priority processing. The Bad: A bit more complex to implement, sharing resources leads to priority inversion, naïve implementation can lead to unbounded priority inversion, analysis is more difficult.

The key to using priority-based preemption is the selection of the priority of the tasks. A task's priority is an assigned numeric value that is used to determine which, among a set of tasks currently ready to run, will actually execute. The task with the highest priority “wins” and the other tasks remain in the ready queue until the higher priority task completes or blocks.

Priorities can be assigned on the basis of urgency or criticality, or some combination of the two. The most common priority-based scheduling schema is *Rate Monotonic Scheduling* (RMS). RMS assigns the priority on the basis of the recurrence property metadata. RMS assumes that

- Tasks are periodic (i.e., time-based event arrival initiating the task)¹ and are characterized by
 - Frequency (or, reciprocally, a period)
 - Variation around that frequency (“jitter”)
- The task deadline is at the end of its period.
- The task is infinitely interruptible (i.e., able to be interrupted at any point in its execution by the start of a higher priority task).

RMS assigns a higher priority to a shorter period. RMS is both optimal and stable. By “optimal” we mean that if you can meet the deadlines for the set of tasks by any other scheduling algorithm, you can *also* meet them with RMS. By “stable,” we mean that in an overload situation in which some deadlines will be missed, you can predict which deadlines these will be – the lower-priority task deadlines. RMS is inherently urgency based. Priority-based scheduling methods are more difficult to analyze than cyclic executives. For this reason, most avionic systems use cyclic executives in spite of their demonstrably suboptimal performance because of the relative ease with which they can be flight certified. Most real-time operating systems (RTOSes) support all of these scheduling approaches.

Figure 9.2 graphically shows the important concurrency metadata concepts.

A resource is a typically passive element that must be shared among tasks but is constrained in some way. The most common constraint is that it can only be accessed (safely) by a single

¹ Aperiodic tasks may be modeled as if they were periodic using the minimum interarrival time (the minimum time between task invocations), although this can result in overly strong (and expensive) requirements for hardware.

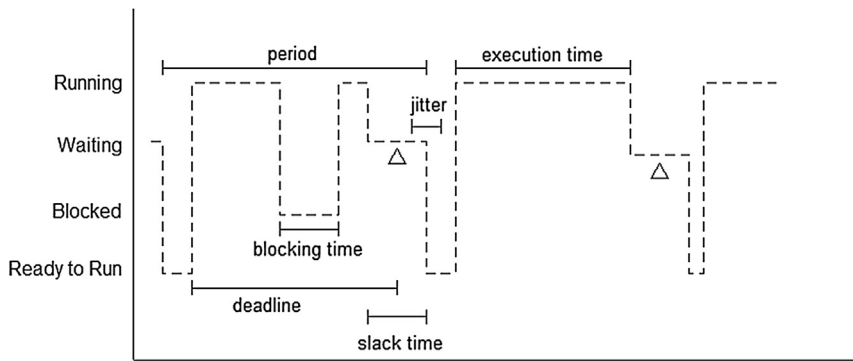


Figure 9.2
Concurrency metadata.

task at a time – the so-called *mutual exclusion problem*. Most of the resource sharing patterns² serialize the access to a resource. The standard ways of doing this are

- **Critical regions**
This approach disables task switching during the access to the resource. The Good: Simple approach solves the mutual exclusion problem. The Bad: Requires that higher-priority tasks cannot preempt the lower-priority task (creating priority inversion) even if the higher-priority task doesn't need the resource, and breaks the “infinite interruptibility” rule.
- **Guarded call**
This approach adds a lock, such as a mutex semaphore, to a resource. This means that a task accessing the resource must block if that resource is currently in use, and becomes unblocked when the use of that resource is removed. The Good: Approach solves mutual exclusion problem while permitting normal priority preemption for tasks that don't need the resource. The Bad: Causes priority inversion because the higher priority task that wants to use the resource must block to allow the lower priority task to finish its use of the resource. The Worse: Naive use can lead to *unbounded priority inversion*. The Worst: Can lead to deadlock in some circumstances.
- **Queuing requests**
This is an asynchronous approach – it serializes all requests for the resource into a queue and handles them when it gets around to it. The Good: Approach solves the mutual exclusion problem and permits tasks to execute in their normal priority scheme. The Bad: Delays access to the resource by queuing the requests, so may not result in a timely response to the request.

A bit of explanation of priority inversion is in order. If you look at the task diagram³ in [Figure 9.3](#) you can see two «active» classes, `PacingTheHeartTask` and the `BuiltInTestTask`.

² See *Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* by Bruce Powel Douglass (Addison-Wesley, 2004).

³ A task diagram is a UML class diagram whose mission is to show some aspect of the concurrency architecture. In the case of Figure 9.3, the active classes represent the tasks while the class with the stereotype «resource» is used by these active classes.

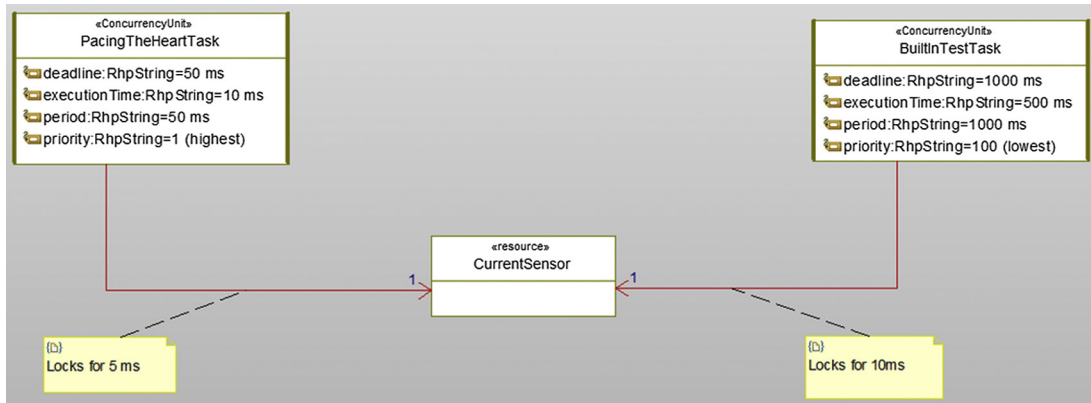


Figure 9.3
Priority inversion example 1.

I've added a stereotype `<<ConcurrencyUnit>>` to add the concurrency metadata to those classes and used Rhapsody display options to show them on the diagram. The relevant metadata are

- **Deadline** – the time after the start of the task by which it *must* be complete or the system fails (in this example, the deadline is at the end of the period for all classes)
- **Execution time** – the length of time the task requires focus to complete execution
- **Period** – the length of time between task invocations
- **Priority** – a scalar value used to select the ready task to execute (the lowest numeric value is the highest priority in this case)

In [Figure 9.3](#), the highest priority task is that represented by `PacingTheHeartTask`. It runs every 50 ms and requires 10 ms to execute, 5 ms of which locks resource `CurrentSensor`. `BuiltInTestTask` is the lowest priority task; it runs every 1000 ms and requires 500 ms to execute, during which it locks the resource for 10 ms. Let's now imagine that `BuiltInTestTask` is running and locks the resource `CurrentSensor` and right then `PacingTheHeartTask` becomes ready to run. What happens?

`BuiltInTestTask` stops running and the `PacingTheHeartTask` runs because it is of higher priority. However, when `PacingTheHeartTask` gets to the point where it needs to invoke services of the resource `CurrentSensor`, it must block (be put on the blocked task queue) and `BuiltInTestTask` must be allowed to run so that it can complete its work with the needed resource. Once `BuiltInTestTask` releases the `CurrentSensor` resource, `PacingTheHeartTask` is pulled off the blocked queue and can then lock the needed resource and complete its work. While the `BuiltInTestTask` is executing, even though `PacingTheHeartTask` is ready to run, the latter task is said to be *blocked* and the system is in a condition of *priority inversion*. Blocking occurs when a higher priority task wants to run but it cannot because a needed resource is unavailable.

Will the high priority task still meet its deadlines? The worst case is that `PacingTheHeartTask` becomes ready to run just as soon as `BuiltInTestTask` locks the resource. That means it will be

blocked for the full 10 ms that `BuiltInTestTask` locks the resource. The total time for completion of `PacingTheHeartTask` is the execution time (10 ms) plus the blocking time (10 ms for a total of 20 ms), which is still less than the deadline (50 ms). So, yes, deadlines are met in this situation.

Let's elaborate this system slightly. [Figure 9.4](#) adds more tasks into the concurrency architecture. They have various periods and execution times but all are assigned priorities based on RMS. None of these tasks use the resource `CurrentSensor`. Can `PacingTheHeartTask` still meet its deadlines?

Let's consider the worst case: `BuiltInTestTask` locks the resource and then the next lowest priority task (`CommTask`) becomes ready to run. It doesn't need the resource so it isn't blocked so it runs. Then, the other tasks become ready to run in quick succession: `QRSWaveAnalysisTask`, `HolterMonitoringTask`, and finally `DataSmoothingTask`. `BuiltInTestTask` cannot run until all the higher priority tasks have completed, which will take a combined time of 370 ms (80 + 100 + 100 + 90). At this point, `PacingTheHeartTask`, the highest priority task in the system, runs. It preempts all the lower priority tasks and runs until it reaches the point at which it needs the resource and blocks. But at this point, it will be 380 ms (370 + 10) before it can unblock. `PacingTheHeartTask` fails to meet its deadline and the pacemaker patient now has a problem.

The fact that there are an unlimited number of intermediate-priority tasks that can prevent `BuiltInTestTask` from releasing its resource is known as *unbounded priority inversion*. This

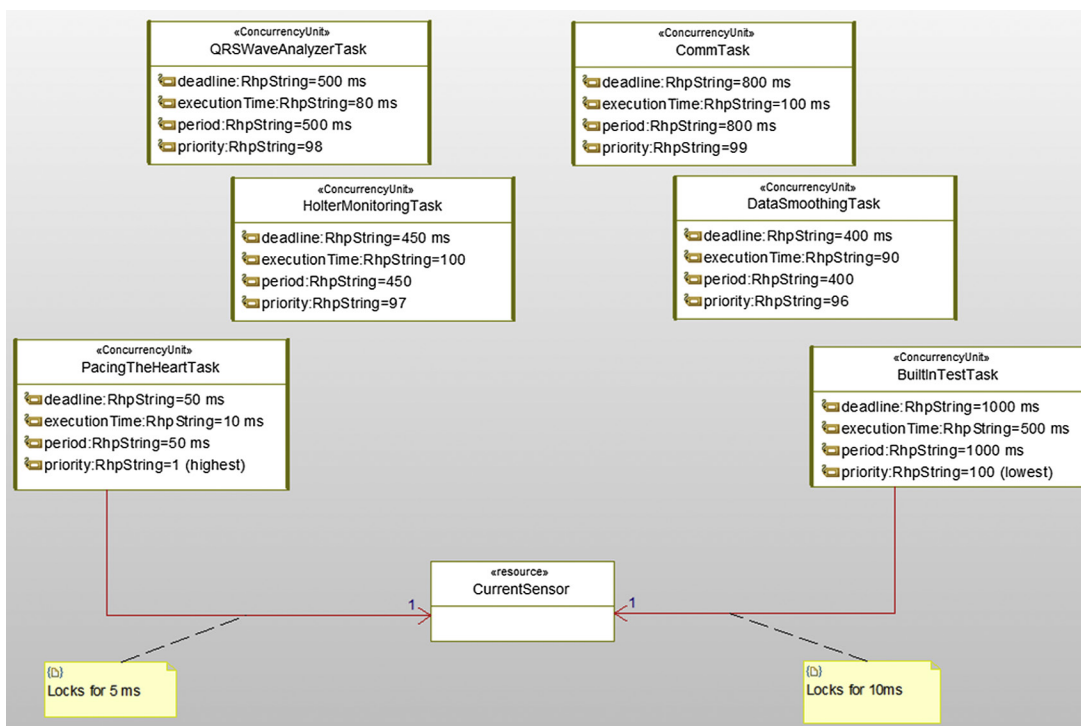


Figure 9.4
Priority inversion example 2.

can be a subtle and elusive problem to detect via testing and can be a serious problem with naïve implementations of priority-based scheduling. There are a number of technical solutions (design patterns) that address this problem, including

- Critical regions – don’t allow task switching during the time a resource is blocked.
- Avoid the use of locking resources; for example, run the resource in a separate thread and use an asynchronous rendezvous to request services.
- Use a priority inheritance-based solution.⁴

All these solutions have pros and cons, but they all (more or less) solve the problem of unbounded priority inversion. You should note, however, that if you use blocking semaphores on resources, you will *always* have priority inversion, but it is possible to limit it to a single level with diligent design. It is certainly easier when tasks are independent and don’t share resources, but that is fairly rare in real systems.

That having been said, in most systems, tasks do depend on the order of execution of other tasks but usually only in limited and well-defined ways. Consider [Figure 9.5](#). In this figure we use an activity diagram to show the overall process for making coffee. Between the fork and the join, we show two task threads, which are independent. One of these is focused on boiling the water while the other is focused on preparing the coffee serving. However, both these tasks must be at the proper synchronization point (i.e., the water is hot and the coffee serving is prepared) before processing (i.e., mixing the coffee into the water) can proceed. Whether the water is heated before the coffee is removed from the freezer or not doesn’t matter – the order of execution of those actions is independent. However, there is an explicit synchronization point at which processing cannot proceed unless both tasks are at a certain point in their execution sequences.

Let us now consider modeling the concurrency architecture of a simple example. [Figure 9.6](#) is a simple control system. At the top left of the figure, the environmental data is acquired, filtered, and averaged from four different sensors. The data is acquired under the control of the `SensorManager` at a rate of 10 Hz (i.e., 100 ms rate) and this is used to update the `AverageSensedData`. The controller oversees the entire system and every 30 seconds it may choose to adjust the control points for the high-speed closed loop controller of the actuators based upon the averaged data. The high-speed closed loop control updates the output at 100 Hz (i.e., 10 ms rate) and does high-speed control to reduce the error between the monitored average values and the control set points. So how do we turn the analysis model in [Figure 9.6](#) into a design model with concurrency?

⁴ There are a number of these, but all are based on the notion of a priority ceiling attribute of the resource, which is based on the priority of the highest-priority task that *could ever* lock the resource. While the lower-priority task locks the resource, its priority is temporarily elevated to the resource’s priority ceiling. The different solutions vary on exactly when this priority elevation takes place. See my book *Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* (Addison-Wesley, 2004) for more details.

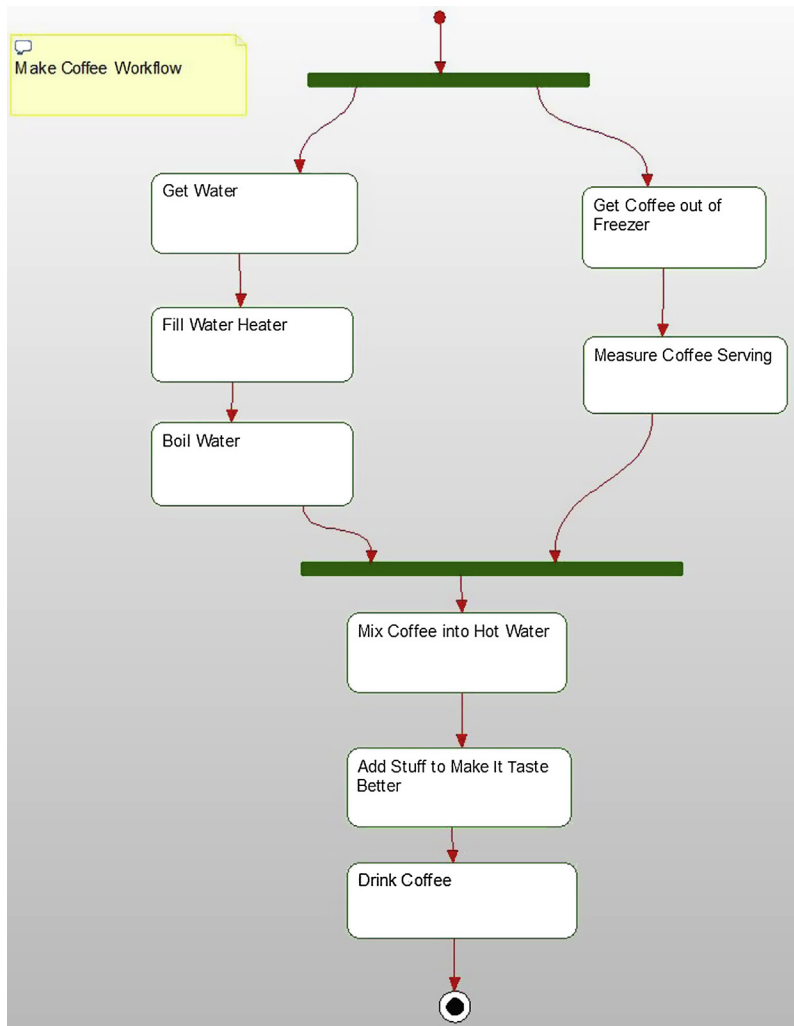


Figure 9.5
Making coffee.

In this example, we'll use the recurrence properties to determine the task set. Actions occur at three different periods: 10 updates/sec for data acquisition, 100 updates/sec for closed loop control, and one update every 30 seconds for the set points. Each of these processing sets will become an «active» object managing its own task. The `AverageSensedData` class is a resource shared among the task threads. For the resource sharing policy, we'll use the *Guarded Call* pattern to enforce serialization of the access to the data.⁵

⁵ In Rhapsody, this is very straightforward. Simply set the concurrency property of the relevant operations from "Sequential" to "Guarded" and Rhapsody will automatically create a semaphore to manage the serialization of access to the methods.

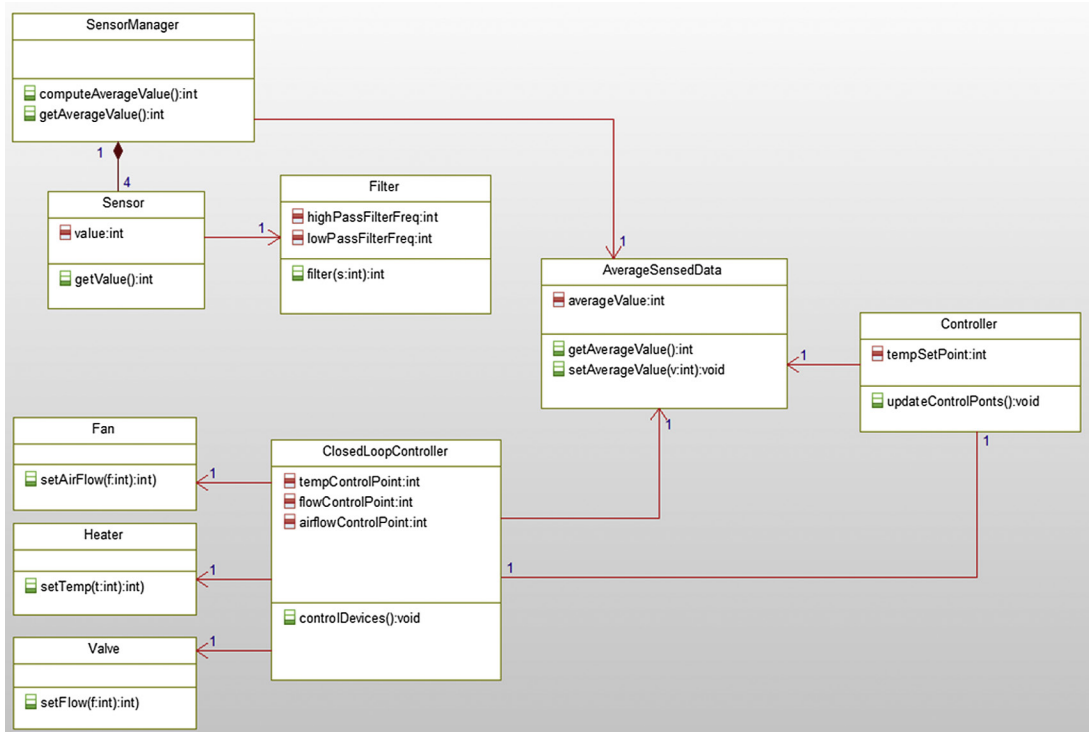


Figure 9.6
Control system.

To make this into a task diagram, these classes must be made parts of the relevant «active» objects.⁶ Lastly, add ports to connect the objects across the thread boundaries.

A feature of UML is that you cannot connect ports of *classes* with associations; you can only connect ports on *instances* with links. Thus, in the resulting task diagram of [Figure 9.7](#), we’ve instantiated the tasks and connected those instances with links to show the connections. We’ve also added the period and priority in constraints anchored to the appropriate «active» objects.⁷ To be complete, we also added the required interface (`iAveragedData`) to specify the ports.⁸

⁶ In Rhapsody this can be done by first showing the structured view of the active class, then dragging the analysis class inside. Once there, right-click on the class you want to make a part and select “Make an Object.” Make the encapsulating class active by setting its concurrency property to “active.”

⁷ In this case, you will note that the larger the value of the priority property, the lower its priority. This is operating system specific.

⁸ The observant reader may notice that we use ports only in a very limited way in the example. Ports are a design pattern and have costs as well as benefits. We tend to only use them between architectural objects that need to delegate to internal parts.

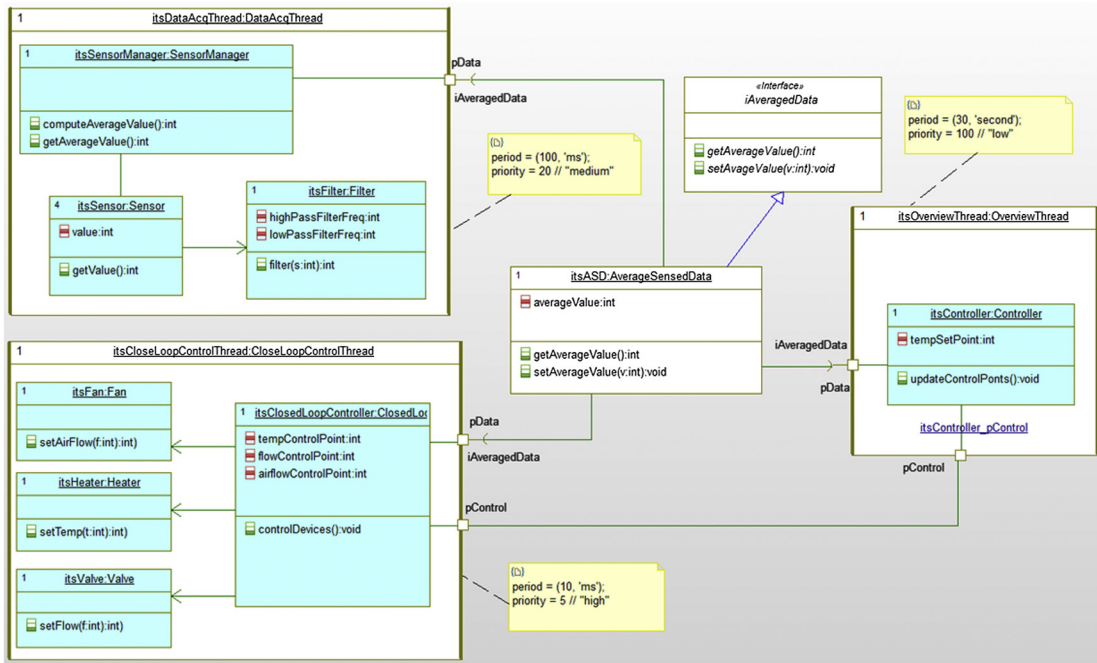


Figure 9.7
Concurrency design model.

Concurrency architecture can be highly complex and the interested reader is referred elsewhere for more details on the ins and outs.⁹ For the purpose of this text, we will select the strategies for you to enter into your design. Nevertheless, it is useful to have a quick glossary of terms important in the concurrency and resource model (Table 9.1)

9.2 Harmony Concurrency and Resource Architecture Workflow

The focus of the concurrency and resource management architecture is

- Identify tasks
- Identify resources
- Map semantic objects into task threads
- Identify synchronization points
- Specify synchronization algorithms
- Specify how resources will be shared among threads

⁹ See, for example, *Real-Time UML: Advances in the UML for Real-Time Systems 3rd Edition* (Addison-Wesley, 2004) or *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999), both by the author.

Table 9.1 Some Concurrency Definitions.

Term	Definition
Arrival pattern	The recurrence property that specifies when events of a given type occur; e.g., periodic or aperiodic. Arrival patterns are detailed with quantitative information such as the period, jitter, minimum interarrival time, and/or average interarrival time.
Blocking time	The amount of time a high priority task is prevented from execution because a lower priority task owns a resource.
Criticality	The importance of the completion of an action.
Deadline	In hard real-time systems, the time after an initiating event that the action must be completed to ensure correct system behavior.
Deadlock	A condition in which a system is waiting for a condition that can never, in principle, occur. Can only occur when four conditions are met: (1) tasks can be preempted, (2) resources can be locked while tasks are waiting for other resources, (3) tasks can suspend when owning resources, and (4) a circular waiting condition exists.
Execution time	The amount of time required to execute the response to an initiating event.
Hard real-time	A real-time system characterized by the absolute need to adhere to a set of deadlines.
Interarrival time	The time between the arrival of events of the same type; for periodic tasks this is a constant, but for aperiodic events, it can vary widely.
Jitter	The variation in the actual arrival time of periodic events.
Period	The length of time between the arrival of periodic events.
Priority	In a multitasking preemptive system, the priority is a numeric value that is used to select which task, from a set of tasks currently ready to run, will run preferentially.
Race condition	A condition in which the computational result depends on a sequence of actions, but whose order is inherently unknown or unknowable.
Real-time	A system in which the specification of correctness includes a timeliness measure.
Resource	An element that provides information or services of a quantifiably finite nature, e.g., a set of services which must be invoked atomically or provides a set of objects from a finite pool.
Schedulability	The mathematically demonstrated ability for a set of tasks in a hard real-time system to always meet their deadlines.
Synchronization pattern	The means by which tasks will synchronize their execution at specific, defined points.
Timeliness	The ability of a task to always meet its deadlines.
Urgency	The immediacy of the need to handle an event or to complete an action.

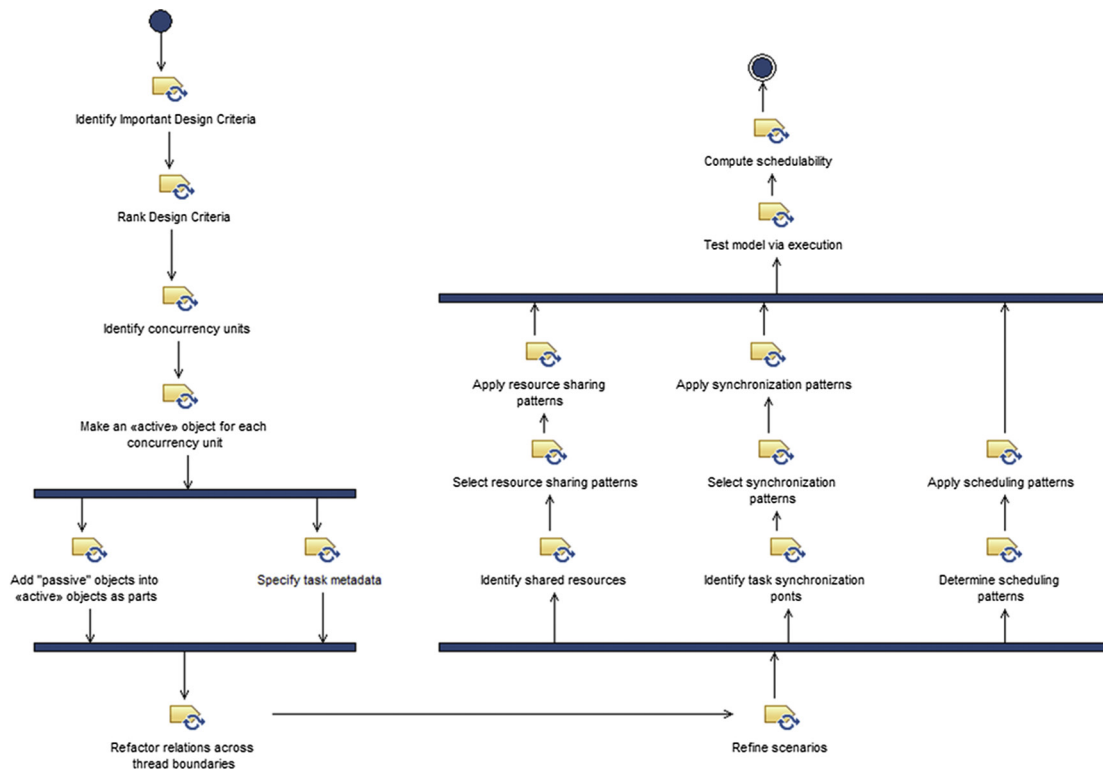


Figure 9.8
Constructing the concurrency architecture.

In UML, «active» objects are used to model concurrency units. An «active» object is a structured composite object that owns its own thread. It usually contains internal parts (object roles typed by classes) that execute nominally in the context of the «active» object's thread. UML itself doesn't distinguish between heavyweight and lightweight threads since they differ in implementation detail but not really in fundamental semantics. The Harmony process identifies the concurrency architecture using the workflow in [Figure 9.8](#).

The four major activities that are completed in the specification of the concurrency and resource architecture are (1) specify the tasks, (2) specify the resources and how they are to be shared, (3) specify the scheduling patterns to be used, and (3) specify how the tasks will synchronize when necessary.

The first job is to find a good set of tasks. In the Harmony process, we do this by using one or more task identification strategies ([Table 9.2](#)).

It is common to mix multiple strategies in the same system. For example, you can use multi-tasking preemption based on recurrence properties as the primary scheduling strategy and also have interrupt handling for highly urgent event handling.

Table 9.2 Task Identification Strategies.

Strategy	Description	Pros	Cons
Single event groups	Use a single event type per task	Very simple threading model	Doesn't scale to many events well; suboptimal performance
Interrupt handler	Use a single event type to raise an interrupt	Simple to implement for handling a small set of incoming event types; highly efficient in terms of handling urgent events quickly	Doesn't scale well to many events; interrupt handles typically must be very short and atomic; possible to drop events; difficult to share data with other tasks
Event source	Group all events from a single source so as to be handled by one task	Simple threading model	Doesn't scale to many event sources well; suboptimal performance
Related information	Group all processing of information (usually around a resource) together in a task	Useful for "background" tasks that perform low-urgency processing	Same as event source
Independent processing	Identify different sequences of actions as threads when there is no dependency between the difference sequences	Leads to simple tasking model	May result in too many or too few threads for optimality; doesn't address sharing of resources or task synchronization
Interface device	A kind of event source	Useful for handling device (e.g., bus) interfaces	Same as event source
Recurrence properties	Group together events of similar recurrence properties such as period, minimum interarrival time, or deadline	Best for schedulability; can lead to optimal scheduling in terms of timely response to events	More complex
Safety assurance	Group special safety and reliability assurance behaviors together (e.g., watchdogs)	Good way to add on safety and reliability processing for safety-critical and high-reliability systems	Suboptimal performance

Another important activity is to identify the shared resources and their methods of access. A shared resource is an entity, normally modeled as a class or object (depending on whether you're referring to a resource type or resource instance) that provides a *quantifiably finite* set of services or data. Normally resources must be accessed by one client at a time, through a process known as *serialization*. This can be done with a variety of different resource sharing patterns¹⁰ such as the Guarded Call (i.e., using a mutex semaphore),

¹⁰ For a detailed description of these patterns, see the author's *Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* (Addison-Wesley, 2002), Chapters 5–7.

Critical Section (i.e., turning task switching off during the use of the resource), or Message Queuing (i.e., queuing asynchronous access requests and handling them on a first-come, first-served basis).

Another issue for the concurrency architecture is to identify the scheduling patterns used. The scheduling pattern determines which tasks will run under what circumstances. The simplest is the Cyclic Executive pattern, in which each task runs in a sequence; once the task set has run through, the scheduler starts back with the first task. While this approach has the benefit of simplicity and ease of certification, it is demonstrably suboptimal in terms of responsiveness to incoming events; further, it is difficult to share data or synchronize with other tasks, and each task must complete before the next task can begin. A Round Robin pattern scheduler is only a bit more complex. In a Round Robin scheduler, the tasks need not complete, but must reach points at which they voluntarily relinquish control back to the scheduler to let other tasks run. A Time Multiplexed Scheduler (also known as a “Timed Round Robin” or TDMA scheduler) uses a time-based interrupt to stop execution of a task and begin the next in the sequence. These last three scheduling patterns are all “fairness doctrine” patterns, meaning that every task gets an equal chance to run; they don’t contain any of the notions of priority, urgency, or criticality.

There are a number of patterns that support these notions – notably the Static Priority Pattern and the Dynamic Priority Pattern. In these patterns, each task is assigned a numeric value, called a *priority*, that determines when the task will run. When a set of tasks is ready to run, the task with the highest priority runs. When it completes, the waiting task with the next highest priority runs. If, while a task is running, a higher priority task becomes ready to run, then the scheduler stops it and places it back on the ready queue, and the higher priority task runs instead. The Static Priority Pattern assigns the priority at design-time. A common specialized form of the Static Priority Pattern, as previously mentioned, is Rate Monotonic Scheduling (RMS).

Dynamic scheduling patterns assign the task priority at runtime based on some runtime criteria. The most common strategy is called Earliest Deadline First (EDF). EDF has the same basic assumptions as RMS but assigns priority on the basis deadline – the task with the nearest deadline has the highest priority. Thus, when a task becomes ready to run, its next deadline is calculated and then its priority is assigned on the basis of which tasks have nearer or farther deadlines. EDF is optimal but not stable because deadlines are computed during runtime it is not possible to predict which tasks will fail in an overload situation.

All these scheduling approaches assume task independence but have been extended to support the inevitable cases in which tasks are not, in fact, truly independent. In actual systems, most tasks must synchronize their execution at explicit synchronization points or share data and services from resources.

The last major concern in the concurrency architecture is to identify the task synchronization points and decide how to perform the synchronization. It's important to understand the basic independence-of-execution semantics of task execution. Each task can be thought of as a fully ordered series of actions. The whole point of using tasks is that the relative order of execution of the action sequences in the tasks is unimportant to the correct execution of the other tasks. The other tasks neither know, nor care, where some other task is in its sequence of actions. If the actions between two different tasks are tightly coupled, then they should not be modeled as independent tasks in the first place.

Problem 9.1 Roadrunner Concurrency Architecture

Now on to the main event ...

For the Roadrunner Traffic Light Control System, we begin this process with the intersection controller model modified to support distribution. The collaboration we will work with for this exercise is shown in [Figure 9.9](#).

For this problem, take the collaboration shown in [Figure 9.9](#) and make it into a design-level collaboration with concurrency. Remember to apply the task identification strategies (that's why they're there!); walk through the concurrency architecture workflow and make sure you either do all the steps or have convinced yourself that you don't need to. Decide what scheduling pattern

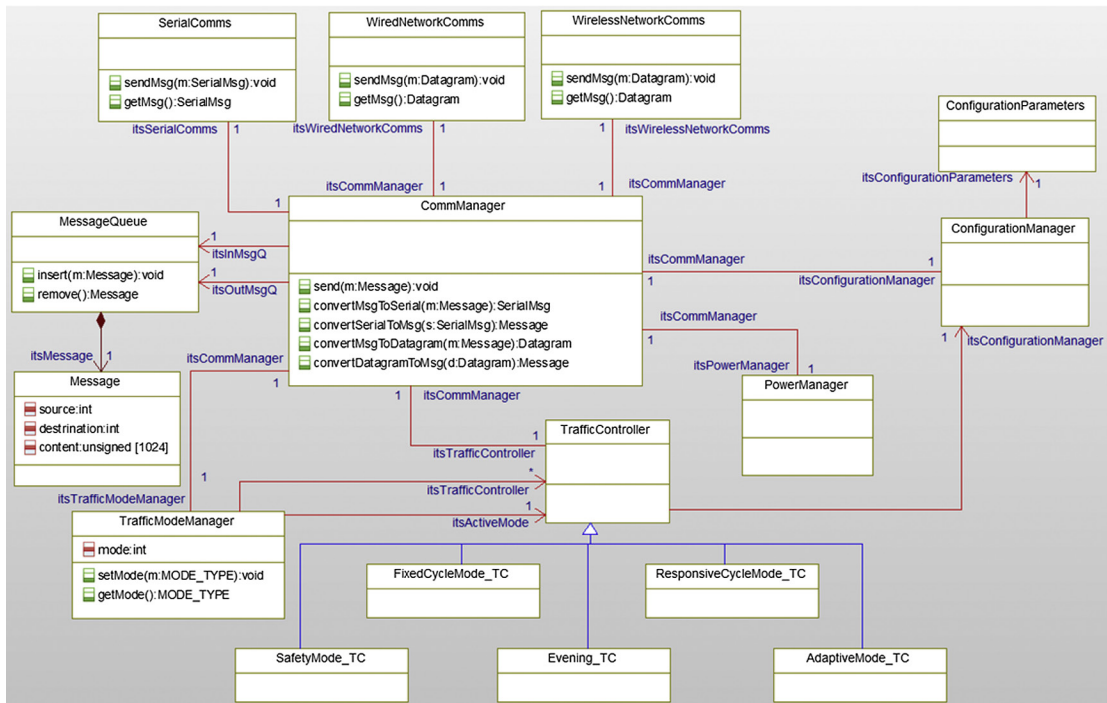


Figure 9.9
Roadrunner intersection controller.

¹¹ As a general rule of thumb, I do not show elements on diagrams that are provided by the OS or by the included frameworks – I only show things that I actually need to design and for which I need to create code. That means when I'm using Rhapsody, I usually do not show event queues or semaphores because Rhapsody automatically creates them for me. I sometimes show them for pedagogical reasons but rarely in real designs.

9.3 Summary

We've seen that architecture is all about taking an analysis model, also known as the Platform Independent Model (PIM), and performing systemwide gross-level optimization. In the Harmony process, there are at least five independent but interacting areas in which these architectural design decisions are made:

- Subsystem and component architecture
- Concurrency and resource management architecture
- Distribution architecture
- Dependability (safety, reliability, and security) architecture
- Deployment architecture

Architecture was categorized in this way because each of these areas has its own rich set of independent design patterns and solutions. The overall architecture is the sum of one or more patterns from each of these architectural areas of concern.

Design – including architectural design – is all about optimization. As such, it is imperative to first identify the design criteria and then rank them in order of criticality. This is because many of the design criteria (such as worst-case execution time and memory utilization) are traded off against each other by the design patterns. The ranking allows us to clearly identify and optimize the design criteria that are most important to us.

Design is discussed in terms of patterns because patterns are a way of codifying and reusing design solutions that apply to problems that reoccur in different contexts. Patterns, then, are generalized solutions of commonly occurring problems. In this chapter, we've seen how to apply design patterns in these architectural areas of concern, and just a few of the patterns that are available for use.

In the next chapter, we'll look at the next two levels down in design abstraction – collaboration and detailed design. Design solutions in the collaboration level of abstraction have a more limited scope – a collaboration realizing a single use case. Although not as “strategic” in their scope, collaboration design is still critical for good real-time and embedded systems development. Detailed design decisions optimize individual classes at a time, and so are much smaller than even collaboration design patterns.

Collaboration and Detailed Design

What You Will Learn

- What is “collaboration design”?
- Optimizing analysis collaborations
 - Optimizing interaction
 - Optimizing reusability
 - Optimizing safety and reliability
- What is “detailed design”?
- Optimizing classes and objects
 - Optimizing for space complexity
 - Optimizing for time complexity
 - Optimizing State Behavior
 - Optimizing for Reusability

10.1 Overview

In this chapter, we’ll deal with the two remaining levels of design abstraction: collaboration¹ and detailed design. The first thing to remember is that while analysis is about semantic correctness (hence construction of the “essential model”), design is all about optimization of that analysis model against the various design criteria. Collaboration design takes an existing analysis-level collaboration (as discussed in Chapter 7) and, based on the weighted set of design criteria, optimizes that collaboration by applying appropriate design patterns.

Collaboration design optimizes the model at the collaboration level. A *collaboration* is a set of classes and objects that work together to realize a larger scale behavior, such as the realization of a use case. Since a typical system has one to a few dozen use cases (at the high level, at least), the scope of the design decisions made for collaboration design is an order of magnitude smaller than for architectural design. If there are 25 use case (“analysis-level”) collaborations, then collaboration design must be applied 25 times, once per use case. Depending on the specific needs of a specific collaboration, it will use different design

¹ In earlier versions of the Harmony process, this level of design was known as “mechanistic design,” but it has been changed in the latest release to “collaboration design” to be more descriptive.

patterns because the context may be different and because the design criteria are likely to be different as well.

Detailed design optimizes the system at an even lower level of abstraction. Detailed design applies idioms to optimize individual class features.² The scope of detailed design is about an order of magnitude smaller than collaboration design. While this sounds daunting because of the sheer number of classes in a typical system, practice has shown that most classes in a well-designed system are very straightforward and require little optimization. There is usually a small set of the classes – typically 3% to 5%– that require special effort. It is for these “special needs” classes that we devote most of our effort in detailed design.

10.2 Collaboration Design

A collaboration is a set of classes and objects working together to realize a larger-scale purpose. A collaboration is specified in terms of specific classes playing named roles (called *classifier roles*) that interact to produce larger-scale behavior and functionality. Solutions to design optimization problems can usually be generalized and reused in different specific contexts. Such reusable design solutions are called *design patterns*. The roles define, in an abstract sense, the formal parameters for the pattern. When bound with actual parameters (the classes from your analysis collaboration), the pattern is said to be *instantiated*. A *mechanism*³ is a type of pattern that is limited in scope to a few classes (i.e., it does not have architectural scope), but is generally applicable in many circumstances. Thus, collaboration design is the term in the Harmony process for the application of mechanisms to produce design-level collaborations.

The Harmony design process is very pattern oriented. [Figure 10.1](#) shows the collaboration design workflow for the Harmony process with the details of the activities shown in [Figure 10.2](#). To understand the workflow, it is important to remember that the recommended lifecycle in the Harmony process is incremental; an incremental lifecycle creates the end product as a series of progressively more capable versions, called *iterative prototypes*.⁴ Normally, only one or a small number of use cases are realized in any given increment. As more increments are completed, the prototype becomes increasingly complete and capable, until at last, the project is complete and the prototype is shipped to the customer.

In the high-fidelity modeling activity of the process (see Chapter 7), one analysis-level collaboration is produced per use case, so that for any given increment, only a small number (typically one to three) of such collaborations are generated. In collaboration design, each of

² A class feature is an internal element of a class, such as a method, attribute, port, or state.

³ See Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User's Guide*. (Addison Wesley Longman, 1999).

⁴ By *prototype*, we do not mean something hacked together to demonstrate some concept. This prototype contains production code, but may be incomplete.

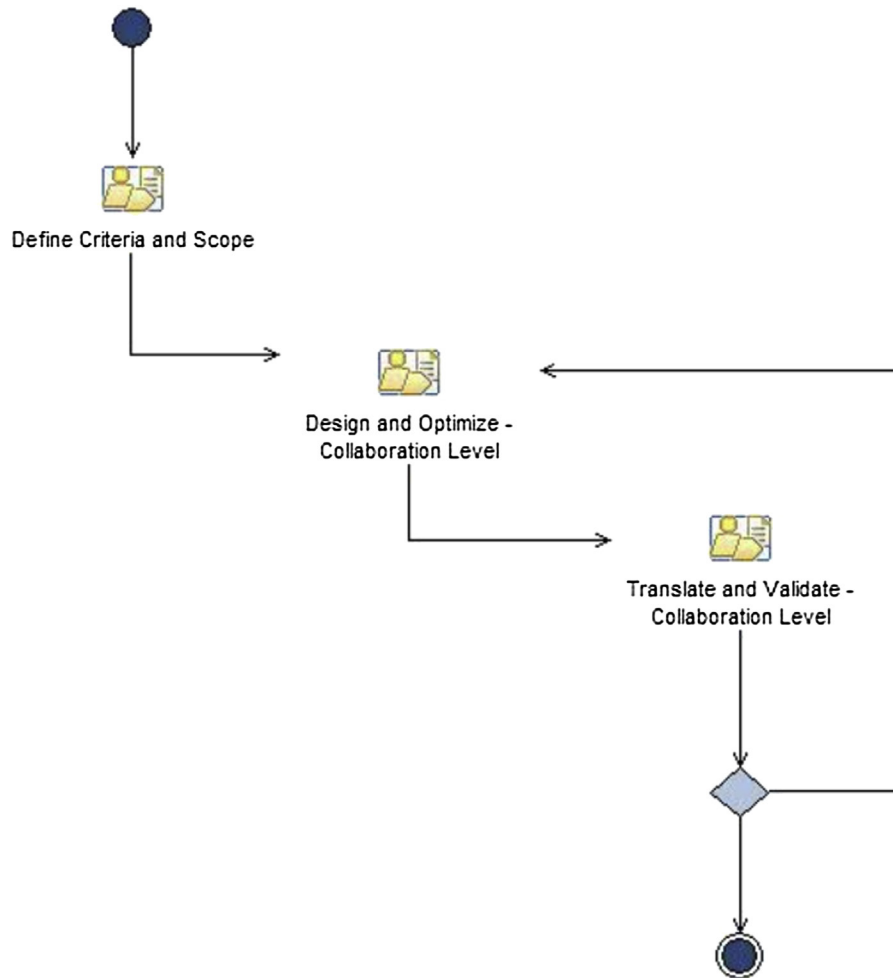
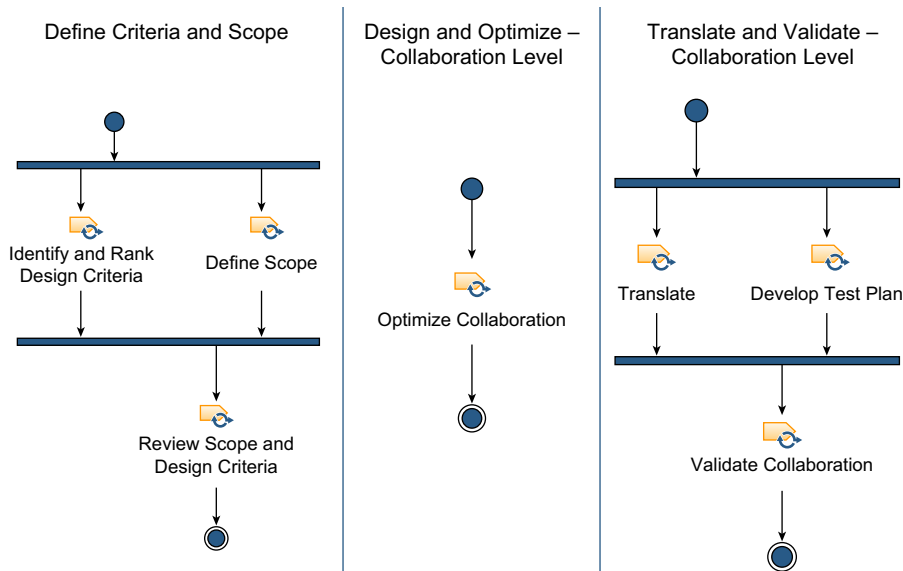


Figure 10.1
Harmony collaboration design workflow.

these analysis-level collaborations is converted into a design-level collaboration by applying the workflow shown in [Figure 10.1](#) and [Figure 10.2](#). As discussed in Chapter 2, these increments are executed frequently, generally every four to six weeks. In addition to generating new collaborations, existing collaborations may be changed in small ways to account for the new functionality added in the current increment. This process is known as *refactoring*.

Just as with architectural design, it is important to explicitly specify the design criteria so that appropriate design patterns may be selected. These criteria are ranked in order of criticality. This ranking is important because many of the design criteria will be in conflict with each other, and the proper application of design patterns requires that you optimize the *most important* design criteria at the expense of the *least important*.

**Figure 10.2**

Collaboration design workflow details.

Once the design criteria are identified and ranked, the collaboration design pattern literature can be searched for patterns that perform the desired optimization. Unfortunately, most references on design patterns mention pattern pros and cons only in passing as a secondary concern. In our view, the pros and cons are the primary selection criteria to identify appropriate design patterns. That is, we don't apply patterns because they are "cool," but because they achieve the desired optimizations at an acceptable trade-off cost.

Once the design patterns are selected, they must be *instantiated* into the collaboration. This, as discussed previously, is a matter of inserting the classes from the pattern into the analysis collaboration, and connecting the analysis classes to the pattern classes by either subclassing from or associating with the pattern's classes.

The classic reference on collaboration design patterns is by Gamma et al.⁵ It classifies patterns into three categories:

- Creational patterns
These patterns deal with the object instantiation process.
- Structural patterns
These patterns deal with how objects are composed into larger structures.
- Behavioral patterns
These patterns deal with algorithms and assignment of responsibilities among sets of objects.

⁵ Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

This is an excellent reference and highly recommended, even though the representation of the patterns is not in UML.

10.2.1 Delegation Pattern Strategy

Almost all patterns use either delegation or interface abstraction to achieve their goals. Delegation strips out some functionality from one class and places it in another. This allows the delegated class to focus on a more specific responsibility and often makes it more reusable in different contexts. The *container-iterator pattern* works in this way. The “whole” class owns a set of parts and must manage their containment (add, remove, find, sort, etc.). By abstracting the containment responsibility into a separate container class, the “whole” is simplified and it becomes easy to manage the parts, and to replace the container type when necessary. Iterators provide “bookmarks” into the containment so that multiple clients of the collection can easily track where they are without stepping on the toes of other clients.

Figure 10.3 shows the basic structure of the container-iterator pattern.⁶ Note that it uses both generalization and parameterization.⁷ Figure 10.4 shows how you might go from an analysis class that must manage the containment of many messages to the separation of that concern into a separate queue class. Lastly, Figure 10.5 shows the details of how the pattern is instantiated to achieve this separation of concerns.

10.2.2 Interface Abstraction Pattern Strategy

The interface abstraction strategy separates the things that change from the things that don’t – specifically, the specification of services (operations and events) are separated from the implementation of the desired services. This allows the services to be provided in different flavors, each of which optimizes a different set of criteria, such as worst-case performance, average performance, space complexity (i.e., memory usage), resource usage, etc.

The *adaptor pattern* is a prime example of the interface abstraction strategy. The situation arises when the client needs or expects a certain interface, but it is different than what is provided by the underlying realization. A simple example occurs when a track manager needs to get data from a variety of different sensors, but the sensors are from many different vendors and even nationalities. The solution is to require a common interface and write adaptors that convert service requests to the sensors and responses coming from the sensors to meet the expectations of the track manager. Figure 10.6 shows the structure of the adaptor pattern.

⁶ The container-iterator pattern is so common and useful that Rhapsody provides the pattern out-of-the-box. Whenever a model contains a * multiplicity, Rhapsody (by default – you can turn it off if desired) generates containers and iterators to manage the containment.

⁷ Generalization (aka “inheritance”) is used when you want to specialize the behavior but work on data of the same type. Parameterization (aka “template classes” in C++ or “generics” in Ada) is used when you want to have exactly the same behavior but work on different data types.

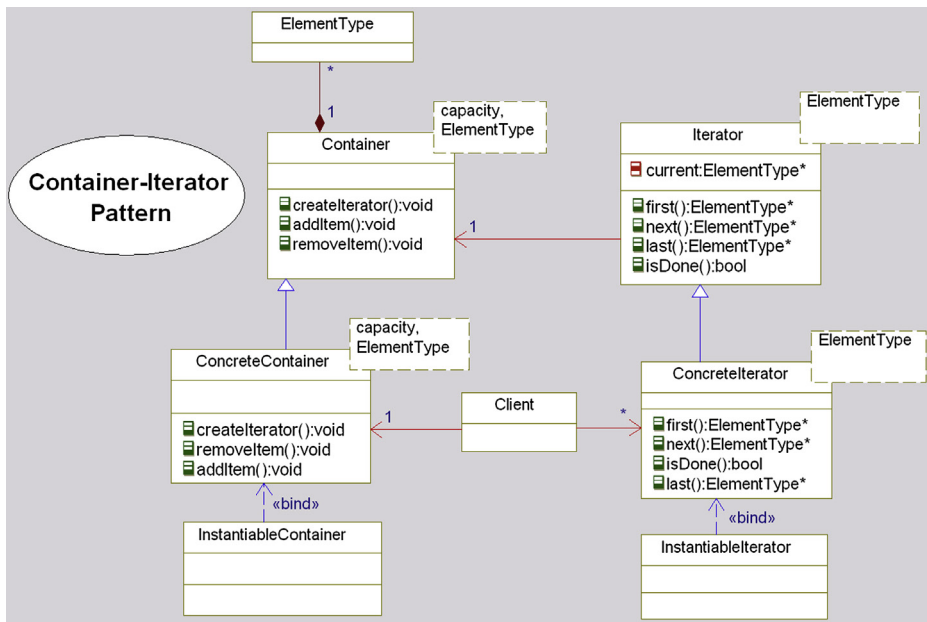


Figure 10.3

Delegation example: container-iterator pattern.

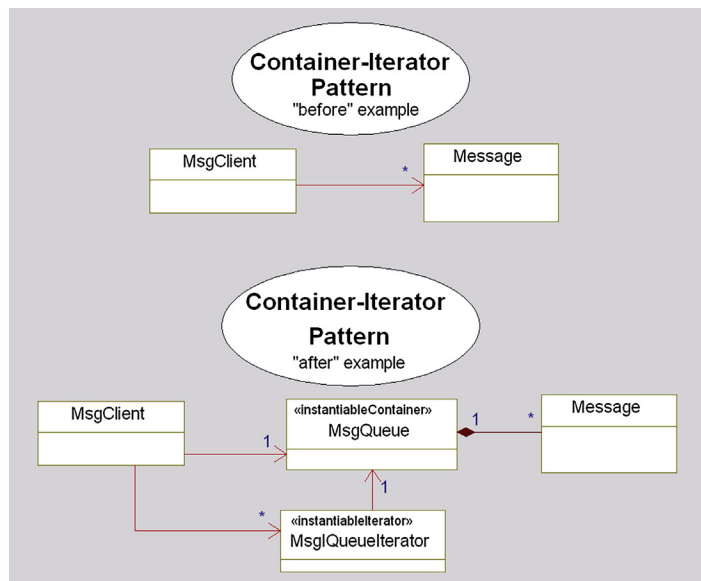


Figure 10.4

Use of container-iterator pattern.

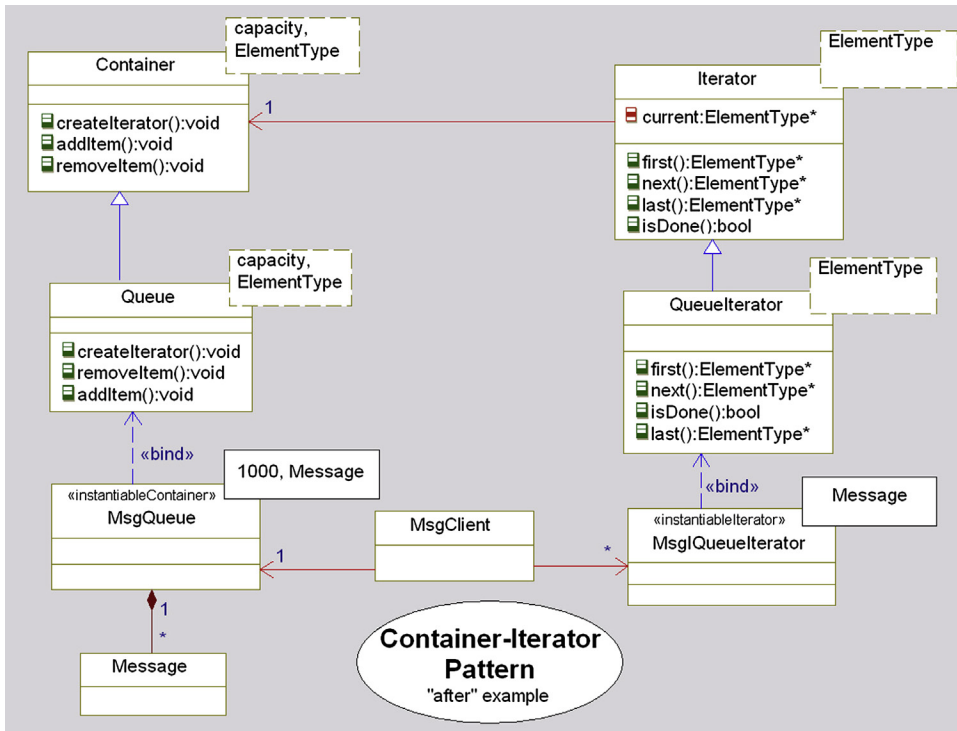


Figure 10.5
Details of use of container-iterator pattern.

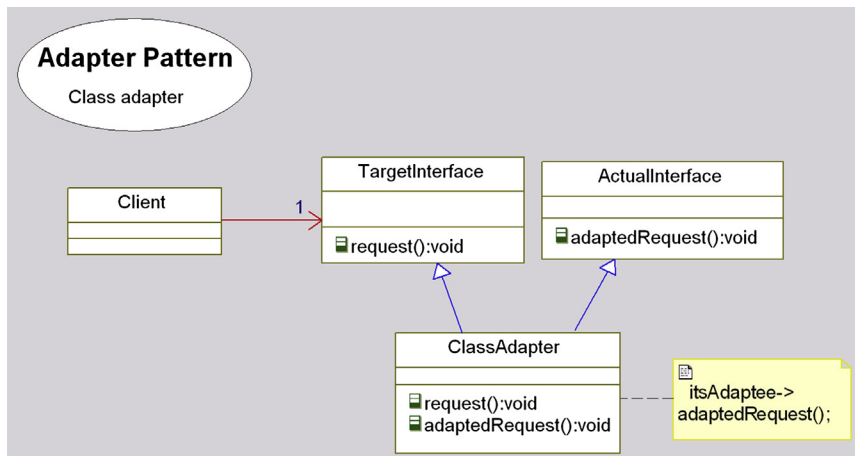


Figure 10.6
Interface example: adaptor pattern.

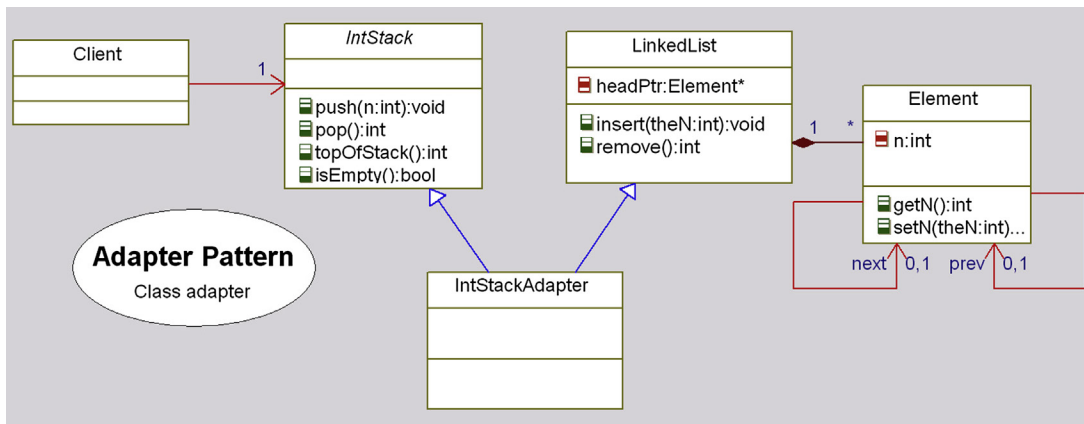


Figure 10.7
Use of adaptor pattern.

You can see the client expects a certain interface (`TargetInterface`). The adaptor subclasses both the desired interface and the implementation (`ActualInterface`). Then the implementation of the `TargetInterface` services is specialized to implement the expected interface in terms of the actual implementation offered by the `ActualInterface`.

Figure 10.7 provides another simple example of the use of the adaptor pattern. In this case, the client needs a stack, but what we have is a linked list. The `IntStackAdapter` class subclasses both the `IntStack` and the linked list. The inherited operations from `IntStack` are overridden to provide the desired functionality using the `insert()` and `remove()` operations of the linked list implementation.

Design patterns consist of four primary aspects:⁸

- Problem
 - What problem does the pattern address?
- Applicability
 - Which design criteria does the pattern optimize?
 - When is the pattern applicable (i.e., what is the common problem context)?
- Solution
 - The pattern specification
- Consequences of the pattern
 - Pros (benefits) of the pattern
 - Cons (costs) of the pattern

⁸ The exact number of pattern aspects differ depending on the author consulted. I believe these are the most important and general aspects.

Collaboration design patterns are medium scale, involving as few as two or as many as a dozen classes. The interested reader is referred to the references for more patterns.⁹

10.3 Detailed Design

Detailed design optimizes the system at the class level, and so is concerned primarily with class features – attributes, operations, states, activities, and ports. There are anywhere from 10 to 100 classes (and possibly more) involved in a collaboration but only a few of them really require special attention. This is because either they are more complex than their brethren, either in data structuring or in behavioral complexity, or because they are part of a control or data flow that has high optimization needs.

Figure 10.8 shows the detailed design workflow as specified in the Harmony process. You can see that the classes are optimized one at a time.

Like the preceding design phases, detailed design proceeds largely through the application of design patterns, although in the literature they are more commonly referred to as design “idioms.” That is, for the most part, the problems you’re trying to address in this most-detailed level of design have been solved before in different systems and perhaps for different applications. However, those solutions can be abstracted, generalized, and reapplied in new situations, including your current system under development.

The issues normally addressed during detailed design include:

- Ensuring operation pre- and postconditional invariants, with idioms such as
 - Range checking
 - Reasonableness checking
 - Unit checking (e.g., miles vs. kilometers)
- Internal data structuring
- Internal algorithmic structuring and optimization
- Optimization of state machines
- Specifying local error and exception handling (both accepted and thrown)
- Low-level redundancy for safety and reliability
- Numeric round-off error management and correction
- Abstraction of services into interfaces
- Class feature visibility
- Ensuring quality of service budget adherence for services
- Realization of associations, e.g.,
 - Pointers
 - References

⁹ See, for example, the Patterns Home Page at <http://hillside.net/patterns>. The aforementioned book by Gamma et al. is highly recommended, although it is not cast in UML terms. There are collaboration patterns in the author’s books, such as *Doing Hard Time* and *Real-Time UML 3rd Edition*. Another useful book is *Applying UML and Patterns* by Craig Larman (Prentice Hall, 1998).

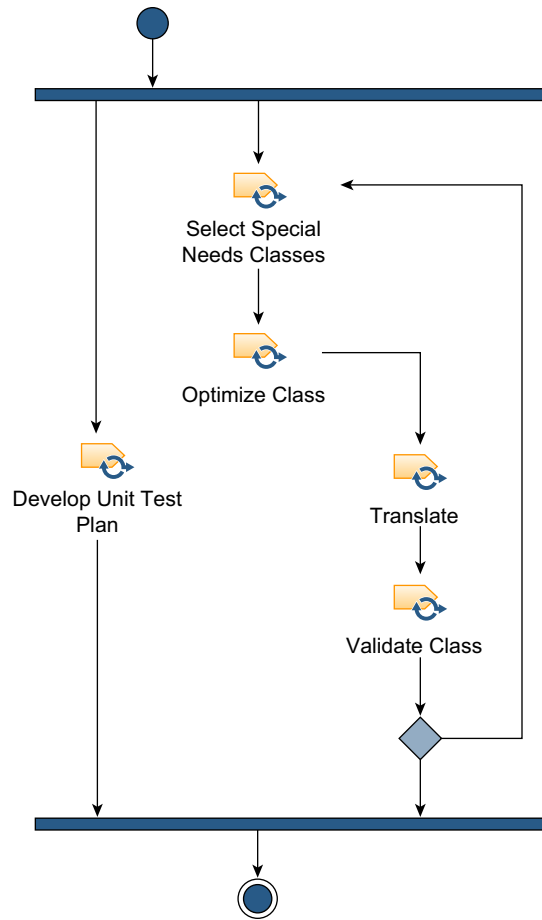


Figure 10.8
Harmony detailed design workflow.

- Object IDs
- Widget handles
- Sockets

The first five of these get most of the attention, but all are important. For example, consider the following general problem for a class state machine: the state machine wants to remember that an event happened in the past and handle it when it is convenient, or relevant, to do so. It is easy to use and-states and see if another and-state is in a particular state with the `IS_IN(stateName)`¹⁰ operator, but how to you remember that an event or state was *ever* visited in the past?

¹⁰`IS_IN(state-name)` is a behavior of a reactive object, returning `TRUE` if the object is currently in the specified state and `FALSE` otherwise. Rhapsody supports this query.

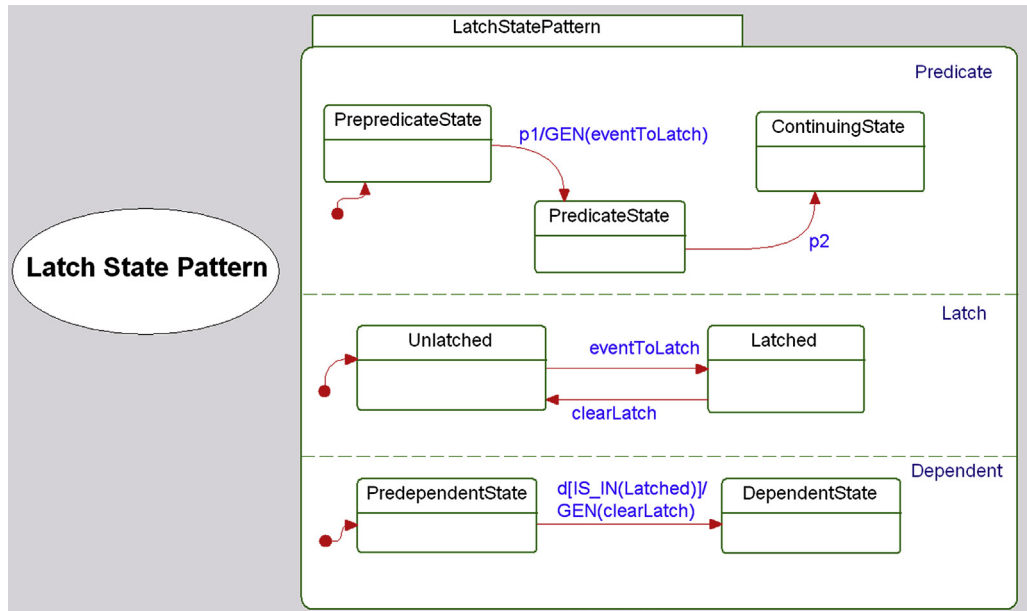


Figure 10.9
Latch state pattern.

One answer is the *latch state pattern*.¹¹ The structure of this pattern is shown in Figure 10.9. The state machine has three and-states. The **Predicate** and-state is responsible for identifying the predicate event, that is, the event that must be remembered. The **Latch** and-state is responsible for remembering that the event occurred. The **Dependent** and-state depends upon the event that was remembered. When the **Predicate** state receives an event that should be remembered, it generates a different event (the **GEN()** action in the predicate action list) that causes the **Latch** and-state to transition from the **Unlatched** to the **Latched** state. Later when the **Dependent** state receives an event that requires the original event to have been received at some point in the past, a guard checks to see if the latch is in the **Latched** state. If it is, then the guard is **TRUE**, and the transition can be taken. When the transition is taken, another event is generated that clears the latch by sending a **clearLatch** event, causing the latch to transition back to the **Unlatched** state.

Figure 10.10 shows a simple usage of the pattern. A sensor runs three and-states. One and-state has the responsibility for acquiring the data when available. The bottom and-state takes the data and performs some long computation on the data. Because this computation can take a long time, and because the bottom and-state doesn't want to miss any incoming events the event is remembered by the latch and-state in the middle.

¹¹ The concept of state behavior patterns was introduced in a previous book by the author: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999).

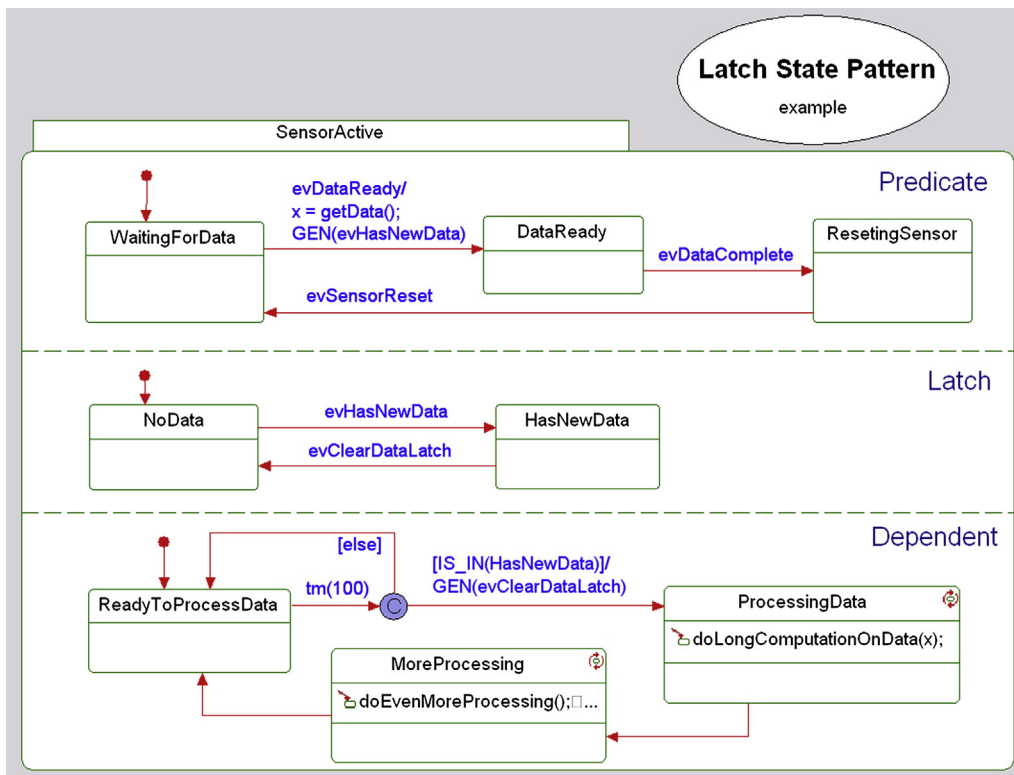


Figure 10.10
Use of latch state pattern.

Other than the use of state machine behavioral patterns, most of the optimization done within detailed design is done “below” the UML modeling level. For example, performing range and validity checking is done by adding actions to methods, activity diagrams, and state machines.¹² There are a few other optimizations that manifest themselves in the model as well, such as the identification of interfaces, additional class-level data redundancy, and feature visibility, but much of the optimization takes place by adding actions to achieve specific optimizations.

For example, suppose we have a class `PatientInfo` that has attributes of `height`, `weight`, `age`, `sex`, `ID`, `address`, and `phone`. Some of these are safety-critical (`weight`, `age`, `sex`, and `ID`), and so

¹² The *action language* is the language used to specify primitive behavioral statements (“actions”) in UML. UML specifies the kinds of things that an action language must be able to state in the UML action semantics specification, but does not currently specify or define a particular action language. It is most common is to use the implementation language (e.g., C, C++, Ada, or Java) as the action language in the model. Some tools have proprietary action languages but these are not without problems, including difficulty in debugging and lack of reverse engineering code changes when necessary.

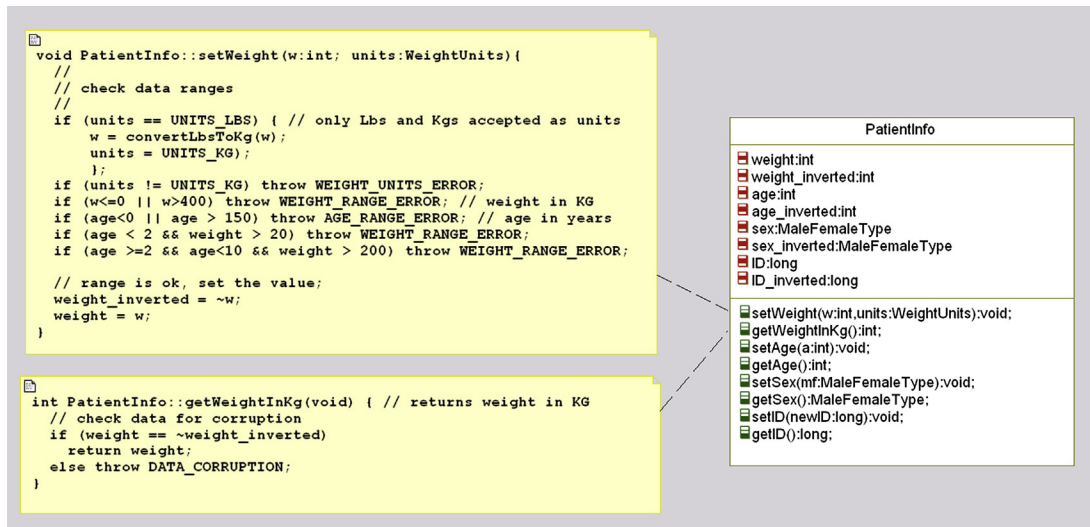


Figure 10.11
Safety checks.

they can be stored redundantly to identify data corruption problems. Further, additional checks can be made for reasonableness. Figure 10.11 shows how range and data corruption checks might be added in detailed design. The `PatientInfo` class has redundant one's complement storage of the safety-related values.¹³ The comments show the actions within the mutator (`setWeight()`) and accessor (`getWeight()`) operations that perform range checking and check that the stored data was not corrupted.¹⁴

Problem 10.1 Applying Collaboration Design Patterns: Part 1

In this first problem, we'll apply certain collaboration design patterns to the Roadrunner Traffic Light Control System and the Coyote UAV (CUAV). In the next problem, we'll expand the solution some more and add more design patterns to resolve more design issues and further optimize the solution.

There are many design patterns that can be applied to the systems we've been modeling – hundreds of possibilities, at least. In this problem, we'll discuss a couple of design problems and patterns that address it, and then you'll incorporate them in your model.

The Roadrunner Traffic Light Control System must interface with many different sensors from different manufacturers. Also, the system needs to be able to easily adopt new sensor modalities. To do this we need to support a general interface for both vehicle and pedestrian sensors

¹³ One's complement (bit-wise inverted) storage is used in case the problem is with stuck-at-0 or stuck-at-1 memory faults.

¹⁴ Corrupted data is a problem in electronically noisy environments. The author has personally witnessed memory bits flipping within a medical treatment device while in a medical operating room environment due to the operation of electrosurgical equipment.

and use adapters to support different specific sensors and sensor technologies. The task for this part of the exercise is “Optimize the collaboration for support of different sensor modalities and technologies.” So, for the first part of this exercise, apply the adapter pattern, discussed earlier in this chapter (see Figures 10.6 and 10.7), to optimize the Detect Vehicle collaboration to support different sensors. Figure 10.12 shows the starting point – the object analysis model of the Detect Vehicle use case. Show the resulting collaboration design-level class diagram.

For the second part of this problem, we’ll address noise reduction and image enhancement in the image processing of the Reconnaissance Management subsystem (RMS) of the CUAV. This is a crucial problem that must be solved for effective reconnaissance and track management. It may be necessary to add or delete different image processing algorithms dynamically, based either on commands from the Mission Ops personnel or onboard by trying a best-fit or error minimization function. Some noise reduction and image enhancement algorithms are:

- Gaussian sharpening
 - Gaussian sharpening removes noise by convolving the original image with a mask. This brings the value of each pixel closer to that of its neighbors.

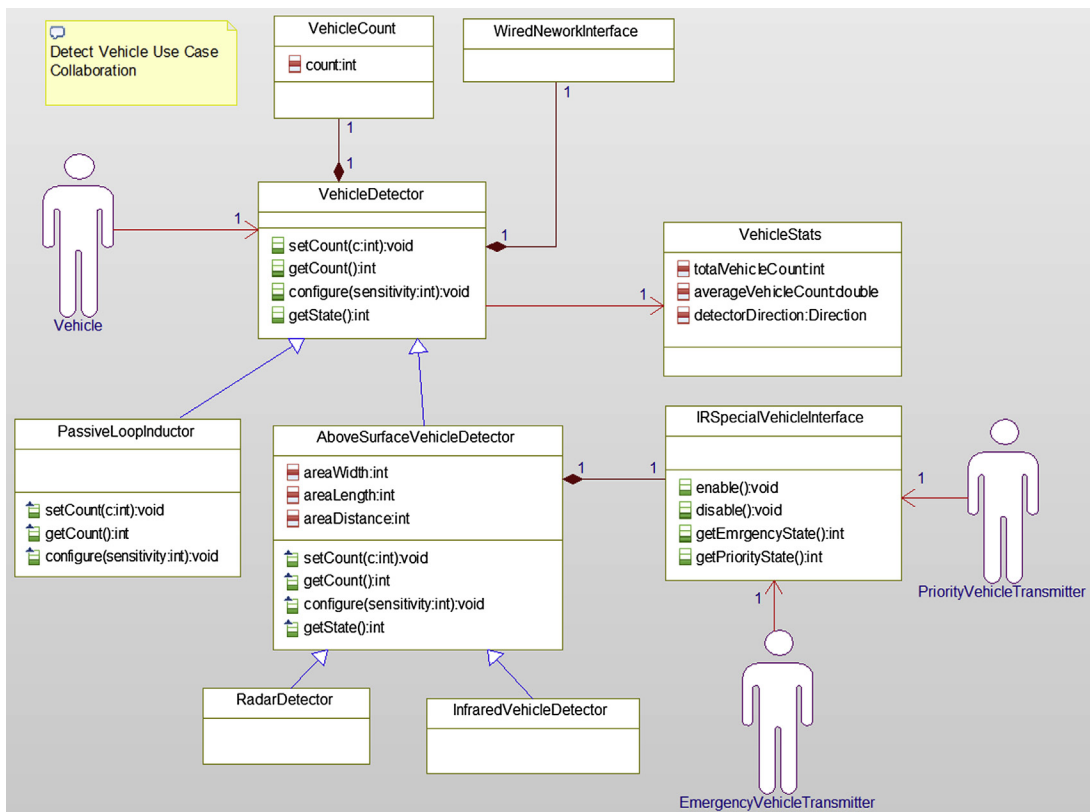


Figure 10.12
Roadrunner Detect Vehicle collaboration.

- Averaging filter
 - Averaging is a degenerate case of Gaussian filtering, where the function defining the mask values has an infinite standard deviation.
- Median filter
 - A median filter is a nonlinear filter that can preserve image detail. A median filter works by
 - Considering each pixel in the image
 - Sorting the neighbouring pixels into order based upon their intensities
 - Replacing the original value of the pixel with the median value from the list
- Center-surround neural network error minimization
 - A neural network is an ordered collection of rather simple processing units, arranged to perform error minimization. Neural networks usually require training (by applying examples with known results, hundreds or thousands of times). Neural networks come in several types. A center-surround neural network is the computational basis for image sharpening in the human visual system and works by sharpening edge boundaries in images.

There are, of course many more image enhancement algorithms from which to choose. The RMS may have a dozen or more available from which to choose. The RMS needs some way to efficiently switch algorithms or to chain algorithms together to provide optimal image processing. The goal for this part of the problem is to “Optimize the system for flexibility in supporting multiple image processing algorithms in sequence.”

To realize this goal, instantiate the *chain of responsibility pattern*, discussed in [Box 10.1](#). Structurally, the chain of responsibility is a very simple pattern, as we can see in [Figure 10.13](#).

For this exercise, use the chain of responsibility pattern to improve the algorithmic flexibility of the image processing for the Reconnaissance Management subsystem, whose analysis class

Box 10.1 Chain of Responsibility Pattern

Problem

You want to have flexibility in the addition of new and possibly multiple algorithms (“handlers”).

Applicability

- When the best place to handle a request isn’t known at design time
- When you want to support a multilevel response
- When you want to be able to dynamically modify the response
- When you want to be able to give a set of possible responses each a chance to handle the request

Solution

Decouple the request handling from the receiver of the request by creating a chain of handlers and pass the request from handler to handler so that they each get a chance to respond until either the list of handlers has responded or all handlers have been given the chance to respond. Any specific handler can

Box 10.1 Chain of Responsibility Pattern—cont'd

- Choose to handle the request and terminate
- Choose to handle the request and pass on the request to the next handler
- Choose not to handle the request and pass on the request to the next handler

Consequences

- Reduced design-time coupling of the class that receives the request from the algorithms that process the request.
- Ability to add request processing dynamically.
- There is no guarantee that a request will be handled because there is no grand oversight.

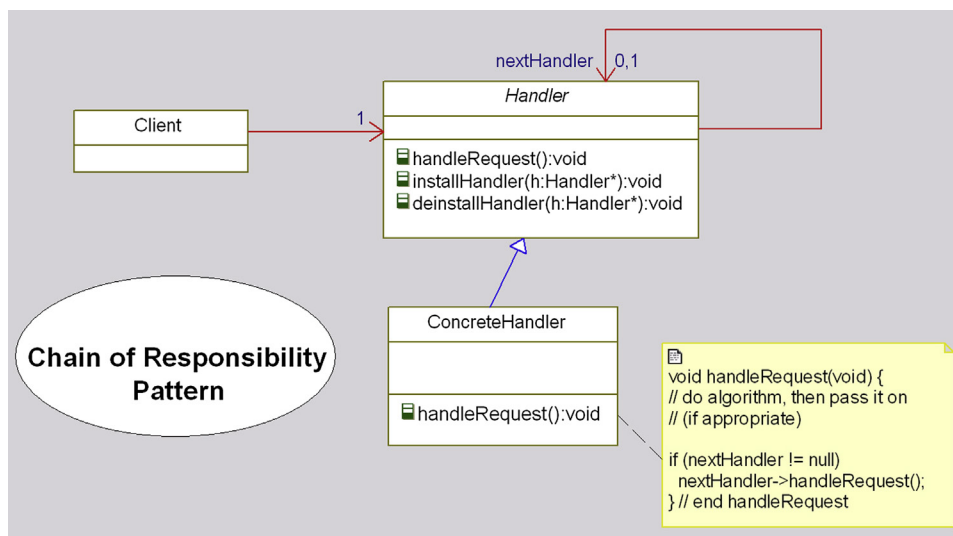


Figure 10.13
Chain of responsibility pattern.

diagram is shown in Figure 9.10. Show the resulting collaboration design-level class diagram. Hint: For this exercise, you need only show the elements from the collaboration that relate to the application of the design pattern.

Problem 10.2 Applying Collaboration Design Patterns: Part 2

In this next problem, we're going to continue applying collaboration design patterns to the collaborations we've been working on. For the Roadrunner system, we want to optimize the timely update of change of state information of the lights to the Intersection Controller, as well as to the other vehicle lights (from the safety architecture, it was determined that the lights

need to share contextual state information to ensure an appropriate overall system state). It is true that the light assemblies can, whenever they change state, notify the Intersection Controller and the other clients. However, we may want to dynamically add other clients of that information, even at runtime. In such cases, there must be a mechanism in place to let the light assemblies know to whom the state information should be sent.

A common solution to this problem is provided by the *observer pattern* (also known as the “publish-subscribe pattern”). In this pattern, the servers (in this case, the light assemblies) don’t know their clients *a priori*, but instead are instructed, at runtime, to add clients who request the information. They do this by providing a subscription mechanism in which the clients subscribe and provide their contact information. When it is appropriate (in this case, when the light assembly state changes), the client list is walked and the information is sent to each subscriber. The observer pattern is one design solution that could be used in this case. However, we have a deeply cross-connected system. In the safety analysis done in the previous chapter, it was decided that each light will talk to the others and in this way ensure that the system overall will always remain in a safe state (that is, traffic is never allowed to go in crossing directions at the same time).

Another option is the *data bus pattern* (see [Box 10.2](#)).¹⁵ The data bus pattern optimizes the connection topology of deeply cross-connected architectures by providing an object that serves as a repository for shared data. In the “push” version, when the server changes state (or changes the value of interest), the data is pushed to the data bus, which in turn sends it to all the subscribers. This simplifies the construction and maintenance of the cross-connecting topology.

[Figure 10.14](#) shows the structure of the data bus pattern. The composition shows that the different composites own their own copies of the data. The pattern can be mixed with other patterns, such as proxy or broker, to enable the pattern in distributed architectures.

[Figure 10.15](#) shows a simple application of the pattern. In this case, two data elements are put into the data bus. The train speed sensor creates an instance of the `VelocityData` class

Box 10.2 Data Bus Pattern (Push Variant)

Problem

Need to be able to share data among many different clients.

Solution

This is a variant of the observer pattern, where a data bus object holds shared data and distributes the datum whenever a new value is “pushed” to the data bus. Clients subscribe to data of interest owned by the data bus.

Consequences

- A simple pattern that allows the addition of clients at runtime and simplifies deeply cross-connected topologies.
- Additional memory is required to hold copies of the data and the notification handles (addresses) of the clients .

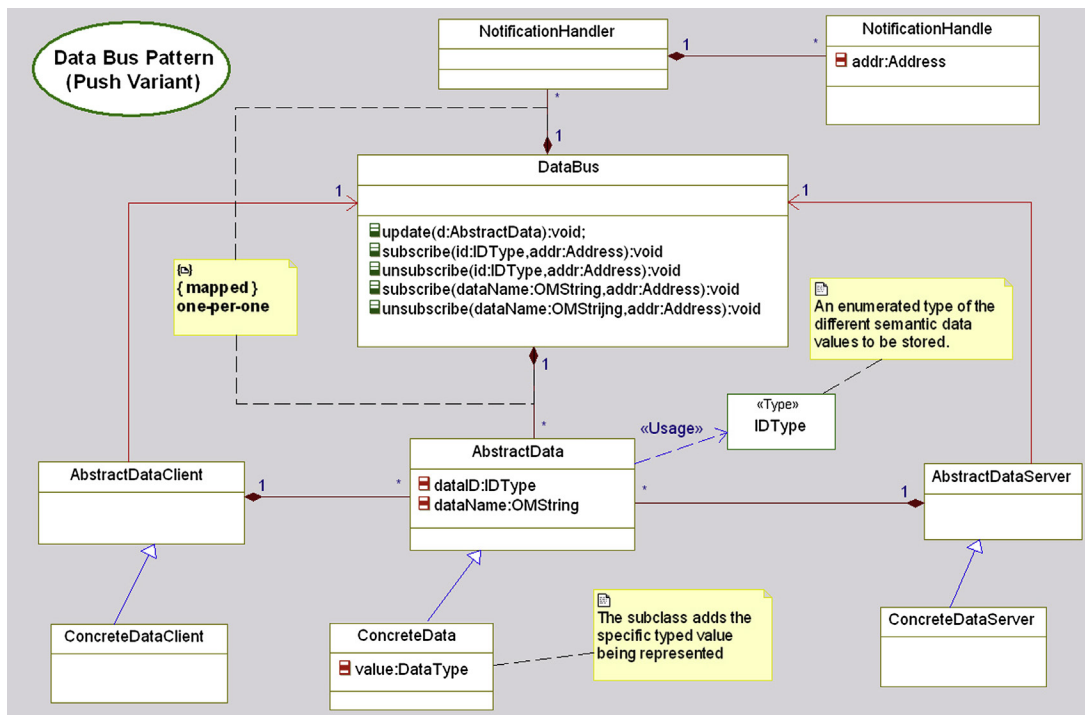


Figure 10.14
Data bus (push) pattern.

and calls `update()` to update the value. The GPS object creates an instance of the `PositionData` class and similarly updates the copy owned by the data bus. The engine closed loop control needs, and subscribes to, the velocity data while the display subscribes to both velocity and position. When the data is updated, the clients are updated automatically by the data bus. The `DataBus::update()` operation locates the appropriate instance to update; if the ID or name can't be found, a new one is added to the composite along with a new notification handler.

Now to apply it in our problem ... For this part of the exercise, apply the data bus pattern to manage the update of the states of the individual light assembly instances and distribute them to all the interested clients.

For the second part, let's consider again the CUAV Acquire Image collaboration in the RMS in Figure 9.10. We see that commands are sent from the Mission Ops actor to command the acquisition process in a variety of ways:

- Configure sensor
- Orient sensor
- Request image
- Configure image processing
- Set target types and their properties

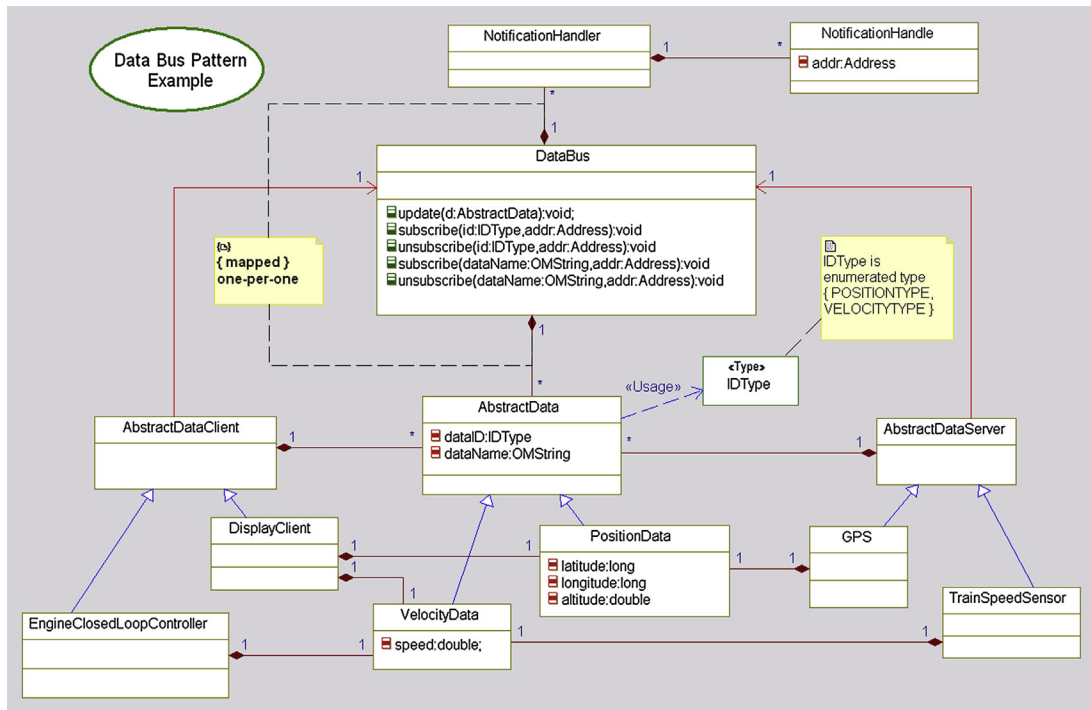


Figure 10.15
Data bus pattern example.

These commands must persist until handled. The *command pattern* provides a design means to address this kind of object interaction. The pattern is described in [Box 10.3](#). It works basically by reifying the command or request itself as an object, allowing it to be remembered and manipulated. [Figure 10.16](#) shows the basic structure of the command pattern, while [Figure 10.17](#) shows a simple example.

Box 10.3 Command Pattern

Problem

You want to manage requests that require logging, queuing, or take a period of time to fulfill.

Applicability

- When you want to take a request through a series of states (steps)
- When you want to manage multiple requests simultaneously, each in possibly different states
- When you want to support undo
- When you want to log or queue requests

Solution

- Reify the request as an object, possibly with internal state
- Also known as the Transaction Pattern.

Continued

Box 10.3 Command Pattern—cont'd**Consequences**

- Allows simultaneous management of multiple requests easily.
- Commands are first-class objects and can be subclassed and extended.
- “High-level” commands may be assembled from smaller command parts.
- Easy to add new commands.

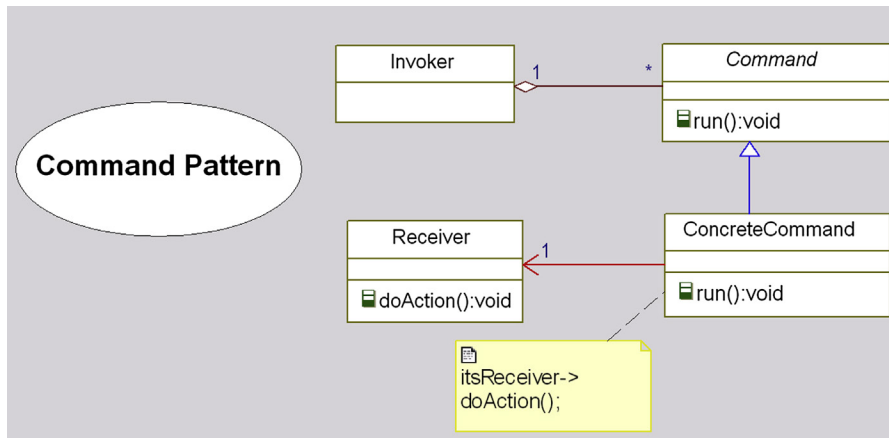


Figure 10.16
Command pattern.

For this latter part of this problem, apply the command pattern to the CUAV to specify and optimize how it handles and manipulations commands.

¹⁵ See the author's *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2002). The pattern shown here is slightly modified from that reference.

Problem 10.3 Applying Detailed Design State Behavioral Patterns

The next problem brings us into the detailed design phase. As discussed early in the chapter, detailed design is all about optimization at the class or object level of scope. This problem will deal with the optimization of class state machines to solve particular detailed design issues. That is, we will look for opportunities to apply state design patterns to the state machines in the classes of the collaborations to optimize or simplify their structure and/or behavior.

If you look at [Figure 16.1](#), you see an instance of the class `TrafficModeManager`. This instance is responsible for changing amongst the modes of operation within the traffic light system. This can be done under direct command or at a predetermined time of day.

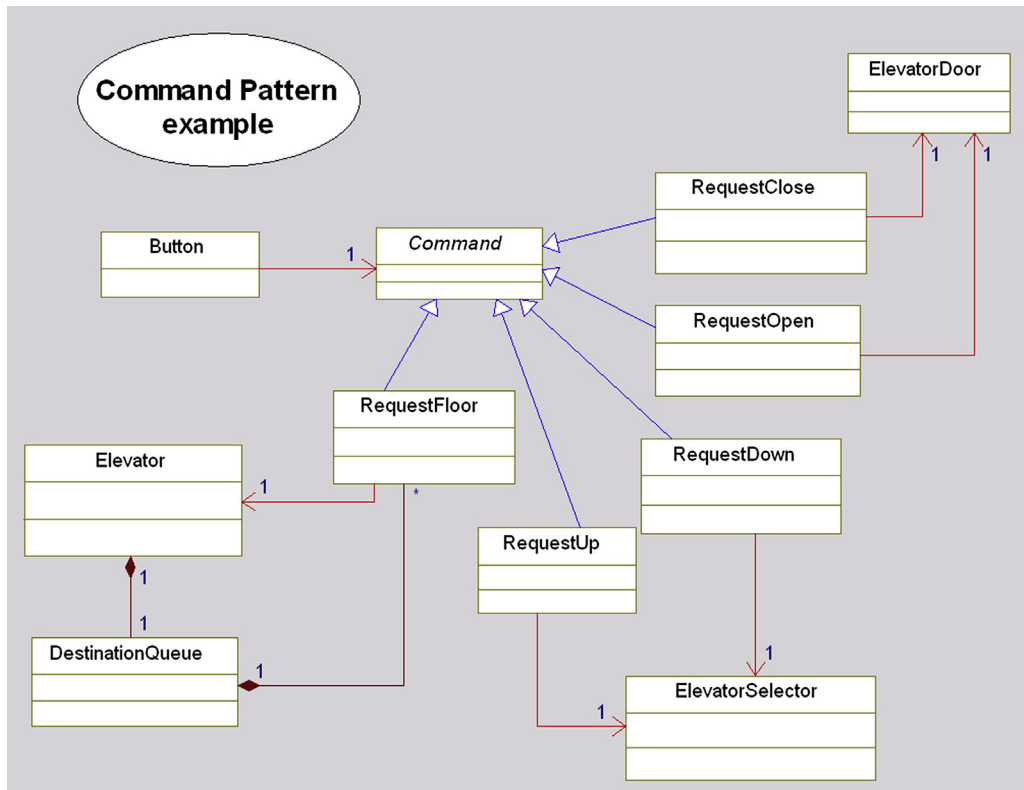


Figure 10.17
Command pattern example.

Figure 10.18 shows the state machine of the class `TrafficModeManager`. We note that the modes supported include:

- Safe Intersection Mode (all lights flash red)
- Evening Low Volume Mode (primary flashes yellow, secondary flashes red)
- Fixed Cycle Time Mode (fixed time cycling of lights)¹⁶
- Responsive Cycle Mode (lights cycle according to traffic)
- Adaptive Mode (like Fixed Time Cycle Mode except that the length of the cycle times vary depending on traffic load)

We must be able to transition from any mode to any other mode (note that we send the event `evStop` to the current mode instance, update our active mode instance, and then command it to start with an `evStart` event. This results in a “rat’s nest” state machine with $n-1$ transitions exiting and $n-1$ transitions entering each of n states. Isn’t there a cleaner, simpler way to represent exactly the same behavior?

The *any state pattern* uses composite states to simplify the transition structure while providing *exactly* the same behavior. It accomplishes this by placing the semantic states inside a composite and drawing a *single* transition from the composite state to the internal semantic state.

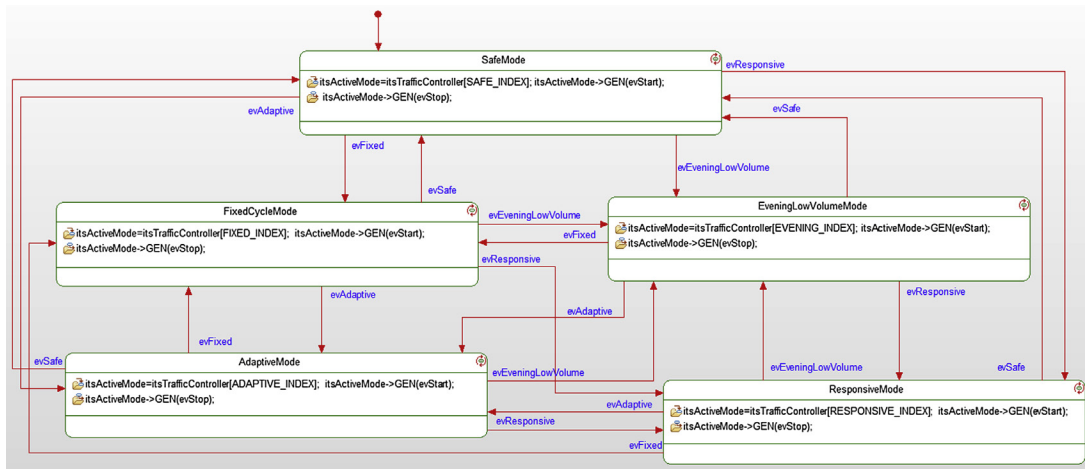


Figure 10.18
TrafficModeManager state machine.

That transition represents all transitions from the peer semantic states because whenever the object is “in” the composite state, the transition may be invoked, regardless of exactly which nested semantic state is active. Figure 10.19 shows the rat’s nest state machine before, and the streamlined and *behaviorally equivalent* state machine after encapsulating the semantic states with the composite.

For this part of the exercise, apply the any state pattern to the `TrafficModeManager` state machine; be sure to include the appropriate events, guards, and actions (as necessary) on the state machine.

For the Coyote UAV, we haven’t built any state machines yet for the Acquire Image collaboration (shown in Figure 9.10). In this exercise, you will construct a statechart for the `ImageManager` to acquire and process image data from the radar, FLIR, and optical payloads. To solve this problem, do the following:

- Treat the `ImageManager` as a composite class and create a structure diagram for this class.
- Add in three parts to manage the acquisition and image processing of the payload image data:
 - `theRadarDeviceDriver:RadarDeviceDriver`
 - `theFLIRDeviceDriver:FLIRDeviceDriver`
 - `theOpticalDeviceDriver:OpticalDeviceDriver`
- In a separate class diagram, create a class called `SensorPayloadDeviceDriver` as the base class for these three classes and make the specific device driver classes subclasses of this.
- In the `SensorPayloadDeviceDriver` class, add a statechart that has the following states:
 - Off
 - WaitingToGetRawImage
 - ConstructingImage
 - PreprocessingImage
 - AddingMetaData

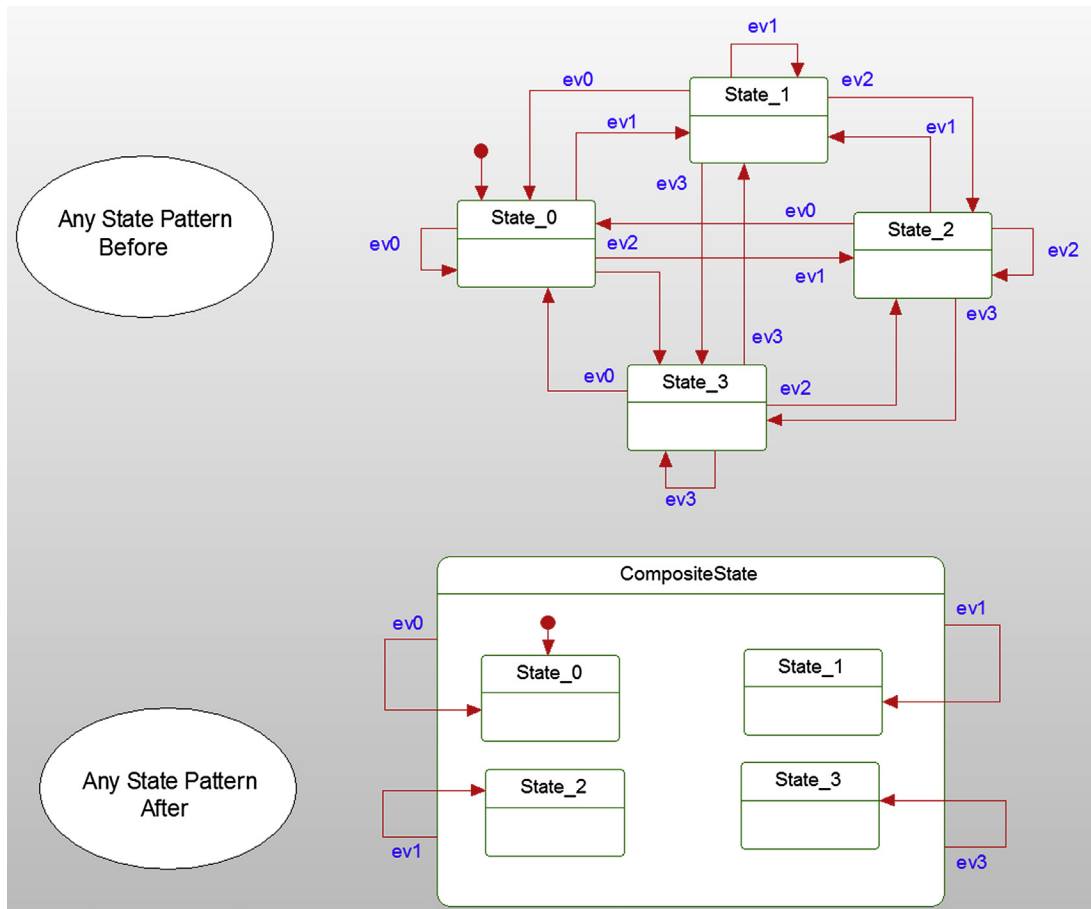


Figure 10.19
Transitions among high-level states with the any state pattern.

- Link the states with the following transitions:
 - When this state machine is Off, an evEnableSensor event takes it to the WaitingToGetRawImage state.
 - From there, the evImageStart event takes the machine to the ConstructingImage state, executing action initializeImage().
 - From that state, an evImageData event transitions back to the ConstructingImage state, executing the action addDataToImage().
 - From this point, an evImageComplete event transitions to the PreprocessingImage state, invoking the handlers created when we added the chain of responsibility pattern earlier in the chapter by executing the action imageHandler->handle().
 - When that processing is complete, transition to the AddingMetaData state and invoke the action theImage->addMetaData().
 - Once that is complete, cycle back to the WaitingToGetRawImage state

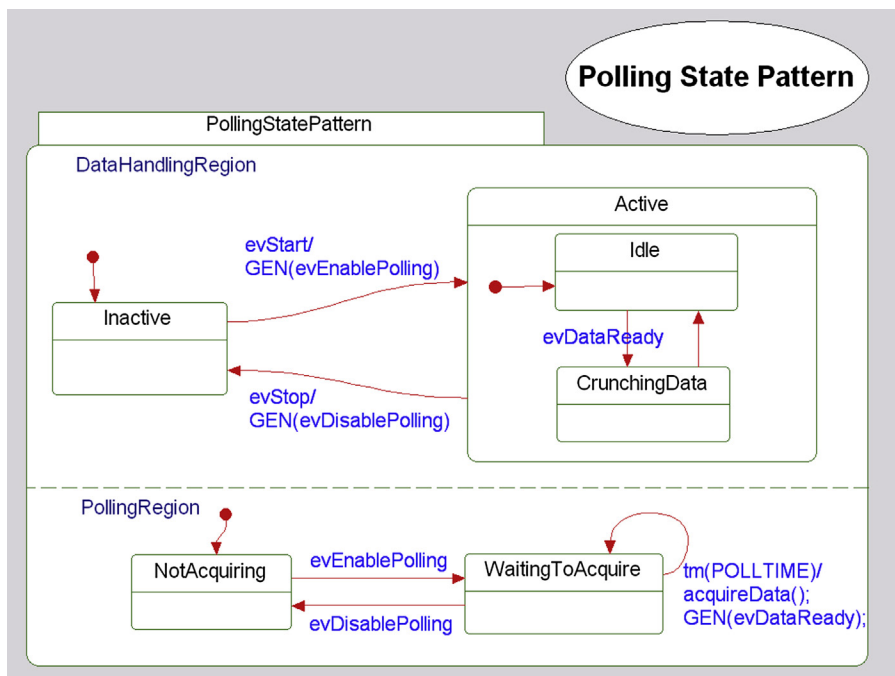


Figure 10.20
Polling state pattern.

- If, at any time (except in the *Off* state), new data becomes available (i.e., an *evImageStart* event is received), abort the current image, cycle back to the *ConstructingImage* state, and execute the action *initializeImage()*.
- If, at any time (except in the *Off* state), an *evDisableSensor* event is received, abort whatever image processing is proceeding and transition to the *Off* state.

This is the first half of the problem. Now we have to address the concern of how we get the data. Let's suppose that we want to get the data by checking periodically, a process known as *polling*. How do we effectively poll on a periodic basis?

One solution is the *polling state pattern*. This pattern consists of two and-states. The first, called the *DataHandlingRegion* in Figure 10.20, is where the semantic handling of the data occurs. When it commanded into the active state, it sends an event, which is consumed by the second and-state. This latter and-state focuses on its responsibility to periodically check for new data. When the *POLLTIME* period has elapsed, it invokes the *acquireData()* action, polling for the data, and then generates the *evDataReady* event. The first and-state consumes that event and transitions to the *CrunchingData* state to handle it.

The polling state pattern simplifies the overall behavior by segregating the semantic data manipulation from the work of acquiring the data periodically.

Figure 10.21 shows a simple application of the polling state pattern for acquiring air speed data. When it starts, it powers the hardware and enables polling. When data is acquired from

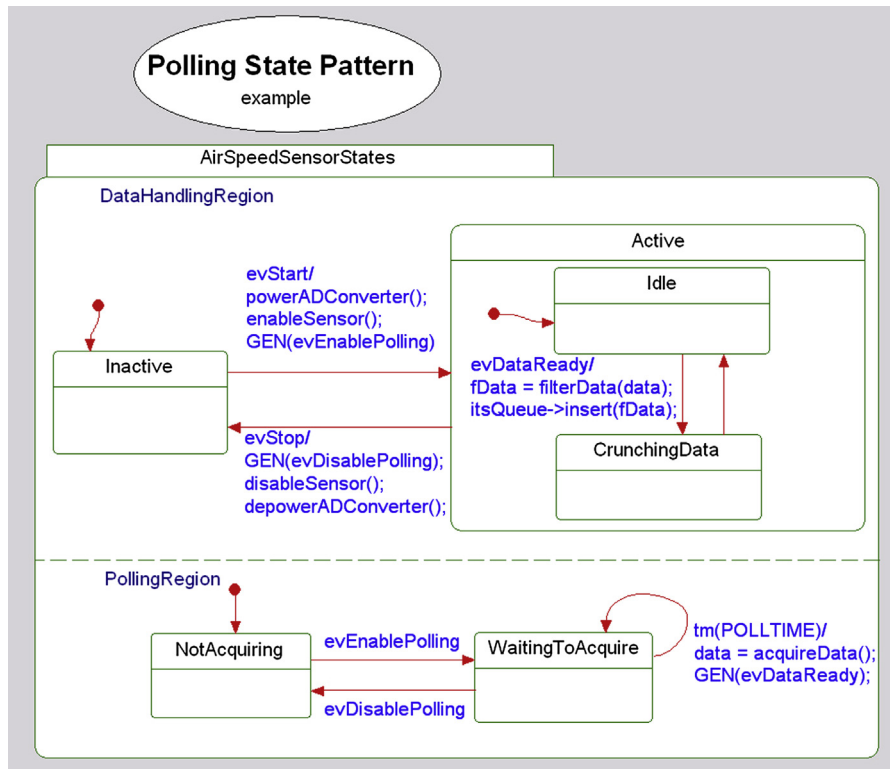


Figure 10.21
Polling state pattern example.

the air speed sensor, the second and-state generates an event that causes the semantic region to filter and then queue the manipulated data.

Let's now apply this pattern in the CUAV model. For the latter part of the exercise, add the polling pattern to the statechart you've constructed to add polling behavior. Just to make it interesting, the `acquireData()` operation returns a value called `success`, which can be one of three values: 0 (`NONE`), in which case data is not ready; 1 (`NEW_IMAGE`), in which a new image is ready to begin (i.e., generate an `evImageStart` event); or 2 (`ADD_DATA`), in which case the next part of the current image is ready (i.e., generate an `evImageData` event). Incorporate this functionality into the resulting state machine.

¹⁶ For the most part at least, as we've seen previously in Chapter 7.

Problem 10.4 Applying Detailed Design Idioms

For this last problem, I'd like to address some detailed design idioms. There are many small detailed optimizations that are typically performed in detailed design – these are listed earlier in

the chapter. In this last problem, we will deal with two detailed design issues. First, how does one handle errors or exceptions in UML, especially in state machines, and second, how do we ensure adherence to preconditional invariants on service requests?

The common idiom in C to handle errors is to return a nonzero value from a function. The problem with this idiom is that every client is free to ignore those error indications, and I've debugged more than my fair share of defects in safety-critical embedded systems in which error indications were simply ignored by the clients. Exception handling in C++ and Java, though, forces the client to handle the thrown exception or the calling function terminates. Once an exception is thrown from a server, the application looks for an appropriate exception handler (i.e., "catch clause") in the caller; if one is found, then the exception is handled. If not, then the caller terminates and the application looks for an exception handler in *its* caller, ad infinitum. This process is commonly called "walking back the stack." Eventually, either an exception handler is found, or the application terminates.

Such language-specific exception handling mechanisms have two things going for them. First, they cannot be ignored, and that is crucial for high-reliability and safety-critical systems. Second, they separate the exception handling from the main functionality, simplifying both. When the two are intermixed, the logic for both the main processing and the exception handling is muddled and hard to understand. By separating out the two, both become simpler and easier to get right.

However, these exception-handling mechanisms are not without their problems. The most significant of these is their inability to handle exceptions across thread boundaries. In C++, if an exception is not caught by the time the stack has unwound to the top of the thread, then the thread must terminate, and then the application must terminate. Another problem is that it isn't entirely clear how to use exceptions in the presence of state machines.

State machines in UML may be synchronous (using call events, or what Rhapsody calls "triggered operations"), asynchronous (using signal events, or what Rhapsody calls "receptions") or a combination of the two. If a series of call events affects multiple state machines and an exception is thrown, then the stack gets walked back just like it would with normal method calls. It's a little more complex because care must be taken to ensure each state machine remains in a valid state, but it is relatively straightforward. It is much less clear what happens if a signal (asynchronous) event occurs. In that case, the stack walks backwards through the active object owning the event queue, but it would be odd indeed to have the active object understand the appropriate exception-handling behavior for all the objects held within that thread boundary. So what to do?

In UML, an exception is considered to be a kind of signal. In the UML 2.0 Final Adopted Specification, it defines an exception as

A special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. The receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified.

Even in this case, the receiver of the exception is implicit – that is, it is the service requester. In general, an asynchronous state machine doesn't know who its sender is, unless that information is provided as a parameter for the passed signal event.¹⁷

So how should we handle exceptions when we have multiple threads and when we have state machines? Whatever mechanism we use, we would very much like it not to depend on whether the source event is synchronous or asynchronous, because having multiple ways to manage exceptions complicates the applications and makes it pathologically tightly coupled. And, while

we're at it, we need to be able to handle normal language exceptions in our state machines as well, because actions may invoke services – some of which are in the standard libraries, such as `new()` – that may throw exceptions.

One solution is the *exception state pattern*. This pattern uses events to indicate not only service invocations but also exceptions. The class diagram in Figure 10.22 shows the structural elements involved. The `StatefulClient` and `StatefulServer` are both reactive classes, that is, they use state machines to specify their behavior. They also connect with ports – this is important because the `StatefulServer` will use the port name to identify along which link the exception event should be sent. The same event could be from different sources, but as long as each connects via a different port, then the source can be identified.

The `StatefulClient` realizes two distinct interfaces. The first, `iStatefulClient`, is the semantic interface – it specifies the events normally received during the execution of the object's life. The second, `iStatefulClientException`, is the exception interface – these are the events that have to do with the handling of faults and unexpected conditions and occurrences. The `StatefulServer` is typed by a single interface, `iStatefulServer`.

The ports connecting the instances of the classes are typed by these interfaces. The `StatefulClient` port `pServer` requires the `iServer` interface and offers both the `iStatefulClient` and `iStatefulClientException` interfaces. Using two distinct interfaces helps segregate the normal semantics from the exception handling. The `StatefulServer` port `pClient` requires both these interfaces, while offering its own semantic interface, `iStatefulServer`.

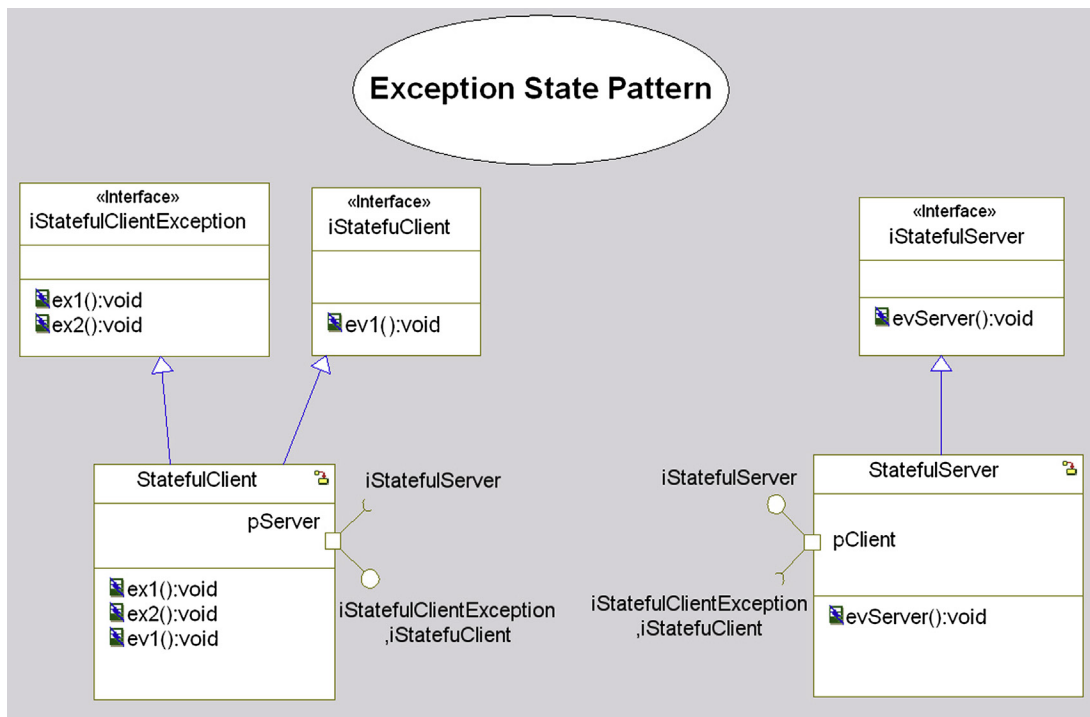


Figure 10.22
Exception state pattern.

The state machine for the `StatefulServer` is provided by Figure 10.23. We see how the exception gets raised in the action list for the `evServer` event. If `doAction` returns an unsuccessful result, it generates an `ex1` event¹⁸ and sends it back to the original sender. The original sender is identified by the guard on the event; `IS_PORT(pClient)` identifies that this event is from the `pClient` port. If the event can come from other ports, then they must be listed in other transitions. Following the sending of the exception event back to the client, an `evRollback` event is generated to return to the original state.

Lastly, the state machine for the `StatefulClient` is shown in Figure 10.24. We notice that the semantic actions are segregated from the exception actions by putting each in their own and-state. When the `ev1` event is handled, it sends an `evServer` event out the `pServer` port. If it later receives an exception event (`ex1` or `ex2`), then it does whatever cleanup is required and then issues an `evRollback` event to return to the `PredecessorState`.

We see that the pattern works by using a separate set of events, collected into the `iStatefulClientExceptions` interface, to represent the exceptions being thrown. The server uses the `IS_PORT()` in a guard to identify the source of the event that led to the exception identification.¹⁹ We create a separate and-state to handle the exception events. And in both the client and the server, we typically want to do a rollback to the previous state in the case of an exception being identified. The exact exception handling behavior is, of course, case and situation dependent but such specific handling behavior can easily be handled.

In the Roadrunner Traffic Light Control System, the light assemblies associate with the actual lights themselves. This association is a composition, indicating strong ownership. In this special case, there is a navigable link between the light and the composite light assembly. Since no other object can directly command the lights, ports aren't required – all that is necessary is that if the `VehicleLight` or `PedestrianLight` has an exception, it passes the appropriate event back to its composite owner.

The relation between the `VehicleLightAssembly` and the `VehicleLight` has * multiplicity, one for each traffic lane in the “through” direction and another for the turn lane. The

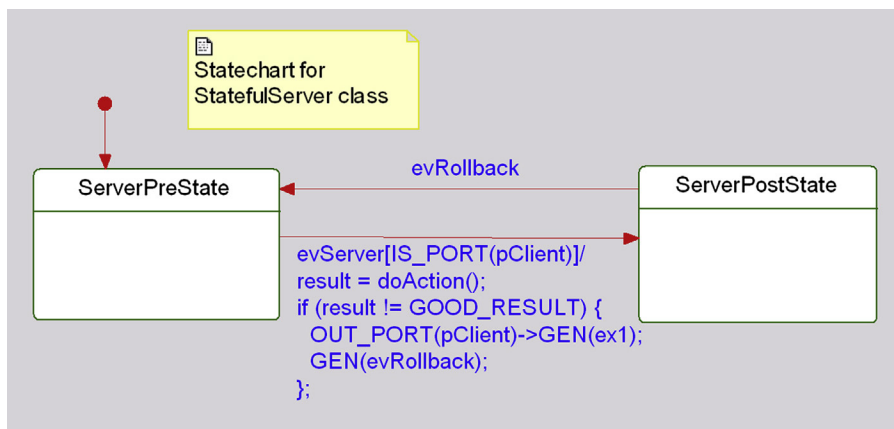


Figure 10.23
State machine for `StatefulServer`.

`PedestrianLightAssembly` has a single light that it controls, so the multiplicity needs to be changed to 1. These changes are reflected in [Figure 10.25](#).

For this exercise, add the statecharts for the vehicle and pedestrian lights (the lights are dumb, so any state may be entered from any other state – sequencing is handled by the light assemblies). Then add the exception state pattern to handle the error condition that when a bulb is illuminated, the photo detector does not detect the light going on. As mentioned, ports aren't required because the light assembly has a composition relation with the lights, and that composition provides a navigable link at runtime.

Next, let's work on the CUAV. Previously, we added the polling state pattern to the Acquire Image collaboration for the Reconnaissance Management subsystem for the CUAV. In this exercise, we want to avoid a problem commonly known as “lockup” – that is, the condition in which a system stops what it is doing, either because of a software or hardware fault. The solution we will add into the collaboration is called the “watchdog” pattern. The watchdog pattern is detailed in [Box 10.4](#).

The watchdog pattern ([Figure 10.26](#)) works by having a watchdog active object (or it can run in its own thread) monitor clients. The client must subscribe to the monitoring services by invoking the `addMonitor(timeout:long)` operation. This operation results in the creation of a monitor (called a `WatchPuppy` in the pattern) with the appropriate timeout and returns the

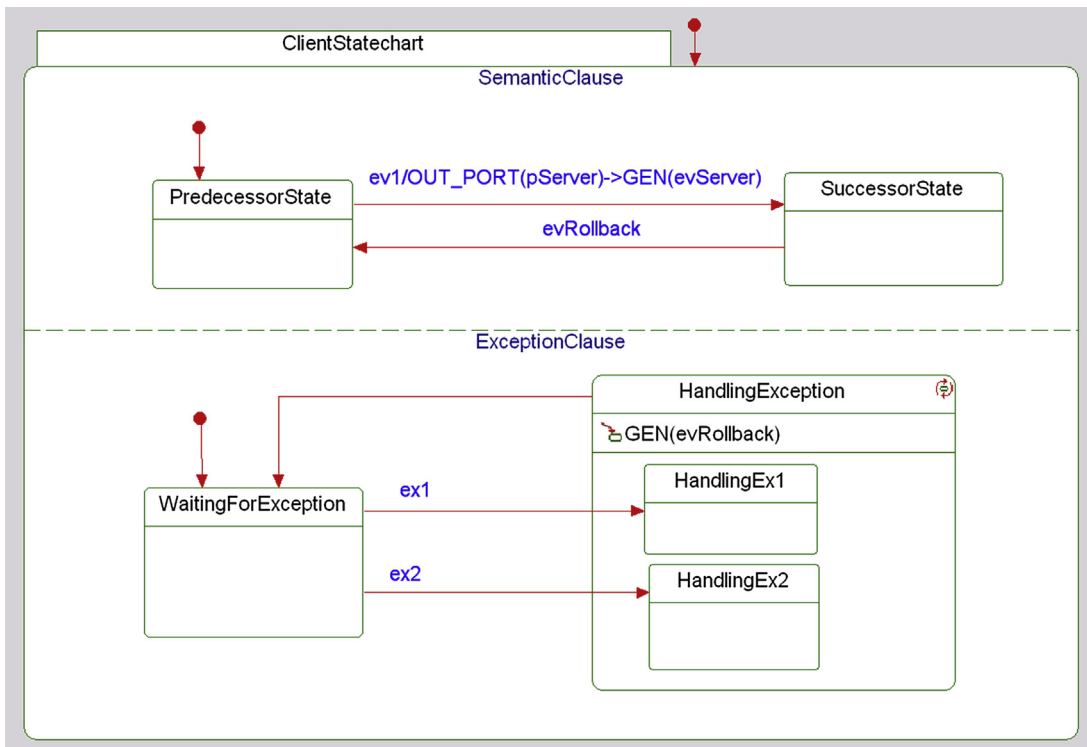


Figure 10.24
State machine for `StatefulClient`.

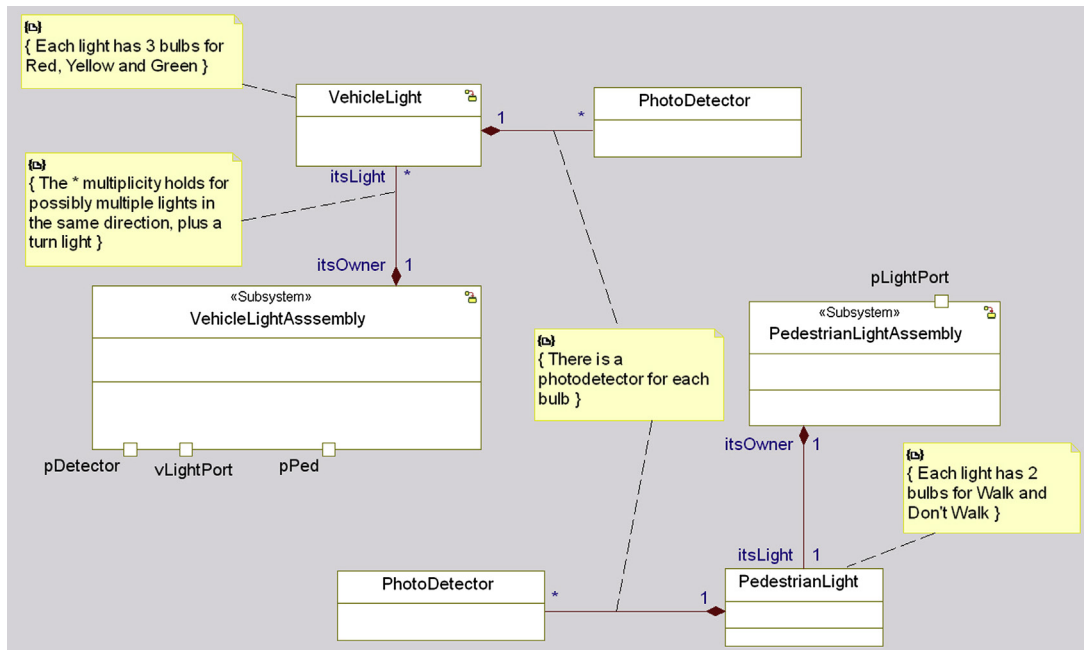


Figure 10.25
Light assemblies and lights.

Box 10.4 Watchdog Pattern

Problem

Need to ensure liveness of periodic processes.

Solution

Run an observer in a separate thread that receives “life ticks” on a periodic basis. If the watchdog doesn’t receive the life tick in a timely fashion, the system is assumed to have locked up and some draconian response, such as system (or subsystem) reset, or transition to fail-safe state, is taken.

Consequences

- Useful only when the timeliness of the monitored processes can be predicted with fair precision.
- Different watchdogs can be used for different processes when specialized responses are required to different process faults.
- Many different processes may be monitored.

ID of the monitor. This ID is used to identify which client is stroking which watchdog via the `stroke(ID)` operation. This results in an event being generated to the appropriate `WatchPuppy`. Should a given `WatchPuppy` timeout, it invokes the `timeoutFault(ID:int)` operation.

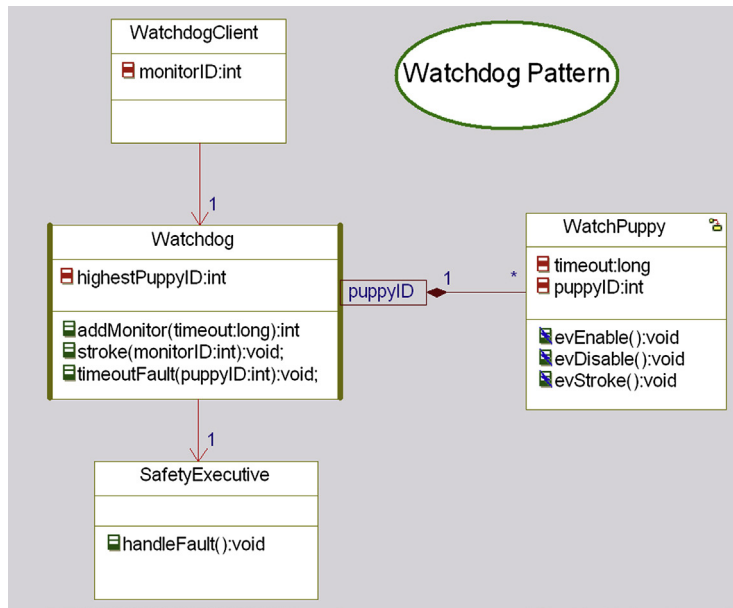


Figure 10.26
Watchdog pattern.

The watchdog then in turn invokes the `handleFault()` operation on the `SafetyExecutive`, which takes whatever corrective action is appropriate. In the simple version, any fault results in the same action, e.g., a subsystem reset, but since the `WatchPuppy ID` is returned, more elaborate client-specific behavior could be easily added. Note that care must be taken that the timeout should be not less than the period of the process plus whatever jitter²⁰ occurs in that period.

Also note the use of the `WatchPuppy puppyID` attribute as a qualifier. This means that the `puppyID` is used to discriminate among multiple `WatchPuppy` instances; the most common implementation is to make an array of `WatchPuppies` and the `puppyID` is simply the index into that array.

The state behavior for the `WatchPuppy` is straightforward. As long as it receives the `evStroke` event frequently enough, the event arrives before the timeout fires. When the transition handling the `evStroke` event is processed, the state is exited and then reentered, causing the timeout to be reset. The timeout fires only when the `WatchPuppy` hasn't received an `evStroke` event soon enough. This state machine is shown in [Figure 10.27](#).

For this last exercise, add the watchdog pattern to the `Acquire Image` collaboration to detect if the sensor device drivers fail for some reason. When this occurs, the `SafetyExecutive` should destroy and restart the `ImageProcessing` object. Show the resulting class diagram, and the updated state machine(s).

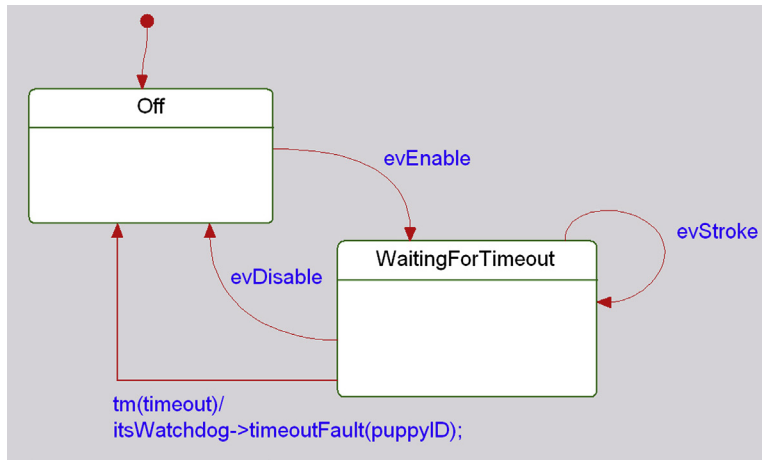


Figure 10.27
WatchPuppy state machine.

¹⁷ An exception (pardon the pun) to that rule is that when ports are used, the source port for an event can be identified, but even then, it is the port that is identified not the actual event source. Nevertheless, if the port is bidirectional, the port can be navigated backwards, provided that the exception is passed as a signal event that is specified in the interface contract of the port.

¹⁸ We typically use the *ev* prefix for normal semantic events, and the *ex* prefix for exception events.

¹⁹ If you don't want to use ports, you can add a "null* Sender" pointer as a parameter to each primary event so that its source can be identified and targeted for an exception event when necessary.

²⁰ Jitter is the variance in the period of a periodic process. It is normally specified as an absolute value. A periodic process might have a period of 30 ms +/- 2 ms, where 30 ms is the period and 2 ms is the jitter.

10.4 Summary

In this chapter, we have explored both collaboration and detailed design. As with architectural design, these processes are about optimizing the system against various design criteria. Collaboration design is focused at the level of the mechanism or collaboration, while detailed design focuses on the individual object. There is, naturally enough, some blurring of the lines among architectural, collaboration, and detailed design. Such is the nature of artificial taxonomies, so you will occasionally come across a design decision that is the software equivalent of the platypus and doesn't fit neatly into the normal categorical schema. However, the taxonomy is useful because the literature is itself implicitly organized that way, and the use of the taxonomy makes finding the relevant patterns easier.

All of design is concerned with optimization and most of the design approaches are not fundamentally innovative. They are adaptations of previously used solutions. When a solution can be abstracted and reused in a different context, it becomes a pattern. The notion of patterns is so helpful that there are dozens of books and thousands²¹ of web pages devoted to them.

This chapter wraps up this workshop. We've walked through the Harmony process from identification of use cases all the way through detailed design. I hope that it's been an interesting, useful, and enjoyable trip! ☺

²¹ A quick Google of “design patterns” returns about 78,000,000 hits!

Specifying Requirements: Answers

11.1 Answer 4.1: Identifying Kinds of Requirements

This exercise should hold no difficulties. It is a simple matter to create the requirements elements and link them together in a taxonomy with stereotyped dependencies. In this case, I split the requirements into two diagrams, as shown in [Figure 11.1](#) and [Figure 11.2](#). Note that I used the Rhapsody feature to insert a bitmap image showing the arrangement of the system elements in context.

11.2 Answer 4.2: Identifying Use Cases for the Roadrunner Traffic Light Control System

[Figure 11.3](#) shows a set of use cases for the Roadrunner Traffic Light Control System. The two primary use cases are Configure System and Manage Traffic. The Configure System use case interacts with either the operator of the front panel or the remote monitor. Independently of how the configuration takes place, the system also, of course, manages traffic.

The relations among the use cases were arranged very deliberately. While it would have been possible to just create a single Manage Traffic use case and have a state machine provide the different modes as high-level composite states for that use case, that state machine would have been highly complex. I felt it was better to create subclasses for the different kinds of traffic management modes and have a simpler state machine for each. In this case, the Intersection Controller can switch among these modes but as far as the operational usage of the system is concerned, they represent separate independent uses of the system, hence different use cases. Note that each of these use cases meets the criteria we set up in Chapter 3, with a couple of possible exceptions:

- Each returns a value to at least one actor.
- Each contains at least three scenarios, each scenario consisting of multiple actor-system messages.
- Each logically contains many operational requirements.
- Each does not reveal or imply anything about the internal structure of the system.
- Each is independent of other use cases and may (or may not) be concurrent with them.

The exceptions are the Detect Vehicle and Detect Pedestrian use cases. The former is likely to be a little complex than the latter because of the need to perform vehicle

The Roadrunner Traffic Light Control System: Overview

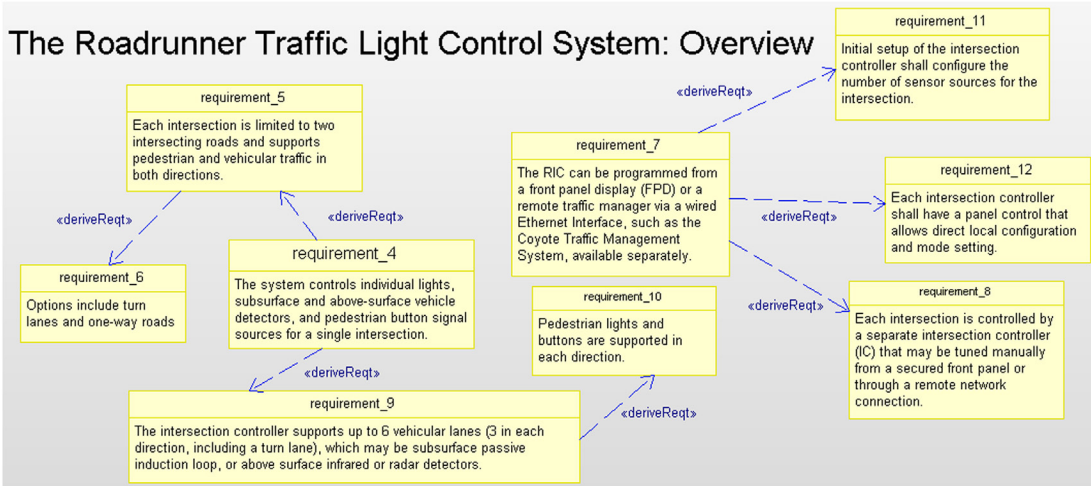


Figure 11.1
Overview requirements diagram.

Roadrunner Traffic Light Control System: Emergency and Priority mode Requirements

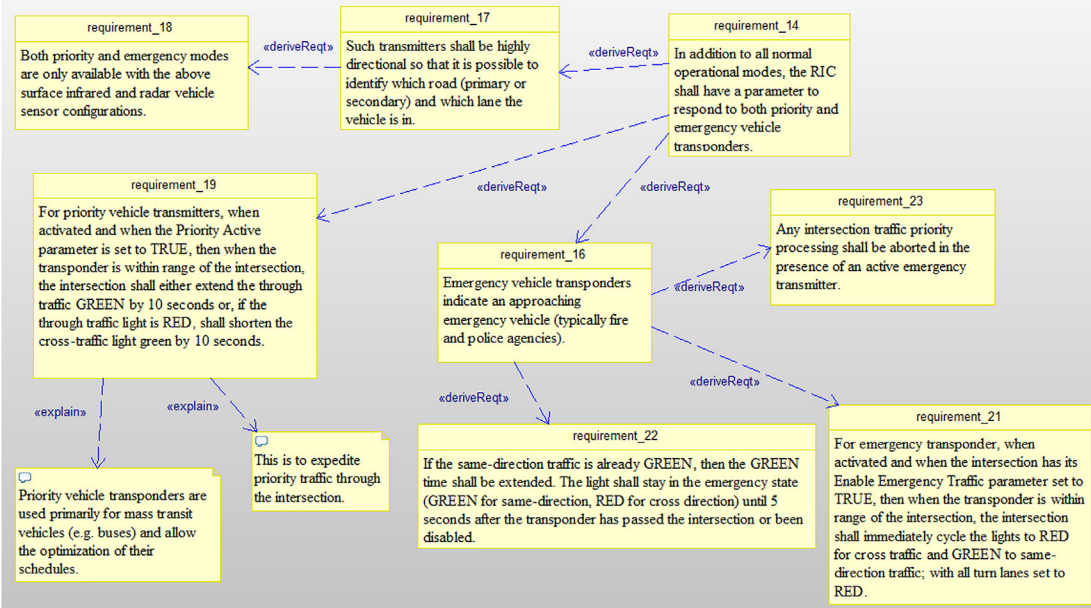


Figure 11.2
Special mode requirements.

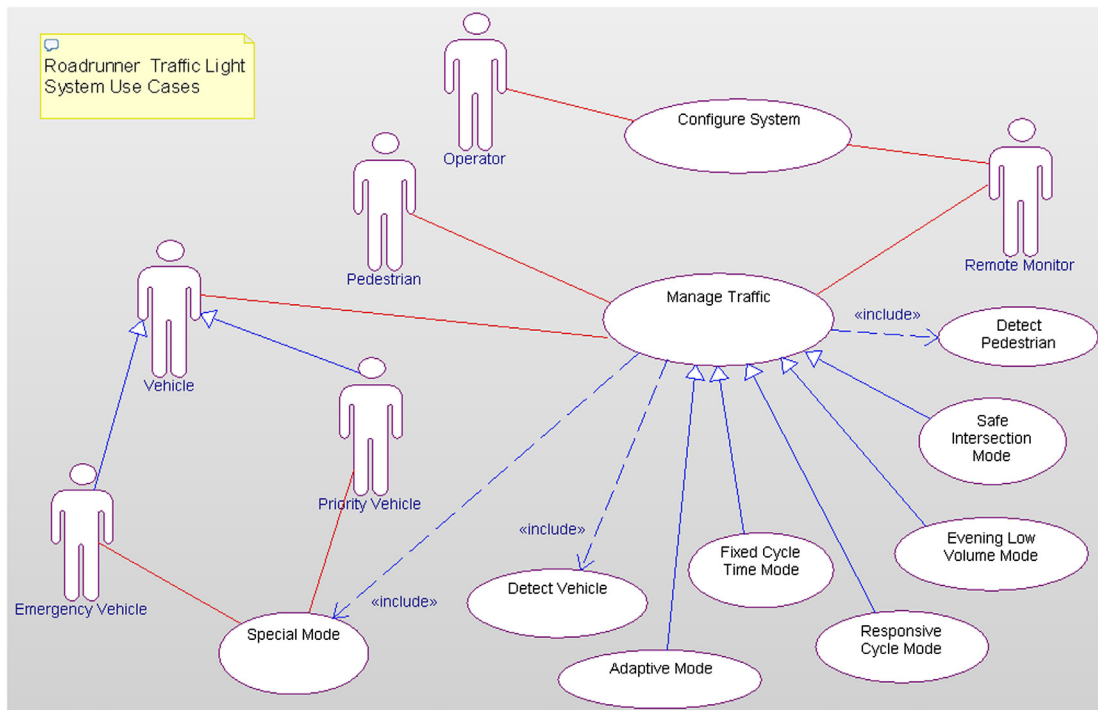


Figure 11.3
Roadrunner use cases.

counting for Responsive Cycle Mode, but the latter will consist mostly of sensing and localizing when a button is pressed. That fact that these small use cases are included by the Manage Traffic use case means that all the specialized traffic management use cases inherit that inclusion as well. If these two use cases turn out to be too small, we can simply absorb them into the Manage Traffic use case and not model vehicle and pedestrian detection as use cases.

The Manage Traffic use case also includes Special Mode, which involves detecting and responding in appropriate fashion to external transponders and modifying the behavior in “normal” operational modes. Hence, inclusion makes sense because that behavior is inherited by all the specialized traffic mode use cases. It was made into a singular Special Mode use case rather than separate Emergency Mode and Priority Mode use cases because the two behaviors share a common requirement (Emergency Mode overrides Priority Mode) and good use cases should have no requirements in common. However, because it was only a single requirement, they could have been broken out because they are *mostly* independent. The point is to model the requirements organization in a way that is consistent, makes sense, supports detailed functional analysis, and can be used to move forward into analysis and design. There are a number of good organizations that can be used for that purpose.

Notice also the specialization of the Vehicle actor. Emergency and priority vehicles are special kinds of vehicles that have special associations with particular use cases as well as the more generic relation due to the fact that they are, in fact, vehicles. While it is fair and reasonable to show generalization/specialization relations among actors, we do *not* show associations between them. Why? First of all, actors are, by definition, outside our scope and we have enough work to do just specifying and designing the parts we need to construct. Secondly, since it is outside our scope, we have little, if any, control over those interactions. If they should change, then our model will be wrong. It is better to focus on the parts we need to build.

What about the choice of the actors? Why these actors? Why not “Button,” “Traffic Light,” “Walk Light” and “Vehicle Sensor” instead? The rule of thumb I use to select actors I call the “In the Box” rule. If the object is inside the box that I ship to the customer – regardless of whether it is electronic, mechanical, or software – then it is an object *internal to the system*. If it is something that either the customer provides or is found in the operational environment, then it is an actor. An actor, then, is the object that interacts with the things found in the box but is itself *not* in the box. The problem statement defines the lights, buttons, and sensors as parts to be designed; they will be in the box shipped to the customer. This is always a crucial question to answer, because as developers and manufacturers we need to clearly understand the system boundary and what we are designing. If the problem statement said, “Interface with Acme Traffic Light Models AAA and BBB,” then those would be the actors and not the vehicles. Having said that, an actor is any object – vegetable, animal (e.g., human), or mineral (e.g., silicon-based hardware device) – that interacts with the system in ways that we care about.

One last point about the actors – what about the Remote Monitor actor? The specification says that the system can be commanded via a network. Note that this actor isn’t a network interface card (NIC), but the element that uses the NIC to do something interesting. We want to capture the proper actor in terms of the semantics of the interaction and not the technology used to communicate; the technology we will deal with in our design.

11.2.1 Additional Questions

What is the problem with having two separate use cases called “Manage Pedestrian Traffic” and “Manage Vehicular Traffic”?

11.3 Answer 4.3: Mapping Requirements to Use Cases

For this problem, I selected the Detect Vehicle use case. I cut and pasted each individual requirement from the problem statement into separate requirements elements and linked them to the use case with dependencies. I felt that it was useful to decompose the use case into two sub-use cases, one for subsurface passive loop detection and one for above-surface detection.

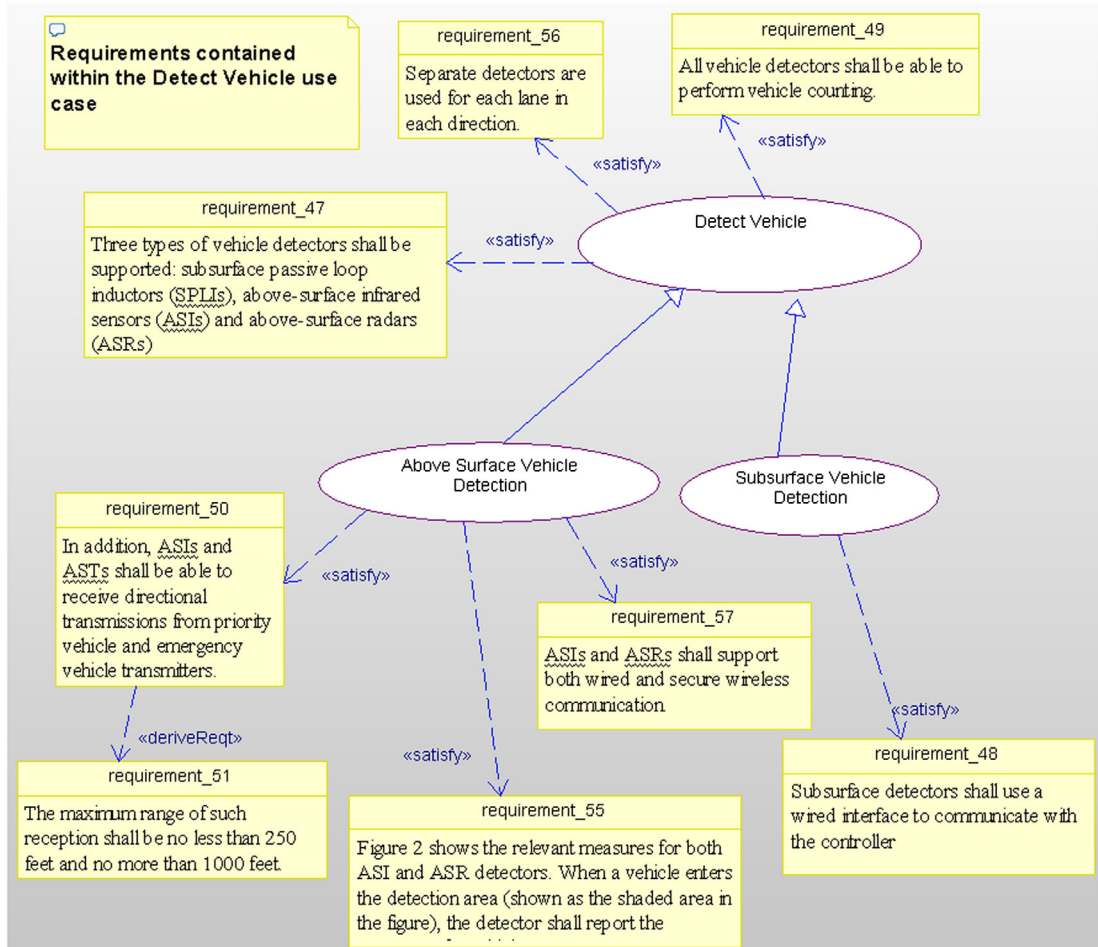


Figure 11.4
Detect Vehicle requirements.

I also pasted in the bitmap to show the ranging of the infrared and radar detectors because I felt it augmented understanding (see [Figure 11.4](#)).

[Figure 11.5](#) is fairly simple, showing these same trace dependencies in tabular form.

11.4 Answer 4.4: Identifying Use Cases for the Coyote UAV System

The Coyote UAV (CUAV) is a large system. If we had all the use cases identified at the same level of abstraction, we might end up with, by the end, hundreds of use cases. Since the purpose of use cases is to organize the operational requirements into usable, coherent chunks, this doesn't meet the need. In the figures below, we have identified the "high-level" use cases

To: Requirement		Scope: System						
From: Use Case		requirement_49	requirement_50	requirement_55	requirement_57	requirement_47	requirement_48	requirement_56
	Above Surface Vehicle Detection							
	Detect Vehicle							
	Subsurface Vehicle Detection							

Figure 11.5

Traceability matrix for Detect Vehicle use case.

and then decomposed them on separate diagrams. We also added a package to hold the parametric requirements.

Figure 11.6 shows the high-level use cases. All but one of these (the Manage Datalink use case) will be decomposed on subsequent diagrams. In Rhapsody, you can add hyperlinks as user-defined navigational links to link the use cases with the more detailed use case diagrams. These high-level use cases may be *abstract* (convenient placeholders in the hierarchy but contain no requirements directly) or *concrete* (contain requirements common to all specialized use cases derived from them). Also note the package `SystemParameters`; it contains a number of nested packages, each of which contains requirements elements and diagrams relevant to their subject matter.

Figure 11.7 shows the decomposition of the Execute Mission use case. This use case contains (via the «include» relation) the Perform Surveillance and Process Surveillance Data use cases. The use case is specialized into two forms – the Execute Reconnaissance Missions and Attack use cases. Each of these use cases is further decomposed. The former is specialized into Execute ECM (electronic countermeasures), Execute Preplanned Reconnaissance, and Execute Remote Controlled Reconnaissance. Preplanned reconnaissance means uploading and following a flight plan that might search an area or a route, or orbit a point target (such as a fixed defensive site).

The Fly UAV use case is decomposed in Figure 11.8. It is important to remember that each of these leaf-level use cases is still a system-level use case. That means that each contains many specific black-box requirements and many scenarios, and they will be detailed with a state machine or activity diagram.

11.4.1 Additional Questions

What would the use case model look like if it were drawn as a flat organization? Draw the diagram of that model and compare and contrast the presented hierarchical model with it. Which is easier to use and navigate?

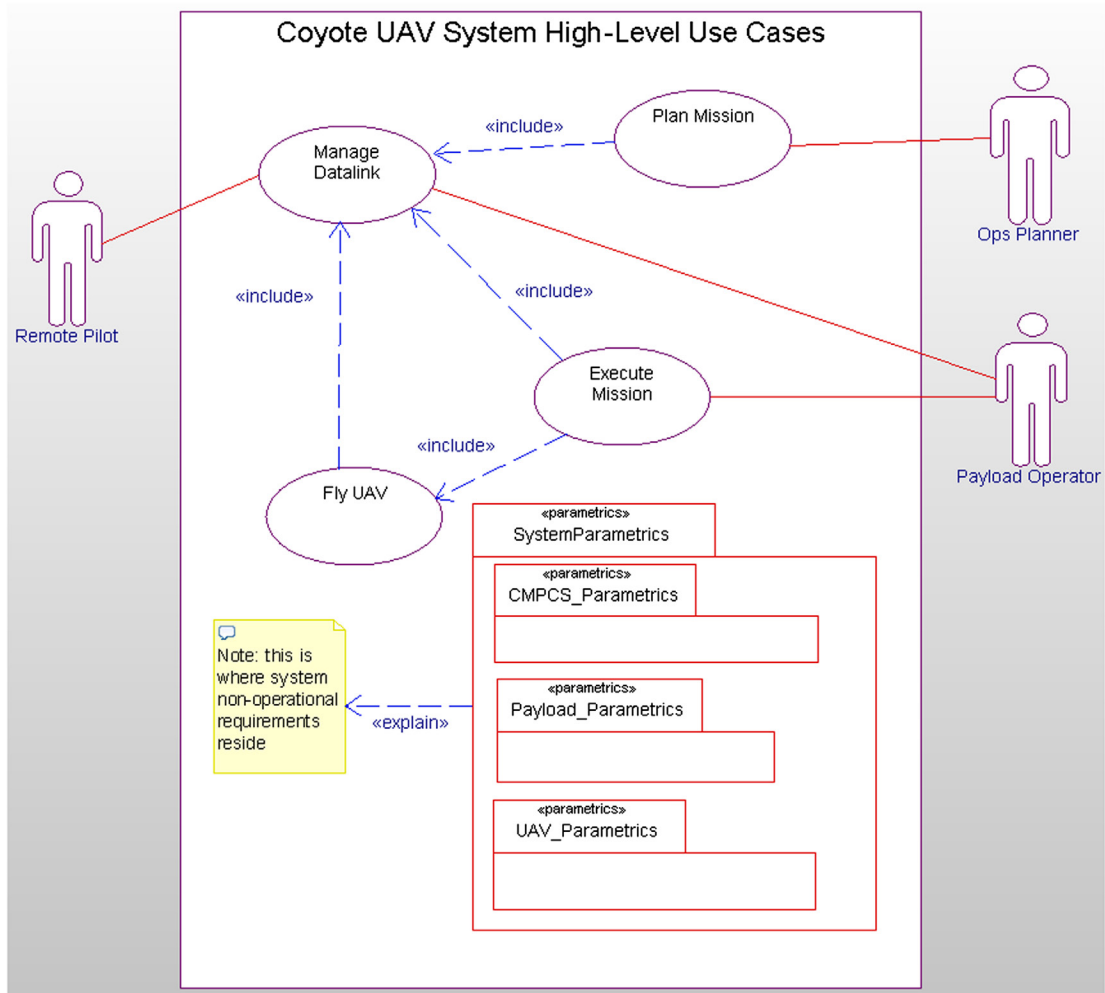


Figure 11.6
Coyote UAV system high-level use cases.

11.5 Answer 4.5: Create a Requirements Table

Figure 11.9 shows a requirements table in Rhapsody. I find that to look at and review many requirements, a tabular form is best. However, a graphical view is best when I want to see the relations between a set of requirements and a single use case, such as we see in Figure 11.4. However, a diagram with hundreds of requirements I find to be unusable.

11.6 Answer 4.6: Capturing Quality of Service Requirements

Quality of service (QoS) requirements are a kind of constraint that is applied to another requirement or to a use case. In real-time systems, QoS requirements are crucial because they

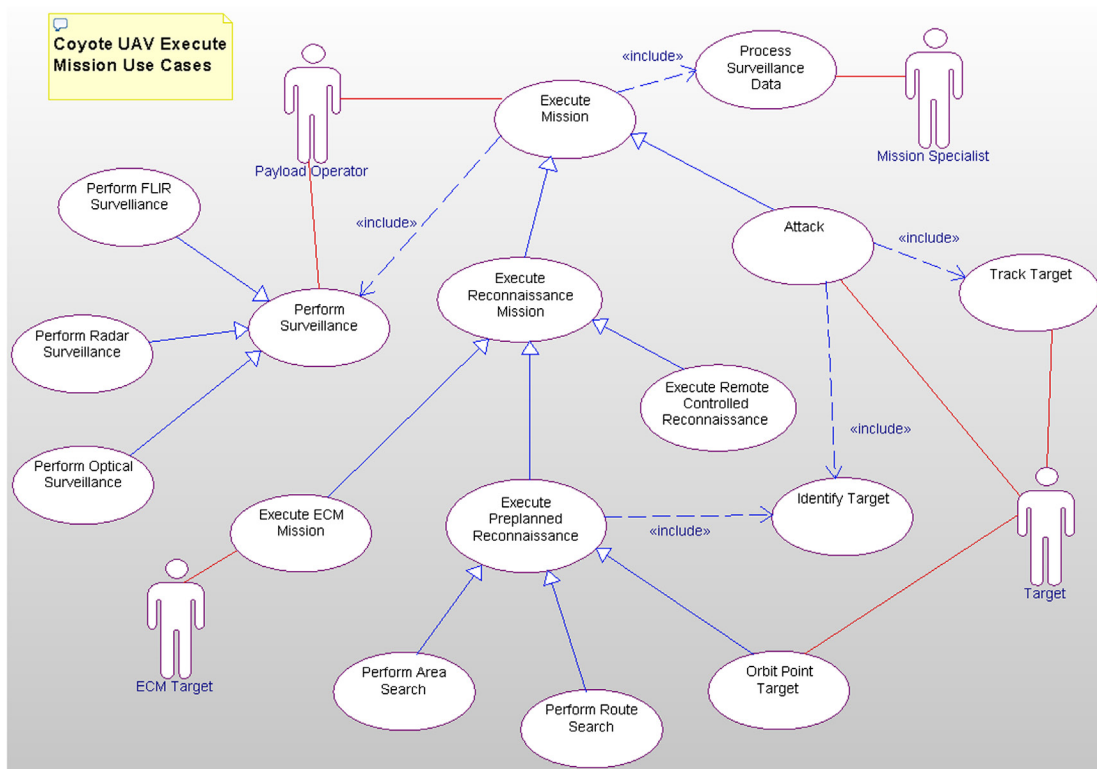


Figure 11.7
Coyote UAV Execute Mission use cases.

represent quantitative requirements such as performance, throughput, and bandwidth – issues vital to any real-time system. In this system, the problem statement identifies a number of QoS requirements around the operation of the vehicle, such as maximum altitude, maximum speed, cruise speed, and so on. If the requirement is not operational, then it should be placed in the `UAV_Parametrics` package, but if they are operational then they should relate to the appropriate use case. In this case, this is the Fly UAV use case. I created a use case diagram, put the previously defined Fly UAV use case on it, and added the QoS requirements to it. They relate to the use case using the «qualifies» stereotype of dependency. This is shown in Figure 11.10.

11.7 Answer 4.7: Operational View: Identifying Traffic Light Scenarios

The use case selected for this exercise was the Responsive Cycle Mode use case. The context diagram for this is shown in Figure 11.11. Note that objects are shown rather than classes as this is necessary to connect them up with links between the ports. These objects are then used as the relevant lifelines on the sequence diagrams.

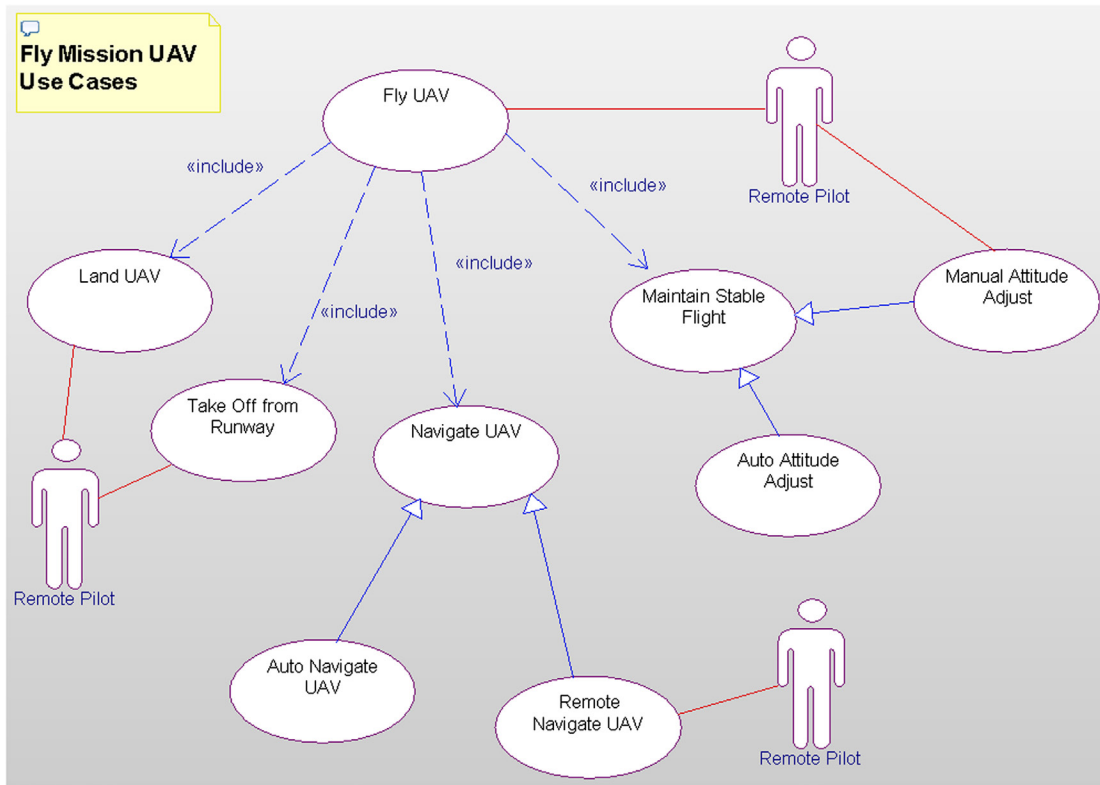


Figure 11.8
Coyote UAV Fly UAV use cases.

Note also that there isn't just a Vehicle and a Pedestrian actor; this is because in the context of the use case, a vehicle along the primary road plays a different role than along the secondary road, just as turn vehicles play a different role in the scenario than through traffic. Note though that in this system south- and north-facing through traffic cars are treated as a single actor. This means they all receive the same message at the same time. Similarly, south-facing-east-turning vehicles along the primary road are treated the same as north-facing-west-turning vehicles. To be clear, the *type* (class) of a primary through or secondary turn vehicle (`Vehicle_a`) is the same, but the instance role is very different. In a system that could send different messages to south-facing-east-turning and north-facing-west-turning traffic, then they would have to be differentiated in the operational context by adding those specific actor instances.

In this use case, the system runs a Fixed Cycle Mode except when a vehicle or pedestrian signals to use the intersection. Many different scenarios suggest themselves, such as:

- No traffic (fixed cycles only)
- Vehicle approaches from road B when road A has green

Name	ID	Specification
Req001 - Suite	Req001	A full CUAVS consists of 4 Coyote Unmanned Air Vehicles (CUAVs) and a ground Coyote Mission Planning and Control system.
Req002 - UAV 1	Req002	The CUAV shall operate at altitude of up to 30,000 feet with ground speeds of up to 100 knots (cruise) and 150 knots (dash) and carry a payload of up to 450 lbs for durations in excess of 24 hours.
Req003 - UAV 2	Req003	The Coyote is meant to fly unimpeded in low-visibility environments while carrying either reconnaissance or attack payloads.
Req004 - UAV 3	Req004	While controllable from the ground station CMPCS, it is also capable of flying complex flight plans with specific operational goals of systematic area search, ground route (road-based) search, and orbit surveillance of point targets.
Req005 - UAV 3	Req005	Coupled with manned control from the ground, the Coyote shall provide sustained 24 hour flight with real-time visual, infra-red or radar telemetry with target recognition preprocessing.
Req006 - Comm 1	Req006	Communications shall be jam-resistant, although need not be anti-jamming in a high ECM environment.
Req007 - Comm2	Req007	Control commands shall be encrypted while telemetry data can be compressed but unprotected.
Req008 - Comm 3	Req008	Telemetry rates of for visual telemetry support 30 frames-per-second (fps) at 640x400 resolution.
Req009 - Flight 1	Req009	Range of flight shall be fully supported within line of sight (LOS) range but since the Coyote also has the ability to be passed among different CMPCSs, its range shall be considerably greater than LOS.
Req010 - Nav 1_	Req010	For navigation, the Coyote has on-board Global Positioning System (GPS) based-navigation as well as being directly controllable from the ground station.
Req011 - Nav2	Req011	For navigation, the Coyote can use an inertial navigation system if GPS is unavailable.
Req012 - Take off and landing	Req012	Unlike many smaller UAVs, the Coyote does not require specialized launch and recovery vehicles. It can use a short runway for either automated or remote-controlled takeoff and landing.
Req100 - CMPCS 1	Req100	Mobile CMPCS shall have capability to control up to 4 UAVs with a manned control station per UAV that fits into a smaller towable trailer.
Req101 - CMPCS 2	Req101	Each control station shall consist of two manned substations - one for controlling the CUAV and one for monitoring and controlling payloads.
Req102 - CMPCS 3	Req102	If desired, both functions can be slaved together into a single control substation.
Req103 - CMPCS 4	Req103	Control of the aircraft shall consist of transferring navigational commands which may be simple (set altitude, speed, direction), operational (fly to coordinate set, orbit point, execute search pattern, etc), planned (upload multi-segment flight plan) or remote controlled with a joystick interface.
Req104 - UAV 4	Req104	Stable flight mechanics shall be managed by the aircraft itself but this can be disabled for remotely controlled flight.
Req105 - CMPCS 5	Req105	The CMPCS displays real-time reconnaissance data as well as maintaining continuous recording and replay capability for up to 96 hours of operation for 4 separate CUAVs.
Req106 - Payload 1	Req106	In addition, with attack payloads, the Coyote can carry up to 4 Hellfire missiles with fire-and-forget navigation systems.
Req107 - Payload 2	Req107	The Coyote UAV shall be easily configured to handle any of several different surveillance and reconnaissance payloads (video, forward-looking infrared (FLIR), radar), countermeasures (electronic countermeasures (ECM) and electronic counter-countermeasures (ECCM)) and attack payloads (Hellfire missiles).
Req108 - Payload 4	Req108	In this latter case, the Coyote performs any of a set of systematic search patterns for a target that matches the target specifications (such as tank, ship, convoy, platoon, soldier, missile launch platform, building) upon which time it notifies the controlling CMPCS.
Req109 - Payload 5	Req109	The coyote can be configured with up to 450 pounds of payload and within that limitation can contain a mixture of surveillance, ECM/ECCM and attack payloads.
Req200 - CDS 1	Req200	The CDS shall communicate between the CUAV and the CMPCS.

Figure 11.9
Coyote UAV system parametric requirements.

- Vehicle approaches from road B when road A has green and primary pedestrian light is WALK
- Vehicle approaches from road B when road A has green for its turn light
- Vehicle approaches from road B when road A has green for its turn light and a pedestrian is waiting to walk parallel with road A
- Pedestrian approaches from road B when road A has green
- Pedestrian approaches from road A when road A has green
- Vehicle approaches from road A to turn when road A has green
- Vehicle approaches from road B when road A has green; once road A turns yellow, a pedestrian approaches on road A
- Vehicle approaches from road B when road A has green; once road A turns yellow, a vehicle approaches on road A

Then, of course, there's dealing with emergency and priority vehicles, system faults, and so on. One can easily think of a few dozen scenarios that show different actor-system interactions entirely contained within this use case.

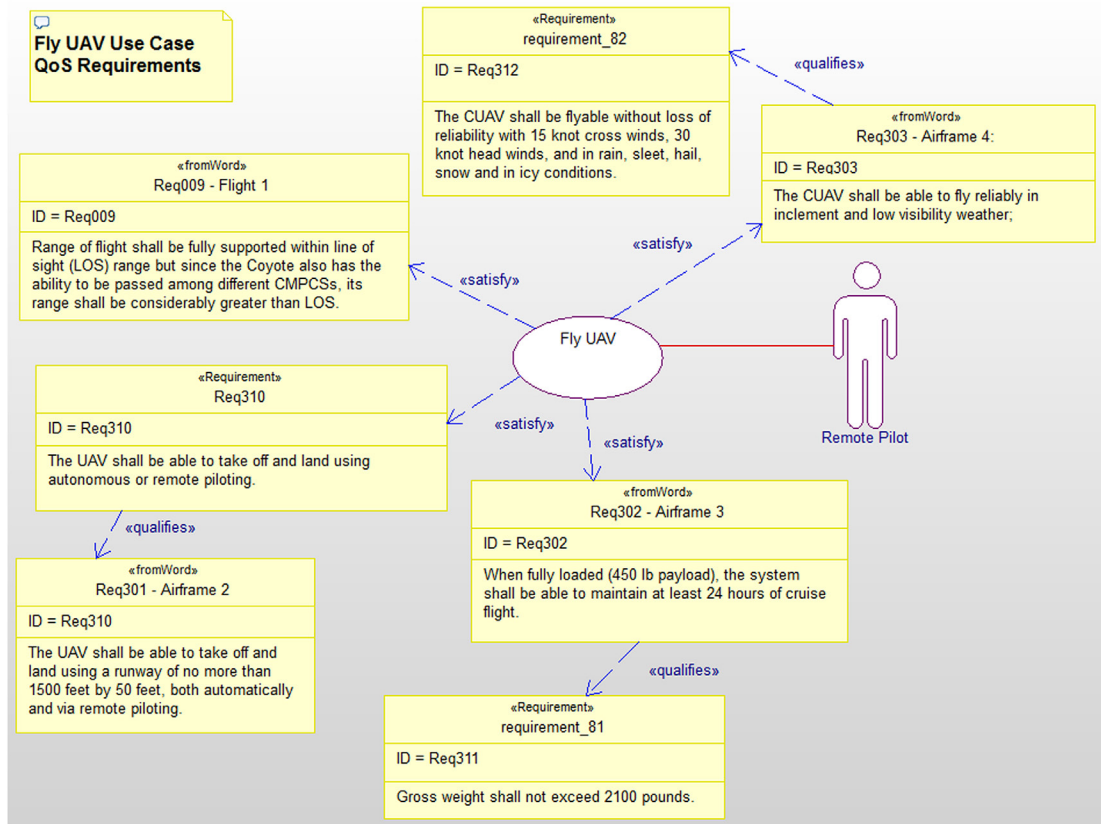


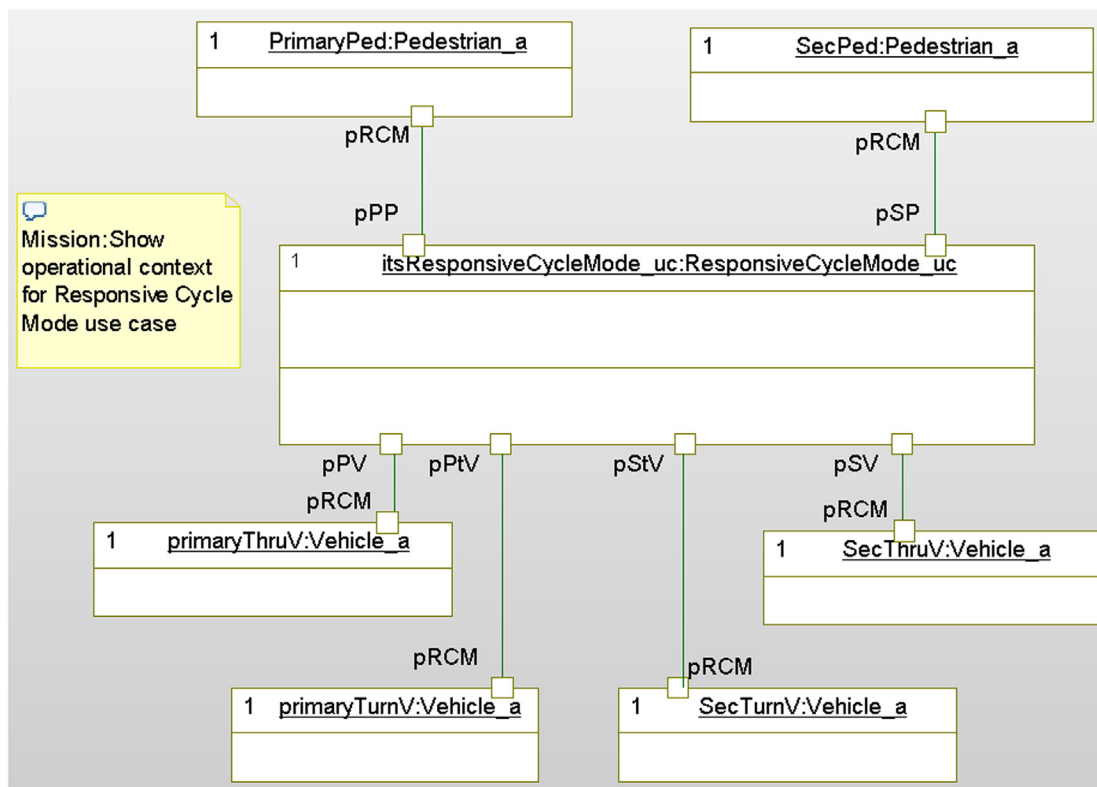
Figure 11.10
QoS requirements.

To answer the problem posed, we only need to construct three scenarios. In our answer, we will elucidate the following scenarios:

- No traffic (fixed cycles only)
- Vehicle approaches on secondary road to turn left when primary road light is green
- Pedestrian approaches from road B when road A has green

There are some interesting things even in the simple, no-traffic scenario, shown in [Figure 11.12](#). First, note that the “system” is represented with the use case as a lifeline. This is perfectly reasonable and it can be interpreted as “the system executing the use case.” Some people find that putting a use case here is counterintuitive, and so they prefer to use the «system» object as the lifeline. This is done in the third scenario; however, the meaning is equivalent. In either case, the lifeline stands for the system executing the use case.

Next, note that the actors in the scenario match the actors from the use case diagram, shown in [Figure 11.3](#). Because the Responsive Cycle Mode is a specialized form of the Manage Traffic

**Figure 11.11**

Operational context for Responsive Cycle Mode use case.

use case, it inherits the actors used by the latter. To create the empty scenario, the use case and the actors were dragged from the Rhapsody browser onto a blank scenario diagram and then the details of the scenario were added. Because the scenario is an exemplar, it refers to specific actor instances. There are four vehicle actors – one each for the primary through, primary turn, secondary through, and secondary turn directions. Similarly, there are two pedestrian actors. In this first simple scenario, the system ignores the turn lanes, so we don't really need the turning vehicle actors to be shown on the diagram. However, we will need them for the other scenarios so I've added them here so that the structure is the same for all of the scenarios.

The astute reader will note that the kind of message sent between the actors and the use case is an asynchronous event.¹ This is common in use case analysis to treat all message exchanges as asynchronous messages as this maps well to a state machine use case specification and simplifies the execution of the use case model. Each one of those messages represents one or

¹ The open arrowhead is the giveaway. A closed arrowhead would indicate a synchronous message, like a function call.

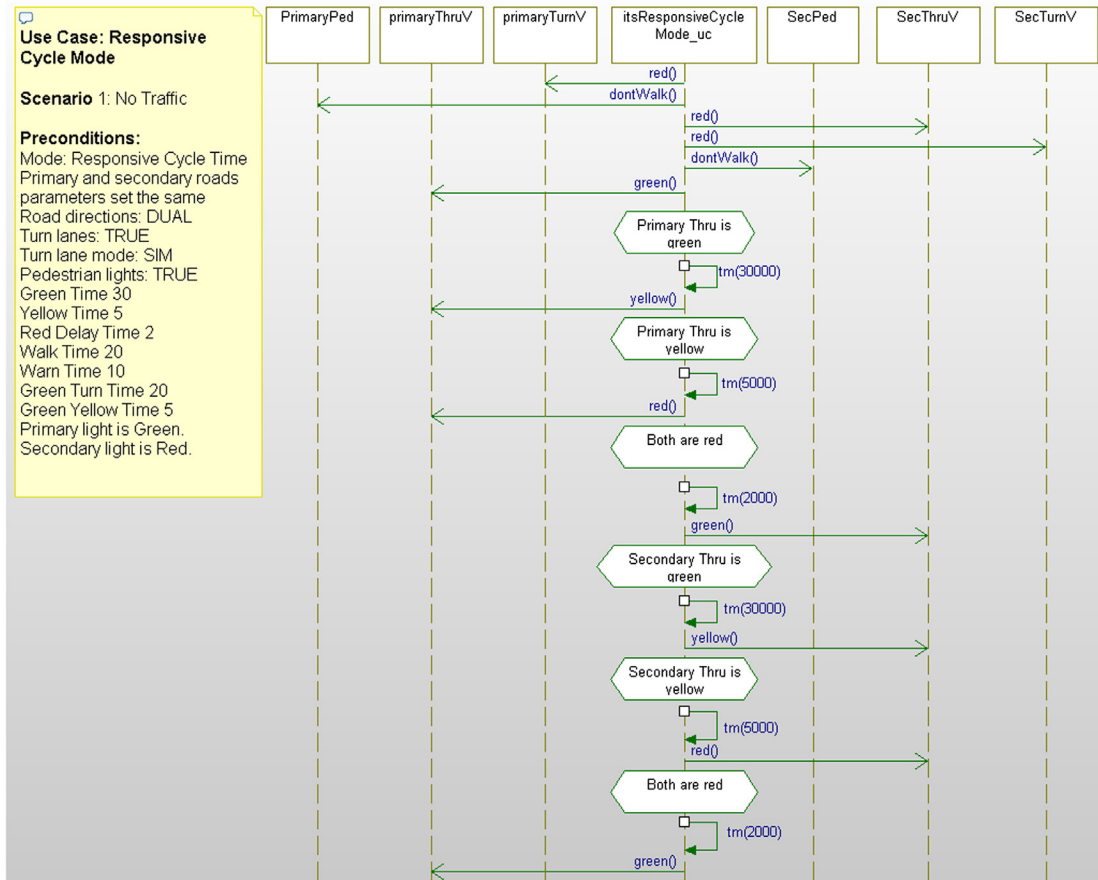


Figure 11.12
 Traffic light Scenario 1.

more textual requirements. There shouldn't be anything "extra" added; if it is needed, then you've identified a missing requirement – one of the valuable outcomes of scenario analysis.

Another aspect to notice is the modeling of time. Some modelers put Time as an actor. However, Time is not an object from which one receives messages. Instead, it is a part of the infrastructure of the universe. Inside the system, we will ultimately have timer objects that provide explicit messages to indicate timeouts, so that these timeout messages are shown as a "message to self." I find that a much more useful representation of time. Rhapsody provides a special icon for a timeout – a message to self that originates from a small box placed on the lifeline. The hexagons in the figure are a UML 2.0 feature called a *condition mark*, which show the condition of the system use case, such as a state or value of an important variable. They are optional but they are included in the first two scenarios because I believe they clarify the scenario. Compare these scenarios with the third and make your own decision as to whether they clarify or obfuscate the scenario.

The first scenario is useful because it shows a very reasonable example of system behavior in the Responsive Cycle Mode – what happens when there is no traffic. A more interesting example is shown in the next scenario, shown in [Figure 11.13](#).

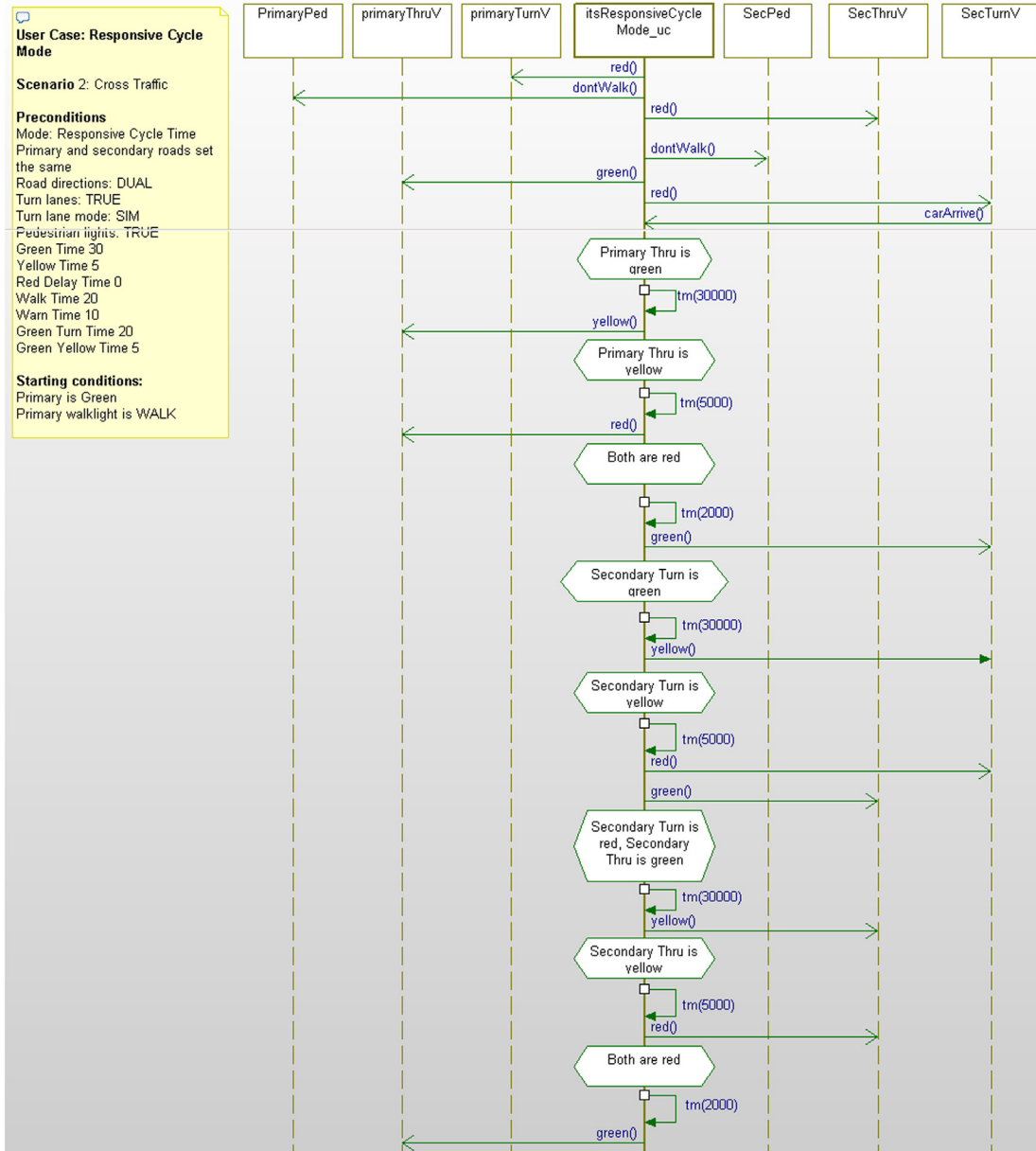


Figure 11.13
 Traffic light Scenario 2.

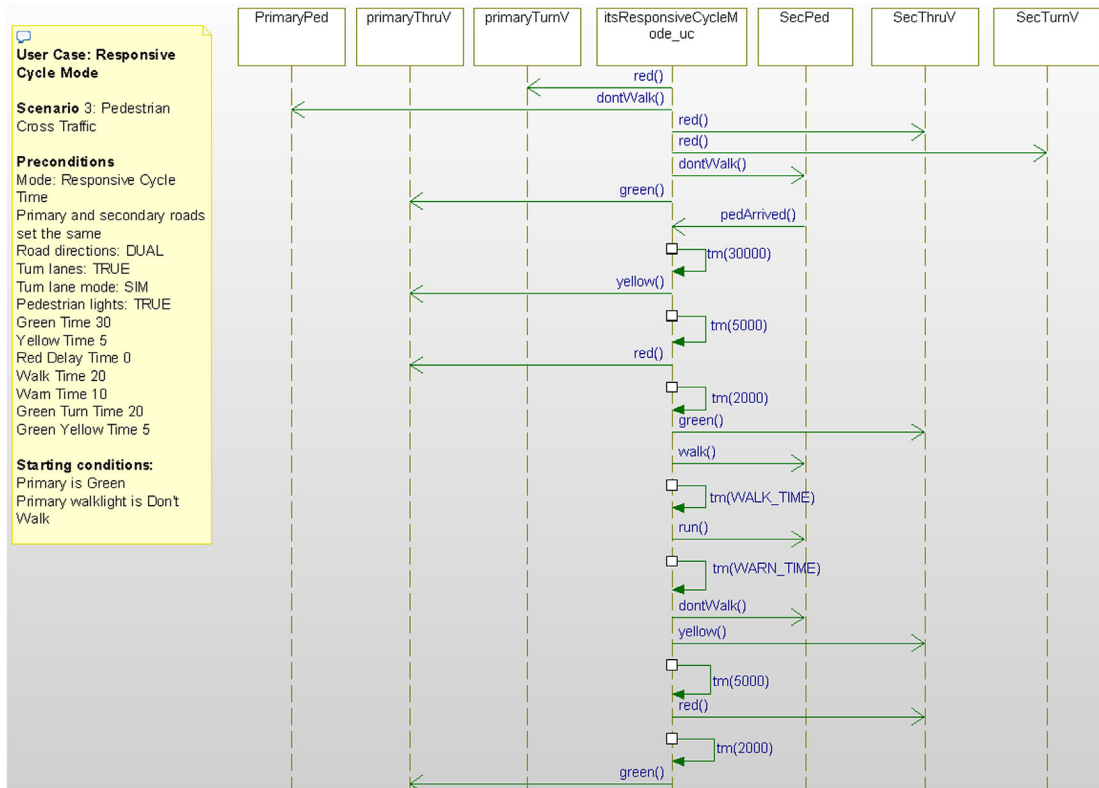


Figure 11.14
Traffic light Scenario 3.

The final scenario in this section occurs when a pedestrian comes from the secondary road when the primary road has green. This scenario is represented a bit differently than the previous two just to show the possibilities. First, the main lifeline in the middle is the «system» object rather than the use case. As previously mentioned, using the use case or the system object is a personal preference. Secondly, the conditional marks are not shown in the use case. Conditional marks can add clarity but they can also obscure the flow of the scenario if overused.

11.8 Answer 4.8: Operational View: Coyote UAV Optical Surveillance Scenarios

These scenarios for the Perform Optical Surveillance use case will seem a bit “high level” because we haven’t identified the internal architecture of the CUAV and its payload. Nevertheless, we can still see how we expect the payload operator (in charge of managing the video camera on the CUAV) to interact with the system. In the next chapter, we will create a system

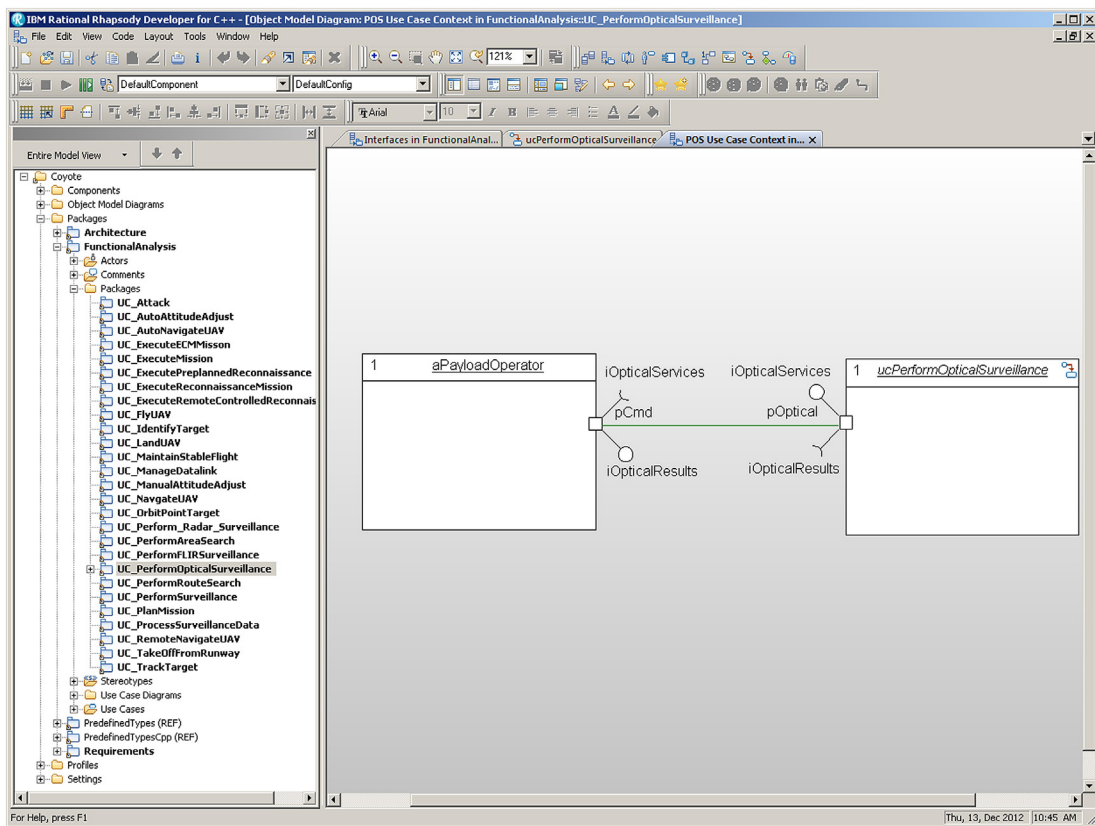


Figure 11.15

Operational context for Perform Optical Surveillance use case.

architecture for the CUAVS and elaborate these scenarios to include that level of detail. For now, however, the CUAVS is a black box that interacts with various identified actors. Figure 11.15 shows the context: the use case has a single actor, the Payload Operator. In this figure, I've shown the named interfaces that group the messages between the system use case and the actor.

The first scenario, in Figure 11.16, shows manual movement of the camera in direct response to commands from the Payload Operator. First, the camera is enabled. Once enabled, the camera begins to transmit video at 30 frames per second at a resolution of 640×480 pixels. The command processing to move the camera is independently processed from the image acquisition. This is shown with the *parallel* interaction fragment operator. The image acquisition occurs repeatedly until the camera is disabled, as shown with the *loop* interaction fragment operator. Note that the process of analyzing this scenario has identified three new requirements: perform a camera power on self-test, auto-position the camera on startup, and limit the camera position to 80 degrees off normal. This identification of missing

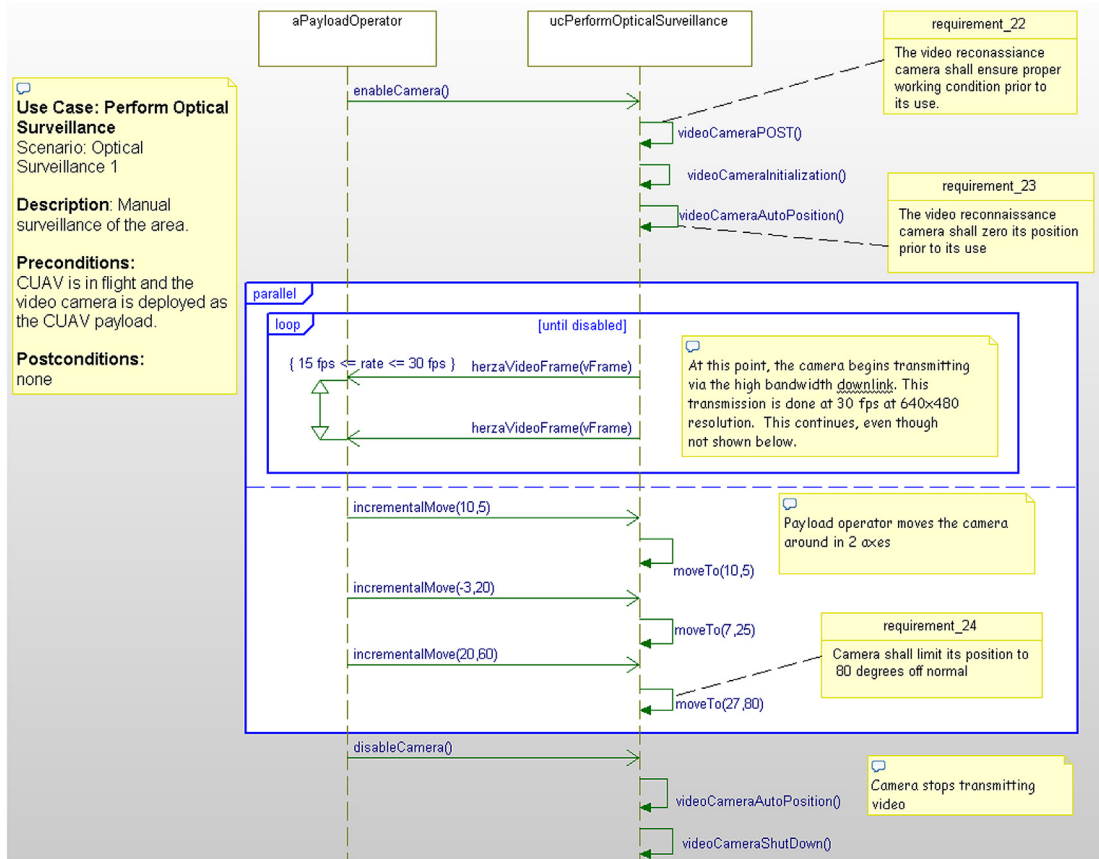


Figure 11.16
CUAV Scenario 1.

requirements is one of the most beneficial outcomes of use case analysis. These requirements are then added to the evolving requirements dataset.

In the second scenario, shown in [Figure 11.17](#), automatic tracking of a target is shown. In this example, the Payload Operator moves the camera around until an “interesting” area is shown. He then selects a portion of the visual display and commands the system to identify targets within that area. The system identifies a target (as specified in `targetSpec`). This target is then highlighted in the visual frame and the potential targets are given target IDs. The operator then selects the target ID for the tank and commands the system to track it. As the tank moves around, the camera automatically adjusts to follow the tank’s movements. At the end of the scenario, the operator commands the system to return to manual tracking operation.

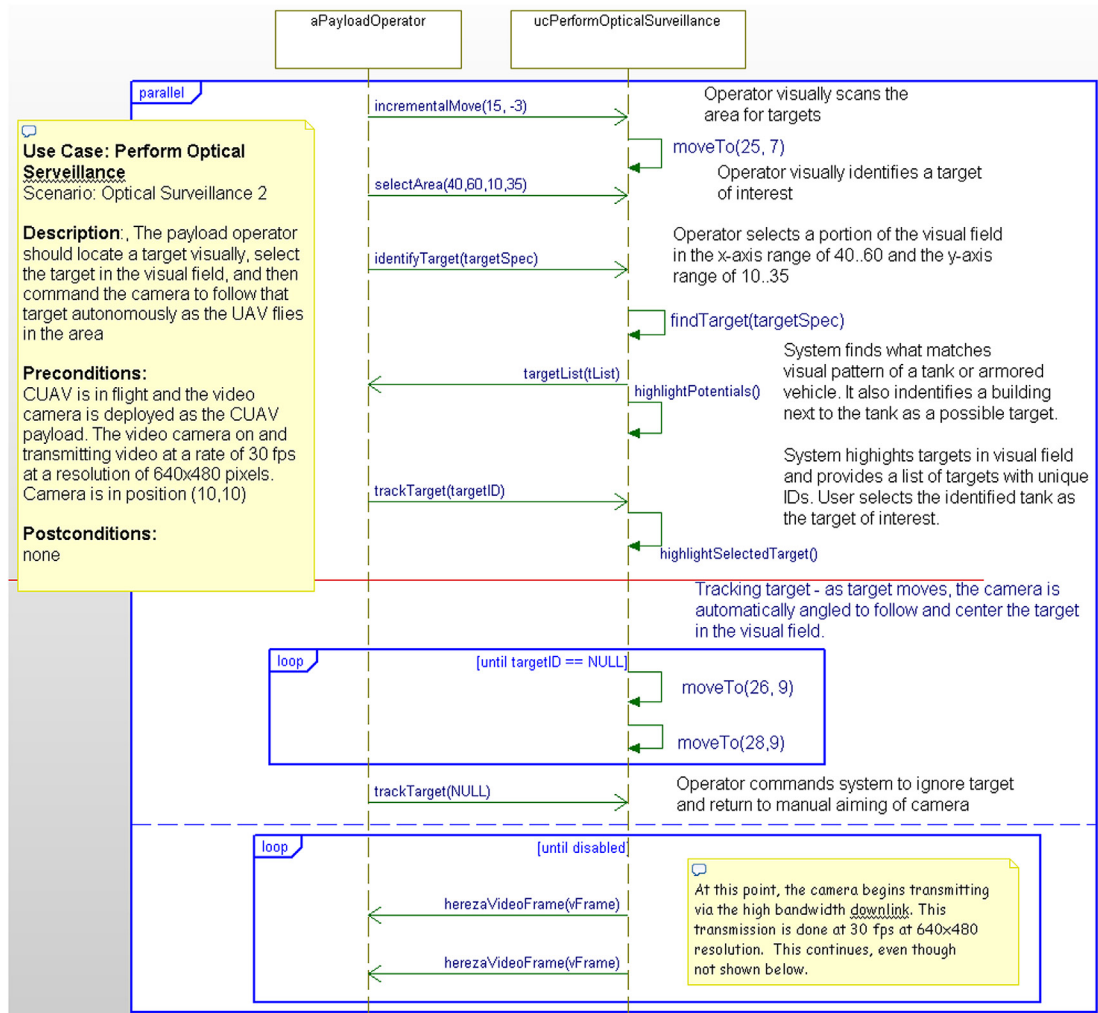


Figure 11.17
CUAV Scenario 2.

The last scenario, shown in Figure 11.18, involves the operator selecting an area, centering it, and then zooming in. Once zoomed in 300%, he pans around, eventually decides there is nothing of interest, and so resets the zoom level to normal (100%).

11.9 Answer 4.9: Specification View: Use Case Descriptions

This problem should pose no difficulties for the reader. The purpose of the problem is just to provide an opportunity to actually create a use case description. Use case descriptions are important because they provide necessary documentation for the understanding of the use

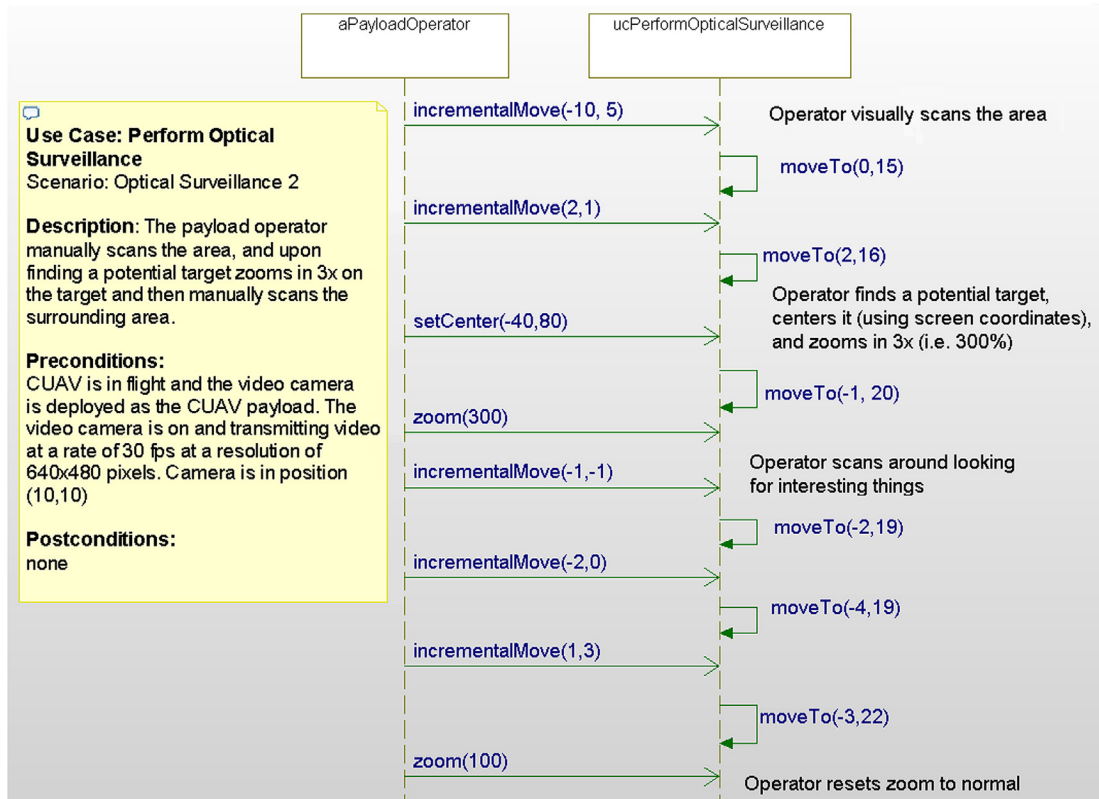


Figure 11.18
CUAV Scenario 3.

case (Figures 11.19 and 11.20). Remember – just because you’re building a model doesn’t mean that you don’t need description and comments!

11.10 Answer 4.10: Simple State Machine Specification

This problem should be straightforward and is meant to be a gentle introduction to the power that state machines bring to the requirements capturing process. This behavior can be expressed with three states – one to provide a delay before you start the flashing (to make sure vehicles in the intersection when this mode is initiated can clear the intersection), one for flashing “on,” and one for flashing “off.” The transitions between the states are all based on time.

Figure 11.21 shows the state machine context diagram and Figure 11.22 shows the state machine. Note that the state machine could have used methods such as `setPrimary(RED)` and hide the mechanism for actually generating the event in that method body or put the event generation directly on the diagram. Since the elements in Figure 11.21 are connected via

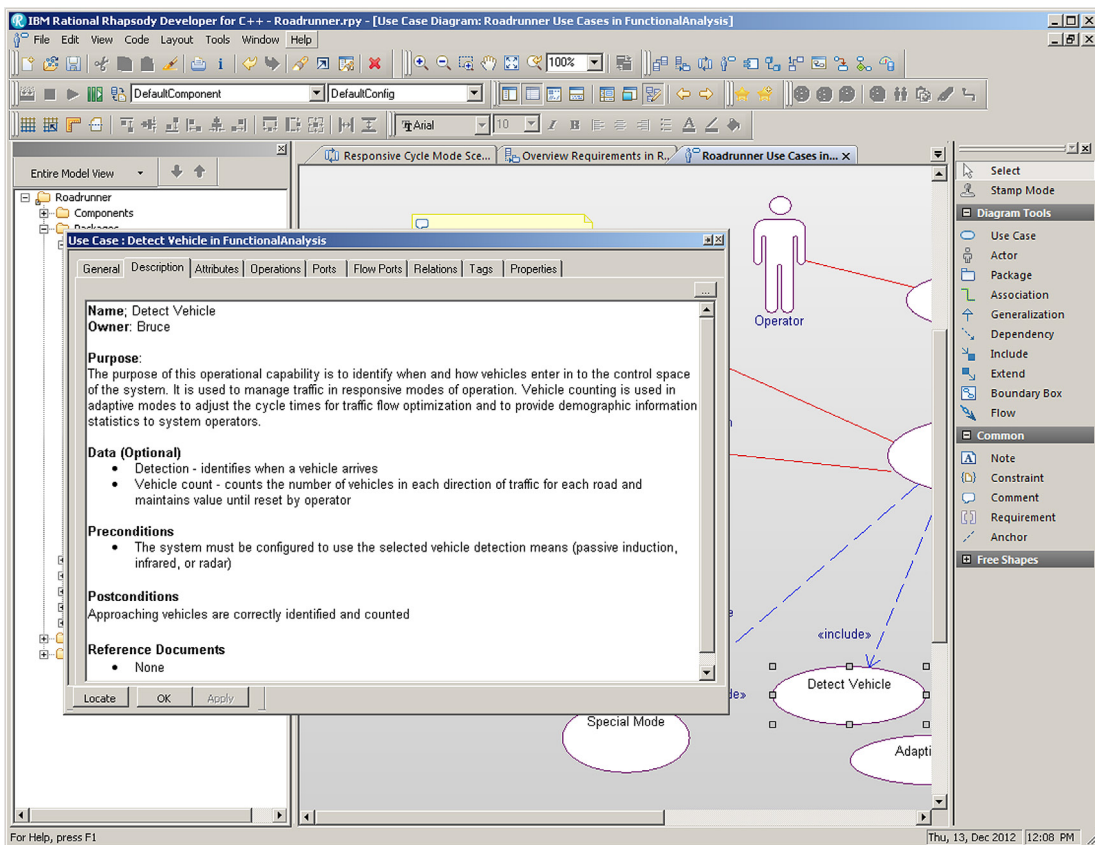


Figure 11.19
Roadrunner use case description.

ports, the events generated to the actors must be sent out via those ports. This example uses the Rhapsody-specific event generation syntax but subsequently we'll hide the event generation in methods to improve state machine diagram readability.

11.11 Answer 4.11: Specification View: Capturing Complex Requirements

The state machine for Fixed Cycle Mode is significantly more complex than for the Evening Low Volume Mode use case. We will incrementally develop and verify this solution using the Harmony nanocycle approach.² Too many modelers fail in their efforts because

² I've used these examples in my teaching many times. In my experience, if I give the problem to a group without forcing them to do a nanocycle development of the use case model, *none* of them will get it completely right and many will be seriously flawed. If, however, I force them to use the nanocycle approach almost all groups will succeed.

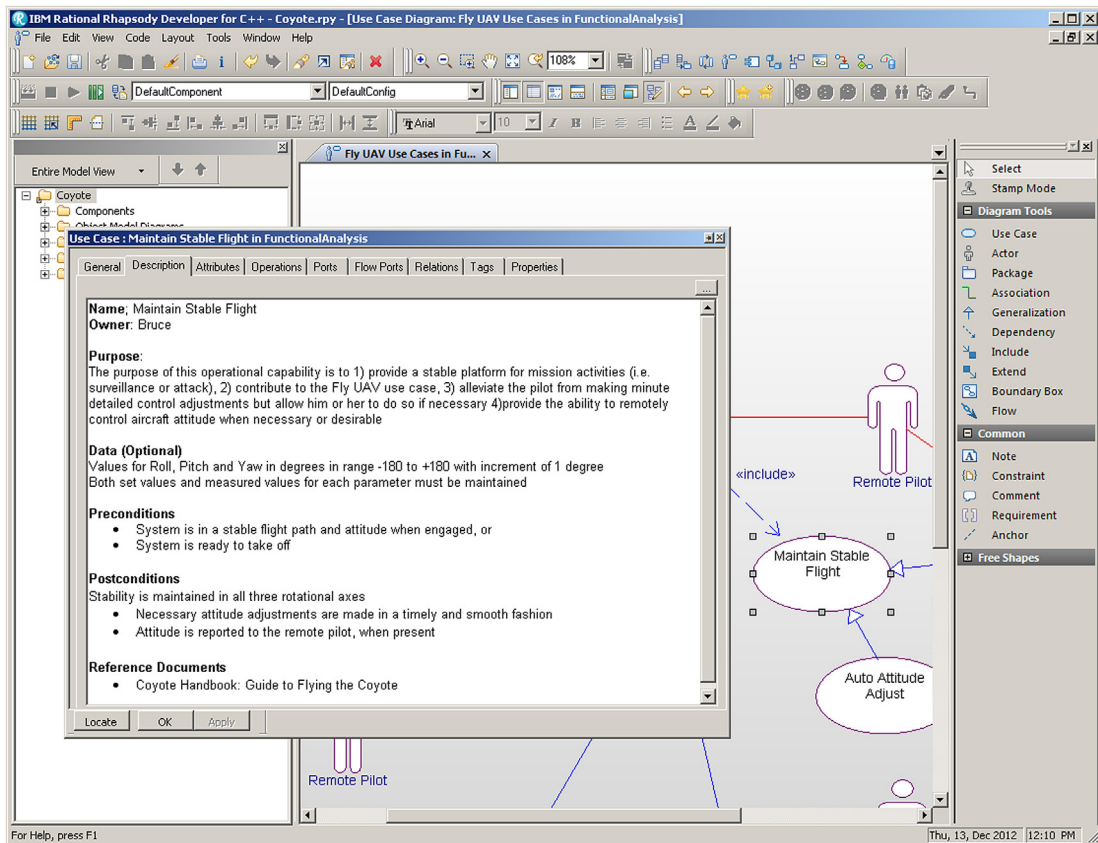


Figure 11.20
Coyote UAV use case description.

they wait too long before they try to execute the model. By solving the problem in small steps and demonstrating that the solution works via execution, we are implementing the nanocycle of the Harmony process, described in Chapter 2. This enables us to achieve a correct solution much more efficiently than the more common OGIHTW approach to modeling.³

To simplify our discussion, we will develop the Fixed Cycle Mode use case in three phases:

1. No cars or pedestrians are considered (just get through traffic working)
2. Add turn lanes
3. Add pedestrians

³ Oh God I Hope This Works – a “problematic” approach to engineering.

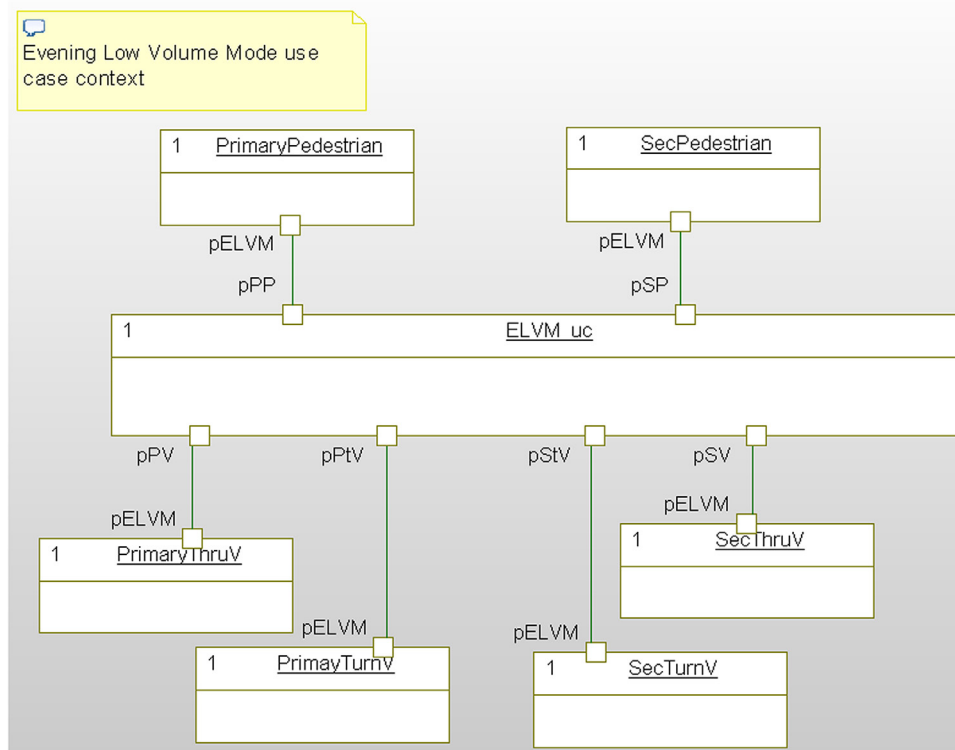


Figure 11.21
Evening Low Volume Mode (ELVM) use case context.

In actual practice, you might have six or more nanocycles, such as

1. Initialize the lights
2. Get primary through road cycling
3. Add secondary through road
4. Add primary turn lane
5. Add secondary turn lane
6. Add primary pedestrian with through traffic
7. Add primary pedestrian with turn traffic
8. Add secondary pedestrian with through traffic
9. Add secondary pedestrian with turn traffic

We'll go with just the first three for the purpose of this book.

The context diagram for this use case looks pretty much identical to [Figure 11.21](#), except that the name of the object in the middle is `itsFixedCycleMode_uc`. In actual practice, the state machine will evolve as we proceed through the nanocycles. In this case, for pedagogical reasons we'll subclass the `itsFixedCycleMode_uc` class and extend it in the subclasses as we proceed through the nanocycles.

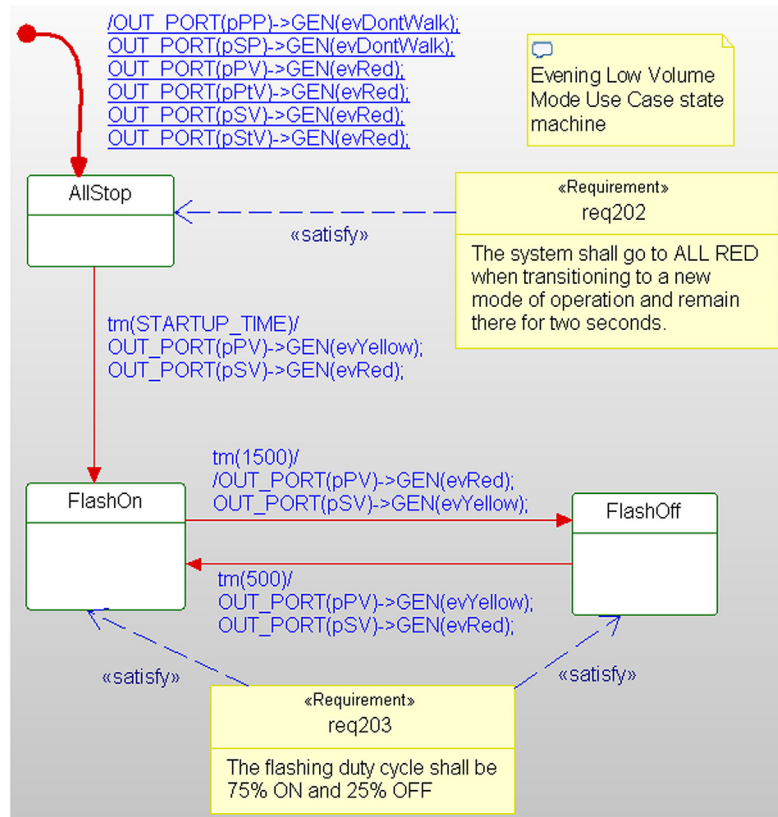


Figure 11.22
ELVM use case state machine.

11.11.1 Step 1: No Cars or Pedestrians Are Considered (Just Get Through Traffic Working)

The first step is to get the basic Fixed Cycle Mode behavior working. This is shown in [Figure 11.23](#). Getting this state machine to work is fairly easy. Note the use of and-states. Looking ahead, I thought about how I might want to extend the state machine for the additional requirements and it appeared that and-states might simplify that progression. However, this can be done without and-states if desired.

The initialization of the actors is done by the `initialize()` operation, whose implementation is presented in [Code Listing 11.1](#).

You will note that (to facilitate debugging), I've added a name and `laneID` to the actors so when I have them print out their state I know which actor is speaking. `laneID` is an enumerated type: `enum LANE_TYPE {PRIMARY_THRU, PRIMARY_TURN, SECONDARY_THRU, SECONDARY_TURN};`

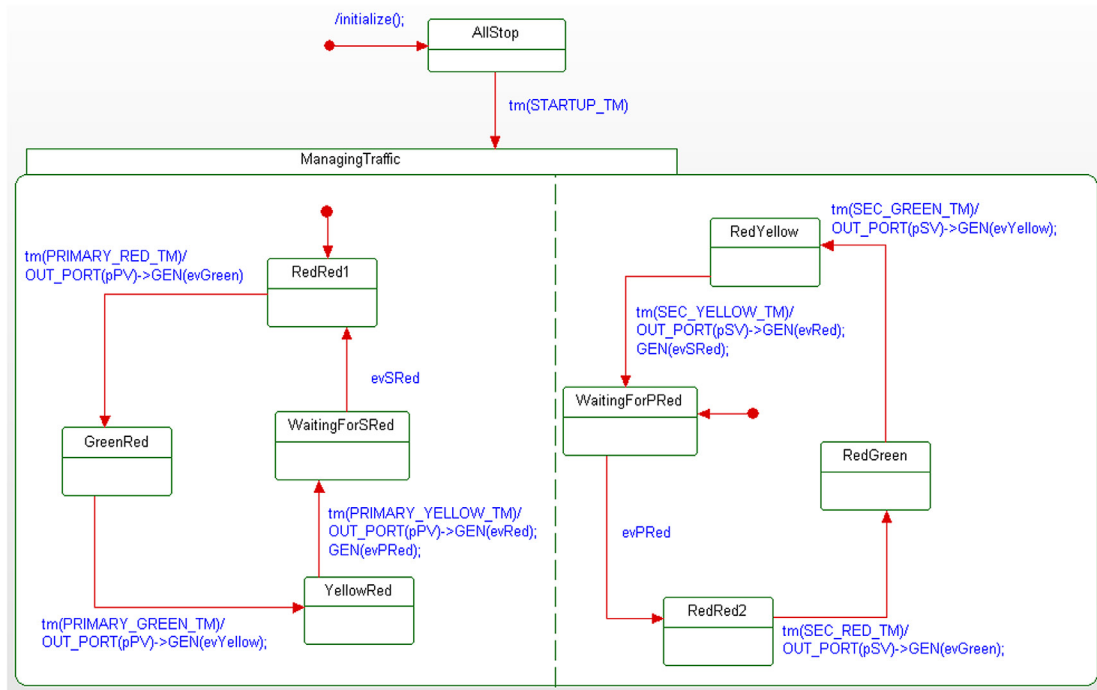


Figure 11.23
Basic Fixed Cycle Mode.

```
void initialize() {
    // Setup the actors with their identify
    // First pedestrian lanes
    OUT_PORT(pPP) -> GEN(evSetName("Primary Through"));
    OUT_PORT(pPP) -> GEN(evSetID(PRIMARY_THRU));
    OUT_PORT(pSP) -> GEN(evSetName("Secondary Through"));
    OUT_PORT(pSP) -> GEN(evSetID(SECONDARY_THRU));
    // Now vehicle lanes
    OUT_PORT(pPV) -> GEN(evSetName("Primary Through"));
    OUT_PORT(pPV) -> GEN(evSetID(PRIMARY_THRU));
    OUT_PORT(pPtV) -> GEN(evSetName("Primary Turn"));
    OUT_PORT(pPtV) -> GEN(evSetID(PRIMARY_TURN));

    OUT_PORT(pSV) -> GEN(evSetName("Secondary Through"));
    OUT_PORT(pSV) -> GEN(evSetID(SECONDARY_THRU));
    OUT_PORT(pStV) -> GEN(evSetName("Secondary Turn"));
    OUT_PORT(pStV) -> GEN(evSetID(SECONDARY_TURN));

    // Now initialize them
    OUT_PORT(pPP) -> GEN(evDontWalk);
    OUT_PORT(pSP) -> GEN(evDontWalk);
    OUT_PORT(pPV) -> GEN(evRed);
    OUT_PORT(pPtV) -> GEN(evRed);
    OUT_PORT(pSV) -> GEN(evRed);
    OUT_PORT(pStV) -> GEN(evRed);
}
```

Code Listing 11.1
Implementation of `FixedCycleMode_uc::initialize` operation.

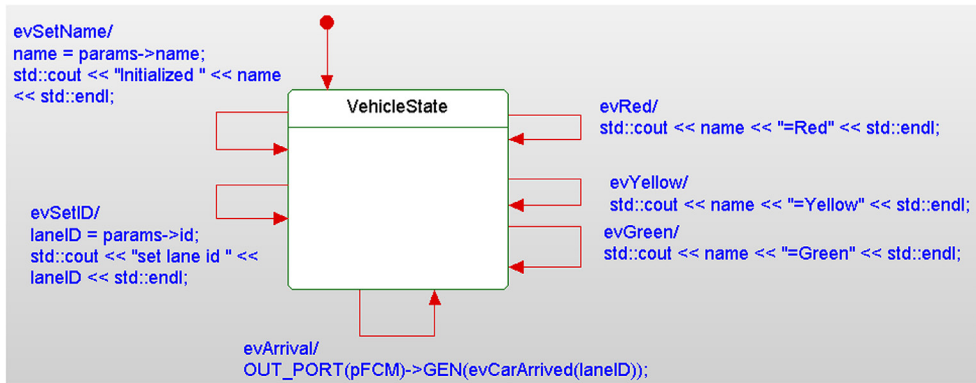


Figure 11.24
State machine for `Vehicle_a` class.

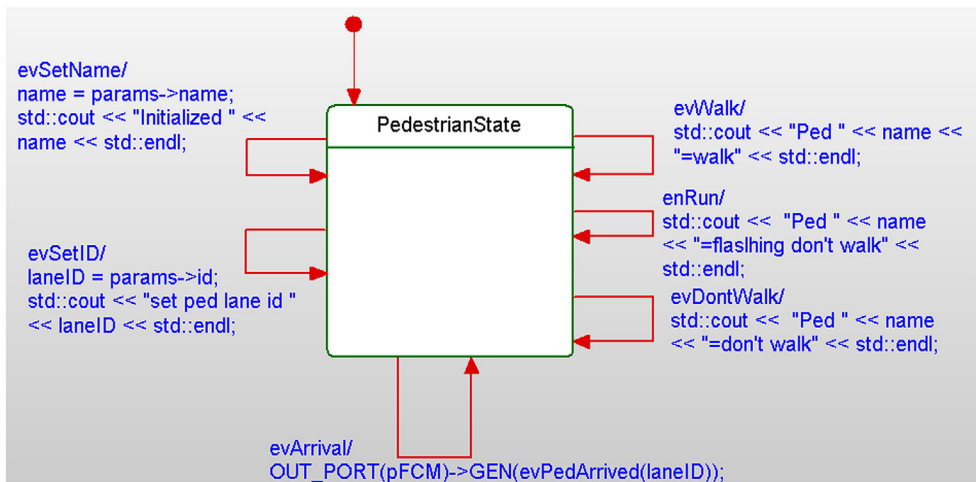


Figure 11.25
State machine for `Pedestrian_a` class.

I have also “instrumented” the actors; that is, I’ve defined a state machine for the vehicle and pedestrian classes to support simulation and execution. The instrumentation is shown in [Figures 11.24 and 11.25](#). This instrumentation just aids in the debugging of the simulation and isn’t strictly required.

We can see the state machine in [Figure 11.23](#) is correct by executing it. In this case, Rhapsody can generate a sequence diagram to capture the execution. It is shown in [Figure 11.26](#). I removed the messages due to initialization to reduce the size of the diagram, as execution of the `initialize()` function creates a number of messages sent to the actors to initialize them.

11.11.2 Step 2: Add Turn Lanes

The next step is to add turn lanes. While I'll show this as a *fait accompli*, in reality I constructed this by first adding the latches, executing that version, and then adding the turn lanes.

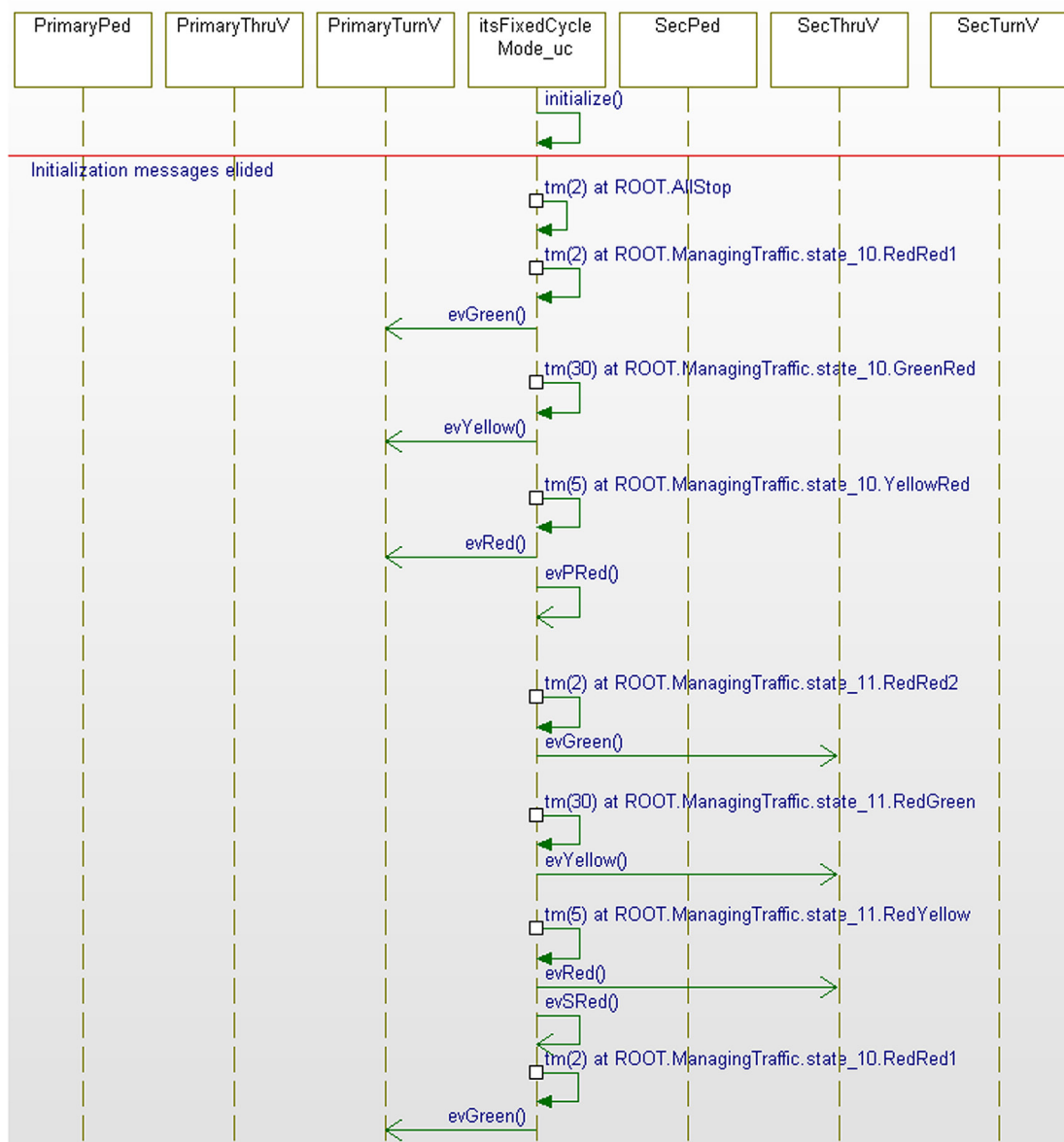


Figure 11.26
Execution of Fixed Cycle Mode state machine.

Figure 11.27 shows the use case state machine that now handles both through and turn traffic. You can see that although the behavior is simple in the abstract sense, the state machine is beginning to appear complex. The downside of writing precise models is that they must be, well, *precise* and address all circumstances. The upside is that through the execution/simulation of this use case, I was able to quickly identify and repair errors that would have likely passed through review without comment.

The state machine uses “latch” and-states to remember the information that a primary turn or secondary turn car has arrived until that information can be acted on. Other and-states handle the turn lanes specifically. Note that the only change to the states defined in [Figure 11.23](#) is the addition of the new states `WaitingForTurn` and `WaitingForSTurn`, and the conditional that

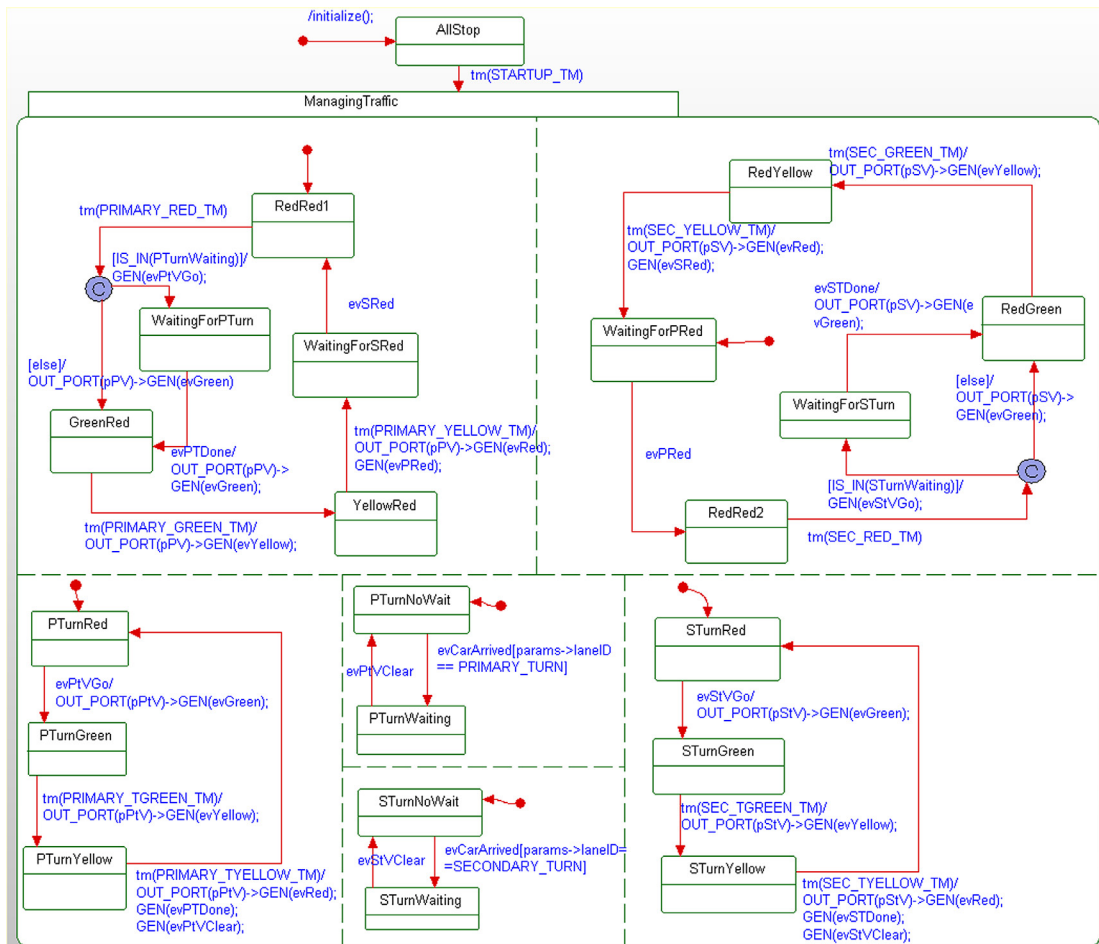


Figure 11.27
Fixed Cycle Mode with turn lanes.

directs the transition to either normal through processing (no cars waiting) or to manage the turn lane. This use case state machine now handles all ways in which through lane cars and turn lane cars are managed, regardless of when they arrive in the cycle. Yet to be added is the integration of pedestrian crossing management.

To support the simulation, I added a testing object (`itsFCMTester1`) to the build that generates `evArrival` events to the `PrimaryTurnV` or `SecTurnV` actors. The block diagram for the context looks like Figure 11.28. Note that in this case, I've used a "builder class" called `FCMBuilder_Turn` to generate all the links for me. It's a simple trick I use a lot to quickly get collaborations instantiated and linked together (Rhapsody generates the initialization code for the links automatically in this case).

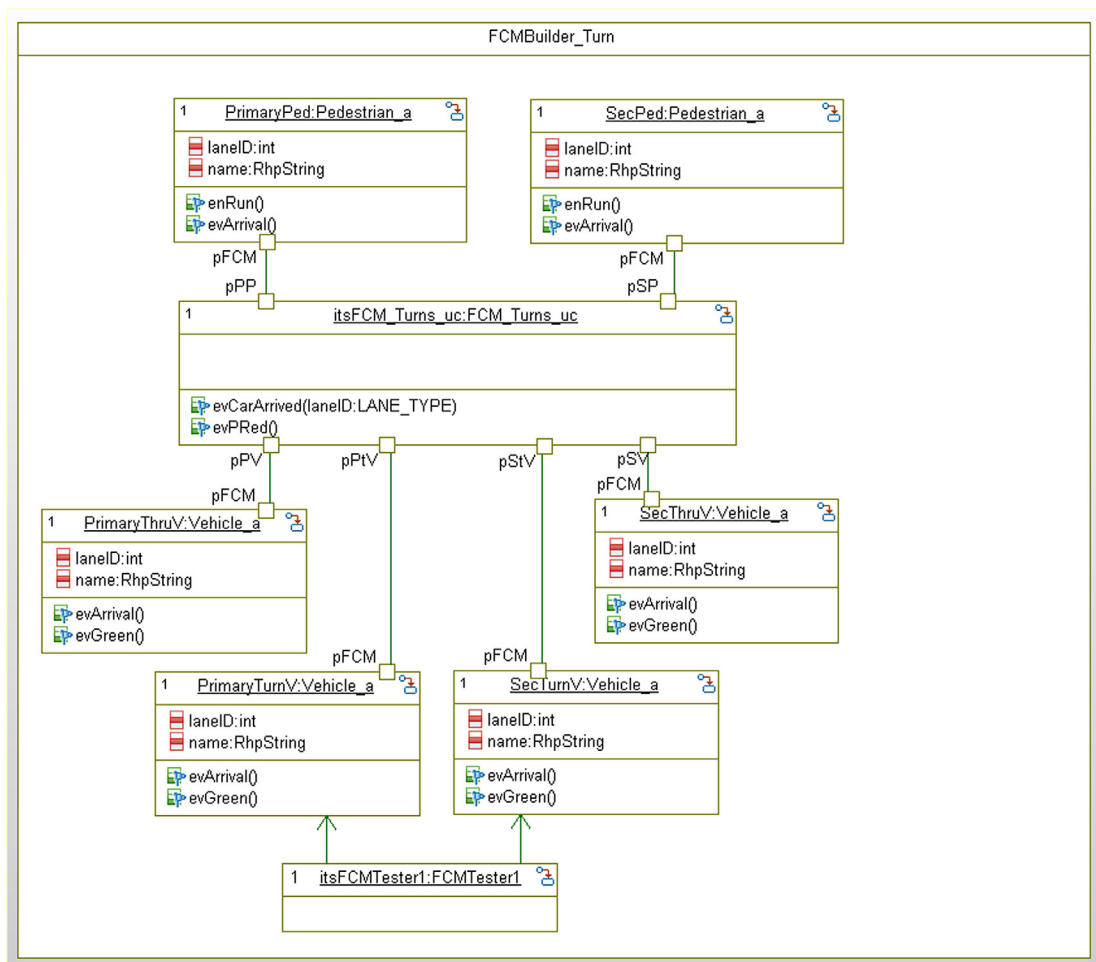


Figure 11.28
Fixed Cycle Mode with turns context.

11.11.3 Step 3: Add Pedestrians

Once this model executes properly, I can move on to add the remaining features. In the last step, I added the pedestrian traffic management. The context view is the same as [Figure 11.28](#) except that links are added from its `FCMTester1` instance to the two pedestrian instances, so they can be stimulated as well during simulation. The complete state machine, including through and turn vehicle traffic, is shown in [Figure 11.29](#).

An example simulation scenario is provided in [Figure 11.30](#). In this case the tester was commanded to insert a secondary pedestrian arrival while the traffic light was about to go GREEN for the primary road. You can see how the pedestrian for the secondary road is properly handled in this case: the secondary through light remains GREEN until the pedestrian light has cycled back to DON'T WALK, and then the secondary light goes to YELLOW.

11.11.4 Additional Questions

- Many other scenarios are needed to form a testing set. Identify at least five scenarios that test different combinations of operating conditions
- Run a scenario via simulation in which a primary turn vehicle, secondary turn vehicle, and a secondary pedestrian are present and ensure the scenario unfolds properly. Were any missing requirements identified?

11.12 Answer 4.12: Operational to Specification View: Capturing Operational Contracts

In this exercise, you were given several distinct tasks to accomplish:

1. Draw the block diagram representing the use case, the actor(s), and the interfaces between them.
2. The messages that come from the actor(s) must be collected into *provided* interfaces on the appropriate port of the use case block and *required* interfaces on the actor. Messages sent to the actor(s) must be collected into *required* interfaces on the use case block and *provided* interfaces on the actor port. Messages to the use case block should have public visibility while reflexive messages (if any) should be protected or private.
3. For each service (event reception) the system or its actors provides, specify pre- and postconditions of the service and the logical data types of parameters, if necessary. Model the services as asynchronous event receptions that pass data as necessary.
4. For each logical data type that defines data passed as a part of the services, specify the important data attributes and properties.
5. Construct a use-case activity diagram representing *all* of the scenarios previously specified for the use case.
6. Define a state machine for the use case block that is consistent with the set of scenarios

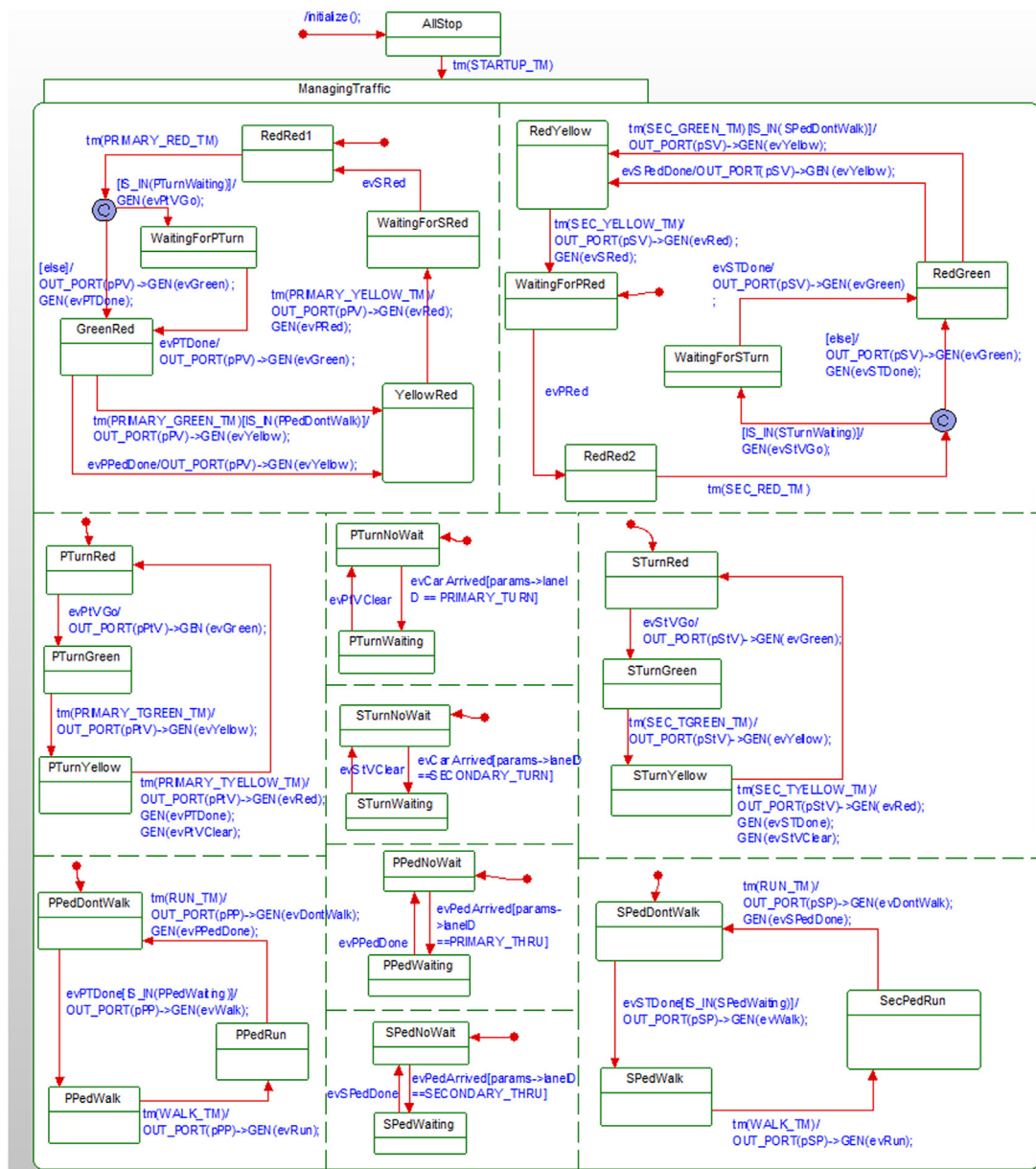


Figure 11.29
Manage primary traffic state machine.

11.12.1 Step 1: Draw the Block Diagram

Figure 11.31 shows the block diagram representing the Perform Optical Surveillance use case. Note that the use case and actor have both become blocks (objects), complete with ports and interfaces. The use case block provides one interface, `iOpticalServices`; this same

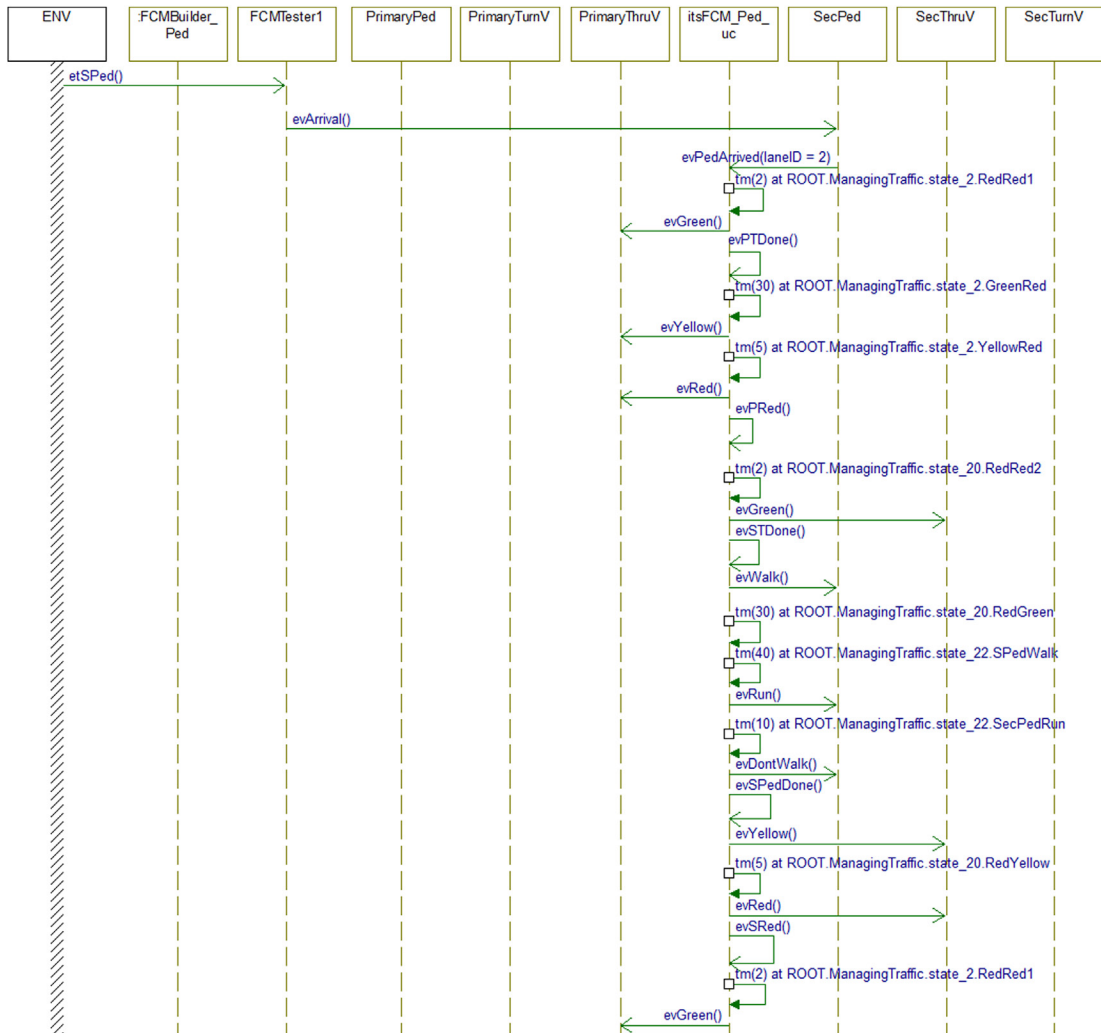


Figure 11.30
Simulation with secondary pedestrian.

interface is required by the actor. Similarly, for messages coming from the use case block to the actor, another interface, *iOpticalResults*, is required by the use case block and provided by the actor. Interfaces used in this way are called interface conjugates pairs; that is, the interfaces offered by one port are required by the other and vice versa.

Remember that the interfaces specify the services provided or required by some element. By providing a set of services in the *iOpticalServices* interface, the use case block is declaring that it promises to perform those activities. By requiring the other interface, *iOpticalResults*, it declares that it needs the actor to provide those services.

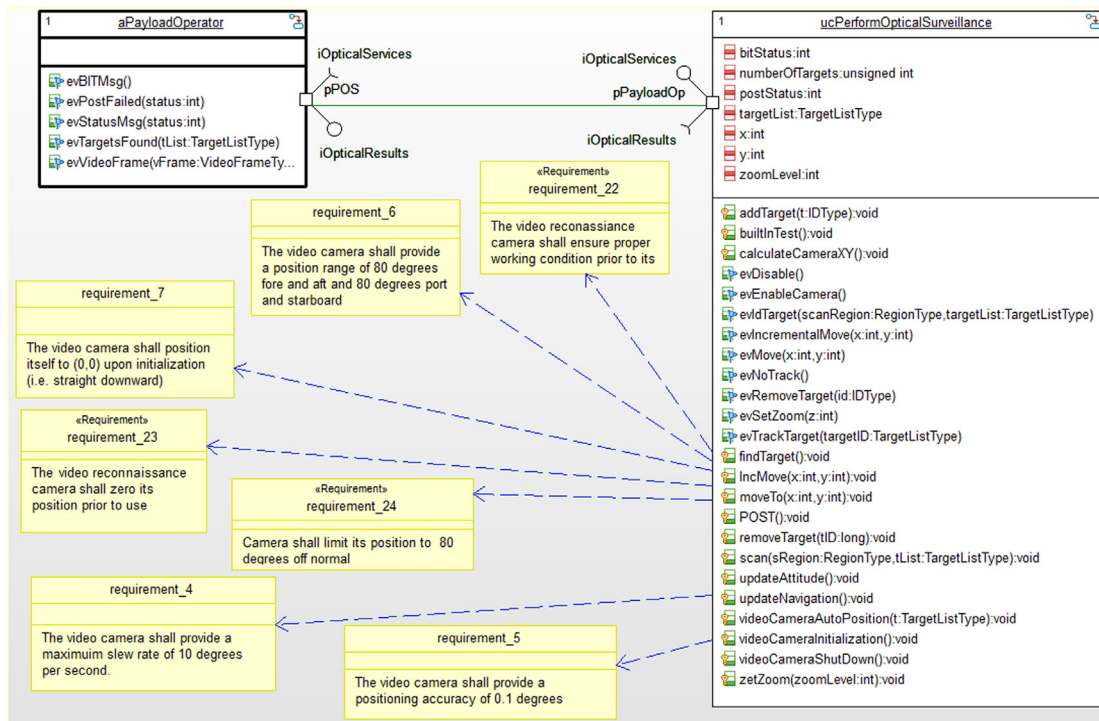


Figure 11.31
Perform Optical Surveillance use case context diagram.

The operations are shown inside the blocks with a key symbol used to indicate “protected” visibility and no symbol to indicate that they are public.

11.12.2 Step 2: Define the Interfaces

The interfaces themselves are shown in [Figure 11.32](#). Of course, they only contain the public event receptions for the use case block and the actor. Specification of the interfaces was a straightforward job of selecting the public event receptions defined on the block and entering them as part of the interface.

11.12.3 Step 3: Defining Pre- and Postconditions for the Interfaces

The next step is to iterate over each of the operations in the interface and specify the pre- and postconditions. Preconditions are things that must be true prior to the invocation of the service. Postconditions are things that are guaranteed to be true by the service provider upon completion of the service.

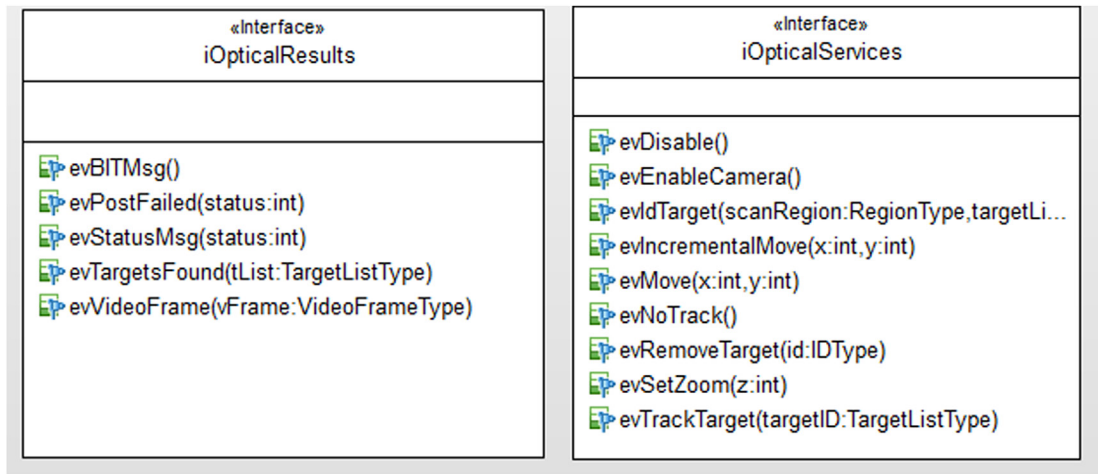


Figure 11.32
Logical use case interfaces.

The common repository for such information is in the description field provided by Rhapsody for each operation. However, it is more parsimonious for our purpose here to collect them all in a table (Table 11.1).

11.12.4 Step 4: Construct a Use-Case Activity Diagram Representing All of The Scenarios Previously Specified for the Use Case

It is useful to have an overview of all the different scenarios in one place. Since a sequence diagram represents a particular scenario, how can this overview be represented? Although in UML it is theoretically possible with the interaction fragment operators to represent all branch and concurrent points in a single sequence diagram, it is not generally practical to do so. One easy way is to use an activity diagram to sum up all the messages that can occur. If two scenarios depart at some particular branch point, this can be easily represented using the activity diagram branch operator.

For the set of scenarios defined for this use case, the derived activity diagram is shown in Figure 11.33.

11.12.5 Step 5: Define a State Machine for the Use Case Block that is Consistent with the Set of Scenarios

The last task from the problem statement was to construct a statechart for the use case block that invoked the behaviors under the right circumstances, at the right time, and in the right order. Note that the activity diagram is recommended to be a documentation-only thing and is

Table 11.1 Interface Pre- and Postconditions.

Interface	Service	Precondition	Postcondition
iOpticalResults	evBITMsg	System passed power on self-test. System has been initialized and is enabled.	System errors are identified to the payload operator. If unrecoverable errors are identified, the system is disabled.
	evPOSTFailed	System has been commanded to be enabled from the disabled state.	POST error has been identified by the payload operator.
	evStatusMsg	System has been initialized and is enabled.	System continues operating in same state.
	evTargetFound	System has been commanded to search for targets.	One or more targets matching the requested target specification have been identified.
	evVideoFrame	System is enabled.	Video frame is passed to payload operator.
iOpticalServices	evDisable	System is enabled.	Camera is returned to starting position and shut down.
	evEnableCamera	System is disabled.	Camera is powered and a power on self-test is performed. If errors occur, then errors codes are sent; otherwise, the camera is initialized and centered.
	evIdTarget	Camera is operating and search area or route has been specified.	System identifies a list of targets, each with a unique target ID.
	evIncrementalMove	Camera is operating.	If the incremental move is within the range of the gimbaled camera mounting, then the camera is moved incrementally +x units and +y units. If the commanded position is out of range, then the camera will move to the closest valid position.
	evMove	Camera is enabled.	If the incremental move is within the range of the gimbaled camera mounting, then the camera is moved incrementally +x units and +y units. If the commanded position is out of range, then the camera will move to the closest valid position
	evNoTrack	System is actively tracking targets.	If specified point is in range, the point is set to be the search center. If not, an error is returned and the center remains unchanged.
	evRemoveTarget	System is actively tracking targets.	If the target ID is in the current target list, it is removed from that list. If the target ID is not in the list, an error message is sent to the payload operator
	evSetZoom	System is enabled.	The zoom level is set as requested. If the requested zoom level is out of range, then the command is ignored.
	evTrackTarget	Camera is enabled and potential targets have been tagged with IDs.	If the target ID is known to the system, then it will track that target within the center of the visual field with a combination of flight maneuvers and camera gimbal movement, depending on the state of the CAUV.

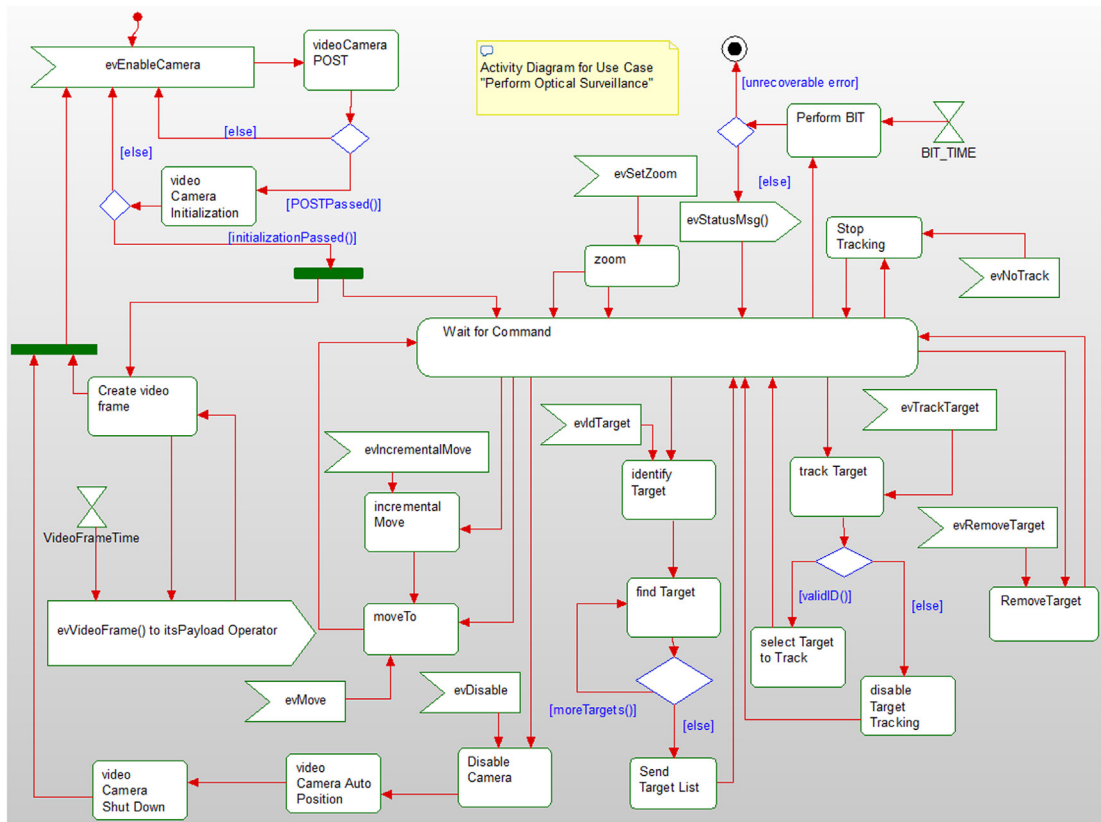


Figure 11.33
Derived use case activity diagram.

used to specify the user case per se; however, the statechart is executable and specifies the use case block.⁴ The advantage of using a statechart over an activity diagram is that the former can use events such as timeouts, arrival of commands, etc., to trigger transitions. This isn't possible in UML 2.0 activity diagrams.

A powerful aspect of using statecharts in this way (namely, to represent scenarios) is that the statechart fully specifies the allowable sequences and shows which sequences are disallowed. A single scenario cannot do that, but a statechart can. In this case, shown in Figure 11.34, we

⁴ Activity diagrams *can be* the executable specification of the use case block if desired; however it is important to only have a single executable specification of the use case. After all, if they differ, then which is right? If they define the same behavior, why pay for the extra effort to maintain them in sync? I personally find state machines far more useful for use case specifications than activity diagrams, but your mileage may vary.

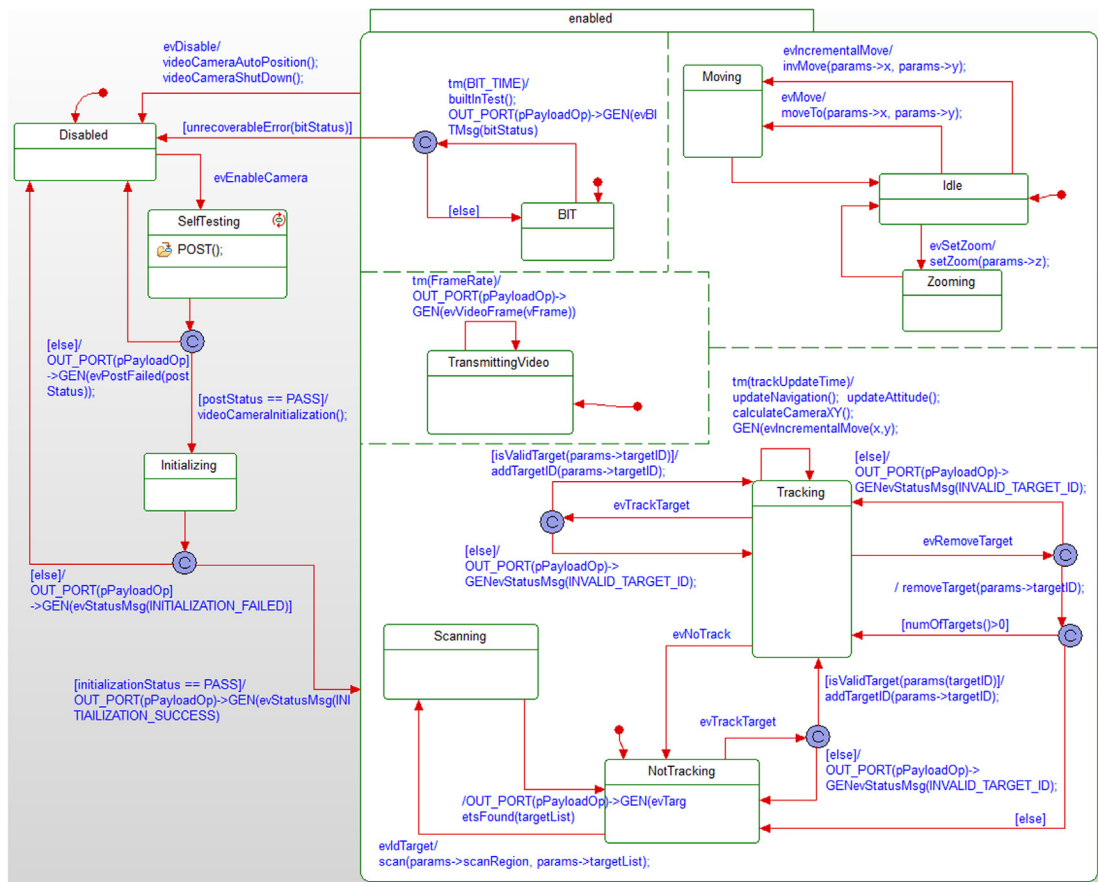


Figure 11.34
Use case block statechart.

show logical concurrency with and-states.⁵ Thus, we see that tracking is independent from, for example, moving of the gimbaled camera and managing optical parameters such as the zoom level. How target tracking takes place is also much more explicitly stated in the statechart than in the more informal sequence diagrams.

References

- Douglass, Powel, Bruce, 2004. Real-Time UML third ed. Advances in the UML for Real-Time Systems. Addison-Wesley.
- Douglass, Powel, Bruce, 1999. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley.

⁵ However, this does not imply or constrain the internal concurrency architecture. Here we are capturing things that are independent in their order of execution, but there are an infinite number of ways that this can map to a set of internal OS threads.

Deployment and Subsystems Architecture: Answers

12.1 Answer 5.1: Organizing the Systems Model

This exercise has two parts, one for each of the two systems we're working with in this book. The first answer shows the organization of the Roadrunner Traffic Light Control System. This is shown in both model browser and package diagram views in [Figure 12.1](#).

12.1.1 The Roadrunner Model

In this figure, we see two high level packages. The first – `SEModel` – holds the systems engineering information and is composed of packages as we expect. The second – `SSModel` – is focused on the subsystem design. `SSModel` has three singular packages – `InterfacesPkg`, `CommonTypesPkg`, and `Domains` – that hold information shared by the subsystems. In addition, `SSModel` has the `SS1Pkg`, which holds the basic structure that will be replicated for all subsystems, once we identify them. I didn't put the otherwise-imported `SSSpec` package in the `SS1Pkg` because it can be directly referenced from its location in the `SEModel` package now, since it is all in one model.

12.1.2 The CUAV Suite of Models

The second part of the exercise is to structure the Coyote UAV model into a set of interacting models along the guidelines set out in Chapter 5. The Coyote UAV system is two or three orders of magnitude larger in scope than the Roadrunner system and in order to be effective, it must be managed in multiple models:

- Systems engineering (SE) model
- Shared model
- (n) Subsystem models

12.1.2.1 The SE Model

The SE model will be used for system level requirements and systems architecture specification. Because of the size of this model, it has more levels of decomposition than the far simpler Roadrunner model (see [Figure 12.2](#)).

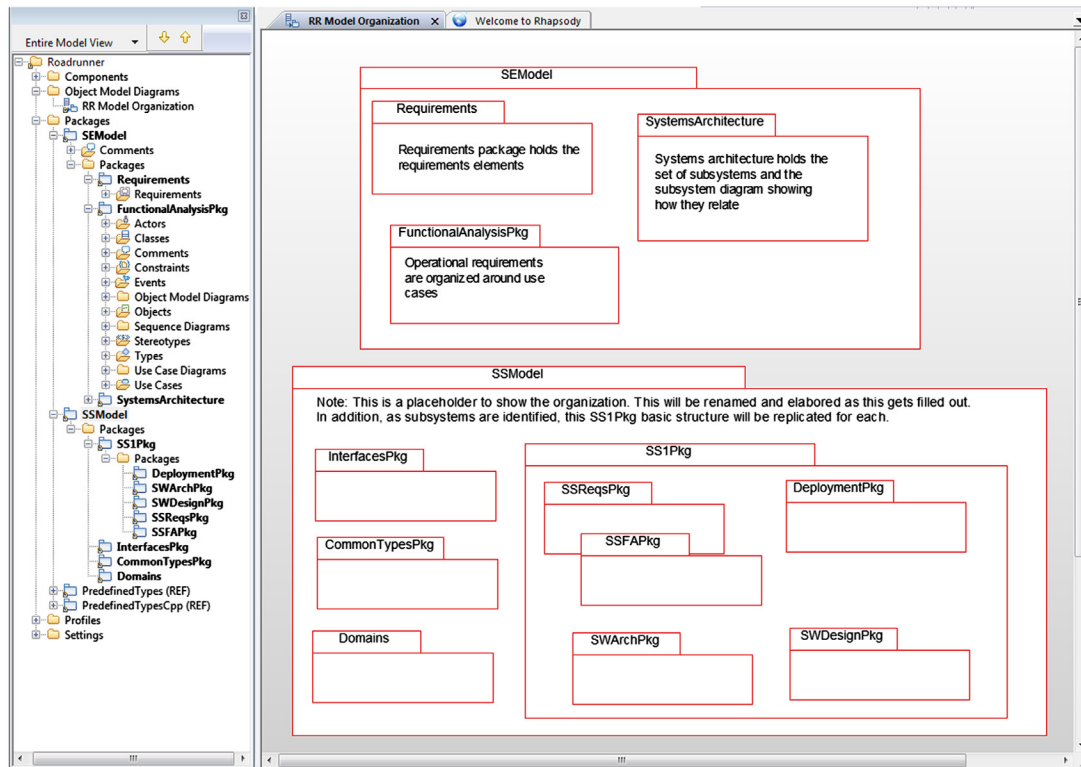


Figure 12.1

Roadrunner overall model organization.

The package holding the requirements is decomposed into behavioral (including quality of service) requirements and parametric (nonbehavioral) requirements for each of the three major pieces of the system – the UAV, the ground system Coyote Mission Planning and Control System (CMPS), and the payload. This is necessary to manage the potentially large volume of requirements in a realistic system.

The functional analysis model has a package for every identified use case in the system (although not for the subsystem use cases).¹ Some of these may be unused because some of the use cases may turn out to be *abstract*, serving as organizational and decomposition units and not holding any real requirements themselves. For example, the Perform Surveillance use case has three sub-use cases – Perform Optical Surveillance, Perform FLIR Surveillance, and Perform Radar Surveillance. If there are requirements common to all of these specific use cases, then they would be placed in the base use case (Perform Surveillance), but it could be that all the requirements occur at the leaves of the generalization tree. If that is true, then we won't need to perform analysis of this abstract use case.

¹ If necessary, the subsystem packages will be elaborated in the same style as the SE model with packages for subsystem use cases, but we're not ready to do that yet.

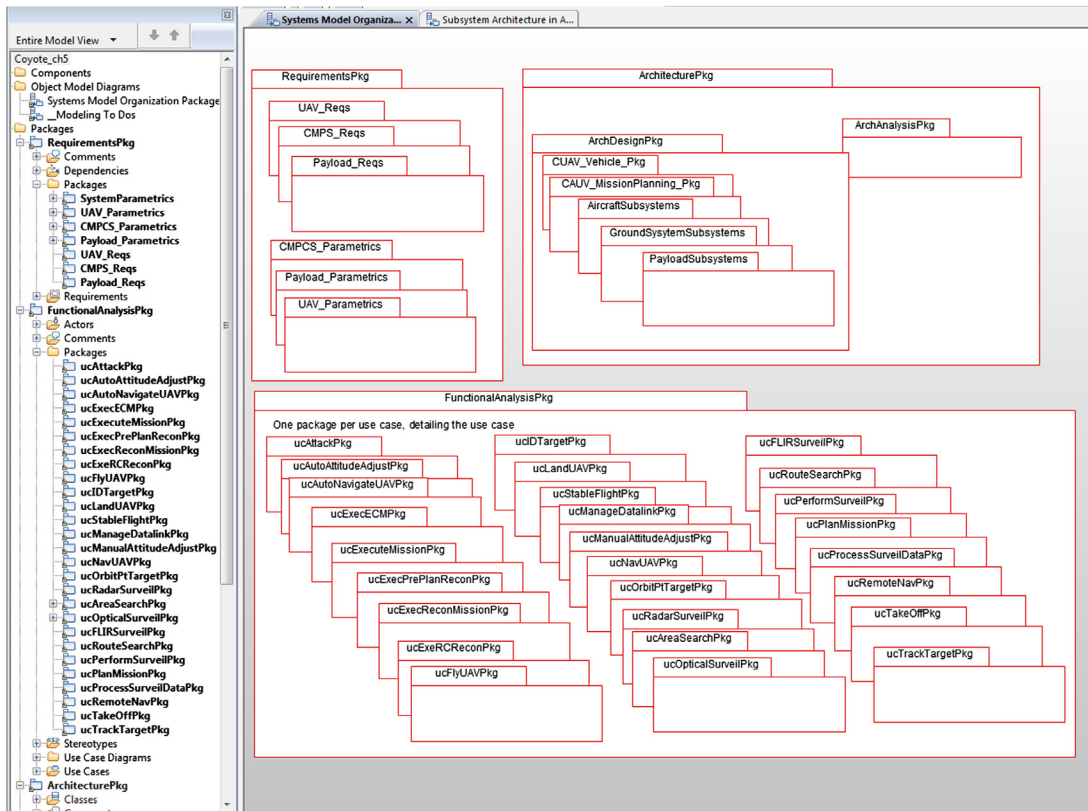


Figure 12.2
CUAV systems model package diagram view.

Similarly, the architecture design package is broken into three packages: `VehicleArchPkg`, `CMPSArchPkg`, and `PayloadArchPkg`. Each of these will be decomposed further as we identify their internal structure. Note that if the model becomes sufficiently complex, we can take these packages and make them separate models and reference them inside the “master” SE model.

The reason that the architecture design model may become very complex is that each one of these systems – the aircraft, the ground station, and the payloads – is a complex system by itself and will have a number of internal subsystems that are themselves complex. To manage the analysis and design of such a complex system is no easy task and we must be able to grow the richness of our modeling structure. Once we’re down to an individual subsystem such as the Navigation subsystem for the aircraft, for example, a package will be created to hold its “software and hardware specification” models, including its subsystem requirements, use cases, and their functional analysis. This will form the specification of that subsystem, which will be then handed off to that subsystem team. Those packages are then handed off to the subsystem team for them to design and develop those subsystems.

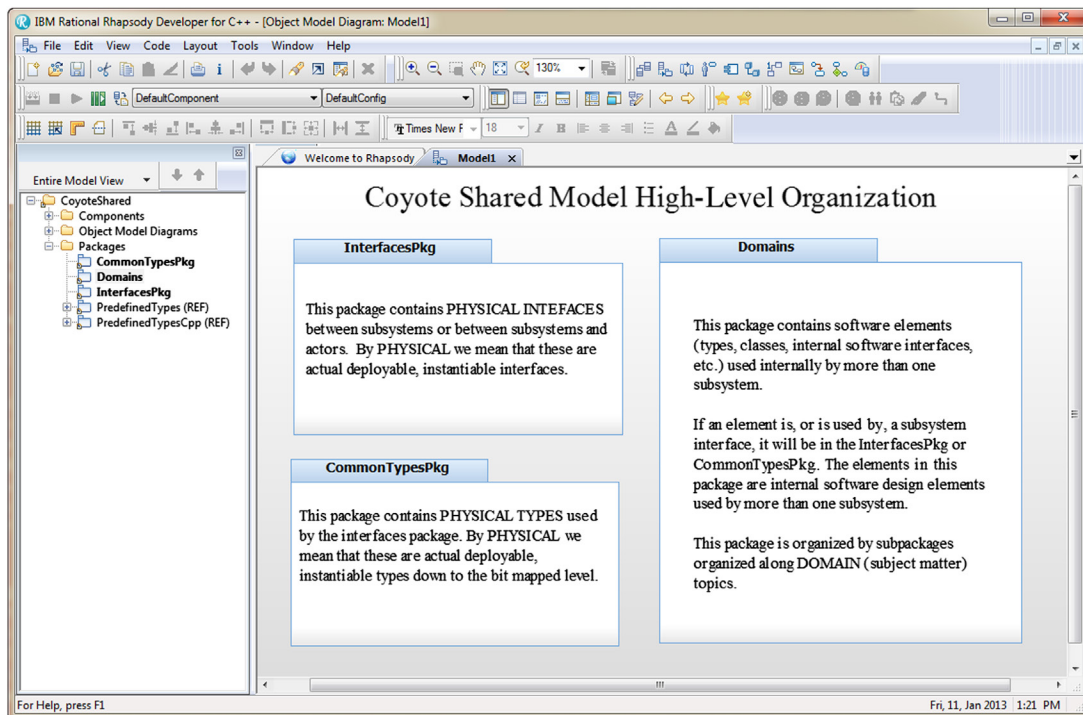


Figure 12.3
CUAV common model.

12.1.2.2 The Shared Model

The common model will hold the elements to be shared among the subsystem teams, such as the subsystem interfaces, the data types they use, and a place for the design elements used by more than one subsystem. The basic organization of the shared model is shown in [Figure 12.3](#).

12.1.2.3 The Subsystem Models

Lastly, for each subsystem team, there is a separate subsystem model that can be thought of as a systems-model-in-the-small. That is, the subsystem model is organized along the same lines and using the same principles as the system model. It has a package for subsystem requirements and a package for the internals of the subsystem, specified primarily as collaborations among the objects held within the subsystem. The model browser view of the subsystem model structure is shown in [Figure 12.4](#).

12.2 Answer 5.2: Subsystem Identification

The Roadrunner Traffic Light Control System is a fairly small system, so the system architecture diagram is straightforward. I used a UML structure diagram to represent the system architecture. Note the use of stereotypes «System» and «Subsystem» to mark the

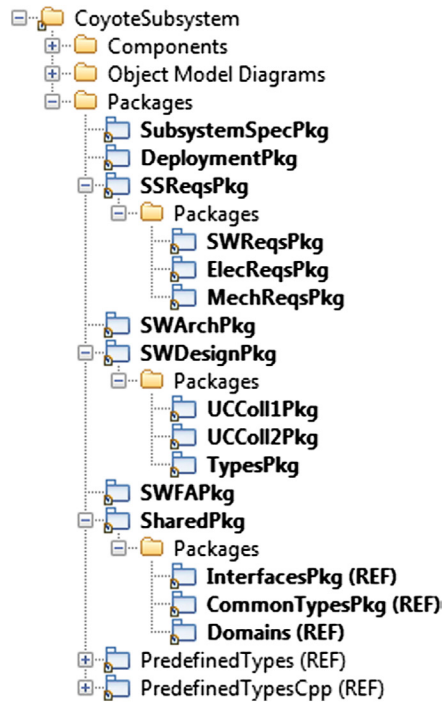


Figure 12.4
CUAV subsystem model.

architectural elements. This is completely optional but I find it useful to clarify the usage of elements in the model. Note that none of the elements shown in [Figure 12.5](#) are “primitive” – they are all composites containing a possibly rich set of internal parts.

The ports between the Roadrunner subsystems are not (yet) typed by interfaces, but the links among the elements are straightforward. One can certainly imagine these connections are required for system operation and with a little thought one could definitely define a set of services provided and required across these ports. The definition of these interfaces is dealt with in an exercise later in this chapter.

Notice the set of subsystems for the traffic light control system in [Figure 12.5](#). There is a different subsystem for the primary vehicle light assembly (`P_VehicleLightAssembly`) than for the secondary road (`S_VehicleLightAssembly`). The two subsystems are, at this point, expected to be identical in their structure and behavior but play different roles; that is, both of these instances are typed by the same class: `VehicleLightAssembly`. The architecture in this case is shown as objects.

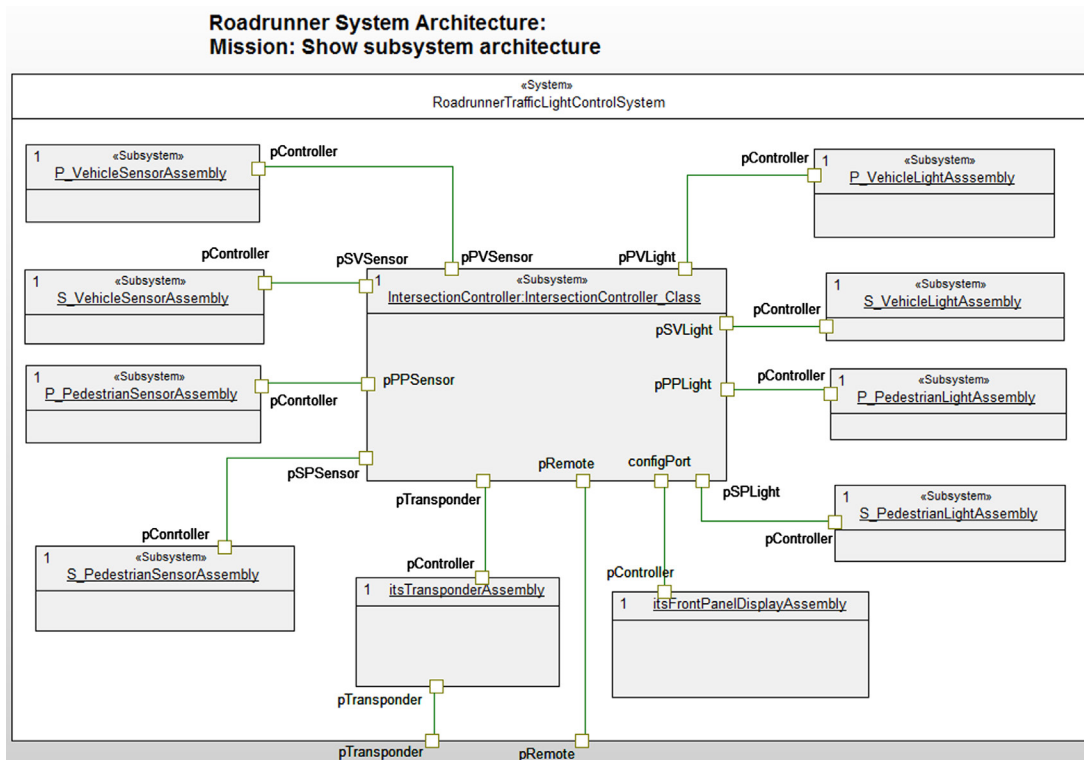


Figure 12.5
Roadrunner system architecture diagram.

The same architecture could be shown with a class diagram instead. However, the explicit connections cannot be shown in ports in class diagrams (the end points must be objects not classes). It is not unreasonable to avoid the use of ports and use class diagrams instead, but it is a different style. Although we won't develop this view further, [Figure 12.6](#) shows what it would look like.

The Coyote Unmanned Aerial Vehicle (CUAV) is about two orders of magnitude larger in scope than the traffic light control system. For this reason, the architecture is depicted on a set of diagrams rather than a single diagram. The first of these, [Figure 12.7](#), shows the interconnected systems comprising the CUAV conglomerate. We can see that there are three kinds of systems identified – the CMPS ground systems (for mission planning and operations control), the Coyote (the air vehicle itself), and the various payloads. There are four different payloads shown – missiles, forward-looking infrared (FLIR), optical video, and synthetic aperture radar (SAR). Each of these is stereotyped «System» and they will be decomposed into separate subsystem diagrams. While it is possible to put all this information in a single diagram, it makes that diagram far more difficult to comprehend.

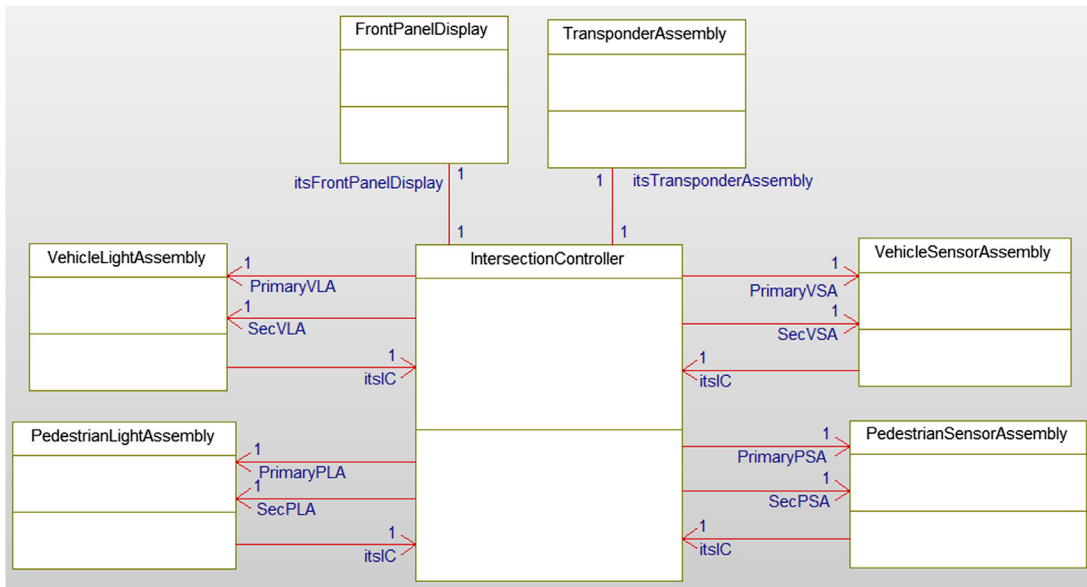


Figure 12.6
Roadrunner architecture as a class diagram.

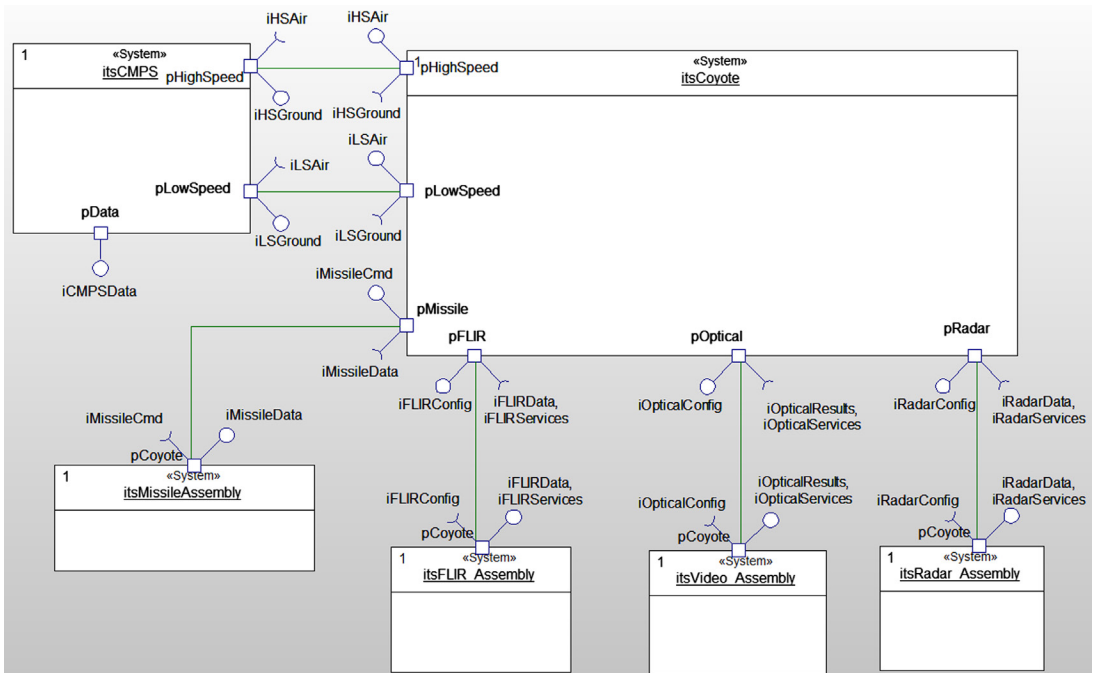


Figure 12.7
CUAV system architecture diagram.

Figure 12.7 is an object diagram, allowing the links to connect the systems at their ports. In this case, the first cut interfaces are shown as well, although not the details of those interfaces. Between the two major systems – the CMPS and the Coyote – there are two ports: a high-speed link for video, radar, and FLIR data, and a low-speed connection for commands and low-frequency data. This latter link is expected to be encrypted. The CMPS also has a port to which unspecified devices can connect to get logs, reports, videos, and other configuration and operational data.

The ports between the Coyote and its (potential) payloads are organized so that the Coyote provides services to configure and command the payload while requiring services that provide data and results from other services. For the sensor payloads, two different interfaces are required from the payload – data services and other services. Also note that all the connected port pairs are *port conjugates*; that is, what one offers, its pair requires and vice versa. This arrangement is typical.

The systems defined within the CUAV project must be be decomposed into their primary subsystems as well. This is done in the following set of diagrams. Figure 12.8 shows the subsystem architecture of the ground station. Note that the Ground Datalink subsystem and the Data Storage subsystem are singletons (i.e., have a single instance) within the ground station, but there are up to four manned stations, and each of these has two stations within them – one for

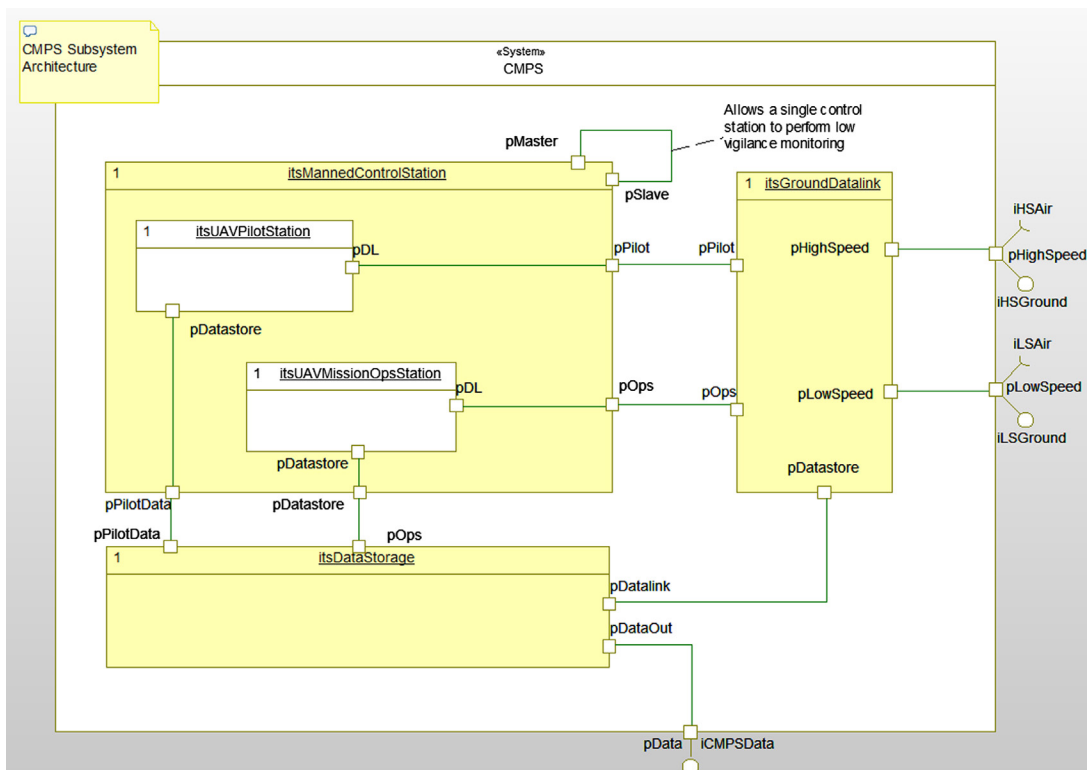


Figure 12.8
CUAV ground station subsystem architecture.

controlling a UAV and one for mission operations, such as receiving and processing the reconnaissance data or targeting and launching missiles. The association among the manned stations allows them to be slaved together into a single station for low-vigilance monitoring.

The guidelines used to link together the subsystems are simple. If one subsystem requires services or data from another, a link is emplaced between them. All links (at the subsystem level) employ ports as a means to help ensure strong encapsulation. Although interfaces on the ports aren't shown for the internal CPMS links in [Figure 12.8](#), they must exist to specify the services and data passed among the subsystems.

[Figure 12.9](#) shows the vehicle itself. It is organized into a set of subsystems as well. The subsystems are summarized in [Table 12.1](#).

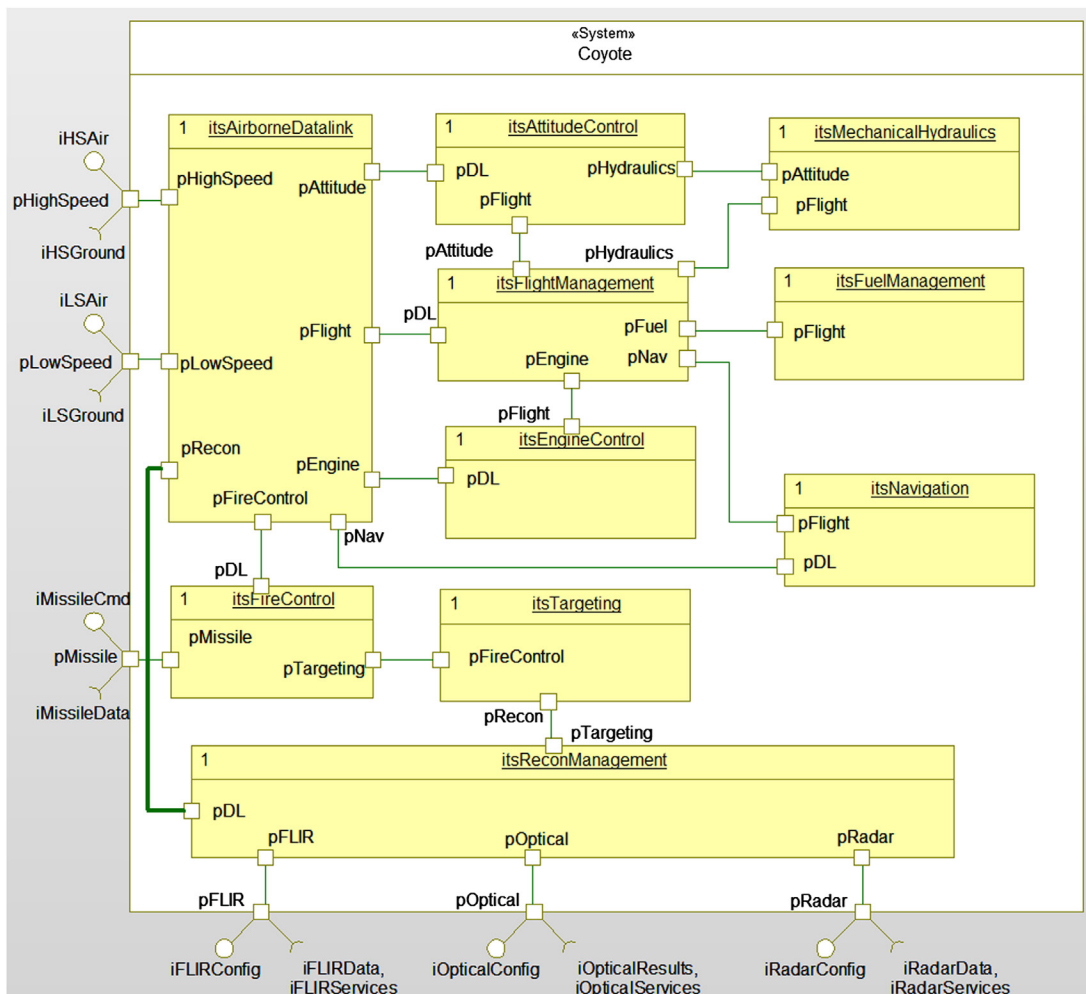


Figure 12.9
CUAV vehicle subsystem architecture.

Table 12.1 CUAV Subsystem Summary.

Vehicle Subsystem	Links With	Purpose
Airborne Datalink	High-speed external port Low-speed external port Flight Management Fire Control Attitude Control Engine Control Navigation Reconnaissance	Links vehicle to ground station for both low-speed (command) data and status and high-speed data; also unmarshalls messages received over the communications links and delivers them to the appropriate subsystem(s)
Attitude Control	Airborne Datalink Mechanical Hydraulics Flight Management	Controls and monitors the attitude (roll, pitch, yaw) of the vehicle by commanding position changes in the vehicle's control surfaces
Engine Control	Airborne Datalink Flight Management	Controls engine output and monitors engine status
Fire Control	Airborne Datalink Targeting Missile external port	Manages on-board weapons systems, both status and control, including the delivery of fly-by-wire Hellfire missiles
Flight Management	Airborne Datalink Attitude Control Engine Control Navigation Hydraulic Control Fuel Management	Manages general flight, which can be fully or partially automated or may be managed completely from the ground
Fuel Management	Flight Management	Monitors fuel status and shunts fuel as necessary for weight balance
Mechanical Hydraulics	Attitude Control Flight Management	Provides direct control over air control surfaces – wings, rudders, elevators, and ailerons; also used to control landing gear.
Navigation	Airborne Datalink Flight Management	Used for control and monitoring of position in the airspace; also contains navigation data and tools, such as digital maps and flight plans.
Reconnaissance Management	Airborne Datalink Targeting FLIR external port Optical external port Radar external port	Gathers and processes reconnaissance data for both on-board target tracking and for remote surveillance (transmitted to the ground station)
Targeting	Fire Control Reconnaissance Management	Identifies and tracks targets of interest

The payloads are not broken down into their subsystems here. One reason is that it is likely that a single team will be developing each of the payloads, so it is less crucial to do that decomposition early. Secondly, the payloads are clearly less complex than either the ground station or the vehicle.

One thing to note about the system-to-subsystem decomposition performed in this answer – the subsystems themselves are multidisciplinary. That is, each subsystem contains software,

electronic, mechanical, and possibly chemical engineering aspects. That decomposition is discussed in a later problem in this chapter.

12.3 Answer 5.3: Mapping Operational Contracts into the Subsystem Architecture

In the process of developing systems, the tradition is to first create a set of requirements – in the MDA/UML/SysML world, these requirements are organized into and around use cases. Next, we define a system architecture. This has been addressed for both sample problems in previous problems in this chapter. The next step is to map the requirements into the various subsystems. This consists of two primary steps:

1. We must map the operational contracts (services) into the internal architecture to define the subsystem interfaces.
2. We must group these services at the subsystem level into use cases.

This problem deals with the first of these two steps. Theoretically, it is a simple process.²

A use case is realized (i.e., “implemented”) by a set of collaborating system elements. At the architectural level these are systems, subsystems, and components. Each service at the outermost system level (“system” level for the Roadrunner Traffic Light Control System, and “system of systems” level for the Coyote UAV) is decomposed into services provided by the various subsystems. In this step of the process, we decompose the system level services into subsystem-level services. We represent these service invocations primarily as sequence diagram messages.

Mechanically, the decomposition of the system services can be done “in place” or by decomposing the use case lifeline. By “in place,” we mean that the original sequence diagram(s) are copied and the subsystem lifelines are added. This is only useful for small systems. For larger scale subsystems, or systems with more parts at the next level of abstraction down, it is better to use the lifeline on the original sequence diagram as a reference to a more detailed sequence diagram; this latter sequence diagram represents the interaction of the lifeline’s internal parts in the very same scenario. In Rhapsody, creating the sequence diagram representing the decomposed lifeline is easy – simply double-click the lifeline to get the “Features” dialog, and then select <new> in the “Decomposed” list control (or select the name of an already existing sequence diagram). This creates (or selects) the sequence diagram to be used as decomposition of the lifeline. To navigate to the referenced sequence diagram is easy – simply select the lifeline in question, right-click, and click on “Open reference sequence diagram.” Other tools will likely have different mechanisms for creating and traversing the linked diagrams.

When you decompose a sequence diagram lifeline, the first question that arises is, “How do I represent messages entering and leaving the sequence diagram?” The most common way is to use the system border, a special lifeline that represents all instances other than the ones

² Although remember the Law of Douglass that states “The difference between Theory and Practice is greater in Practice than it is in Theory!”

explicitly shown. All messages entering or leaving the nested sequence diagram go to or from that system border lifeline. If you visualize the nested sequence diagram as being inside the boundaries of the lifeline on the original sequence diagram, this is the same as saying the incoming messages are coming from the “edge of the owner lifeline” and outgoing messages are sent to that same edge.

An alternative to using the system border is to copy the actors from the original sequence diagram. If all the elements sending or receiving events are actors, it is perfectly fine to copy them into the nested sequence diagram if desired. This is equivalent to the previous solution.

While other tools may work slightly differently, Rhapsody has an “analysis mode” for sequence diagram editing – this is useful for white-boarding scenarios since nothing is populated into the repository in this mode. When the scenario looks stable, the sequence diagram can be turned into a “design mode” diagram that commits the messages to the repository. In design mode, all the messages being received by a lifeline (representing a classifier) become operations (for synchronous messages) or events (for asynchronous messages) on that classifier.

First, let us work on the traffic light control system. The smarts in this system are centered largely in the Intersection Controller subsystem (see [Figure 12.5](#)). The first step is to take the sequence diagram for Scenario 2 and decompose the Responsive Cycle Mode use case. That creates an empty sequence diagram that is referenced from the original lifeline. Then we do the following:

- Open the new sequence diagram and drag the elements (parts of the Roadrunner Traffic Light Control System object) onto the diagram to create the lifelines.
- Add a system border lifeline for messages to enter and leave the sequence diagram.
- Now, working from the original, for every message coming from an actor to the use case lifeline, create a matching message from the system border lifeline on the nested diagram to the appropriate lifeline.
- Similarly, for every message from the use case lifeline to an actor, create a corresponding message from some element in the nested sequence diagram to the system border.
- Add internal messages among the elements as appropriate to glue the collaboration of these elements together. A nested sequence diagram for Scenario 2 ([Figure 5.6](#)) is shown in [Figure 12.10](#). Similarly, the more detailed view for Scenario 3 ([Figure 5.7](#)) is shown in [Figure 12.11](#).

So what does this mean to the subsystems? Once the messages are realized by the subsystems, corresponding services (operations and events) are added to the subsystems. In this particular case, we made the subsystems instances of classes, and those realized messages become operations or event receptions defined on those classes. Once that is done, they can be added to the appropriate ports by collecting those services up into interfaces that specify the contracts of the ports. In [Figure 12.12](#), the interfaces on the ports are not shown to minimize diagram clutter.

The traffic light controller is a relatively simple system, but the example illustrates the process of elaborating the services within the subsystems. In this case, the interfaces between the subsystems are not complex but this clearly allocates responsibilities to the architectural parts of the system. In Figure 12.12, only the services identified so far (event receptions) are shown. Other services will show up as we round out the analysis of other scenarios and other use cases.

Let us now repeat the same process on the CUAV system. Of course, this is a *much* larger system so we will need to use both forms of sequence diagram decomposition to manage the diagrammatic information.

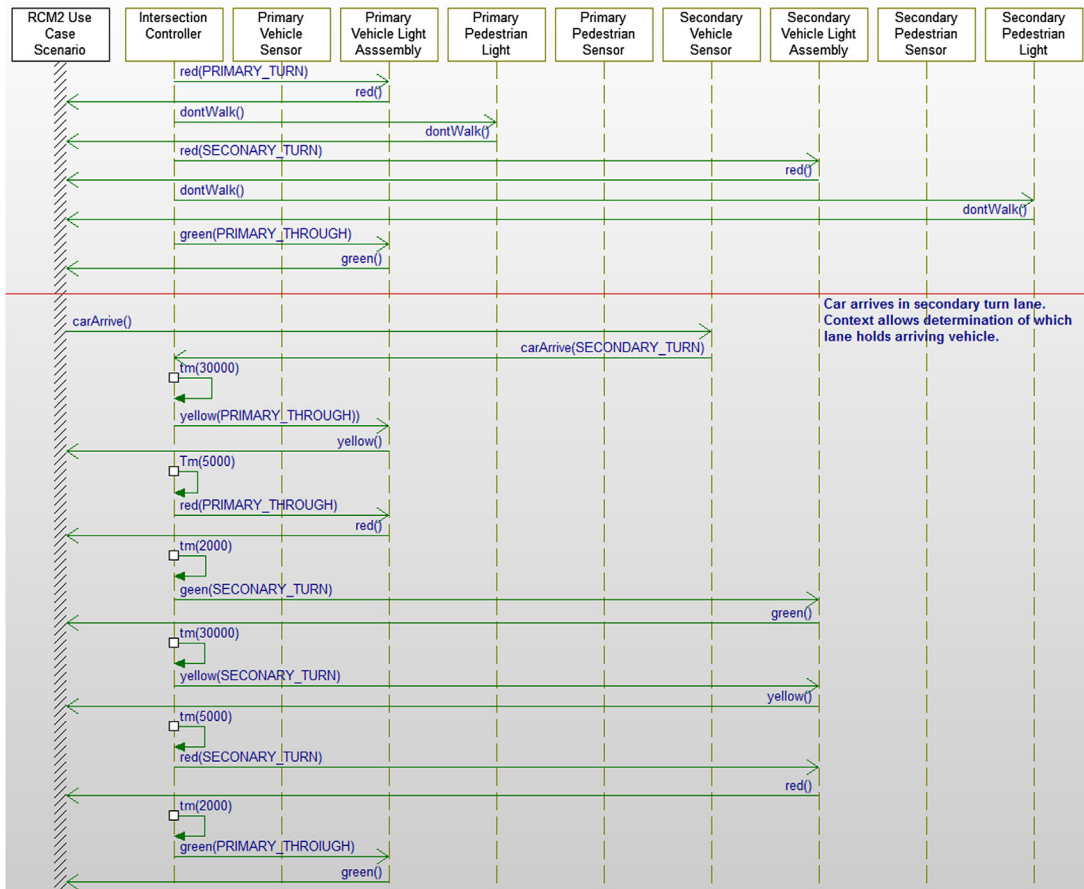


Figure 12.10
Roadrunner Scenario 2 subsystem details.

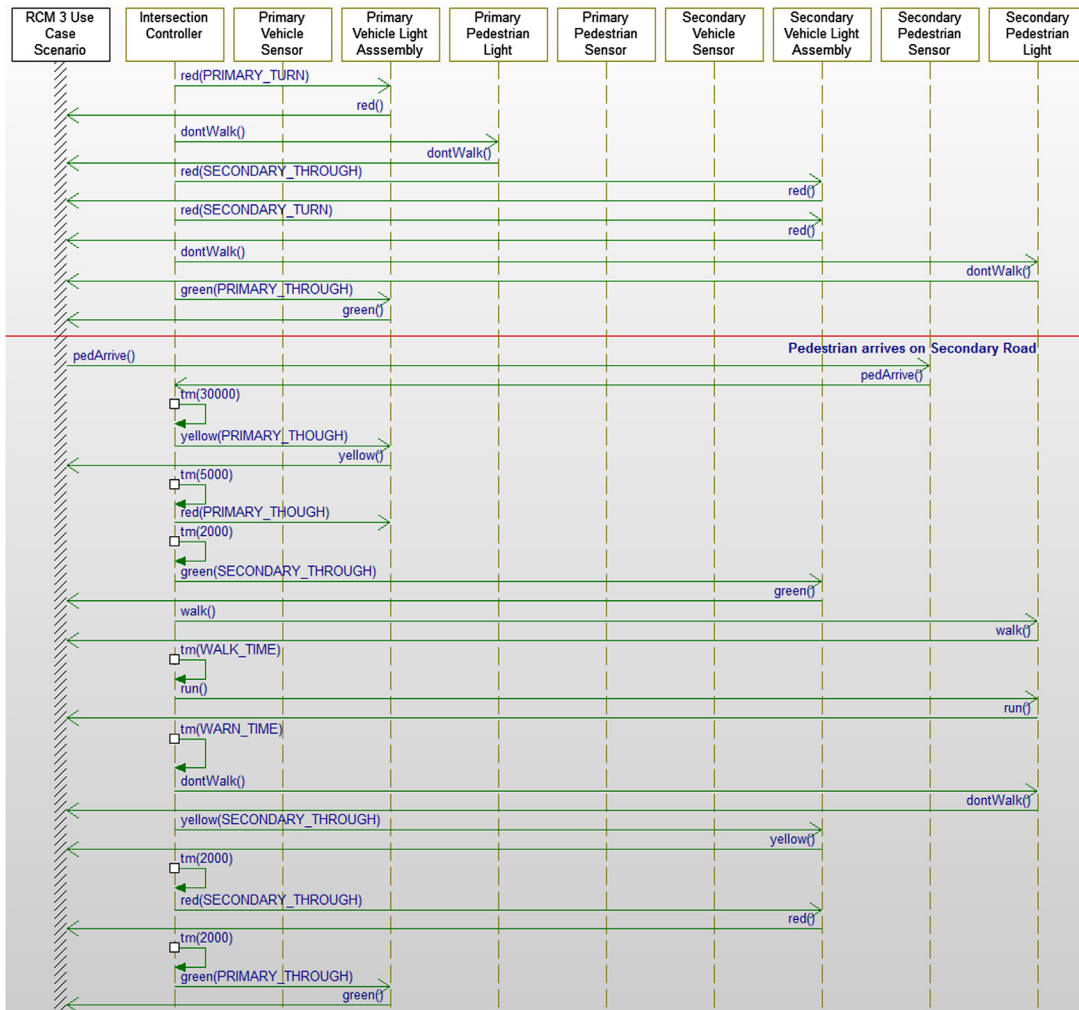


Figure 12.11
Roadrunner Scenario 3 subsystem details.

We will start with looking at the interaction between the ground station (and its Ground Datalink), the aircraft (and its Airborne Datalink), and one of the payloads – in this case the video assembly. As stated in the original problem, we’ll focus on the Perform Area Search use case and, more specifically, on Scenario 1 shown in Figure 5.11. Since that figure just references more elaborated sequence diagrams, we’ll start with Figure 5.12, which details how the aircraft takes off. The resulting figure, which is split across [Figures 12.13 and 12.14](#), shows the actors of the system with the high-level systems and subsystems elaborating the Perform

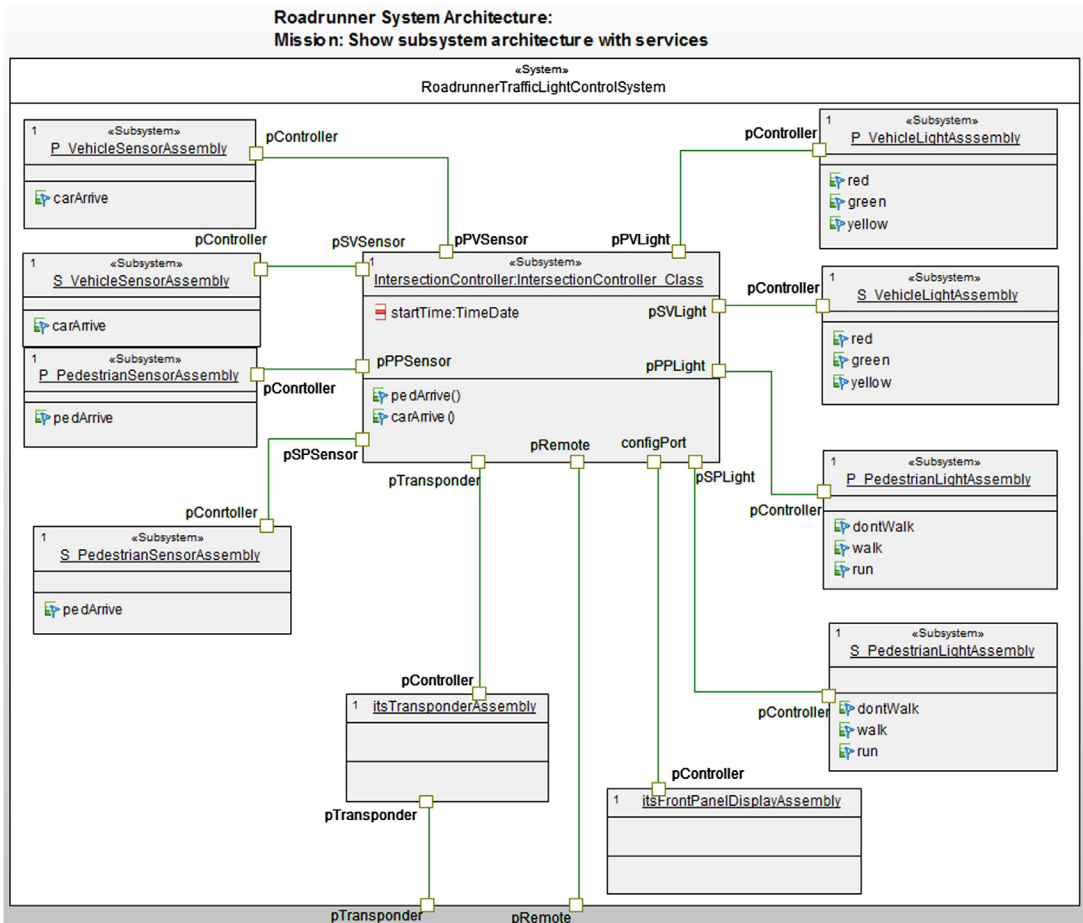
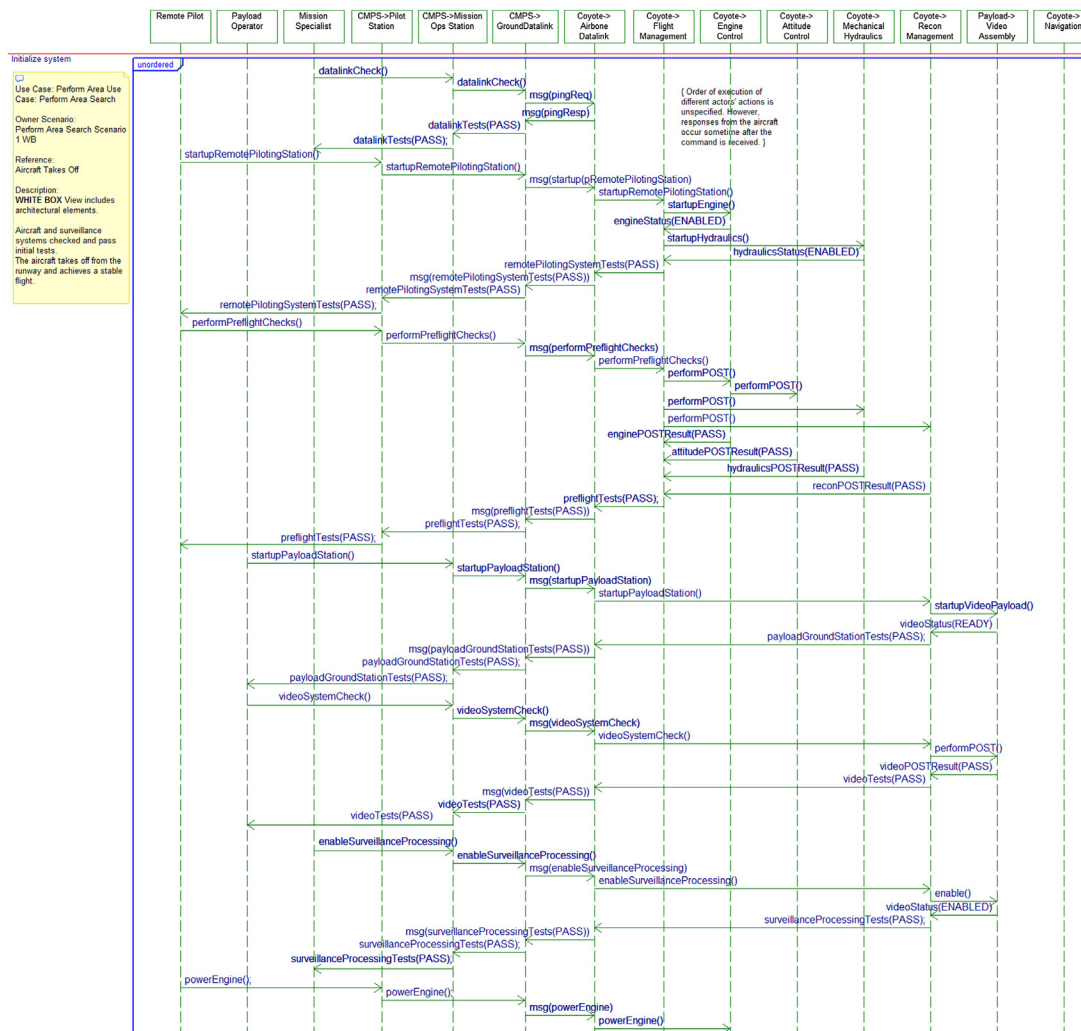


Figure 12.12
Roadrunner architecture with operations.

Area Search use case lifeline in Figure 5.12.³ You can see that elaborating these scenarios quickly makes them difficult to visualize on a single printed page; at this level of complexity, it behooves you to use the decomposition facilities of UML to scope the diagram contents to maintain their readability.

Unlike the previous example for the traffic control light, notice that the messages coming in from the actors are not summarized by the system border (named ENV in the figure) that

³ Where to put this and other detailed allocation diagrams is an interesting question for a large-scale model such as the CUAV model. I recommend you create a package in Functional Analysis -> Perform Area Search for such architectural “white box” work. In this case, I’ll name this nested package ucPAS_WBPkg, where the WB is short for “White Box.”



connects the messages to the actors in the owning sequence diagram – although that is a viable option. Here we used the original actors, just to show a possible alternative.

Realizing these messages populates events representing service requests and responses into the corresponding subsystem elements as shown in [Figures 12.15 through 12.17](#). Note that this service allocation is from realizing the messages from a single sequence diagram. Adding more sequence diagrams from the same use case and other use cases will flesh out the services and responsibilities allocated to the subsystems.

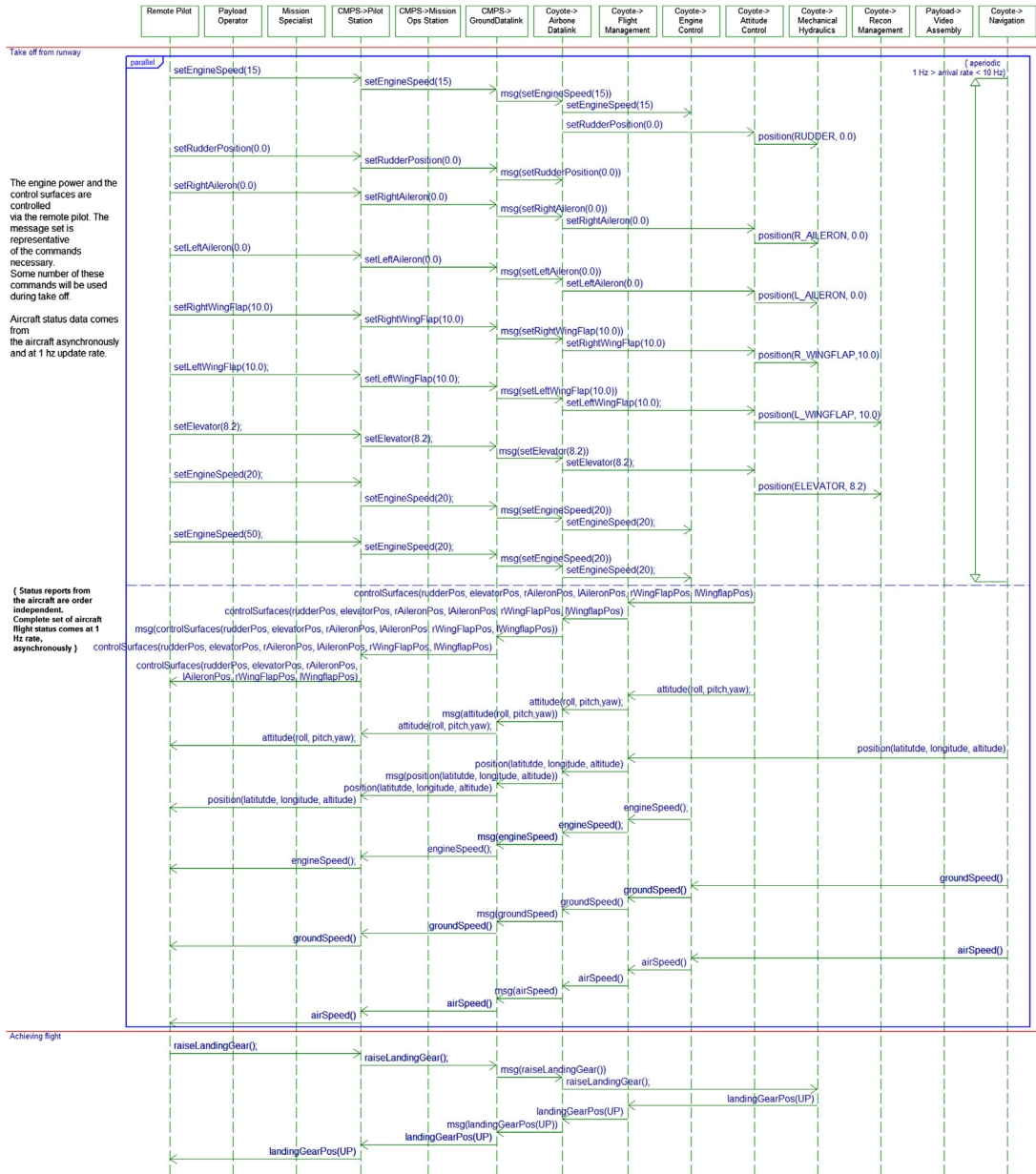


Figure 12.14
Perform Area Search Scenario 1 white box view (part 2).

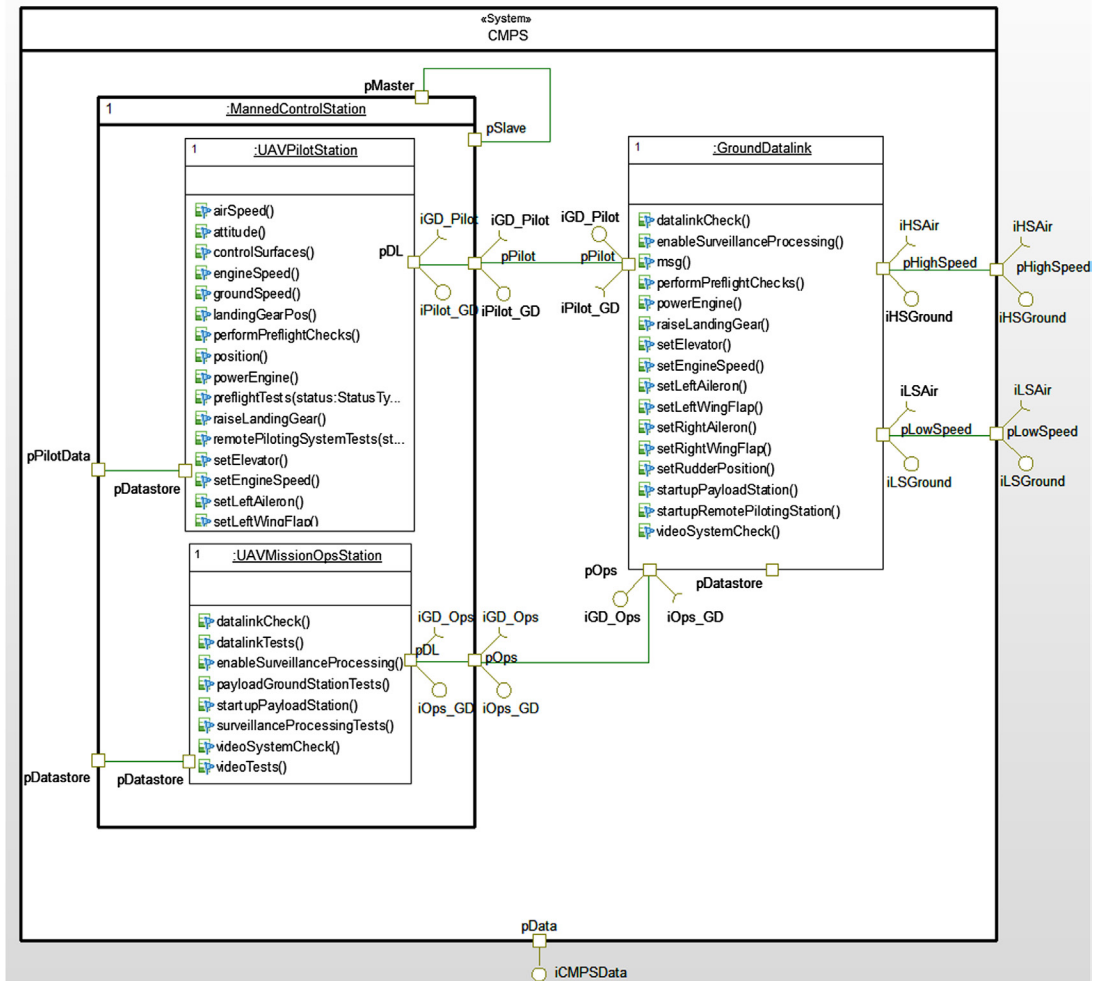


Figure 12.15
CUAV system allocated operations.

Ultimately, these services will be clustered into interfaces that define the messages that traverse the ports between the subsystems, providing the detailed content for the Interface Control Document (ICD).

This allows us to create views similar to [Figure 12.18](#), which I find to be of tremendous value. This diagram is not complete – it is only elaborated to the extent that the use cases and their scenarios have been elaborated so far – but it shows the subsystem (Flight Management in this case) surrounded by the interfaces it supports. Each interface is depicted showing the set of services defined within it.

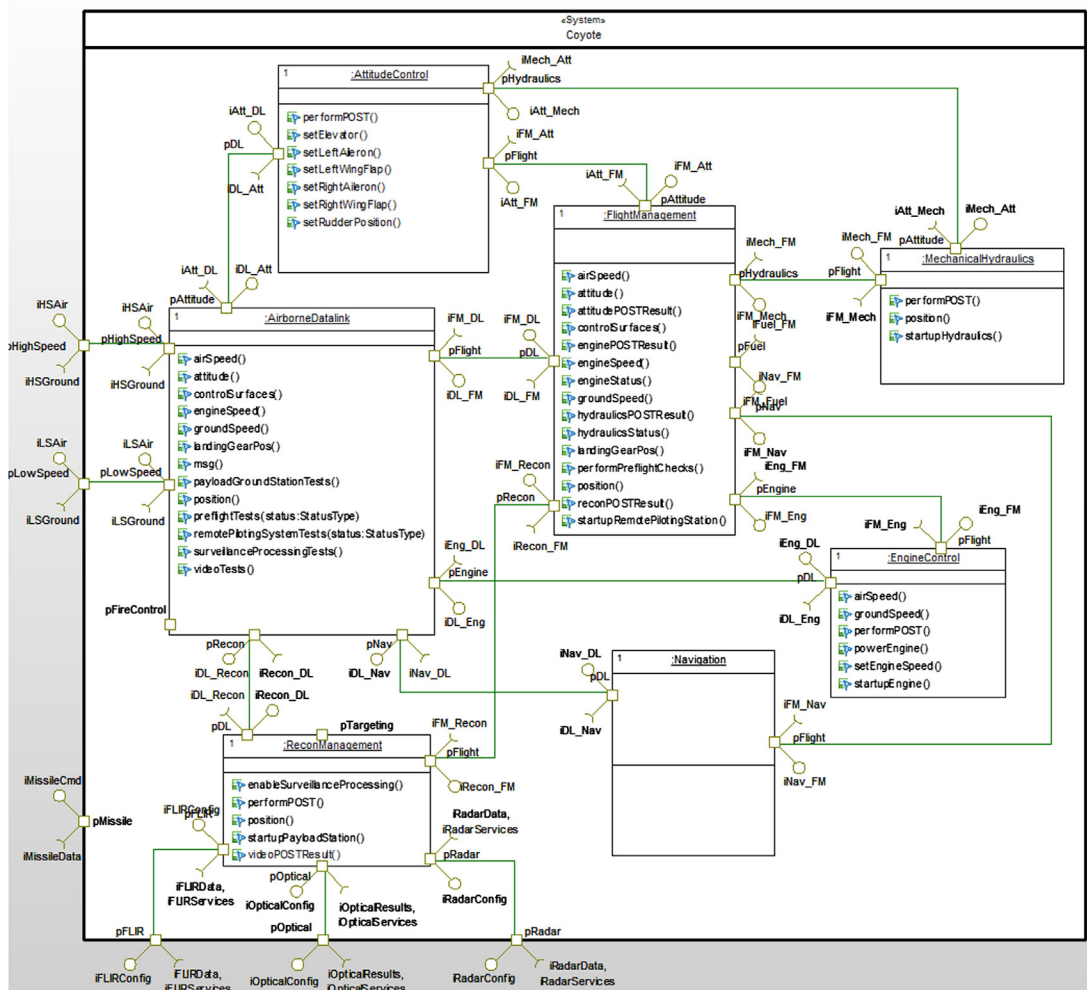


Figure 12.16
Coyote vehicle service allocations.

12.4 Answer 5.4: Identifying Subsystem Use Cases

The first part of this exercise identifies the use cases for the Roadrunner Traffic Light system, however we limit ourselves to just two system-level use cases – Detect Vehicle and Configure System. In a real project, we would, of course, decompose *all* of the system-level use cases and map them onto all of the subsystems. But, for this exercise, we limited ourselves to just two system-level use cases.

The first of these, Detect Vehicle, has two specialized use cases – Above Surface Vehicle Detection and Subsurface Vehicle Detection. Given the subsystem architecture of the traffic

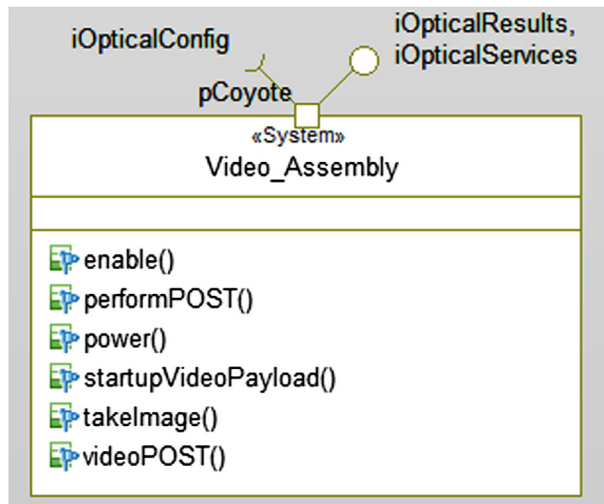


Figure 12.17
Video assembly service allocations.

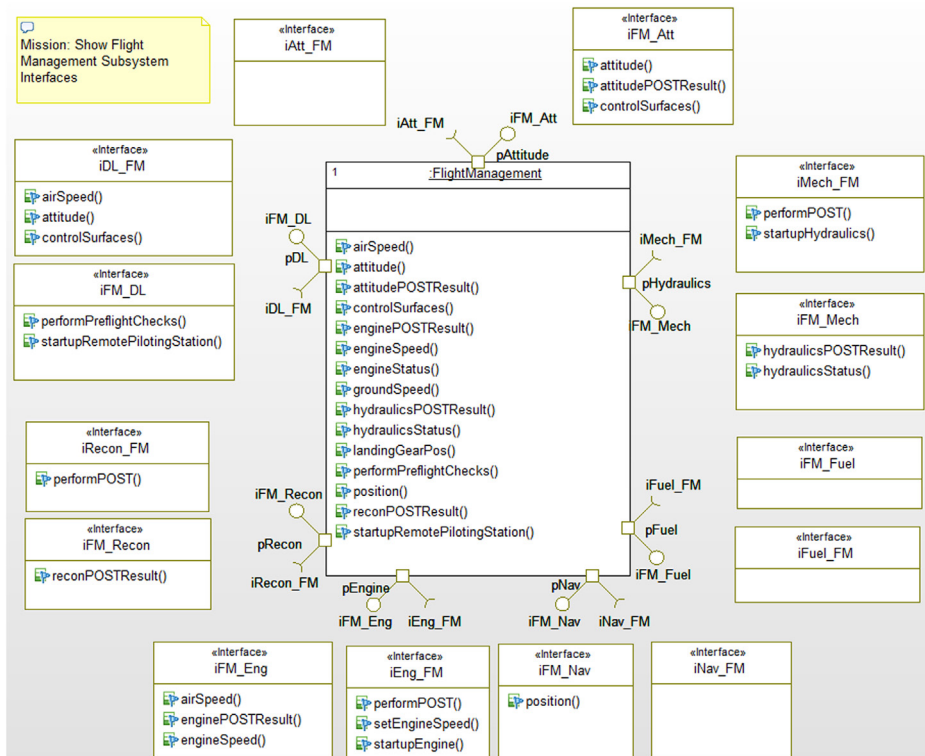


Figure 12.18
Flight Management interfaces diagram.

light controller, it is clear that most of the action for these use cases will take place in two subsystems – the Vehicle Sensor Assembly and the Intersection Controller. Some of the requirements apply to both the specialized subsystems. Specifically, regardless of how the vehicle is detected, vehicle count and related statistics must be kept and managed. Further, all the detectors must be able to communicate over a wired bus. Therefore, in [Figure 12.19](#), the Detect Vehicle system use case is decomposed into two subsystem level use cases – Manage Vehicular Traffic Statistics (mapped to the Intersection Controller subsystem) and Communicate via Wire (mapped to both the Intersection Controller and the Vehicle Sensor Assembly subsystems). The decomposition of the use cases is represented with the «include» dependency. The mapping is indicated in the figure with constraints; in addition, the use cases reside in the relevant subsystem areas of the model. Use cases shared by multiple subsystems can be stored in the system area or in the Common package, or replicated in the targeted subsystems, as desired.

The specialized use cases are likewise decomposed into subsystem-level use cases with the «include» dependency.

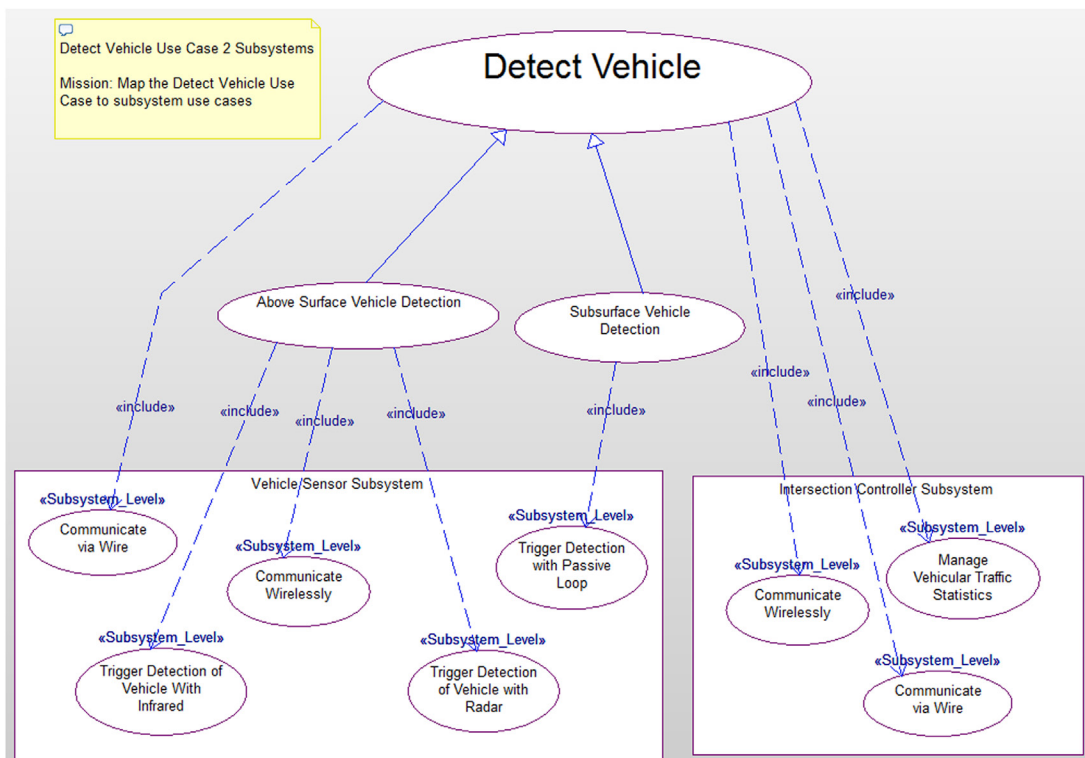


Figure 12.19
Roadrunner Detect Vehicle Use Case maps to subsystems.

The Configure System use case is similarly decomposed to a set of smaller use cases, each of which maps to a specific subsystem. The only exception is the Configure Wireless use case, which is at the system level, but is decomposed into three part use cases that are assigned to subsystems.

A couple of things to note about Figure 12.20. First, just so show a diagrammatic alternative, the mapping to subsystems is shown on the diagram by anchoring them to notes rather than using system boundary boxes. Secondly, the Configure Operational Modes use case is assigned to the Intersection Controller subsystem. This implies that all the more specialized forms of this use case, such as Configure Emergency Mode, are mapped to that subsystem as well. Finally, the more specialized forms for configuring the wireless networks are mapped to the Transponder Assembly, the Vehicle Sensor Assembly, and the Intersection Controller. Remember that the transponder receives signals from priority and emergency vehicles while the vehicle sensor assembly may have either wired or wireless connection to the Intersection Controller. Therefore, each of the systems must be able to configure its wireless communications.

The latter part of the exercise takes the CUAV system and maps the Manual Attitude Adjust and Perform Area Search use cases to the subsystems. We show the mapping of the Manual

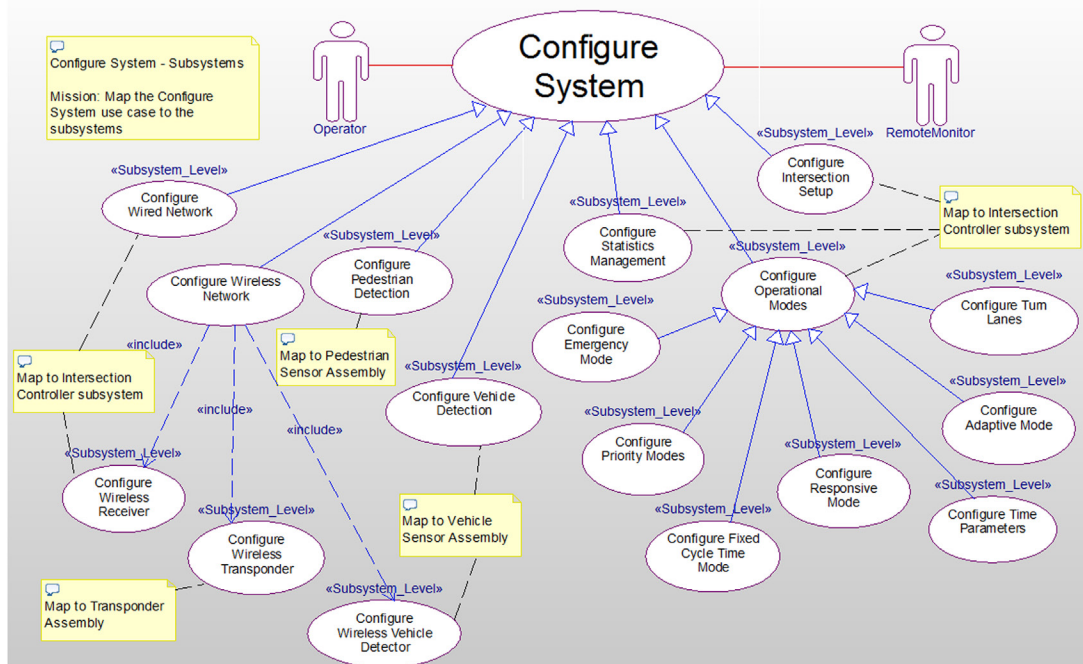


Figure 12.20
Configure System use case maps to subsystems.

Attitude Adjust use case to the subsystems in [Figure 12.21](#). In this figure, we use a different means to depict the allocation of use cases to the subsystems. In the previous example, we used constraints to show the allocation. In [Figure 12.21](#), we show the allocation with system boundaries, one per subsystem.

I create a use case diagram for each system-level use case to show the decomposition to subsystem-level use cases and their allocation to the various subsystems. I call this a “use case allocation diagram,” but it is simply a use case diagram whose mission is to show how the system-level use case maps on to the subsystem architecture.

[Figure 12.22](#) shows the same information as the previous figure, but now for the Perform Area Search use case. If you review the system use case diagrams, you will find that Perform Area Search is a specialized kind of Execute Reconnaissance Mission use case, which is in turn a specialized kind of Execute Mission use case. That latter use case includes the Fly UAV use case. Therefore, rather than directly represent all the included use case mappings as well, [Figure 12.22](#) shows the inclusion of the Fly UAV use case (which in turn includes the Manual Attitude Adjust use case previously shown).

At the end of this allocation process, each subsystem has a set of use cases allocated to it. For example, the use case diagram for the Navigation subsystem might look like [Figure 12.23](#). In this figure, the actors are stereotyped «internal». This indicates that from the perspective of

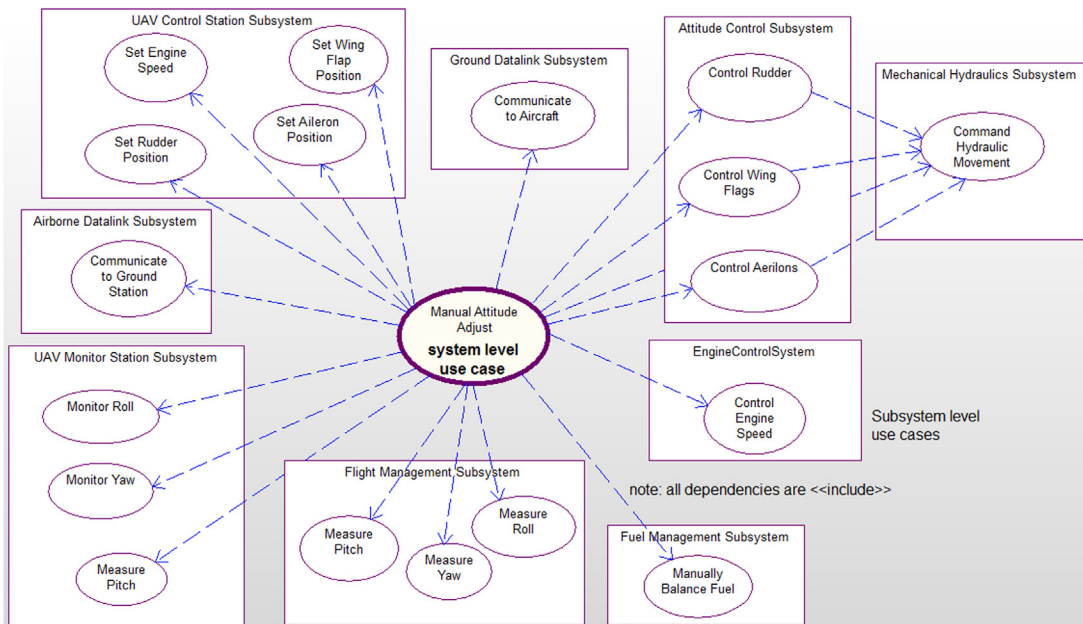


Figure 12-21

CUAV Manual Attitude Adjust use case mapped to subsystems.

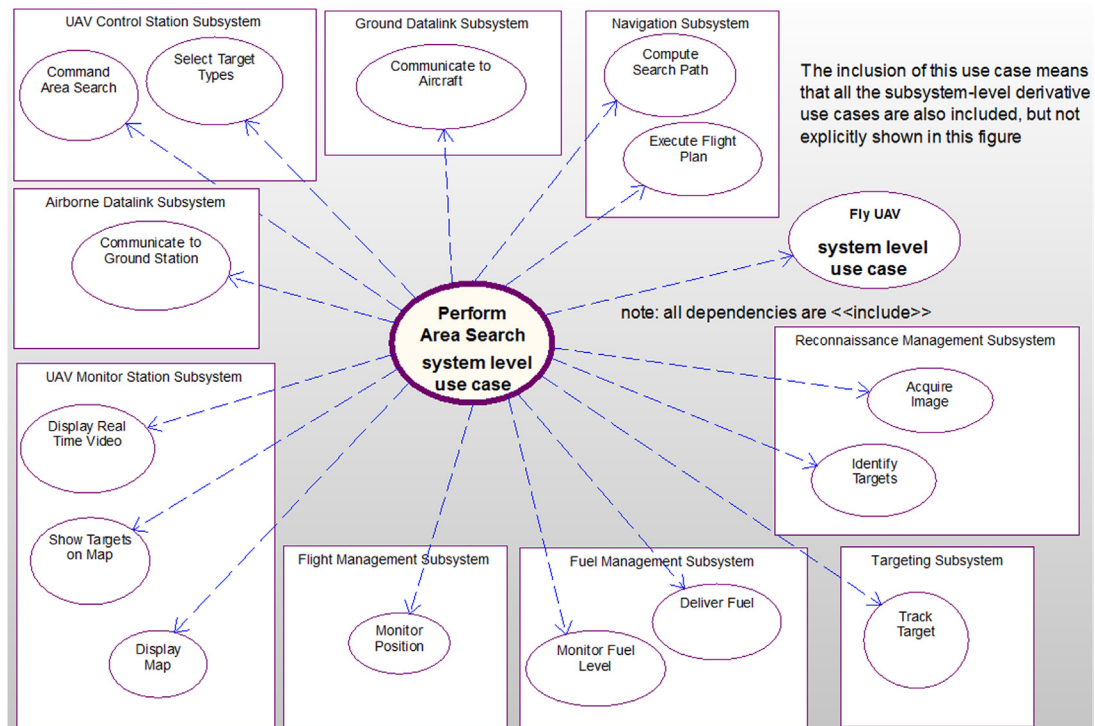


Figure 12.22

CUAV Perform Area Search use case mapped to subsystems.

the subsystem, the elements are actors, but they are internal to the system being developed. That is, the peer subsystems become actors to the Navigation subsystem. In the UML language, strictly speaking, actors are not merely a view of an object – they are its metaclassification. Therefore, Rhapsody requires the actors to actually be different elements than the subsystems they represent. For that reason, I preface the name with an “a” (for “actor”) to distinguish it from the subsystem of the (otherwise) same name, even though in usage, the “actor-ness” of an object is a result of perspective only.

Also note that the use cases are stereotyped «Subsystem_Level» as well. This stereotyping is optional but allows the level of abstraction to be seen at a glance.

The next example, the Flight Management subsystem, has more use cases and certainly more actors (Figure 12.24). These are, again, derived from the system-level use cases, mapping a portion of the system-level use cases to the subsystem in question. Because some of the use cases (such as the Manage BIT⁴ use case) connect to many actors, some of the actors are

⁴ Built-In Test

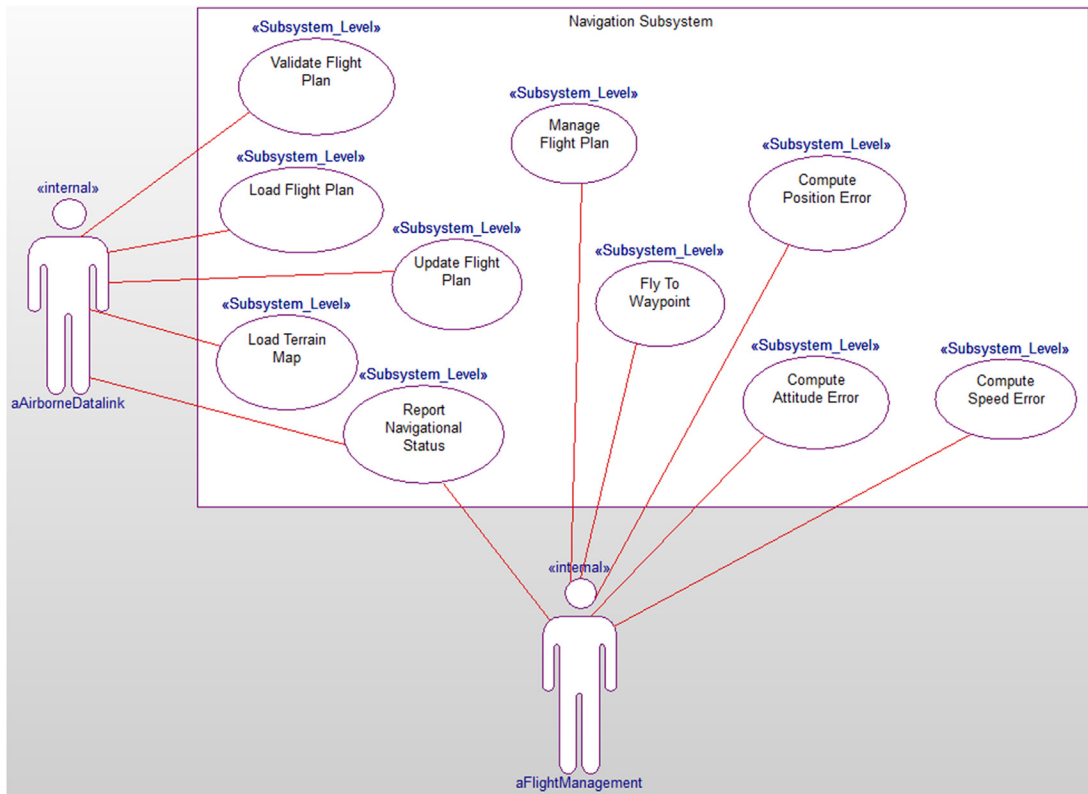


Figure 12.23
Navigation subsystem use cases.

replicated. This means only that they appear multiple times on the diagram. This allows the diagram to minimize line crossing and simplifies the diagram's appearance.

Lastly, once the subsystem use cases have been identified, they can be dragged to the subsystem specifications area of the model. In the case of the Flight Management use case, the browser view of the Flight Management subsystem package looks like [Figure 12.25](#).

12.5 Answer 5.5: Creating the Shared Model

The Roadrunner system is easiest because everything is kept in a single model. This model has two high-level packages. The `SEModel` package holds the systems engineering work and the `SSModel` (for SubSystem model) package holds the hand off and downstream engineering work. [Figure 12.6](#) shows the architecture of the Roadrunner system. While the system has multiple instances of the subsystem designs (notably, the `VehicleLightAssembly`, `VehicleSensorAssembly`, `PedestrianLightAssembly`, and `PedestrianSensorAssembly`), they only need to be designed once. Thus the nested package structure of the `SSModel` package is as shown in

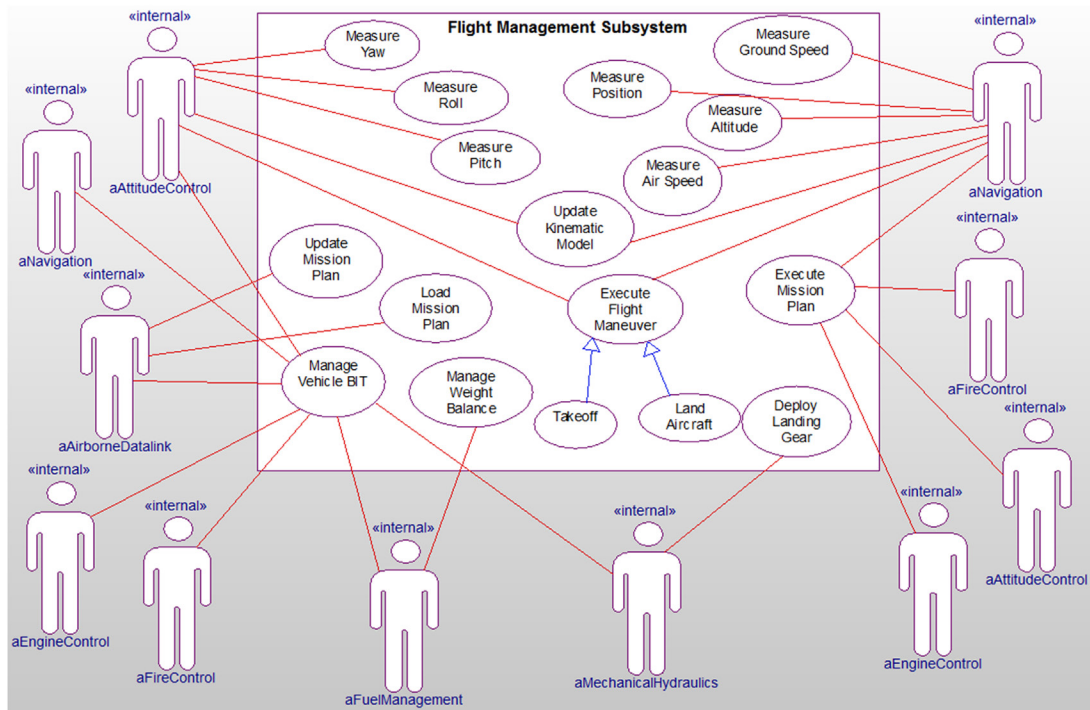


Figure 12.24
Flight Management subsystem use cases.

Figure 12.26. Each of the subsystem packages (such as VLAP_{kg} (short for Vehicle Light Assembly subsystem package)) have a similar basic structure with nested packages for subsystem requirements, subsystem functional analysis, deployment (allocation to various engineering disciplines), software architecture, and software design.

The second part of the Roadrunner hand off has to do with the conversion of the logical system engineering interface to the physical system interfaces. The logical interfaces are shown collectively in [Figure 12.27](#), although a more scalable approach is to create diagrams for each of the subsystems and show the ports and interfaces provided by each subsystem there, just as we did earlier for the logical interfaces in [Figure 12.18](#). This approach is shown in [Figures 12.28 and 12.29](#). I prefer this way of showing interfaces because it scales well to large systems and clusters the information with the context to understand it.

In this case, we will build our own protocol to send the messages between the subsystems and use the same message format (above the network physical layer) with the exception of the link to the remote controller and the front panel – these latter subsystems will connect via Ethernet. We will use a consistent message grammar and use the simple expedient of just

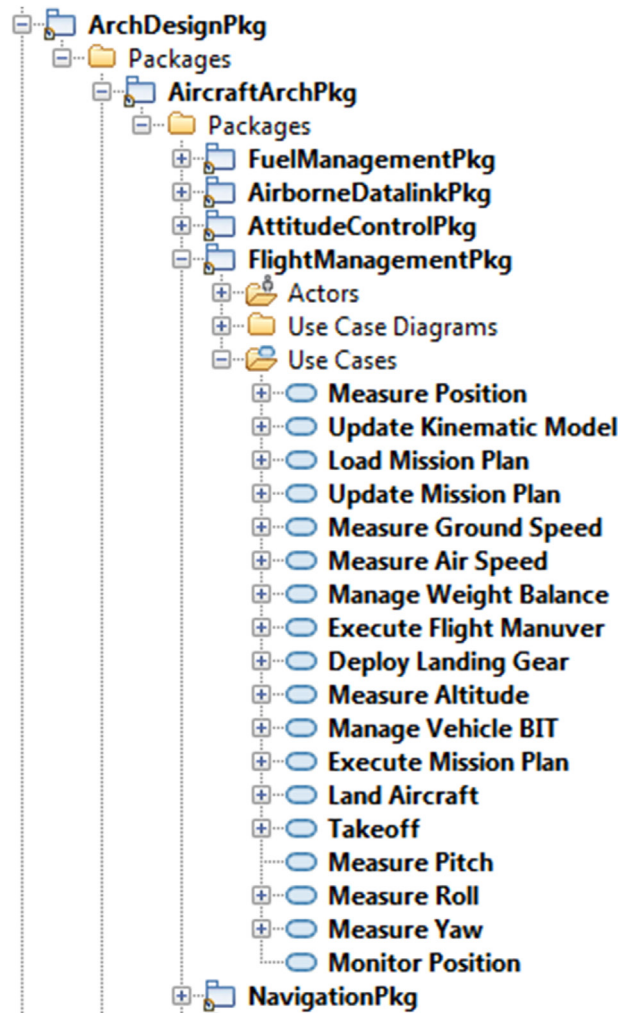


Figure 12.25

Flight Management subsystem specification package contents.

embedding the message within a TCP datagram for Ethernet communication. The message grammar is defined in [Table 12.2](#).

So the task of specifying the interfaces is simply a matter of specifying the electrical signals (for the hardware implementation) and the message command bytes and formats for all messages passed via those interfaces.

For the simple case of the `VehicleLightAssembly` interfaces shown in [Figure 12.28](#), the outcome is shown in [Figure 12.30](#). The advantage of this physical specification is that the other

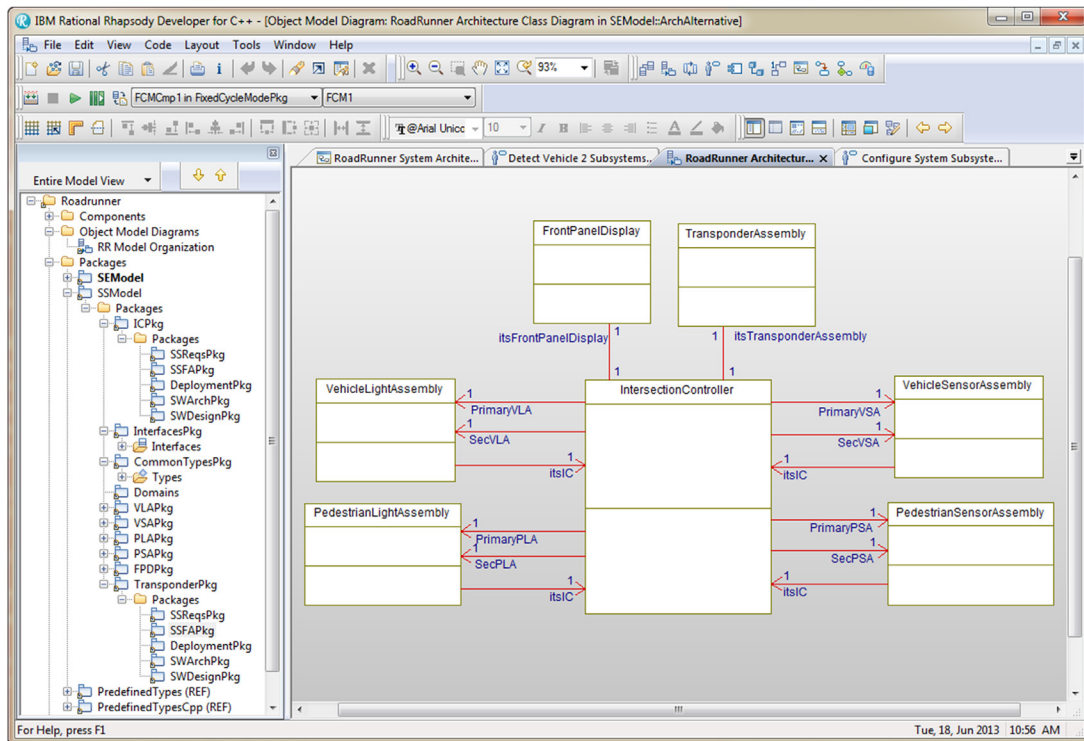


Figure 12.26
Roadrunner downstream engineering package structure.

subsystems, such as the Intersection Controller, can develop their system with an understanding of how their system will actually interface with theirs. You can see the package structure in the model browser on the left-hand side of [Figure 12.30](#).

For the Coyote UAV, the problem was to create a new shared model and structure it to hold the shared information. The result was discussed earlier in this chapter and shown in [Figure 12.3](#).

12.6 Answer 5.6: Initiating the Subsystem Model

The first part of the exercise is to elaborate the subsystem structure in the Roadrunner model for all the subsystems. This should create no particular difficulties. The basic subsystem organization is shown in [Figure 12.1](#) and need only be replicated for all the subsystems (IntersectionController, VehicleLightAssembly, VehicleSensorAssembly, PedestrianLight Assembly, PedestrianSensorAssembly, Transponder, and FrontPanel).

The Navigation subsystem was selected for elaboration in the CUAV model. The browser view is shown on the left-hand side of [Figure 12.31](#). Note that the packages within the

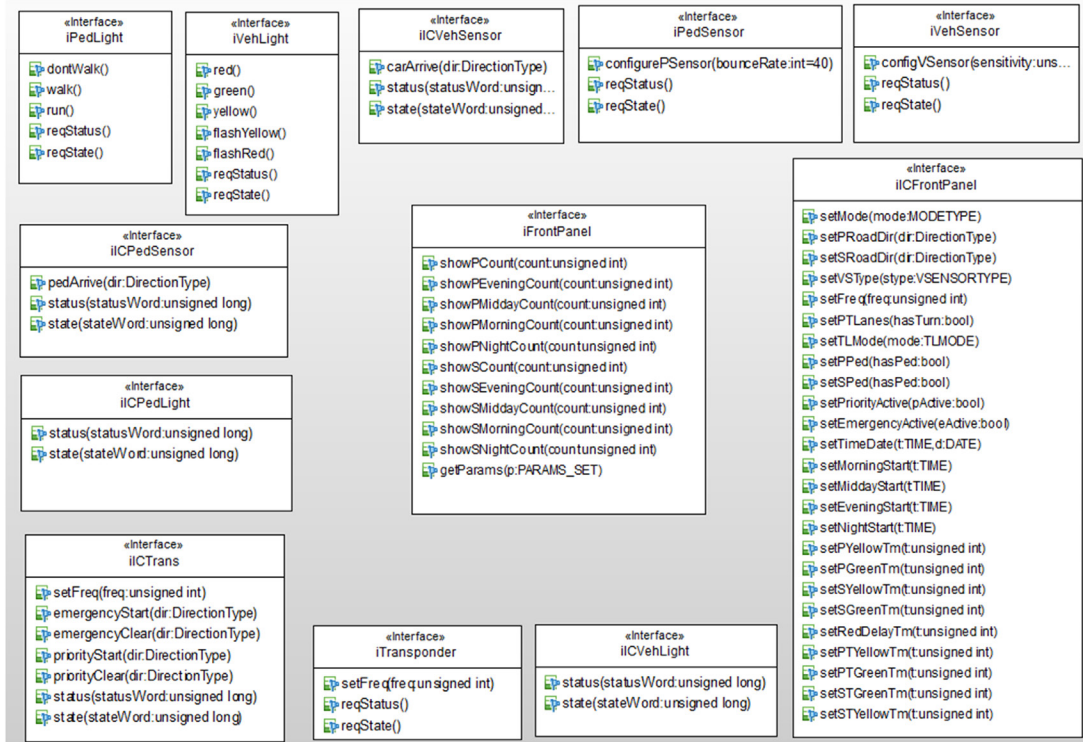


Figure 12.27
Roadrunner interfaces.

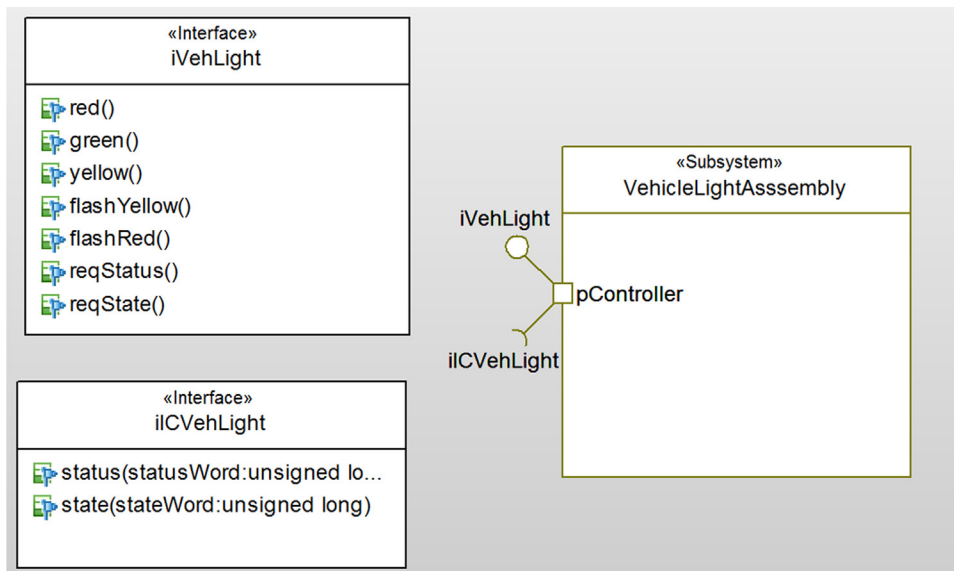


Figure 12.28
VehicleLightAssembly interface diagram.

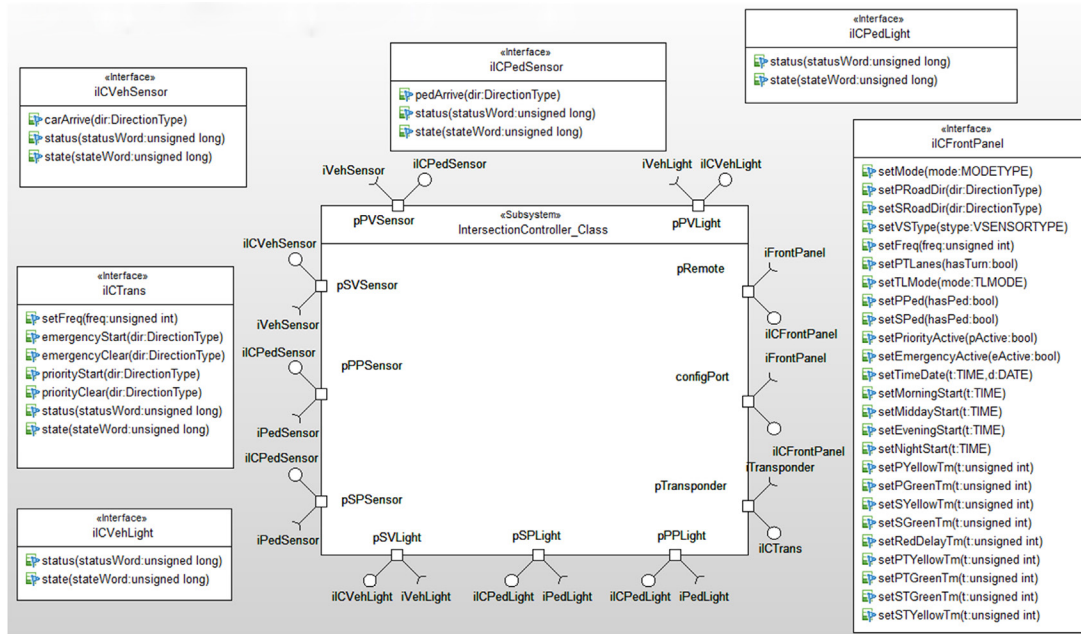


Figure 12.29

IntersectionController_Class interface diagram.

Table 12.2 Roadrunner Message Format.

Word	Name	Description
0	Message length	Number of words in the entire message including the length and CRC words
1	Command	Command word uniquely identifying the data or message being passed
2 .. n	Message content	Message per se
CRC	Message CRC	16-bit CRC

`SharedPkg` are used *by reference* from the separated shared model. That means that the elements within are available for use by reference but cannot be modified except by opening the shared model. Also shown are the contents of the `NavReqsPkg`. This is imported by copy from the system engineering model.⁵

And yes, there was some engineering done in the background to identify the subsystem requirements that was omitted to keep the length of the book under control (it can be considered an extra credit exercise for the reader).

⁵ I use *by copy* for importing the specification from the system engineering model to get better separation from any changes to the SE model that might be done. Should there come a time when I need to use updated specifications, I will reimport the specification to the Navigation subsystem model. Other people prefer to use the specification from the SE model *by reference*. Either approach can work.

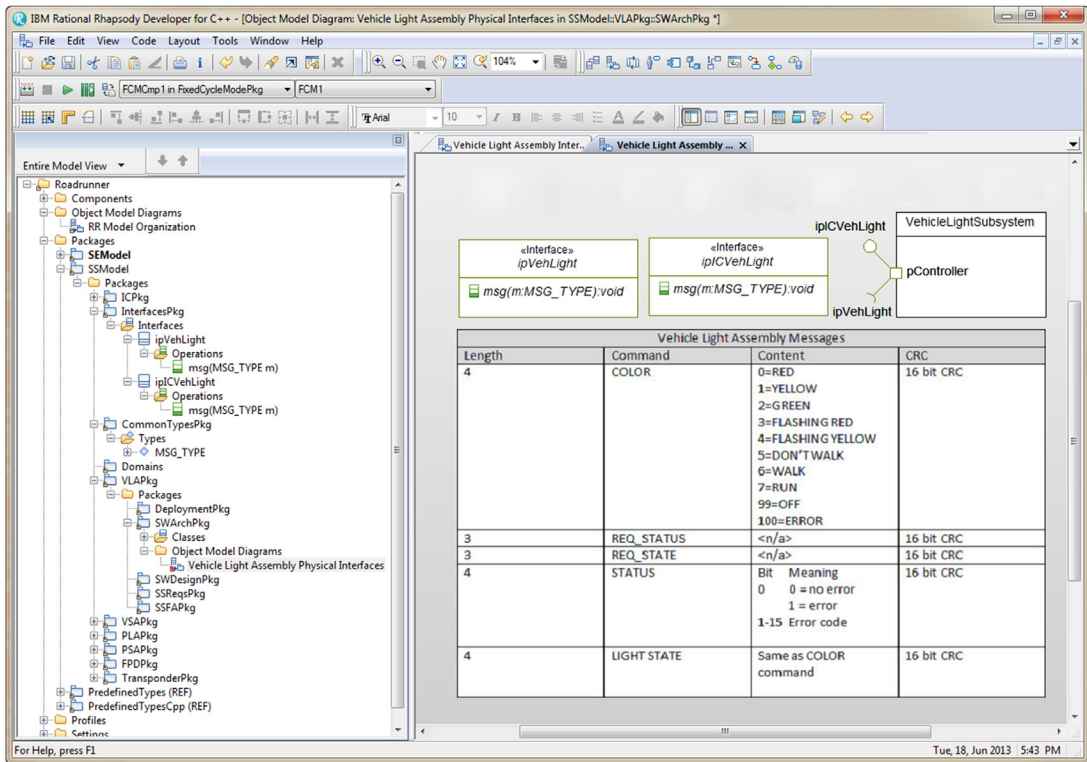


Figure 12.30
Vehicle Light Assembly physical interfaces.

The next step for the problem is to allocate the requirements to the different engineering disciplines and create the cross-discipline interfaces. For the Roadrunner system, I have selected the Vehicle Light Assembly for this purpose (see [Figure 12.32](#)).

The interface to the electronics in this subsystem is pretty simple. The software controls the light electronics with the `enableLight(unsigned char lightID, bool enable)` function, which maps to the hardware in an 8-bit port-mapped interface (bit 0 enables or disables the lights, bits 2-4 select the green, yellow, and red lights, and bits 5-8 are unused). Each light subassembly has three lights (labeled green, yellow, and red) and may be turned off or on together or separately by setting the bits). Flashing is done within the software, although this is done by using the timer interface of the hardware (port mapped for timer setup but interrupt mapped for the timeout signal generated by the electronics). This latter interface is shown in [Figure 12.33](#).

For the CUAV, I selected the Navigation subsystem deployment architecture. This is shown in [Figure 12.34](#). This diagram is a little busy and if it were much more complicated, I would break it up into multiple diagrams.

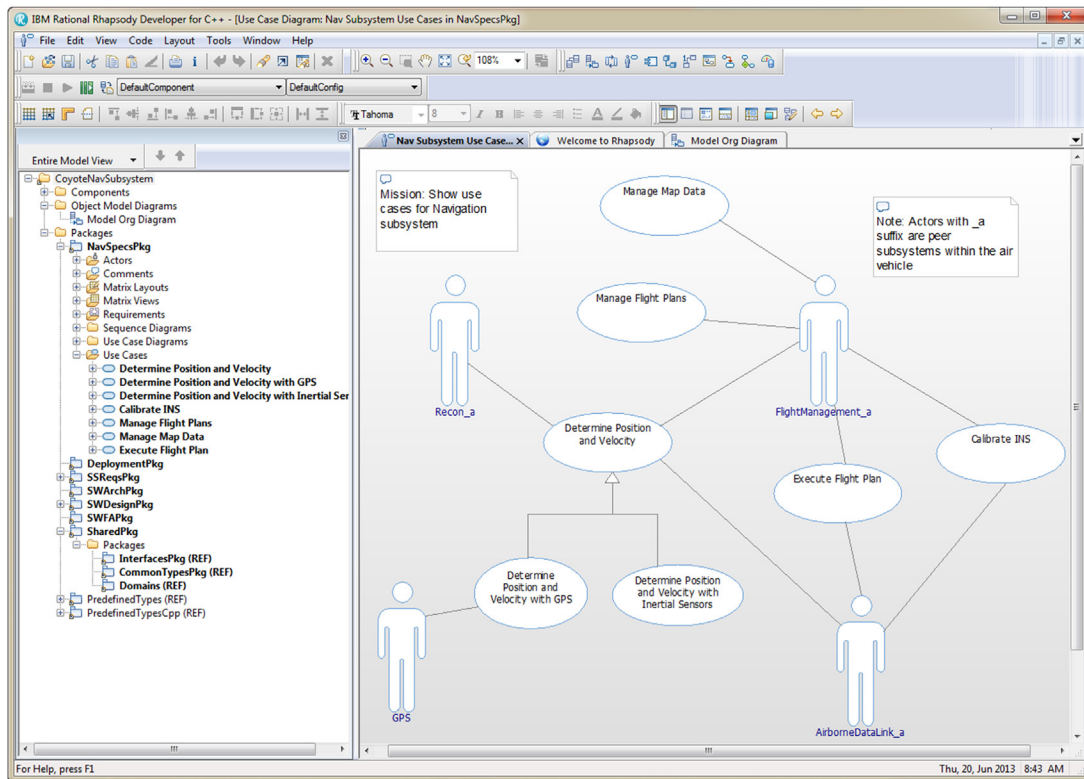


Figure 12.31
Navigation subsystem model organization.

The most common way to decompose such a deployment diagram into multiple views is by the type of connection – such as SW-EE, EE-ME. Other strategies can be employed, such as separating the diagram by levels of abstraction; in this case, [Figure 12.34](#) should be shown in three diagrams, one depicting the connections among the big components of the subsystem, one showing the internal breakdown of the INS, and the third showing the breakdown of the GPS.

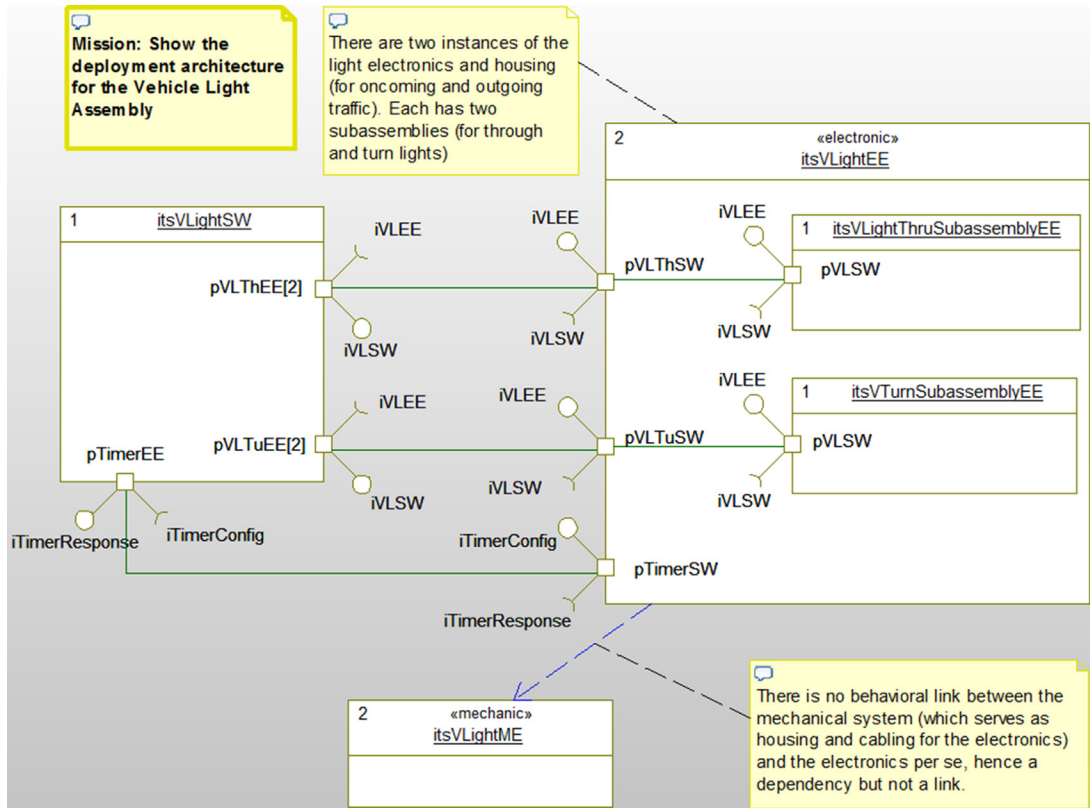


Figure 12.32
Vehicle Light Assembly deployment architecture.

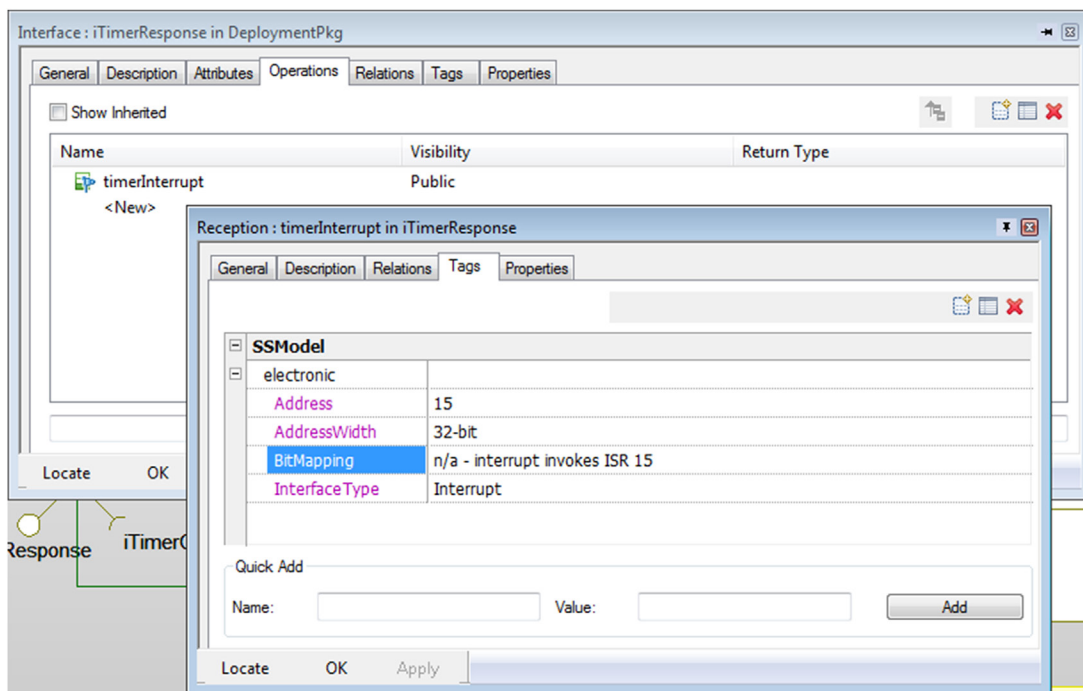


Figure 12.33
Roadrunner Vehicle Light Assembly electronic interface.

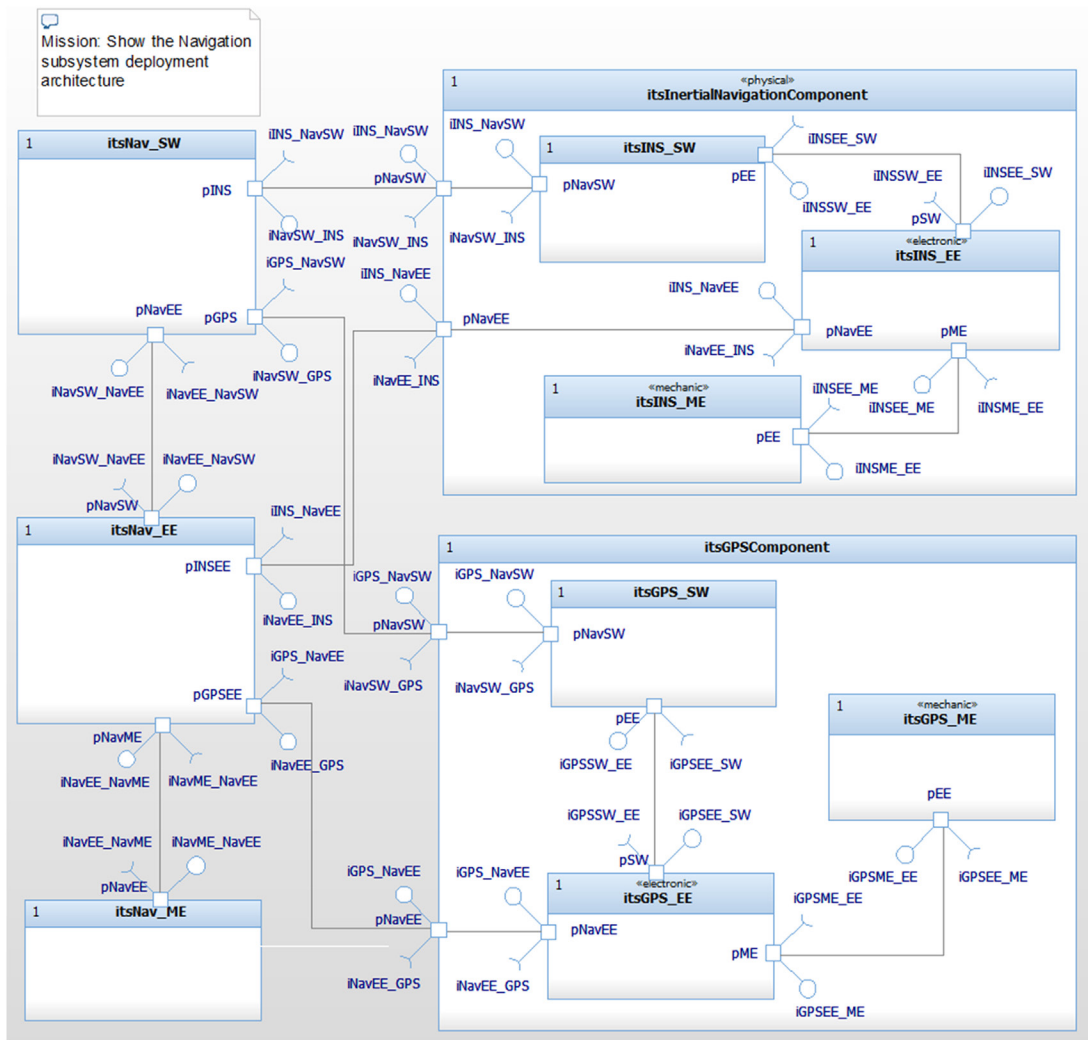


Figure 12.34
Navigation subsystem deployment architecture.

Dependability Architecture: Answers

13.1 Answer 6.1: Safety Architecture

For the Roadrunner Traffic Light Control System, the exercise involves doing the following (taken from Problem 6.1):

- Take the Roadrunner subsystem architecture previously completed (shown in Figures 12.5 and 12.6) and do a separate fault tree analysis (FTA) on each of the fault situations below:
 - Both primary and secondary through traffic have GREEN lights.
 - Pedestrian traffic has a WALK light at the same time that the orthogonal vehicle traffic has a GREEN light.
 - All lights are off.
- From the result of this analysis, create a hazard analysis in which hazards and faults are identified and control measures are added, including estimated fault tolerance times.
- Determine how architectural redundancy should be added to make the system single point fault safe for the analyzed fault situations.
- Draw the safety architecture class or object diagram showing how the elements collaborate to achieve the desired safety.

Figure 13.1 shows an FTA diagram for the All Lights Green case. The easiest way to read this diagram is from the top (the resulting hazardous condition) down; the only way to get both lights green is for one of them to be green and then the other turns green. For one of them to turn green in this case, one of three conditions must occur: either the command to the light is corrupted, the command was issued inappropriately, or a fault in the light causes it to turn green; and so on, down to more primitive fault events.

The next figure, Figure 13.2, is a fault tree for getting a green light for the road traffic in one direction while getting a walk light for the pedestrian traffic in the orthogonal direction. The two primary conditions analyzed are (1) the pedestrian light is already WALK and the vehicle traffic light for the orthogonal road turns green (left side of the figure), and (2) the light is already green and then the walk light turns to WALK (right side of the figure).

Figure 13.3 looks at the last analyzed condition, All Lights Off. If only one light turned off, the situation might not be unsafe, if the light for the orthogonal direction turns to RED; however, the situation is clearly unsafe if all lights are off. Of course, it is not *as* unsafe as in the first hazardous condition analyzed (All Lights Green) because most people, seeing that the light is

Note: Road A and B are arbitrary, orthogonal roads, so the logic applies equally well to the situation in which Road A light is GREEN and Road B light becomes GREEN

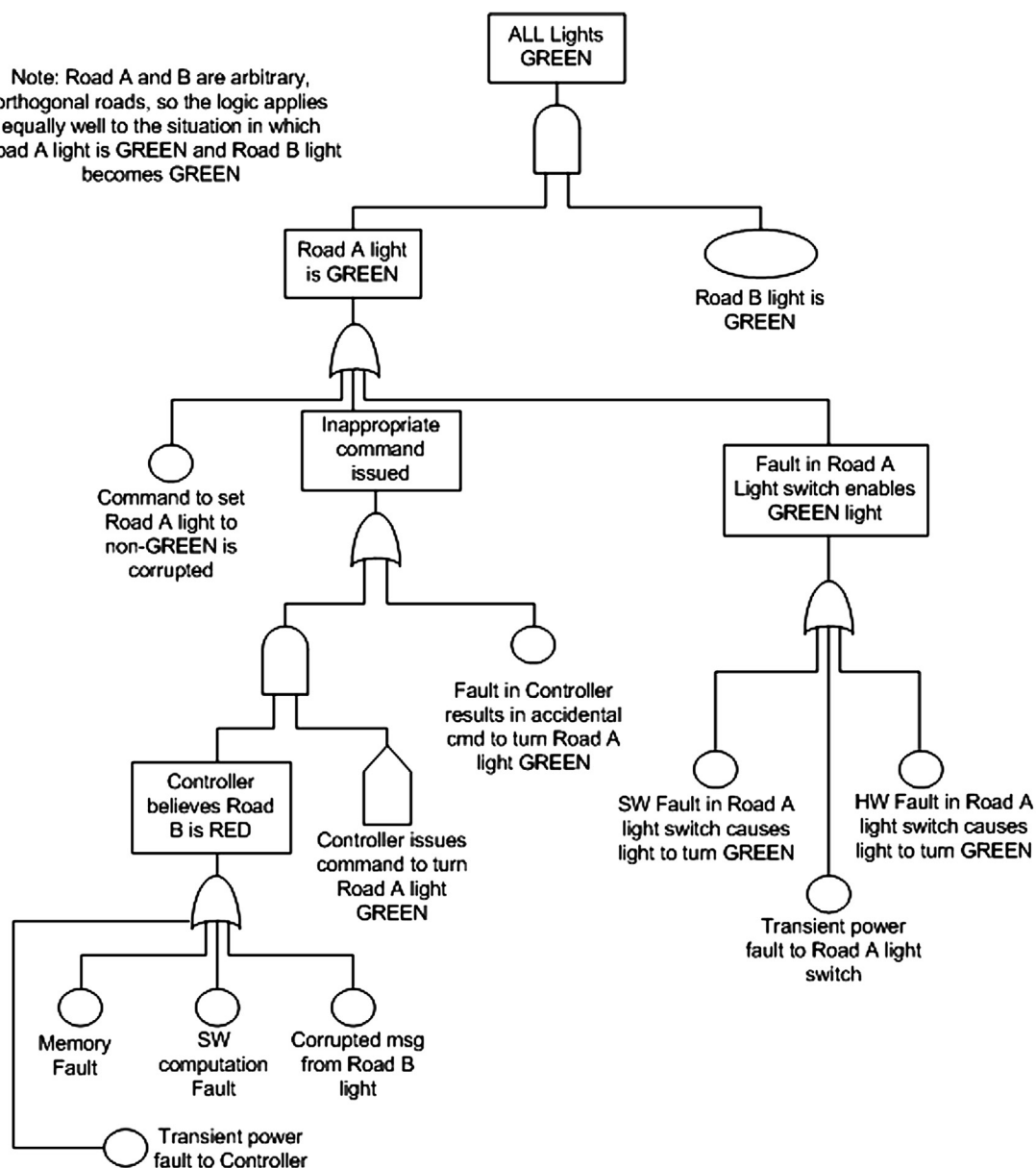


Figure 13.1
Roadrunner FTA for All Lights Green fault.

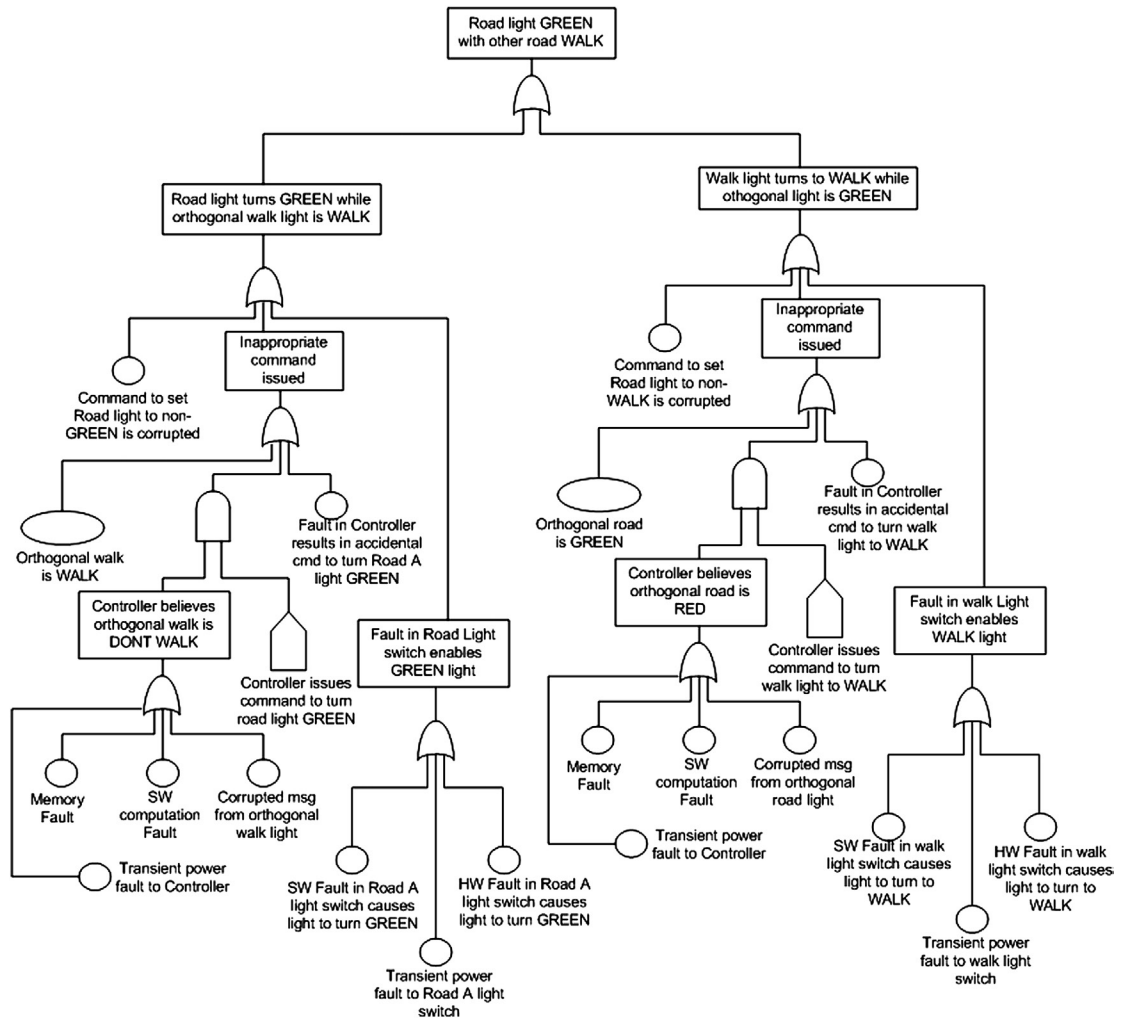


Figure 13.2

Roadrunner FTA for Walk and Cross Green fault.

off, will slow down and proceed cautiously.¹ If they see the light is green, however, they will approach the intersection at full speed and with far less caution. We see in Figure 13.3 that either a general power failure to all lights or commands to turn off the lights (sent because of a controller or communications fault) could result in the situation in which all lights are off.

The advantage of the FTA is that we can clearly see the logic of the confluence of conditions and events necessary to create the hazardous situation. It becomes simple (in principle)² to make

¹ Except in Boston and parts of (ok, *most of*) Italy, where I was once honked at for stopping at a red light.

² Remember, though, the 97th Law of Douglass: “The difference between Theory and Practice is Greater in Practice than it is in Theory.” And, of course, the 98th Law as well: “The purpose of Theory is to improve Practice.”

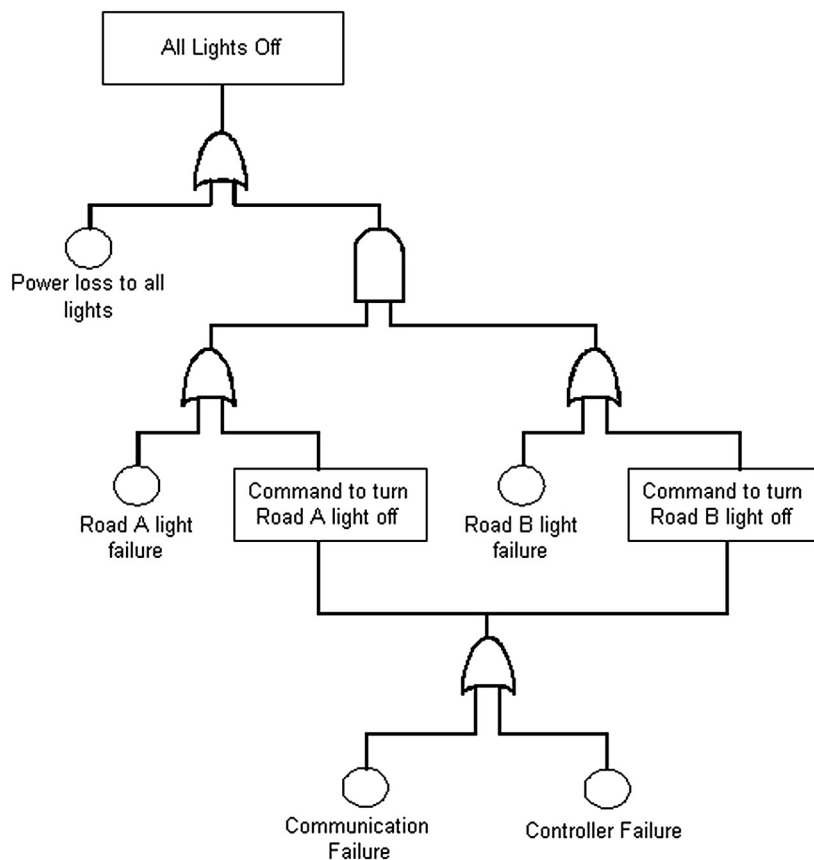


Figure 13.3
Roadrunner FTA for All Lights Off fault.

the system safe by creating other conditions that must be ANDed with the identified faults and conditions to result in an unsafe condition. These “other conditions” are called “safety measures.” For example, in [Figure 13.3](#), there is a fault called “Power loss to all lights.” Clearly, if the lights have no power, then the condition arises. However, if we add a UPS (uninterruptible power supply) to each individual light, that condition can’t arise unless *both* the main power *and* the UPS fail. We can modify the light behavior to make the system safe by making each light go to RED while on the UPS as well as notify the remote monitoring station of the power loss; in addition, we can add “lifeticks” between the lights so that if a light manifests a fault (and so doesn’t respond to the lifetick), the light expecting the lifetick goes into its “fail-safe state.” Thus, to arrive at the unsafe condition under consideration, two independent faults must manifest themselves at the same time – this is what is meant by the term “single point failure safe.”

The hazard analysis ([Figures 13.4 through 13.6](#)) below summarizes these faults and adds safety measures. These safety measures, which are typically the instantiation of safety

Hazard	Fault	Severity (1 [low] –10 [high])	Likelihood (0–10)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
[All Lights Green]	Transient power fault to controller	10	8	80	seconds	2	0.01	UPS for controller; notify remote monitor upon power failure	0	0.01	TRUE
	Memory fault	10	5	50	seconds	2	0.01	CRC set and checked on each state variable access	0.01	0.02	TRUE
	SW computation fault	10	3	30	seconds	2	0.1	Explicit check for condition	0.01	0.11	TRUE
	Corrupted message	10	8	80	seconds	2	0.05	Message CRC with ACK required before state change	0.02	0.07	TRUE
	Accidental message transmission	10	3	30	seconds	2	0.01	Buffers zeroed after message send	0.01	0.02	TRUE
	SW fault in light assembly	10	2	20	seconds	2	0.5	Lifeticks containing light state shared among all lights; if not received, fail-safe state initiated	0.1	0.6	TRUE
	HW fault in light	10	3	30	seconds	2	0.5	UPS for controller; notify remote monitor upon power failure	0.1	0.6	TRUE
	Transient power fault to light assembly	10	8	80	seconds	2	0.01	UPS for light assemblies- either individually or for entire system	0	0.01	TRUE

Figure 13.4

Hazard analysis for All Lights Green hazard.

patterns, constitute the safety architecture. This hazard analysis is shown in three tables, one per fault. In many such analyses, the hazard analysis is a single huge table of hazards, faults, and safety measures.

We see in Figure 13.4 the hazard All Lights Green and the various faults we identified in the previous FTA. We then elaborated the hazard analysis with control measures, and added in (what we hope are) reasonable detection and control action times. From this, the exposure time and safety is computed.

Figure 13.5 presents the same sort of hazard analysis for the hazard of having the walk lights in the WALK state while at the same time having the road light for the orthogonal road in the GREEN state. Many of the faults are the same, but others are added, because faults in the walk light system or commands can also lead to the hazard.

The last hazard analysis, in Figure 13.6, shows the safety analysis for the All Lights Off hazard. From this analysis, the model can be updated to include the safety measures, and the FTA can be redone. Figure 13.7 shows the FTA for the All Lights Off condition with the added safety measures. These safety measures are ultimately ANDed with the original fault's events, so that to achieve the hazardous condition both the original fault must occur and the safety measure must fail.

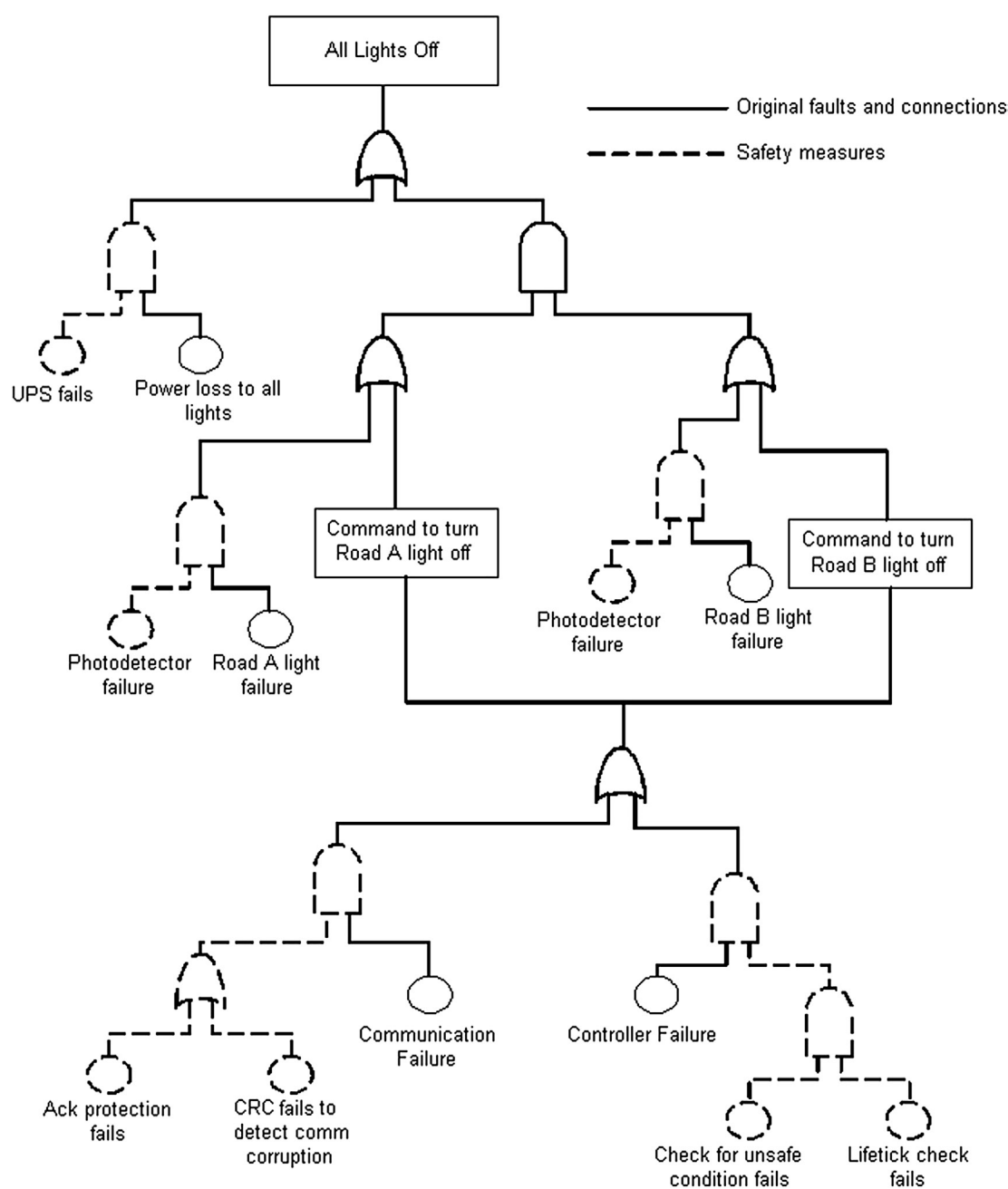
Having completed this analysis, we can update our architecture to include the safety measures. It is interesting to note that the vehicle detectors do not have an impact on safety – if

Hazard	Fault	Severity (1 [low] – 10 [high])	Likelihood (0 – 10)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
Walk and Cross Green	Transient power fault to controller	10	8	80	seconds	2	0.01	UPS for controller; notify remote monitor upon power fault	0	0.01	TRUE
	Memory fault	10	5	50	seconds	2	0.01	CRC set and checked for every access	0.01	0.02	TRUE
	SW computation fault	10	3	30	seconds	2	0.1	Explicit check for condition	0.01	0.11	TRUE
	Corrupted message	10	8	80	seconds	2	0.05	Message CRC with ACK required	0.02	0.07	TRUE
	Accidental message transmission	10	3	30	seconds	2	0.01	Buffer zeroed after message send	0.01	0.02	TRUE
	SW fault in light assembly	10	2	20	seconds	2	0.5	Lifeticks containing light state sent among lights; if not received, fail-safe initiated and broadcast	0.1	0.6	TRUE
	HW fault in light assembly	10	3	30	seconds	2	0.5	Lifeticks containing light state sent among lights; if not received, fail-safe initiated and broadcast	0.1	0.6	TRUE
	Transient power fault in light	10	8	80	seconds	2	0.01	UPS for each light or complete system	0	0.01	TRUE
	SW fault in walk light	10	2	20	seconds	2	0.5	Lifeticks containing light state sent among lights; if not received, fail-safe initiated and broadcast	0.1	0.6	TRUE
	HW fault in walk light	10	3	30	seconds	2	0.5	Lifeticks containing light state sent among lights; if not received, fail-safe initiated and broadcast	0.1	0.6	TRUE
	Transient power fault in walk light	10	8	80	seconds	2	0.01	UPS for each light or complete system	0	0.01	TRUE

Figure 13.5
Hazard analysis for Walk and Cross Green hazard.

Hazard	Fault	Severity (1 [low] – 10 [high])	Likelihood (0 – 10)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
All Lights Off	Power loss to all lights	5	5	25	seconds	2	0.01	UPS to each light or to entire system	0.1	0.11	TRUE
	Light failure	5	9	45	seconds	2	0.01	Photodetector in light detects fault; when detected broadcast fault to initiate fail-safe	0.01	0.02	TRUE
	Communication failure	5	6	30	seconds	2	0.05	CRC on message; ACK required	0.02	0.07	TRUE
	Controller failure	5	3	15	seconds	2	0.05	Explicit check for condition	0.02	0.07	TRUE

Figure 13.6
Hazard analysis for All Lights Off fault.

**Figure 13.7**

Roadrunner FTA for All Lights Off with safety measures.

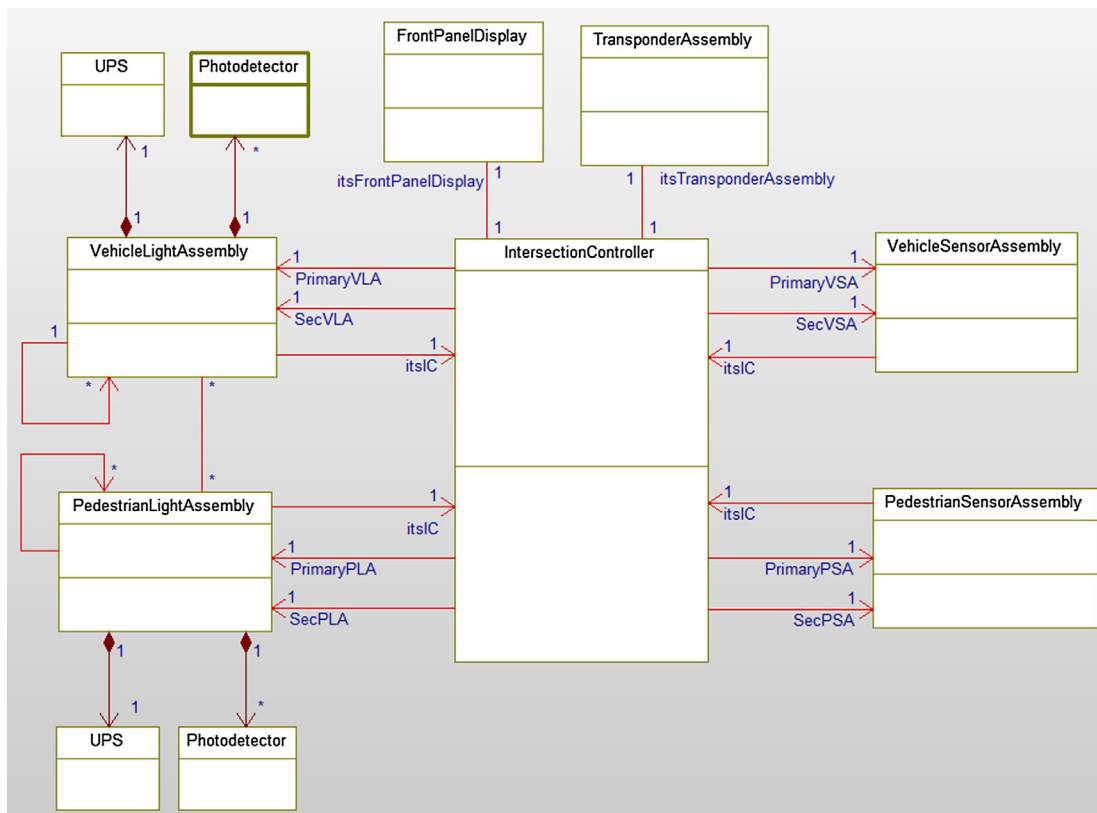


Figure 13.8
Roadrunner safety architecture.

they malfunction, either by missing detections or by providing false detections, the lights will still ensure proper traffic flow. Similarly, the interaction with emergency and priority vehicles are a convenience, not a safety issue. We still want them to work and so may want to add redundancy for these elements, as we did for the lights and controller, but their failure is a reliability, rather than a safety, concern. The updated Roadrunner class diagram is shown in [Figure 13.8](#). Of course, the safety-related behavior, such as checking the CRCs and lifeticks and notifying the remote monitor of faults, isn't shown in the class diagram, so some of these behaviors are indicated with constraints.

It should be noted that we've done both a heterogeneous redundancy specialization of the channel architecture pattern here by providing diverse power supplies to various system elements, as well as inserting data validity checks in the manner of the single channel protected pattern.

The next part of the exercise repeats this work for the Coyote UAV. Since we've focused previously on developing the reconnaissance management subsystem – involved in target identification, tracking, and targeting – we'll continue in that vein.

- Create an FTA for the reconnaissance management subsystem that identifies the safety hazards and risks associated with this subsystem, one per hazard.
- From the result of this analysis, create a hazard analysis in which hazards and faults are identified and control measures are added, including estimated fault tolerance times.
- Determine how architectural redundancy should be added to make the system single point fault safe for the analyzed fault situations.
- Draw the safety architecture class or object diagram for the subsystem showing how the elements collaborate to achieve the desired safety.

For the CUAV, we'll focus on three views with respect to the safety architecture. First is the system architecture – how the overall system, including the ground station, aircraft, and payloads work together. Next is the aircraft architecture – how the aircraft subsystems interact. Lastly, we'll look inside the Reconnaissance Management subsystem (RMS) to see how the internals of that subsystem function, and how they relate to safety and reliability.

We will, of necessity, severely limit the scope of our analysis. In a real system, we would have to look at hundreds of hazards and even more faults, to ensure system safety and reliability. In this answer, we will limit our scope to hazards directly due to faults with the reconnaissance data or management.

To begin with, let's look at the structure of the system under consideration. Figure 12.7 shows the overall CUAV architecture including the ground system, aircraft, and payloads. Figure 12.8 shows the subsystem architecture of the ground system, although not a great deal of detail about the internal structure of the control or monitoring stations. Lastly, Figure 12.9 shows the subsystem architecture of the aircraft itself.

Given that structural overview, let's now list the hazards we are specifically addressing:

- Target misidentification (i.e., “false positive”)
- Lack of (valid) target identification (i.e., “false negative”)
- Loss or corruption of target information either in storage or transmission, e.g.
 - Incorrect target classification or Combat ID (CID)
 - Incorrect location or coordinates
 - Loss of information about target
 - Incorrect target kinematics (vector, flight path, etc.)
 - Incorrect target management (e.g., failure in sensor fusion)

We have focused previously on the reconnaissance management and not the targeting system, so it makes sense to understand how the responsibilities of the RMS and Targeting subsystems differ. The RMS does low-level surveillance of the physical environment and then identifies elements in the environment that are likely to be targets of interest – this is the job of the `TargetFinder` class. The `TargetFinder` has a set of `TargetSpecs`, each of which identifies the observable characteristics of something to look for – size, shape, image template, color,

speed, etc. Once a target is identified and classified by the `TargetFinder`, it is put into the list of targets, managed by the `TargetManager`. When appropriate, it can be passed to the Targeting subsystem as a target at which to fire a missile. The Fire Control subsystem is responsible for guiding the missile to the target under direction from Mission Ops personnel.

These faults can be analyzed with FTA. [Figure 13.9](#) looks at the Target Misidentification and Failure to Identify Target hazards; to save space, the safety measures have already been added (as dashed lines) to this FTA. The primary culprits that could lead to misidentification, either false or negative, are

- Communication corruption
- Memory corruption
- Computation fault
- Noise reduction algorithm insufficient to resolve the noise

Other than the last, these issues are addressed by adding CRC protection to messages to be transmitted and to data *in situ*, on a per-class basis. This protection must be added to communication links within the aircraft, within the ground station, and to the command (low-speed data) and reconnaissance (high-speed data) links between the two.

Note the use of the single channel protected pattern in the inclusion of lightweight data validity checks. Heterogeneous redundancy improves safety through the use of two computational paths (using different algorithms) for the computation of target classification, location, kinematics, and CID. In addition, the problem itself also offers heterogeneous redundancy in that it has two wireless links. It is expected that the high-speed link will fail due to jamming or noise before the low-speed link. The low-speed link is to be designed with broad spectrum and built-in redundancy (e.g., CRCs) and encryption to handle security. The high-speed link does not have these measures because (1) there isn't enough bandwidth to add much redundancy, but 32-bit CRCs are added to the video frames for error detection; 2) the loss of data isn't as critical as undetected corruption; and 3) practice has shown that real-time telemetry interception is not a security risk, since the enemy presumably already knows where they are ;-).

Our detailed design of the RMS might internally use the heterogeneous redundancy and single channel protected patterns to achieve the desired safety level:

- Heterogeneous redundancy pattern
 - Target identification software is replicated using different algorithms to perform preprocessing and target identification (which also computes CID).
- Single channel protected pattern
 - Command processor validates messages with CRCs.
 - `ImageManager` uses CRCs on navigation data messages from the Navigation subsystem.

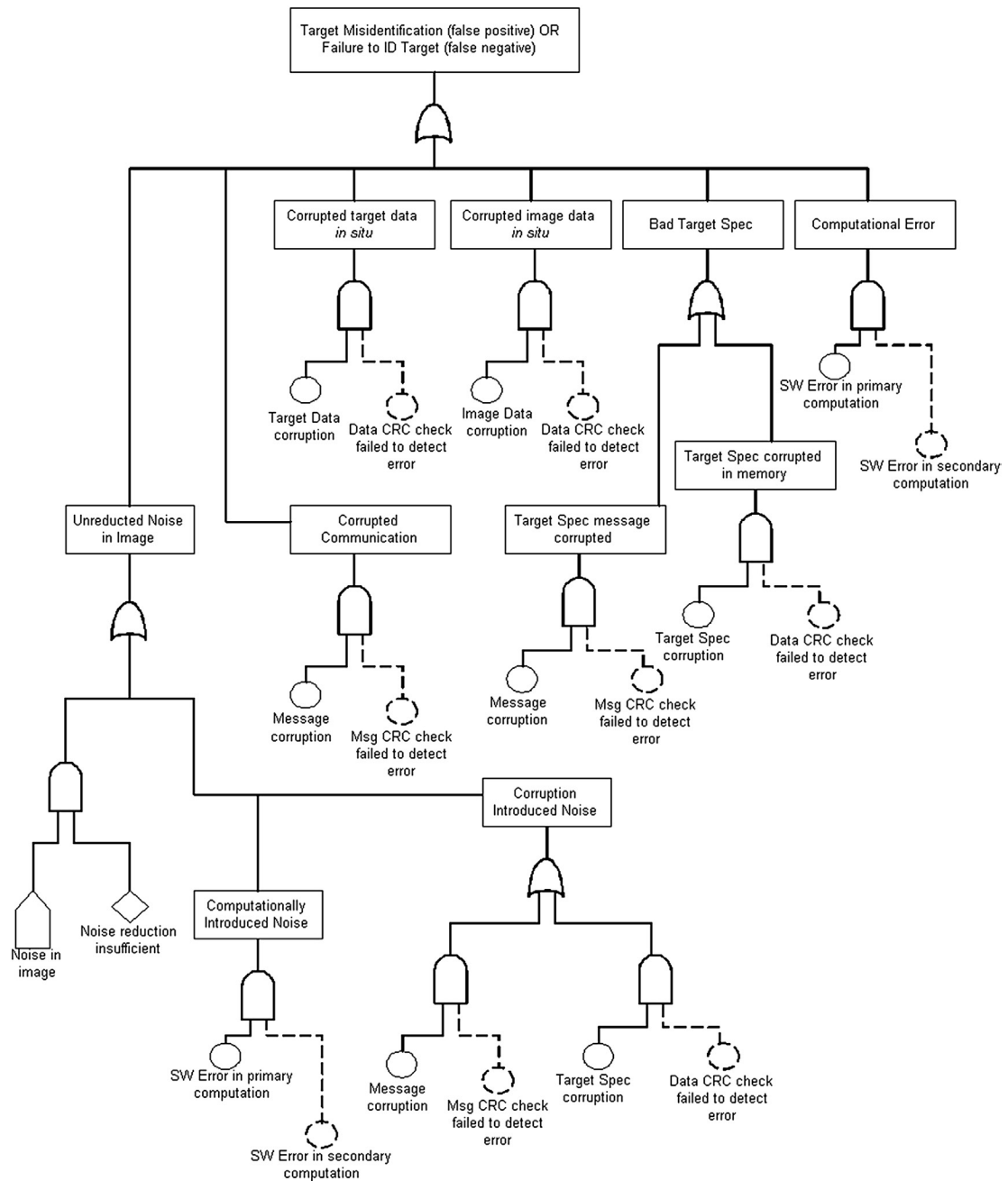


Figure 13.9
CUAV FTA for target misidentification.

- Targets are stored with one's complement (bit-inverted) copies of location and direction attributes so that *in vivo* corruption can be identified.
- Images contain a CRC, which is used to identify image corruption.
- Image management uses algorithmic inversion to “back compute” the original data from the computation for both compression and encryption operations.
- The BIT class performs built-in test procedures in a separate thread to check the stack (using “high water marks”)³, runs a parity memory test (supported by parity in hardware), and checks that all threads are running and issuing lifeticks at a minimal rate.

These collaboration-level design patterns are not entirely visible at the architectural level, but the overall safety architecture for the RMS supports these mechanisms. This is shown in [Figure 13.10](#). Attributes and operations explicitly supporting these safety concerns are shown within the classes.

The hazard analysis adds safety-relevant metadata to the faults to support reasoning about the adequacy of the control measures ([Figure 13.11](#)).

13.2 Answer 6.2: Reliability Architecture

Let's look at the Roadrunner system for the delivery of the detect vehicle service. [Figure 12.5](#) provides the overview of the entire architecture. The two subsystems relevant to this discussion are the Vehicle Sensor Assembly and the Intersection Controller. We can draw up first-cut architecture for the former subsystem easily ([Figure 13.12](#)).

[Figure 13.13](#) shows an FMEA for the Roadrunner Vehicle Sensor Assembly service for detection of arriving vehicles. Architectural actions include the addition of a system-level UPS. Other actions include thickening the housing for the induction cable, updating internal software design, and updating the operations manual to include monthly tests.

For the CUAV vehicle, the subsystems from [Figure 12.9](#) are the Flight Management, Attitude Control, and Mechanical Hydraulics subsystems. The Flight Management system computes the kinematic model of the aircraft, computing the correct orientation of the aircraft (roll, pitch, and yaw). Based on sensor data and current flight surface positions, this results in commands to actuators to update control surface position to either maintain stable flight or to execute a change in flight control.

[Figure 13.14](#) shows a somewhat simplified FMEA for the Control Aircraft Attitude service. A real aircraft would have many more rows as well as many different services to be analyzed than is shown in the figure. You can see, however, how the FMEA supports the identification

³ The stack “high water mark” test simply fills the stack for each task with a known pattern (such as “0xA5”) and checks periodically that these values have not been overwritten above (or below, as appropriate) a certain position in the stack.

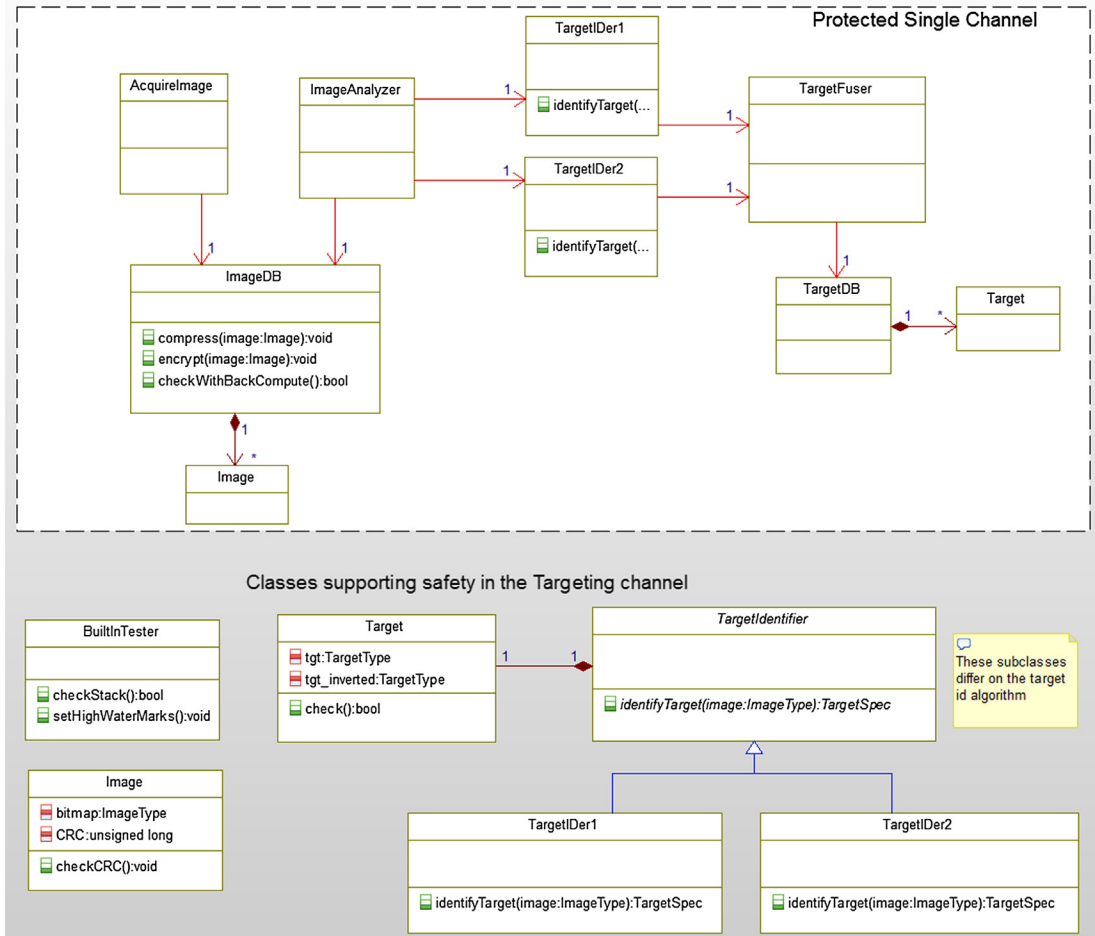


Figure 13.10
Reconnaissance Management collaboration with safety patterns.

of redundancy to support the high-reliability goals of the aircraft. Using multiple control surfaces (such as two independently controlled rudders) permits the aircraft to control attitude, if somewhat less precisely, in the event of certain kinds of faults.

13.3 Answer 6.3: Security Architecture

This problem was to apply model-based threat analysis to some portion of the Roadrunner and Coyote systems.

For the Roadrunner system, let's consider the wireless links (Figure 13.15). The Roadrunner system has multiple wireless connections, not all of which we will concern ourselves with. First, the connection between emergency and priority vehicles and the system is done via a

Hazard	Fault	Severity (1 [low] –10 [high])	Likelihood (0 –10)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe ?
Target Misidentification	Noise reduction insufficient	10	8	80	seconds	1.00	0.1	Use multiple algorithms for noise reduction	0.05	0.15	TRUE
	SW error in primary computation	10	4	40	seconds	0.20	0.001	Use two computational algorithms	0.05	0.051	TRUE
	SW error in secondary computation	10	4	40	seconds	0.20	0.001	Use two computational algorithms	0.05	0.051	TRUE
	Message corruption	10	7	70	seconds	0.20	0.001	CRC on messages	0.001	0.002	TRUE
	Target spec corruption	10	5	50	seconds	0.20	0.05	CRC	0.001	0.051	TRUE
	Target data corruption	10	5	50	seconds	0.20	0.05	Bit inversion redundancy	0.002	0.052	TRUE
	Image data corruption	10	5	50	seconds	0.20	0.05	CRC	0.001	0.051	TRUE

Figure 13.11
Hazard analysis for CUAV Target Misidentification hazard.

directional IR sensor, and only transmits the presence of the vehicle. The second set of wireless links are links to the above-surface vehicle detectors, which include the radar and infrared detectors. These are true connections permitting bidirectional information and command transfer and so are subjects of concern.

For the CUAV, let us consider the GPS navigation system (Figure 13.16). This system is used by the aircraft to (1) report to the ground where it is and (2) execute flight plans. Let us suppose that it were possible to spoof the GPS system to convince the aircraft it was someplace else entirely – not that this could ever happen.⁴ But if it did, the aircraft could be stolen or even convinced to fire munitions at friendly targets.

⁴ See Iran-U.S. RQ-170 Incident, https://en.wikipedia.org/wiki/Iran%E2%80%93U.S._RQ-170_incident, and GPS Spoofing, https://en.wikipedia.org/wiki/Spoofing_attack#GPS_Spoofing.

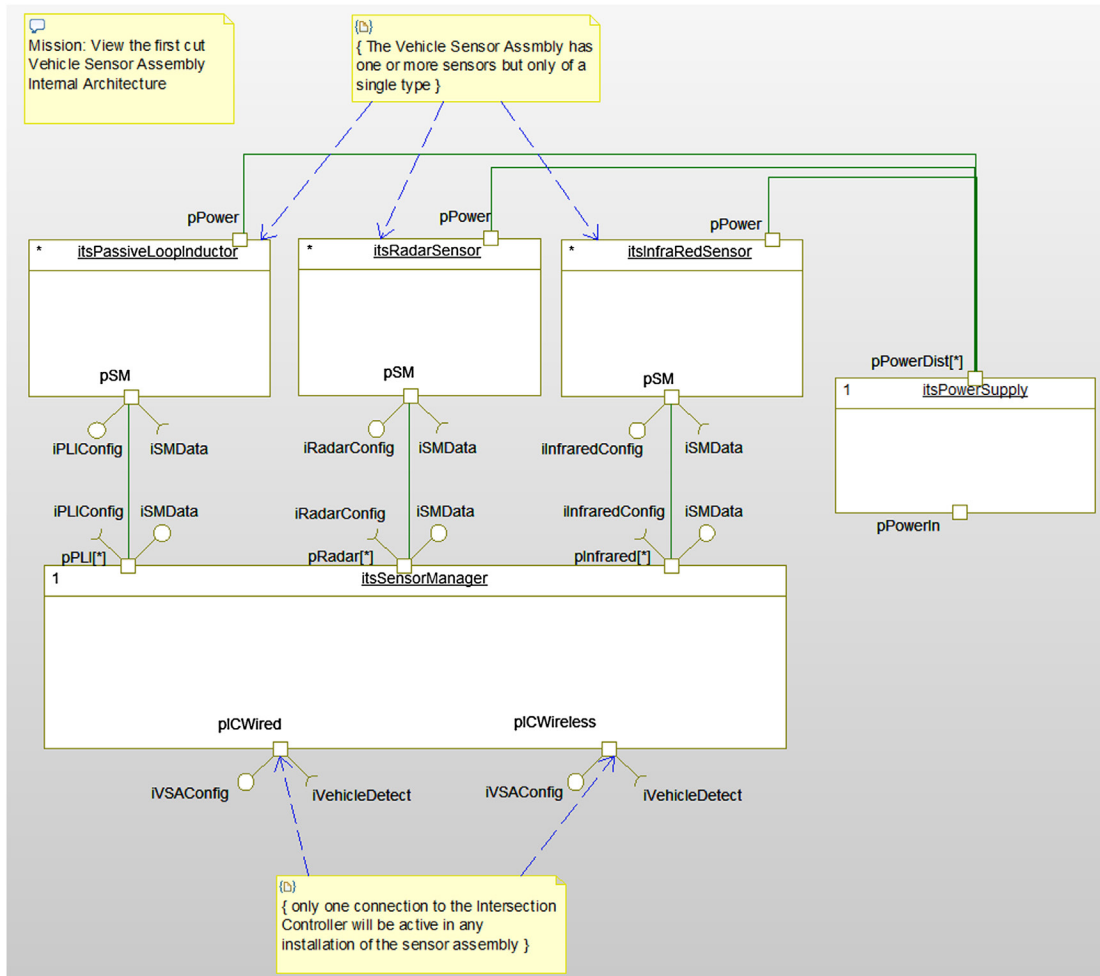


Figure 13.12
Vehicle Sensor Assembly first-cut architecture.

Failure Means and Effects Analysis														
				Pre-action								Postaction		
Service / Function	Failure Mode	Faults	Failure Effects	Likelihood (1=impossible, 10=certain)	Severity (1=no effect, 10=catastrophe)	Detectability (1=certain, 10=no detection)	RPN (= sev * likely * detect)	Existing Control Measures	Recommendations	Responsible	Actions	Likelihood	Severity	RPN
Vehicle detection with passive loop inductor (PLI)	PLI coil unresponsive	Power fault	No power to coil, voltage sensor, or monitor circuit	3	6	3	54	Voltage sensor on incoming power line	Add UPS	George	UPS added to system	2	6	36
		Configuration corrupted	Voltage threshold is wrong leading to incorrect detection	2	6	10	120	None, Can only be detected at extreme values	Redundant threshold with compare	Therese	Software updated to include reliability measures	1	6	60
		Voltage sensor fault	Sensor can't detect induced current in coil	4	6	8	192	None	Built-in test for voltage sensor	Therese	Added BIT to sensor manager for voltage sensors	3	6	144
		Coil breakage	Current cannot flow and therefore cannot detect vehicle	3	6	8	144	None	Thicken coil housing	Matt	Cable house thickened to 2,5mm	2	6	96
	PLI coil too responsive	Ground fault	Ground fault can lead to hypersensitive voltage detector	3	6	8	144	None	Thicken coil housing	Matt	Cable house thickened to 2,5mm	2	6	96
		Configuration corrupted	Voltage threshold is wrong leading to incorrect detection	2	6	10	120	None	Redundant threshold with compare	Therese	Software updated to include reliability measures	1	6	60
Vehicle detection with radar	Radar can't detect	Radar emitter fault	No radar pulse emitted or out of frequency	5	6	9	270	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	162
		Radar receiver fault	Radar pulse cannot be detected at emitted frequency	5	6	8	240	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	144
		Radar processing fault	Processing fails to detect arrival of vehicles	4	6	9	216	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	162
		Power fault	No power, no radar	8	6	3	144	Voltage sensor on incoming power line	Add UPS	George	UPS added to system	4	6	72
		Timing fault	Minor timing problem lead to lack of detection	6	6	9	324	None	Add secondary timer	Doris	Second crystal-based timer added	2	6	108
	Radar false detection	Power fault	Intermittent power can cause a surge, creating a false signal	8	6	1	48	Voltage sensor on incoming power line	Add UPS	George	UPS added to system	4	6	24
		Radar processing fault	Processing detects arrival of vehicles even when not present	4	6	9	216	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	162
		Timing fault	Minor timing problems can lead to hypersensitive detection	6	6	9	324	None	Add secondary timer	Doris	Second crystal-based timer added	2	6	108

Figure 13.13
Roadrunner Vehicle Detection FMEA.

Vehicle detection with Infrared		IR receiver fault	Passive IR cannot detect thermal environment	4	6	8	192	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	144
		IR differential processing fault	Processing cannot properly distinguish between pre- and post-arrival states	3	6	9	162	None	Update operations manual to include monthly test; add test mode	Katie	Manuals updated	3	6	162
		Environmental thermal noise rejection fault	In high temperatures, the thermal signature of the car engine may not be different enough from the background temperature	8	6	9	432	None	Add configuration options for high thermal environments	Therese	Updated software to support thermal configuration options	5	6	270
Vehicle detection independent of sensor tech	Comm problem to sensor manager	Disconnection fault	Disconnect to sensor means detection cannot be seen by sensor manager	3	6	7	126	Connection sensor on comm line	Lifeticks	Therese	Updated comm protocol to have 1 sec lifeticks	2	6	84
	Comm problem to Intersection Controller	Disconnection fault	Vehicle detections cannot be sent to the Intersection controller	3	6	7	126	Connection sensor on comm line	Lifeticks	Therese	Updated comm protocol to have 1 sec lifeticks	2	6	84
	Power fault	Power supply fault	Power distribution interrupted	7	7	7	343	Voltage sensor on outgoing power line	none	n/a	n/a	7	7	343
		External power loss	Cannot process sensor data and send to Intersection Controller	8	7	1	56	Voltage sensor on incoming power line	Add UPS	George	UPS added to system	8	7	56
	Vehicle counting wrong	Memory fault	Corrupted storage of vehicle counts	7	3	10	210	none	Use inverted bit redundant storage	Therese	Software updated to include reliability measures	1	3	30
		SW processing fault	Incorrect processing corrupts memory	4	3	10	120	none	None	n/a	n/a	4	3	120

Figure 13.13 Continued

Failure Means and Effect Analysis													
			Pre-action								Post-action		
Service / Function	Failure Mode	Failure Effects	Likelihood (1=impossible, 10=certain)	Severity (1=no effect, 10=catastrophe)	Detectability (1=certain, 10=no detection)	RPN (= sev * likely * detect)	Existing Control Measures	Recommendations	Responsible	Actions	Severity	Likelihood	RPN
Control Aircraft Attitude	Can't control yaw (rudder), roll (ailerons), or pitch (elevators)	Hydraulic fault	4	8	10	320	Control surface sensor	Add pressure sensor in hydraulic system	Jamie	Add pressure sensors on each independent hydraulic pressure line	8	2	160
		Power fault to electric motors	5	8	6	240	Current sensor on power output	Multiple independent power sources	Susan	Add emergency backup power supply	8	2	96
		Motor fault	4	8	10	320	Control surface sensor	Add torque sensor	Susan	Pair motors	8	2	160
		Control surface stuck	5	9	10	450	Control surface sensor	Multiple control surfaces for rudder, ailerons, and elevators	Jamie	Use 2+ control surfaces per dimension	9	4	360
		Control surface broken	3	9	10	270	None	Multiple control surfaces for rudder, ailerons, and elevators	Jamie	Use 2+ control surfaces per dimension	9	2	180
		Command fault	6	6	4	144	Messages must be ACKed	Multiple independent control lines; message CRC	Mary	Update internal aircraft comm protocol; run parallel buses	6	3	72
	Command fault	Comm fault to ground system	8	6	4	192	Messages must be ACKed	message CRC	Mary	Update external aircraft comm protocol	6	3	72
	Computation fault	Control surface sensor fault	7	5	6	210	Aircraft attitude sensor	Lifeticks; sensor BIT	Susan	Add device lifeticks with state information; boot up BIT	5	3	90
		Aircraft attitude sensor fault	7	9	2	126	None	Lifeticks; sensor BIT	Susan	Add device lifeticks with state information; boot up BIT	9	3	54
		Comm fault on aircraft	6	5	4	120	Messages must be ACKed	Multiple independent control lines; message CRC	Mary	Update internal aircraft comm protocol; run parallel buses	5	3	60
		Kinematic model computation error	5	7	7	245	None	Backward computation to reveal computation error; multiple alternative computational paths	Peter	Add heterogeneous redundancy computational channels with three-way vote	7	2	98

Figure 13.14
CUAV Control Aircraft Attitude FMEA.

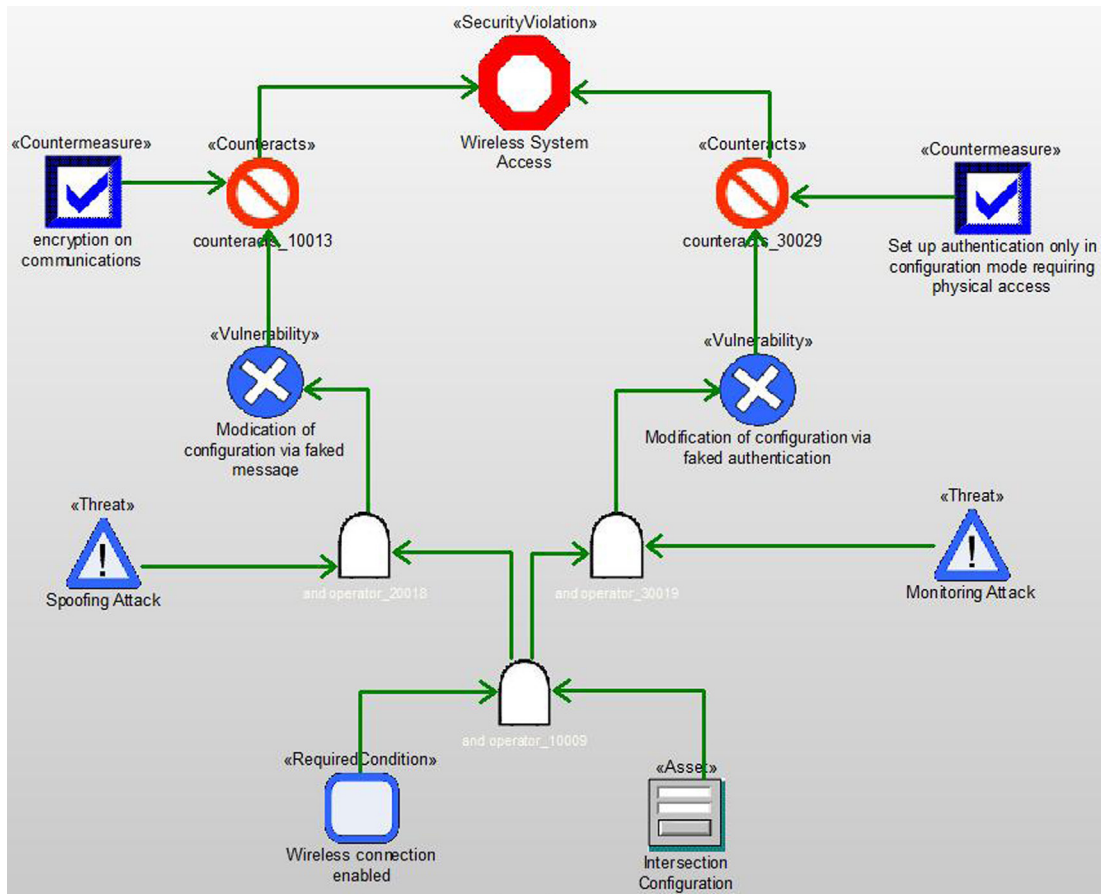


Figure 13.15

Roadrunner security analysis for wireless communication.

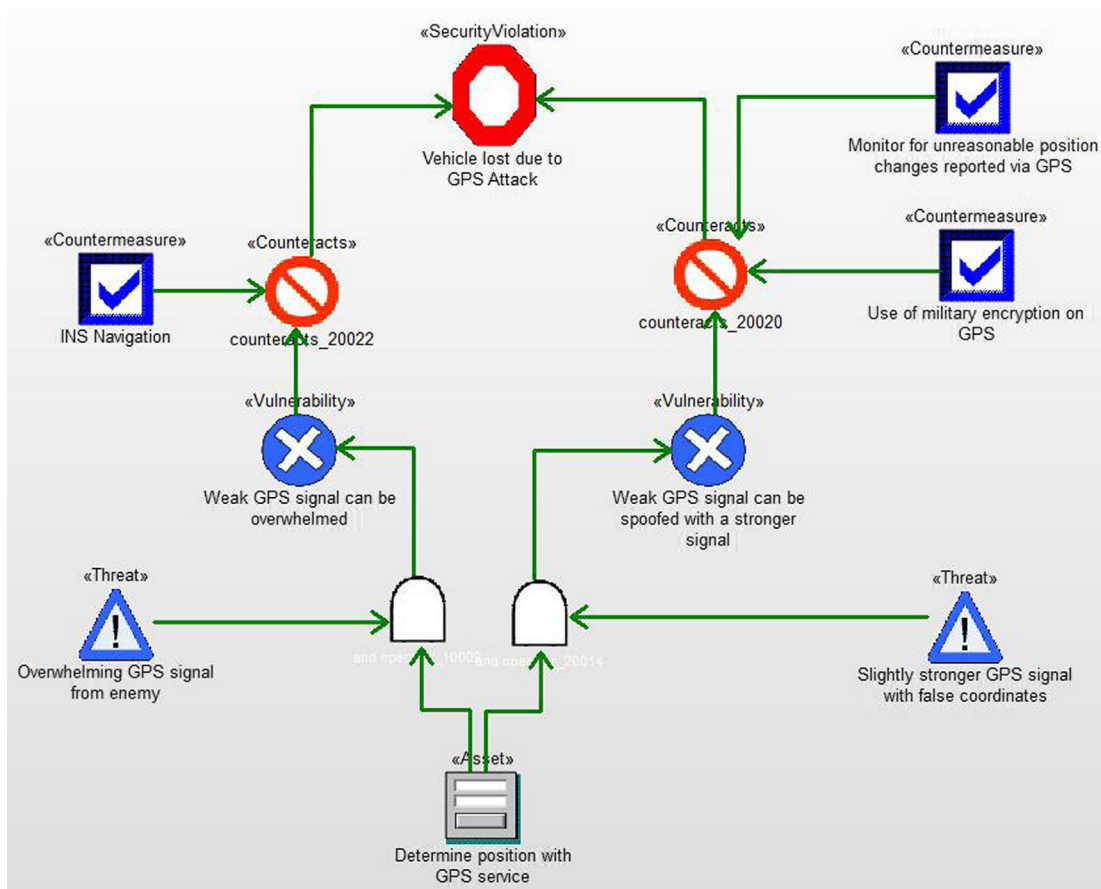


Figure 13.16
Coyote UAV GPS security analysis.

High-Fidelity Modeling: Answers

14.1 Answer 7.1: Apply Nouns and Causal Agents Strategies

The strategy to apply is to underline the nouns and noun phrases. Below, the strategy has been applied.

The Vehicle Detector

Three types of Vehicle Detectors shall be supported: subsurface passive loop inductors (SPLIs), above-surface infrared sensors (ASIs), and above-surface radars (ASRs).

Subsurface detectors shall use a wired interface to communicate with the controller, while ASIs and ASRs shall support both wired and secure wireless communication. All vehicle detectors shall be able to perform vehicle counting.

In addition, ASIs and ASRs shall be able to receive directional transmissions from priority vehicle and emergency vehicle transmitters. The maximum range of such reception shall be no less than 250 feet and no more than 1000 feet.

Figure A-3 shows the relevant measures for both ASI and ASR detectors. When a vehicle enters the detection area (shown as the shaded area in the figure), the detector shall report the presence of a vehicle. Separate detectors are used for each lane in each direction.

In Table 14.1, the noun phrases are categorized.

The second strategy is to identify the causal agents. In this case, the agents that ultimately cause events to enter into the system for the Detect Vehicle use case are the Vehicle actors (for a normal detection), and the Priority and Emergency Vehicle actors for the other kinds of vehicle detections. In our system structure, it will be sensors that identify these vehicles. For the normal vehicles, we have passive loop inductors that detect the metallic content of the vehicle and we have infrared and radar detectors that respond to heat and movement, respectively, within the range of the sensor. The specification also states that the infrared and radar detectors are connected via wireless links while the passive loop inductor is connected via a wired link. For the priority and emergency vehicles, the detection is done by receiving a directional wireless (IR, as it happens) transmission that indicates the approaching vehicle. The information about the detection will be used in different operational modes, but that is outside the scope of this strategy. However, this strategy has identified several kinds of vehicle detectors (passive loop inductor, infrared, radar, and wireless reception from a

Table 14.1 Roadrunner Detect Vehicle Nouns.

Noun Phrase	Element Type	Element Name
Types of vehicle detectors	Class	VehicleDetector
1	Class	PassiveLoopInductor
SPLI	Class	PassiveLoopInductor
above-surface infrared sensors	Class	InfraredVehicleDetector
ASIs	Class	InfraredVehicleDetector
Above-surface radars	Class	RadarDetector
ASRs	Class	RadarDetector
Subsurface detectors	Class	PassiveLoopInductor
Wired interface	Class	WiredNetworkInterface
Wired and secure wireless communication	Unknown	Unknown
Vehicle detectors	Class	VehicleDetector
Controller	Class	Intersection Controller
Detection Area Width	UN	Unknowns
Shaded area	UN	Unknown
Directional transmissions	Class	Message
Priority vehicle and emergency vehicle transmitters	Actor	PriorityVehicleTransmitter EmergencyVehicleTransmitter
Maximum range	UN*	Useful for testing but not explicitly represented in collaboration
Reception	Class	Message
250 feet	UN	Useful for testing but not explicitly represented in collaboration
1000 feet	UN	Useful for testing but not explicitly represented in collaboration
Figure A-3	UN	
Detection Area Width	Attribute?	VehicleDetector.areaWidth
Detection Area Length	Attribute?	VehicleDetector.areaLength
Detection Area Distance	Attribute?	VehicleDetector.areaDistance
Relevant measures	UN	
ASI and ASR detectors	Class	AboveSurfaceVehicleDetector
Vehicle	Class	Vehicle
Presence of a vehicle	Event	evVehicleDetect
Separate detectors	Class	VehicleDetector
Each lane	Attribute	VehicleDetector.lane
Each direction	UN	

*UN stands for Uninteresting Noun; in other words, something we're not going to explicitly model.

transponder or transmitter mounted on the vehicle). It has also identified wired and wireless interfaces to get that information to the internal system.

The results of applying these two object identification strategies are shown in [Figure 14.1](#).

For the second use case, Fixed Cycle Mode, we'll employ the same strategies.

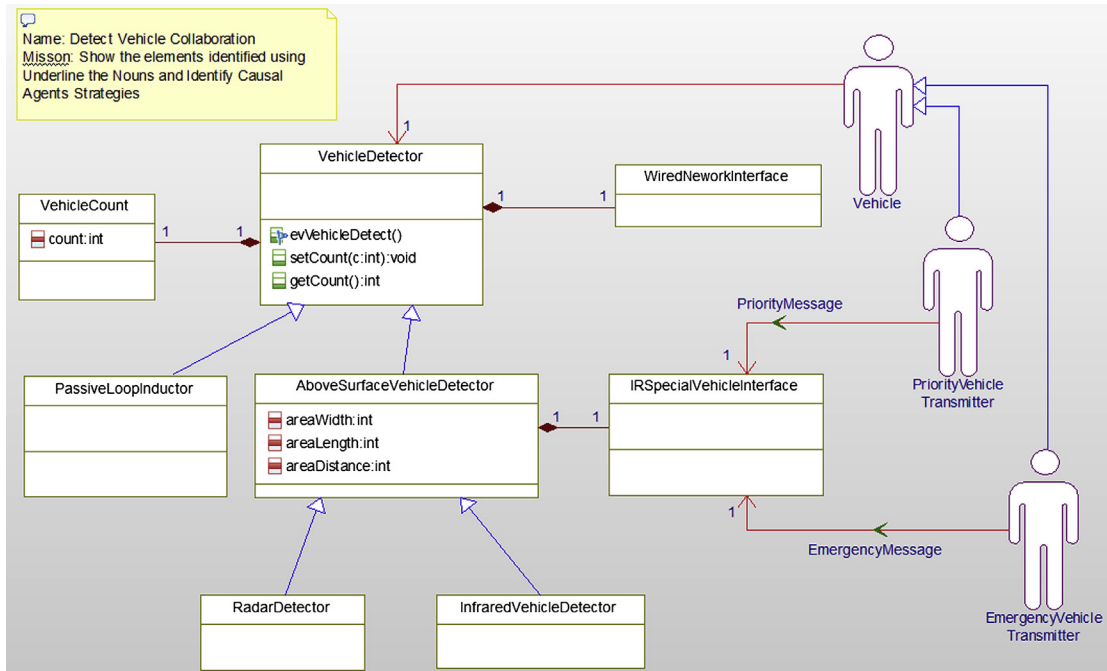


Figure 14.1
 Vehicle Detect collaboration strategy 1.

Mode 2: Fixed Cycle Time

Mode 2 is the most common operational mode. In this mode, the lanes cycle GREEN-YELLOW-RED in opposite sequences with fixed intervals. The system shall ensure that if any traffic light is non-RED, then all the lights for cross traffic shall be RED and pedestrian lights (if any) shall be set to DON'T WALK. Note that the turn lane times and/or pedestrian times are only valid in this mode if (1) the turn lane and/or pedestrian parameter is set TRUE in the RIC system parameters and (2) if a signal from the appropriate detector determines the existence of waiting traffic for the turn or pedestrian light.

The durations of the light times shall be independently adjustable by setting the appropriate parameters (see below). Note that in the table [Table 14.2], the values in parentheses are defaults.

The default values depend on the system configuration.

The values in Table 14.3 are true for each direction, independently. Thus, if the primary road has a car waiting in its turn lane and a pedestrian walking, but the secondary road has neither, then the following timing diagram represents the cycle times for simultaneous turn lane mode (i.e., the turn lanes in both directions for a road turn together and the straight traffic doesn't begin until the turn lanes have cycled to RED).

Table 14.2 Mode 2 Parameters.

Parameter	Value Type	Description
Reset Parameters	FALSE, TRUE	(FALSE) Sets all the <u>parameters</u> for <u>Mode 2</u> to <u>defaults</u>
Primary Green Time (PG2)	10 to 180 seconds	(30) <u>Length of time</u> the <u>primary GREEN light</u> is on
Primary Yellow Time (PY2)	2 to 10 seconds	(5) <u>Length of time</u> the <u>primary YELLOW light</u> is on
Primary Red Delay Time (PR2)	0 to 5 seconds	(0) <u>Length of time</u> between when <u>primary RED light</u> is turned on and the <u>secondary GREEN light</u> is activated
Primary Walk Time (PW2)	0 to 60 seconds	(20) <u>Length of time</u> the <u>primary WALK light</u> is on when the <u>primary GREEN light</u> is activated
Primary Warn Time (PA2)	0 to 30 seconds	(10) <u>Length of time</u> the <u>primary FLASHING DON'T WALK light</u> is on after the <u>WALK light</u> has been on
Primary Turn Green Time (PT2)	0 to 90 seconds	(20) <u>Length of time</u> the <u>primary turn light</u> is GREEN. Note: only valid when the <u>Primary Turn Light parameter</u> is TRUE.
Primary Turn Yellow Time (PZ2)	0 to 10 seconds	(5) <u>Length of time</u> the <u>primary turn light</u> is YELLOW. Note: only valid when the <u>Primary Turn Light parameter</u> is TRUE.

Table 14.3 Default Cycle Times for Mode 2.

Turn Lane	Ped Signal	Green	Yellow	Red	Walk	Don't Walk	Turn Green	Turn Yellow
F	F	30	5	0	0	0	0	0
T	F	50	5	0	0	0	15	5
F	T	50	5	0	15	5	0	0
T	T	50	5	0	15	5	15	5

As before, let's categorize the noun phrases (see [Table 14.4](#)).

The causal agent strategy for the Fixed Cycle Mode identifies the elements that cause change: these are time (that is, the internal elements themselves decide when to change the light based on time; these elements include the `VehicleLights` and `PedestrianLights`), vehicles (in the turn lane only in this case), and pedestrians.

By the way, note that in Fixed Cycle Mode, we still need to have the sensors present for the vehicle turn lanes and for the pedestrian, although not for the through traffic. At this point, we are assuming that the turn light controls can be all structurally and behaviorally identical but their role will be known to the `TrafficController`. Later, we can validate this notion when we add the state behavior and execute the model.

[Figure 14.2](#) shows the class diagram for the elements that support the collaboration for Fixed Cycle Mode. Each of the classes shown will be instantiated as a part in the subsystems indicated in the attached comments. We can get this collaboration working and then move the

Table 14.4 Roadrunner Fixed Cycle Mode Nouns.

Noun Phrase	Element Type	Element Name
Mode 2	State	Traffic Controller
Operational mode	State	Traffic Controller
Mode	State	Traffic Controller
Lanes	Class	Vehicle Light
Sequences	Set of states	In various statecharts
Fixed intervals	Timeout transitions	In various statecharts
System	Object	System object
Traffic light	Class	Vehicle Light
Lights	Class	Vehicle Light
		Pedestrian Light
Cross traffic	Actor	Vehicle
Pedestrian lights	Class	Pedestrian Light
Turn lane times	Timeout transition	In statechart of Vehicle Light
Pedestrian times	Timeout transition	In statechart of Pedestrian Light
Turn lane	Object	Instance of Vehicle Detector
		Instance of Vehicle Light
Pedestrian parameter	Attribute	Intersection Configuration
RIC system parameters	Class	Intersection Configuration
Signal	Event	evVehicleDetect
Appropriate detector	Object	Vehicle Detector
Waiting traffic	State	Vehicle Detector
Turn or pedestrian light	Object	Vehicle Light
		Pedestrian Light
Durations	Parameter for timeout transitions	Various statecharts
Light times	Parameter for timeout transitions	Various statecharts
Appropriate parameters	Attributes	Intersection Configuration
Table	UN	
Values	UN	
Defaults	UN	
Mode 2 parameters	Class	Intersection Configuration
Reset Parameters	Operation	Intersection Configuration
Parameter	Attribute	Intersection Configuration
Value Type	UN	
Description	UN	
Primary Green Time	Attribute	Intersection Configuration
xx to yyy seconds	Attributes	Intersection Configuration
Length of time	UN	
Primary green light	Object	Instance of Vehicle Light
Primary Yellow Time	Attribute	
Primary yellow light	Object	Instance of Vehicle Light
Primary Red Delay Time	Attribute	
Primary red light	Object	Instance of Vehicle Light
Primary Walk Time	Attribute	
Secondary green light	Object	Instance of Vehicle Light
Primary WALK light	Object	Instance of Pedestrian Light
Primary Warn Time	Attribute	Intersection Configuration

Continued

Table 14.4 Roadrunner Fixed Cycle Mode Nouns.—cont'd

Noun Phrase	Element Type	Element Name
Primary FLASHING DON'T WALK light	State	Of an instance of Pedestrian Light
WALK light	Object	Instance of Pedestrian Light
Primary Turn Green Time	Attribute	Intersection Configuration
Primary turn light	Object	Instance of Vehicle Light
Primary Turn Light parameter	Attribute	Intersection Configuration
Primary Turn Yellow Time	Attribute	Intersection Configuration
Default values	Attributes	Various classes
System configuration	Class	Intersection Configuration
Default Cycle Times	Attributes	Intersection Configuration
Turn Lane	Object	Instance of Vehicle Light
Ped Signal	Object	Instance of Pedestrian Light
Values	UN	
Table 14.3	UN	
Direction	UN	
Primary road	Object	Instance of Vehicle Light Instance of Vehicle Detector
Car	Actor	Vehicle
Turn lane	Object	Instance of Vehicle Light Instance of Vehicle Detector
Pedestrian	Actor	Pedestrian
Secondary road	Object	Instance of Vehicle Light Instance of Vehicle Detector
Timing diagram	UN	
Cycle times	Attributes	Various statecharts
Simultaneous turn lane mode	State	Traffic Controller
Road	UN	
Straight traffic	Object	Instance of Vehicle Light Instance of Vehicle Detector

elements to their respective subsystems or we can put them there first and then get the collaboration working. I tend to prefer the former approach; get the collaboration working properly in isolation, move the elements to their proper place in the model, and then formally unit test the in-place collaboration to demonstrate its correctness.

The UAV problem is far larger, but the strategy works because the object analysis is performed on use cases of the subsystems rather than the system as a whole. However, it can still be a complex task to model each subsystem-level use case for a complicated system such as the Coyote UAV. In the sample problem statement below, we have only underlined the nouns relevant to the Acquire Image use case.

The Unmanned Aerial Vehicle (UAV)

The Coyote UAV is meant to be a multipurpose reusable UAV with multimission capability. It is meant to operate at altitude of up to 30,000 feet with ground speeds of up to 100

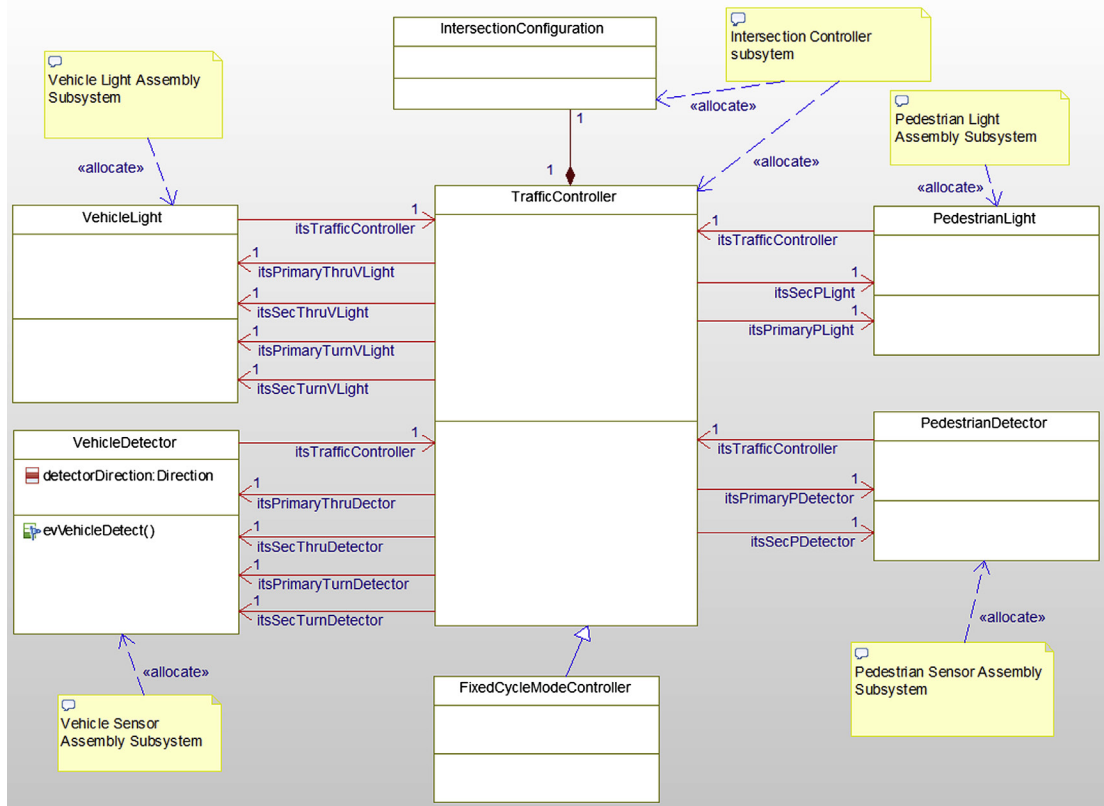


Figure 14.2
Fixed Cycle Mode collaboration strategy 1.

knots (cruise) and 150 knots (dash) and carry a payload of up to 450 lbs for durations in excess of 24 hours. The Coyote is meant to fly unimpeded in low-visibility environments while carrying either reconnaissance or attack payloads. While controllable from the ground station CMPCS, it is also capable of flying complex flight plans with specific operational goals of systematic area search, ground route (road-based) search, and orbit surveillance of point targets. Coupled with manned control from the ground, the Coyote provides sustained 24-hour flight with real-time visual, infrared, or radar telemetry, with target recognition preprocessing. Communications are jam-resistant, although need not be anti-jamming in a high ECM environment. Control commands shall be encrypted while telemetry data can be compressed but unprotected. Telemetry rates for visual telemetry support 30 frames per second (fps) at 640×400 resolution. Range of flight is meant to be fully supported within line of sight (LOS) range but since the Coyote also has the ability to be passed among different CMPCSs, its range is considerably greater than LOS. For navigation, the Coyote has on-board Global Positioning System (GPS) based navigation as well as being directly controllable from the ground station.

Unlike many smaller UAVs, the Coyote does not require specialized launch and recovery vehicles. It can use a short runway for either automated or remote-controlled takeoff and landing.

The Coyote Mission Planning and Control System (CMPCS)

Mobile CMPCS with capability to control up to four UAVs with a manned control station per UAV that fits into a smaller towable trailer. Each control station consists of two manned substations – one for controlling the CUAV and one for monitoring and controlling payloads. If desired, both functions can be slaved together into a single control substation. Control of the aircraft shall consist of transferring navigational commands which may be simple (set altitude, speed, direction), operational (fly to coordinate set, orbit point, execute search pattern, etc.), planned (upload multisegment flight plan) or remote controlled with a joystick interface. Stable flight mechanics shall be managed by the aircraft itself but this can be disabled for remotely controlled flight.

The CMPCS displays real-time reconnaissance data as well as maintaining continuous recording and replay capability for up to 96 hours of operation for four separate CUAVs. In addition, with attack payloads, the Coyote can carry up to four Hellfire missiles with fire-and-forget navigation systems.

The Unmanned Aerial Vehicle (UAV)

...

Mission Modes

Beyond flight modes, the CUAV shall be designed for highly flexible mission parameters. Normal mission modes include:

- *Preplanned reconnaissance*
- *Remote controlled reconnaissance*
- *Area search*
- *Route search*
- *Orbit point target*
- *Attack*

A mission can consist of any number of sequential submissions each operating in a different mission mode, depending on the current payload.

The Coyote Mission Planning and Control System (CMPCS)

The CMPCS is housed in a 30×8×8 triple-axis trailer that contains stations for pilot and payload operations, mission planning, data exploitation, communications, and SAR viewing. The CMPCS connects to multiple directional antennae for communication with the CUAVs. All mission data is recorded at the CMPCS since the CUAV has no on-board recording capability. The CMPCS has a UPS that can operate at full load for up to four hours in addition to use commercial power or power generators.

A single CMPCS can control up to four CUAVs in flight with one station per CUAV. Each CUAV control station provides both pilot and payload operations with separate control substations, although both functions can be slaved to a single substation for low vigilance use.

For the reconnaissance payloads, the CMPCS shall provide enhanced automated target recognition (ATR) capability for all surveillance types – optical, infrared, and radar. While the CUAV has a rudimentary capability, the CMPCS provides much more complete support for the quick identification of high value targets in the battlefield. This capability is specifically design to identify mobile and time-limited targets that may only be exposed for brief periods of time before they go back into hiding. The system is expected to provide high clutter rejection and a low false-positive error rate. The ATR shall able to identify and track up to 20 targets within the surveillance area, with likely identification and probability assessments for each. In addition to the ATR, the payload operator can add targets visually identified from reconnaissance data or gathered from other sources. The battlefield view can be transmitted over links to remote command staff for tactical and strategic assessment.

Image Acquisition and Processing

Images may be acquired from all three sensor platforms – optical, FLIR (forward looking infrared), and SAR (synthetic aperture radar). Optical and FLIR are passive systems using emitted energy from terrain or targets to gather information. The SAR is an active sensor, painting relatively stationary targets with energy (in the microwave range) and using the reflection of these pulses to determine reflectivity and altitude. The optical and FLIR resolution for single images may be as high as 1900×1600 resolution while real-time video for all sensor platforms is limited to 640×480 resolution at a rate of 30 fps. Streaming imagery shall be sent with enough redundancy so that complete loss of random frames shall not affect the quality of other frames. The sensor platforms may be focused at any range from 10 meters to infinite and may be zoomed up to 100× actual. The sensor platforms shall be mounted on a gimbaled assembly so that the system can be aimed without affecting the attitude of the CUAV. The FLIR includes a laser range finder to determine target range so that it can be used in fire control applications. The SAR shall emit a series of pulses meant to emulate the behavior of much larger physical aperture antenna; the images from the SAR are a combination of timed radar surface reflections to be combined into a SAR image in the ground station using Fourier transforms. Thus, a single SAR image results from a set of images each resulting from a single radar pulse from the SAR platform, but combined in the ground station. Aiming the SAR is done through the use of Doppler sharpening, limiting the amount of information that must be transmitted to the ground station to construct the SAR image. The use of two pulse emitters in the SAR platforms allows the interference patterns to be constructed providing altitude determination as well as radar reflectivity data.

Images may be compressed using lossy or nonlossy methods to minimize communication bandwidth requirements. The JPEG 2000 compression standard shall be used; for streaming video the associated MJPEG standard shall be used. The compression may be set

dynamically by the payload operator to be 0% to 80% with the default setting to be nonlossy 50% compression. The imaging system is required to achieve the desired compression only within 20% of the requested due to the variances in the image contents. The selection of lossy or nonlossy compression shall be determined automatically by the imaging system, switching to lossy compression only when the desired compression rate cannot be achieved using lossless compression.

Table 14.5 shows the underlined noun phrases in the problem statement. We have tried to only underline and include noun phrases that are related to image acquisition and processing. However, this is one of the difficulties in applying this strategy in large-scale systems. Unless the textual description/specification is extraordinarily well organized, it is difficult to just underline the noun phrases for the use case of immediate concern, and the underlining of every noun phrase leads to hundreds or thousands of candidate classes that must be sorted and organized.

In this problem, the Acquire Image use case is a capability of the Reconnaissance Management subsystem. A part of this analysis might spill over into other subsystems (notably Targeting, Airborne Datalink, and the reconnaissance payloads), but primarily we'll be focused on two things:

1. Identifying and linking together the objects, classes, attributes, and algorithms within the Reconnaissance Management subsystem
2. Characterizing the interfaces of the Reconnaissance Management subsystem to its peer subsystems

To that end, it is necessary to decide how to split the functionality between the payloads and the other subsystems. In general, the functional allocation will be thus:

- The recon platforms are responsible for the production of a single uncompressed and unprocessed image at a time.
- The Reconnaissance Management subsystem (RMS) is responsible for processing and compressing recon images; managing the streaming of images; commanding, aiming, and focusing the recon platforms; and identifying targets (as directed by the Targeting subsystem).
- The RMS is responsible for commanding configuration of the recon platforms.
- The Targeting subsystem is responsible for target selection, setting of the target identification parameters, and tracking targets.

The created class diagram is shown in Figure 14.3. You'll note that some of the identified objects in the previous table are not represented in the class diagram. This is because they are allocated to other subsystems; for example, the gimbal is a part of the payload not part of the Reconnaissance Management subsystem. Also, there are a few classes in the diagram that do not appear in the table. They were added to hold operations or attributes, or because

Table 14.5 CUAV Acquire Image Nouns.

Noun Phrase	Element Type	Element Name
Real-time visual, infrared or radar telemetry	Class	Image
Target recognition preprocessing	Algorithm	PreprocessImage()
Communications	UN	
Control commands	Class	Command
Telemetry data	Class	Image
Telemetry rates	Constraint	Telemetry Rates
Visual telemetry	Class	VisualImage
Line of sight (LOS) range	UN	
Coyote	Class	CUAV
CMPCSs	Class	GroundStation
Reconnaissance payloads	Class	ReconAssembly
Enhanced automated target recognition (ATR)	Algorithm	identifyTarget()
capability	Class	Target
Optical	Class	VideoAssembly
Infrared	Class	FLIR_Assembly
Radar	Class	Radar_Assembly
High value targets	Class	Target
Battlefield	UN	
Capability	UN	
Mobile and time-limited targets	Class	Target
System	Class	CUAV_Project
False-positive error rate	UN	
ATR	Algorithm	identifyTarget()
	Class	Target
Targets	Class	Target
Surveillance area	UN	
Payload operator	Actor	MissionSpecialist
Targets	Class	Target
Reconnaissance data	Class	Image
Battlefield view	Class	Image
Links	UN	
Images	Class	Image
Sensor platforms	Class	ReconAssembly
Optical	Class	Video_Assembly
FLIR	Class	FLIR_Assembly
Passive systems	UN	
Emitted energy	UN	
Terrain	Class	TerrainFeature
Targets	Class	Target
Information	Class	Image
SAR	Class	Radar_Assembly
Active sensor	UN	
Targets	Class	Target
Energy	UN	
Microwave range	Attribute	Frequency
Reflection	UN	
Pulses	Attribute	PulseRate PulseLength

Continued

Table 14.5 CUAV Acquire Image Nouns.—cont'd

Noun Phrase	Element Type	Element Name
Reflectivity	Attribute	Color
Altitude	Attribute	Range
Optical and FLIR resolution	Attribute	resolution
Real-time video	Class	ImageStream
Sensor platforms	Class	ReconAssembly
Resolution	Attribute	X_resolution
		Y_resolution
Rate	Attribute	FrameRate
Streaming imagery	Class	ImageStream
Redundancy	UN	
Frames	Class	Image
Quality	UN	
Range	Attribute	Range
Gimbaled assembly	Class	Gimbal
Attitude	UN	
CUAV	Class	CUAV
Laser range finder	Class	LaserRangeFinder
Target range	Attribute	Range
Fire control applications	UN	
SAR	Class	Radar_Assembly
Series of pulses	Class	Pulse
Physical aperture antenna	UN	
Images	Class	Image
Combination of timed radar surface reflections	Class	Radar_Image
SAR image	Class	Radar_Image
Ground station	Class	GroundStation
Fourier transforms	Algorithm	ComputeSARImage()
Single SAR Image	Class	Radar_Image
Set of images	Class	Radar_Image
Single radar pulse	Class	Pulse
SAR platform	Class	Radar_Assembly
Information	UN	
Pulse emitters	Class	PulseEmitter
Interference patterns	Class	Radar_image
Altitude determination	Attribute	Altitude
Radar reflectivity data	Attribute	Color
Lossy or nonlossy methods	Algorithm	CompressImage()
Communication bandwidth requirements	UN	
JPEG 2000 compression standard	UN	
MJP2 standard	UN	
Compression	Algorithm	CompressImage()
Payload operator	Actor	PayloadOperator
Default setting	Attribute	DefaultCompression
Imaging system	Class	CUAV
Desired compression rate	Attribute	CompressionRate
Variances in the image contents	UN	
Selection	Attribute	IsLossy
Lossless compression	Attribute	IsLossy

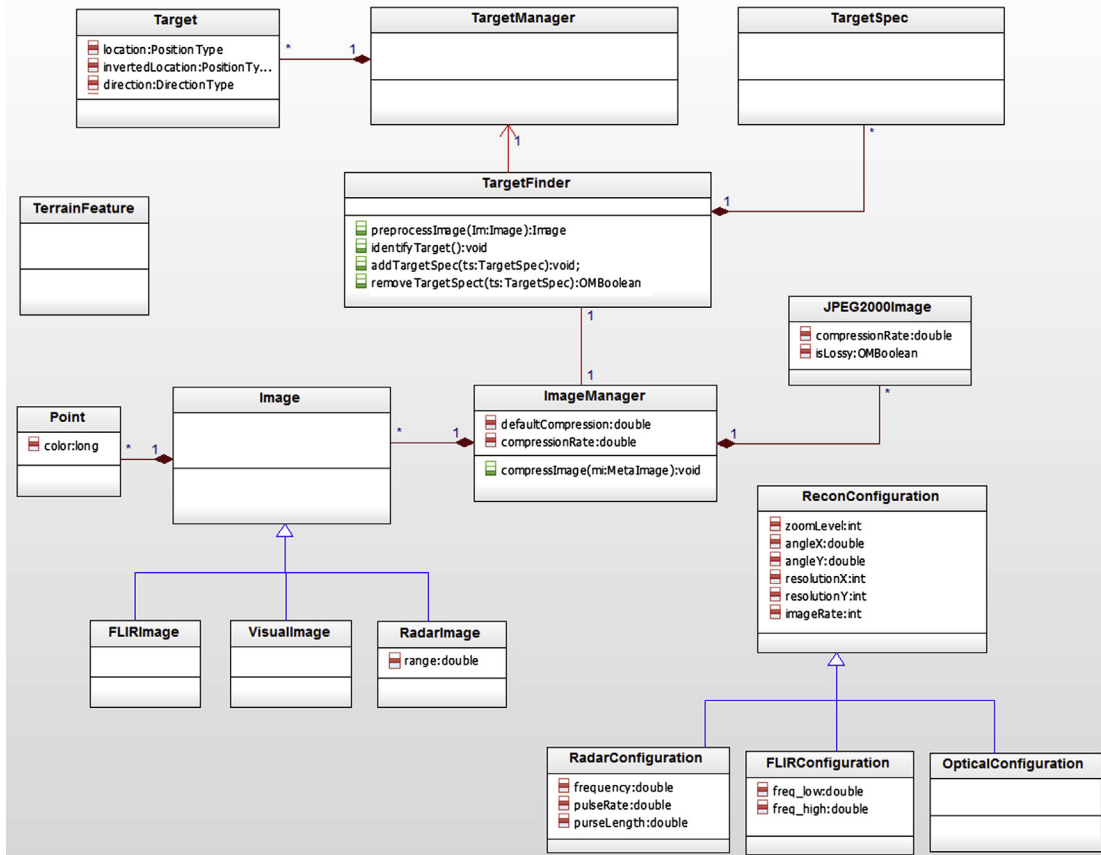


Figure 14.3

Acquire Image collaboration strategy 1.

it seemed likely that they would be needed based on the other classes and how they must interact. The model is certainly not complete, but this is typical when such a strategy is applied. Multiple strategies must be applied to find all the classes and their features to fully realize a use case.

The second strategy, “Causal Agents,” looks for objects that initiate actions. Many of the actions are ultimately caused by the payload operator, such as moving the gimbal, aiming and focusing, setting the resolution, and so on. There must be some facility in the subsystem that accepts and processes those commands. Once initialized, the recon platforms themselves produce images that they send to the RMS. Lastly, the Targeting subsystem specifies the nature of the targets to look for and the RMS must receive and respond to such commands as well. Figure 14.4 shows the classes identified using the Causal Agent strategy. Some operations were added to facilitate the roles of the classes in causing the appropriate actions to occur.

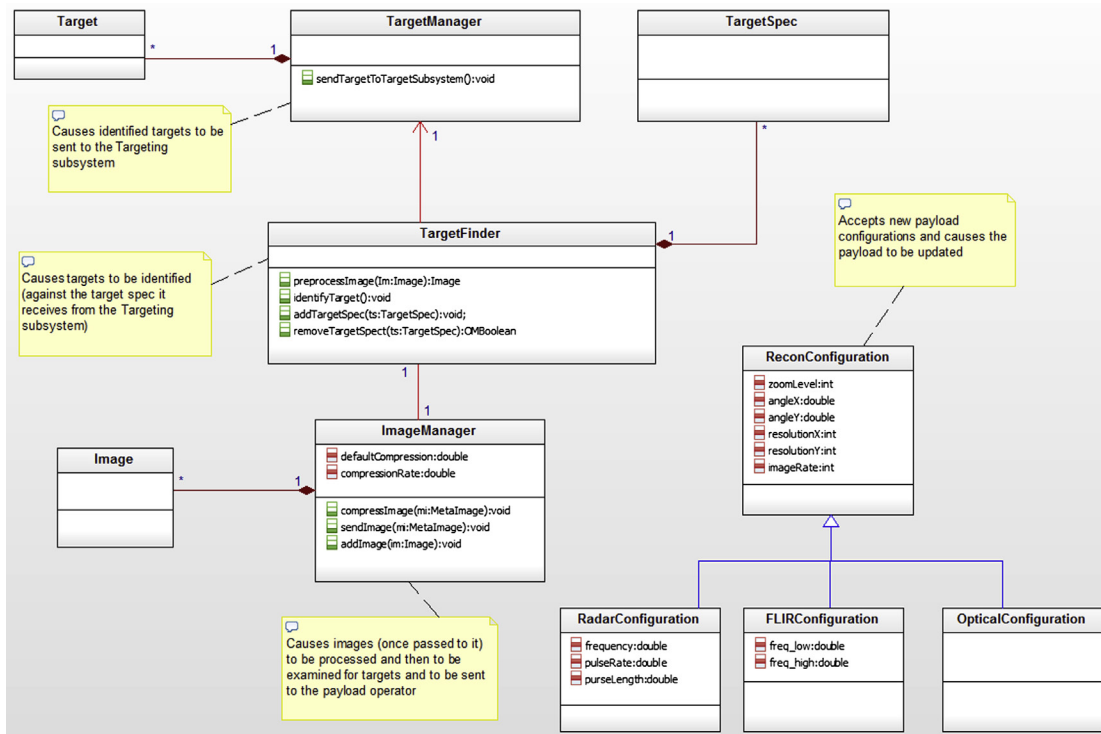


Figure 14.4
Acquire Image collaboration strategy 2.

14.2 Answer 7.2: Apply Services and Messages Strategies

These strategies are all about using the interfaces between the system and its actors or between elements inside the system to identify objects. In the case of a large-scale and abstract service, the service must be decomposed into subservices, and possibly sub-subservices. Ultimately, whether or not services are decomposed, the identified services *must* be provided by the objects within in the system. The purpose of this strategy is to use this fact to identify objects inside the system and then allocate the services to those objects. Since services have at least one caller and one provider, the strategy will also identify associations between the objects because in the majority of cases, to invoke a service on a server object requires an association from a client object.

The strategies work in a different way as well. First of all, most of these services will require data to be passed, either as input, output, or bidirectional parameters, or as return values. Those parameters must come from *somewhere* and many of them are likely to be instances of classes or application-specific data types (which also need identification).

In the first half of this section, we'll look at the same two Roadrunner use cases – Detect Vehicle and Fixed Cycle Mode. In the first case, clearly the messages have to do with a vehicle arriving. The specification states that there are three ways to detect vehicles – passive loop induction, laser range finding, and changes in the infrared field. While it is possible to maintain the distinction between how the detection occurred, we may assume that for the purpose of detecting an oncoming vehicle, *how* the detection occurs isn't as important as *that* it occurred and *where* (as in which traffic lane) it occurred. Thus, we probably have a service such as

```
vehicleDetected(d:Direction)
```

where the type `Direction` in this case may be an enumerated type with values of (`PRIMARY`, `PRIMARY_TURN`, `SECONDARY`, and `SECONDARY_TURN`).^{1,2} The two considerations for this service (in terms of using it as a strategy to identify objects) is *who knows* that a vehicle is detected and can therefore invoke the services, and *who cares* that a vehicle has been detected and therefore provides the service. Obviously some (or possibly a set) of objects are keeping track of whether or not we should turn lights green to permit traffic to flow, depending, of course, on operational mode. The traffic lights control the traffic and so maybe they should be sent the message – but which one? All of them? Or should all the traffic lights be very simple and a centralized controller do all the work? There are clearly a number of distinct good solutions to this problem.

Following this logic, we arrive at a class diagram similar to [Figure 14.5](#).

The other use case under consideration is Fixed Cycle Mode. In this mode, the behavior of the vehicle lights is driven by time-based events for the most part. However, it is important that the behavior of the different lights be coordinated or *bad things* are likely to occur. Following the principle of “allocate responsibilities to elements that have knowledge and context necessary to execute them,” the `FixedCycleModeController` subclass of `TrafficController` will be responsible for driving the light states, while the lights will themselves be responsible for turning on the right lights, blinking, etc.

The `FixedCycleModeController` will receive events from the turn lane detectors but must be able to distinguish between them. To this end, they pass a parameter `dir:DIRECTION_TYPE`, which has values such as `PRIMARY_THRU_DIR`, `SEC_THRU_DIR`, `PRIMARY_TURN_DIR`, and `SEC_TURN_DIR`. This will allow the controller to distinguish which event is being sent and at the same time permits the vehicle detectors to be identical in design (i.e., they don't need to know at design time that they are being used for primary or secondary roads). During system initialization, the controller will tell the detector which direction it represents by passing the

¹ This enumerated type obviously wouldn't work around the Arc de Triomphe, but it will for our problem case.

² There is another way to detect vehicles using transponders for emergency and priority vehicles, however there are other use cases that deal with that functionality.

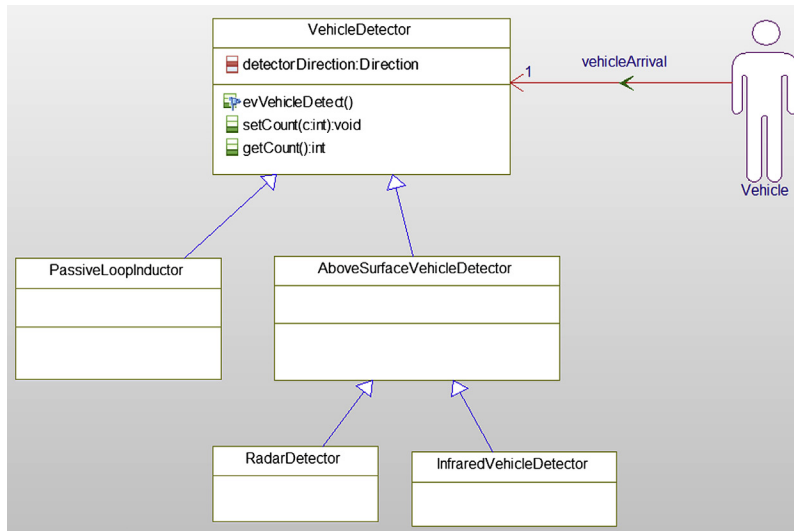


Figure 14.5
Detect Vehicle strategy 3: services.

appropriate value of this type. Similarly, the pedestrian detector also sends an event `evPDetect` with a parameter of this type so that the controller can properly identify which pedestrian lane is requesting a WALK light. These considerations allow us to refine our object model a bit more, as shown in [Figure 14.6](#).

Exactly *how* the controller will use the information behaviorally will be dealt with later in this chapter.

Let us now turn our attention to the UAV. There are many services associated with Acquire Image. Of course, different payloads are involved, but they are outside the scope of the subsystem we’re modeling here. Nevertheless, they must provide services to aim and focus the sensor systems and provide the image. Let us assume that while the RMS may initiate the creation of an image, that image occurs asynchronously. Furthermore, the recon systems may be set up to send a stream of images at some specified rate. This means that the RMS requires a service to receive those images. The image itself is surely an instance of a class and may contain meta-data such as the location (via the navigation subsystem GPS), date and time stamps, the kind of sensor used to generate the image, configuration parameters of the sensors (such as the wavelengths used, or pulse width and frequency from the SAR), the encryption method, etc.

A first-cut solution with the services strategy is shown in [Figure 14.7](#). The only use of the ports in the collaboration is to connect to the ports of the encapsulating RMS. The provided ports are “behavioral” meaning that they provide the services requested over that port. The required interfaces mean that this is the class whose instance will ultimately invoke those services from the other subsystems over those ports. The ports between subsystems are

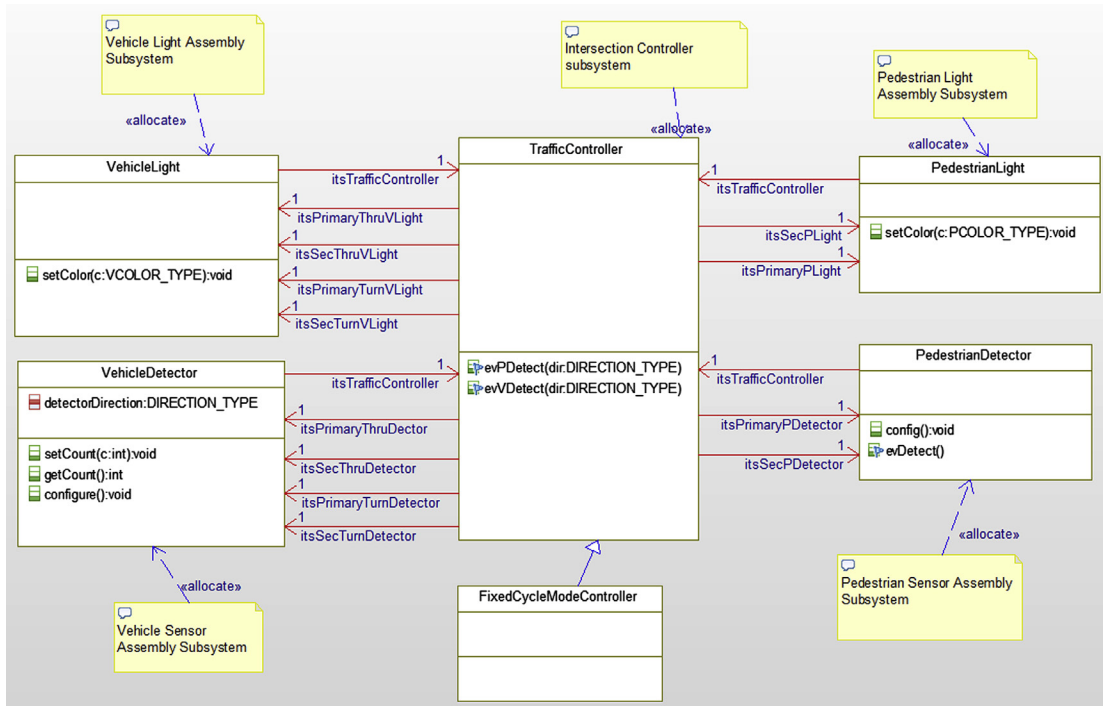


Figure 14.6
Fixed Cycle Mode: services strategy.

typically characterized by *both* provided and required interfaces. In this collaboration, the provided and required interfaces are typically realized by different classes inside the RMS. We've added the interface names to clarify the port usage.

The collaboration is a bit elaborated from the previous strategy. This elaboration includes moving operations and attributes around and creating new classes entirely. This process of elaboration is called “refactoring” and is very common during the creation of an object analysis model.

14.3 Answer 7.3: Apply the Strategies with a Test-Driven Development Approach

The task is to construct the collaboration of elements necessary to support Fixed Cycle Mode (FCM). Some key points about this mode include:

- Through traffic timing is driven entirely by timeouts (with some modifications based on the next few points).
- Turn traffic only activates if a car is waiting. Both turn lane directions for a road turn at the same time, if at all.

Acquire Image collaboration: services strategy.

- Although the behavior is simple, many people try to do it all in one step and end up with defects – both subtle and gross – in their designs and implementations. Our nanocycle approach to implementing test-driven development should ameliorate that.

1. Create a `VehicleLight` class that manages the state machine for a single light.
 - a. This light can go from any light state to any other via commanded event.
 - b. Add a testing buddy class to add unit-level testing.
 - c. Demonstrate through execution.
2. Add a `Fixed Cycle Mode` controller class
 - a. Make a single light go through its cycles with the right timing.
3. Add multiple light instances and add state logic in the `Intersection Controller` to handle through traffic only.

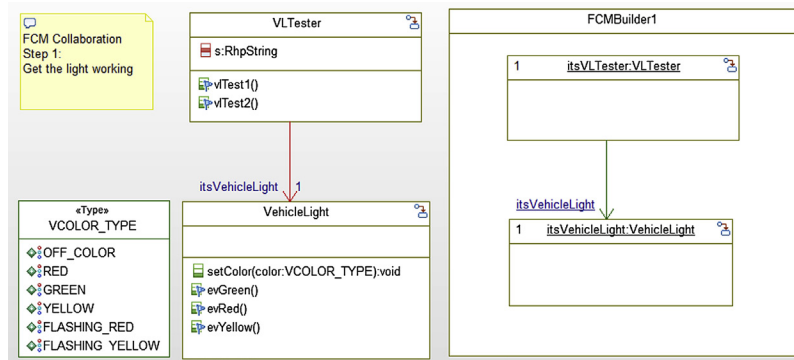


Figure 14.8
Fixed Cycle Mode step 1.

4. Add a car sensor class that can send an event when a car arrives (for turn lanes). Add logic in the Intersection Controller to receive and hold the information for later processing.
5. Add instances for turn lanes and process a case with a car arriving in the turn lane.
6. Add pedestrian buttons and identify where processing of pedestrian requests will be handled.
7. Handle pedestrian requests.

14.3.1 Step 1: Get the Light Working

1. The first step is pretty easy. The structural elements of the collaboration – the `VehicleLight`, `VLTester`, and `FCMBuild1` classes and the `VCOLOR_TYPE` are all shown in [Figure 14.8](#).

The `VehicleLight` class has a state machine that accepts events to change the color of the light. As in our example in Chapter 7, the `VLTester` class is also state-driven. Both state machines are shown in [Figure 14.9](#).

The result of running the two test cases is shown in [Figure 14.10](#) and the output of the tests is shown in [Figure 14.11](#). The initial output of “Red” comes from the default entry of the vehicle light into the Red state. The `setColor()` operation of the light includes output state-ments to show when the state is changed so that the actual output can be easily compared to the expected test outcome.

14.3.2 Step 2: Add a Controller and Step a Single Light through its States

This is a very simple addition but will serve as a foundation for what is to come, so we want to verify its correctness before moving on ([Figure 14.12](#)).

The state machine for the `TrafficController` class is straightforward, as we can see in [Figure 14.13](#).

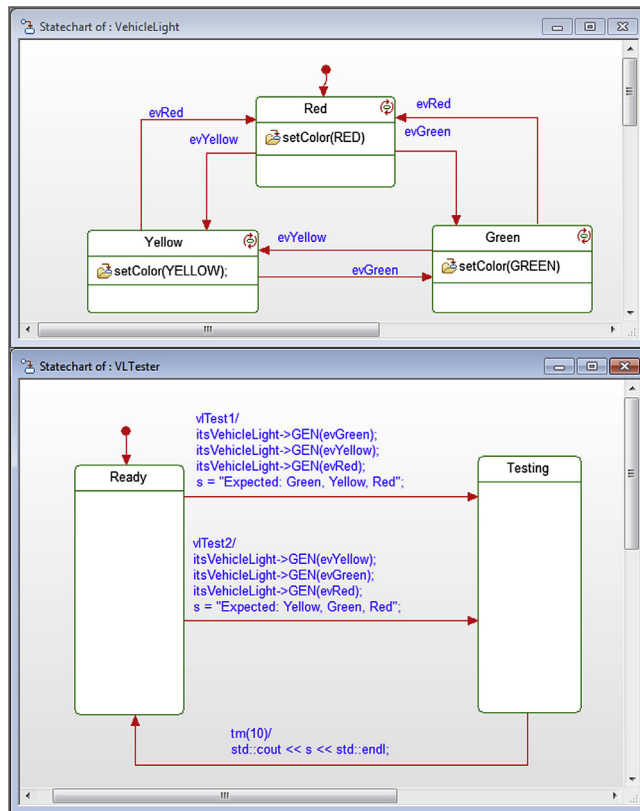


Figure 14.9
State machines for FCM step 1.

This results in correct execution when the event `evGo` is sent to the `TrafficController` instance.

14.3.3 Step 3: Add Through Lights for Both Primary and Secondary Roads and Their Control

With the next step, the system begins to act like the final system in a meaningful way – it controls traffic on both roads. It still doesn't add turn lanes or pedestrians, but it does control through lanes. The modifications in this iteration include:

- The traffic controller has two associations to the `VehicleLight` class, one for the primary through road (`itsPThruVLight`) and one for the secondary road (`itsSThruVLight`). See [Figure 14.14](#).
- The state machine for the `TrafficController` class is updated to control both lights. Notice that the state names were changed as the behavior became more complex ([Figure 14.15](#)).

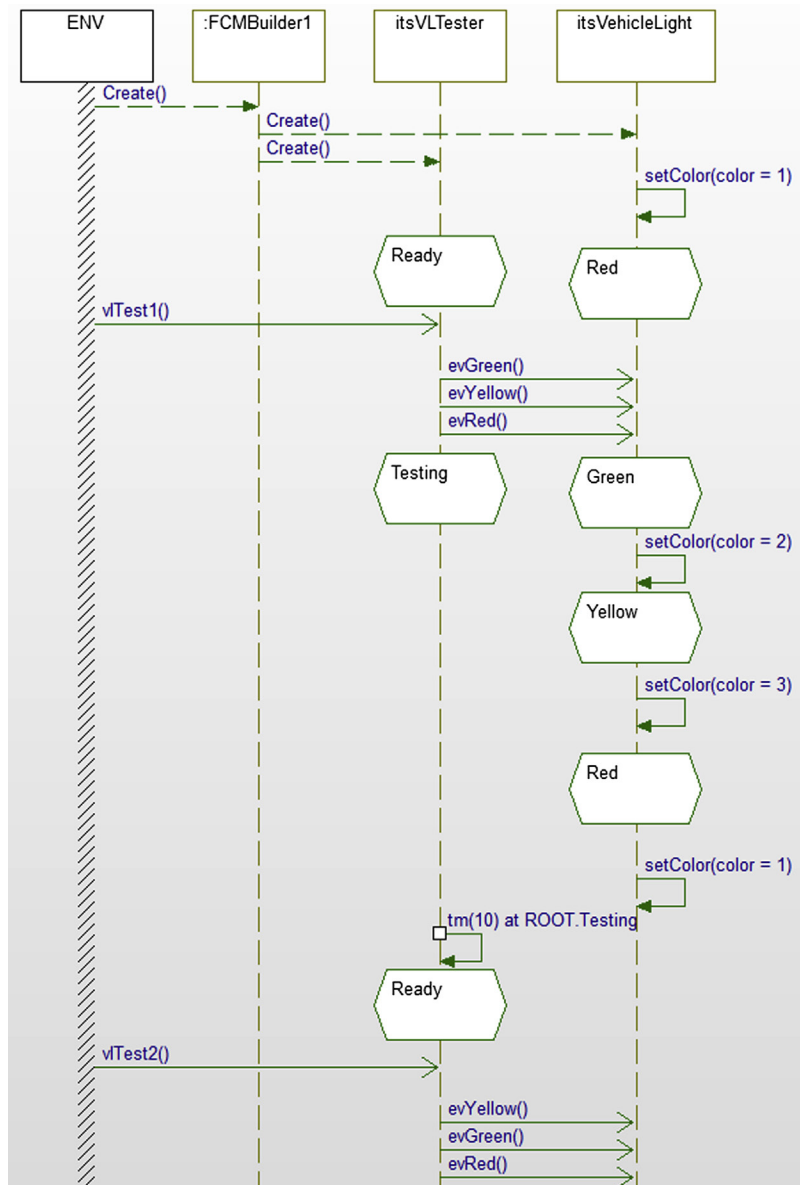


Figure 14.10
Running tests for FCM step 1.

- The `VehicleLight` class now has an attribute that identifies which instance it is (to support debugging) and its `setColor()` operation prints out both the road ID and its state.

The execution of this system results in the output shown in [Figure 14.16](#). The vehicle lights initialize before they are sent their IDs by the `TrafficController`, which is why they first print out UNKNOWN ROAD when they are initialized to Red.

```

C:\Users\Bruce Douglass\AppData\Roaming\Microsoft\Internet Explorer\Quick Launch\IBM\IBM R...
Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cyguinru...
n.bat" FCMCompl.exe
Red
Green
Yellow
Red
Expected: Green, Yellow, Red
Yellow
Green
Red
Expected: Yellow, Green, Red

```

Figure 14.11
Output from tests for FCM step 1.

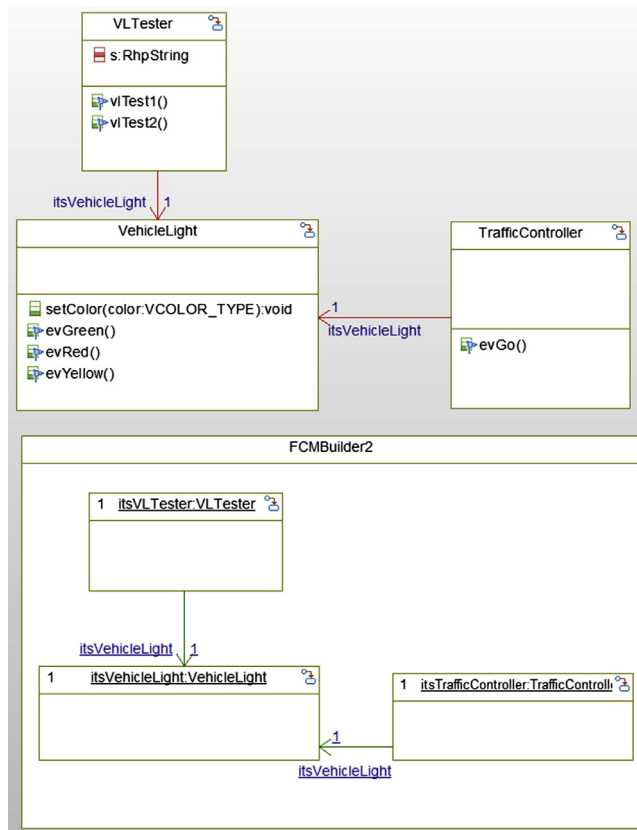


Figure 14.12
Fixed Cycle Mode step 2.

14.3.4 Step 3.1: Convert *TrafficController State Machine to And-States*

Looking ahead, we'll soon be adding turn lanes and pedestrian traffic control. I think it will be simplified if we do this using and-states rather than or-states. For this reason, I'm adding an intermediate iteration just to do that. Everything else is the same except that the

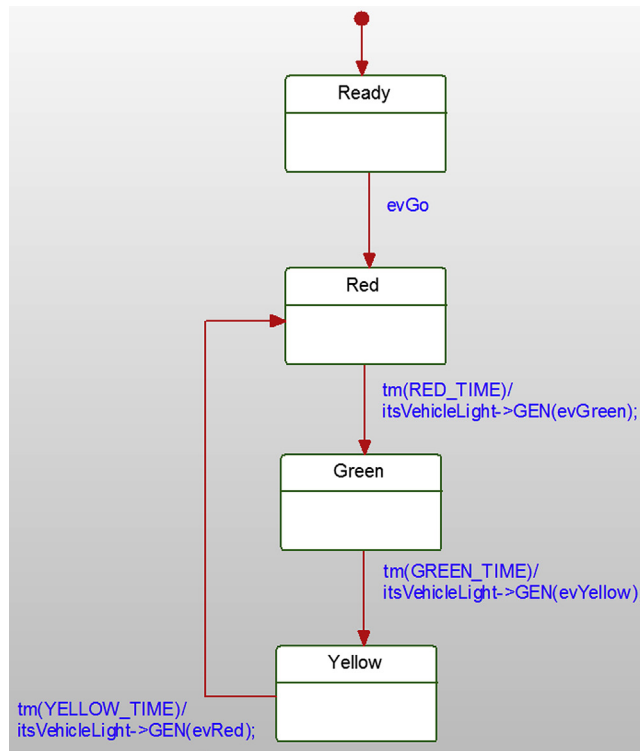


Figure 14.13

TrafficController state machine for step 2.

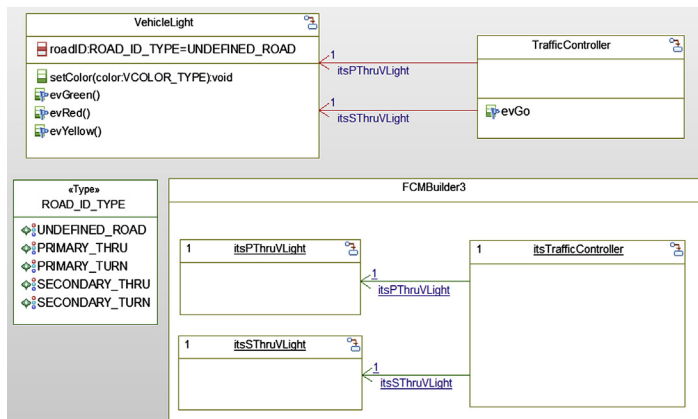


Figure 14.14

Fixed Cycle Mode step 3.

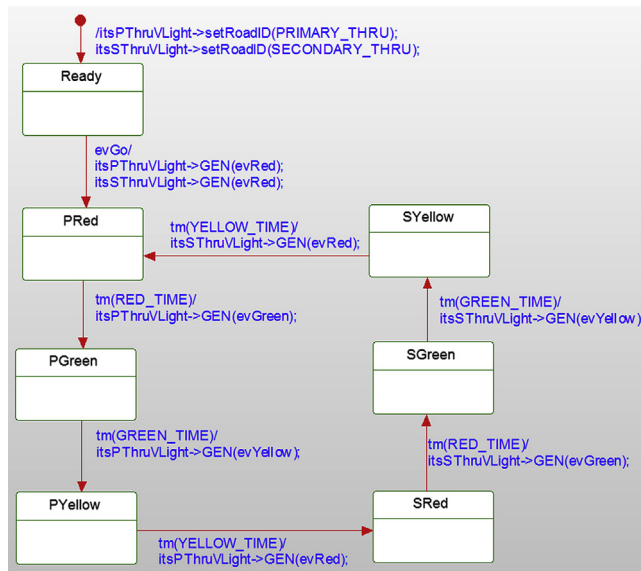


Figure 14.15

TrafficController step 3 state machine.

TrafficController state machine now looks like it does in Figure 14.17. Execution verifies that the execution output is identical to step 3.

14.3.5 Step 4: Add Car Sensor for Turn Lanes

Turn lanes are only activated in Fixed Cycle Mode when a car has arrived. To add this into our collaboration, we'll need to add a vehicle sensor (we can later specialize the vehicle sensor type when it becomes important). The vehicle sensor will have a one-way association to the TrafficController and will send an event when a car arrives. For the TrafficController to know which turn lane has a car, the vehicle sensor will have to know its identity. This will be set up in the TrafficController state machine as it starts up into the Ready state. The structure of the collaboration for step 4 is shown in Figure 14.18.

The most interesting behavioral change is the elaboration of the state machine for the TrafficController class. You can see in Figure 14.19 that the change is relatively minor. Building upon the behavior for the through lanes, a conditional connector is added so that the behavior in state ProcessPTurn is executed if there is a car waiting. Otherwise, the through lane is activated in the same fashion as it was previously.

ProcessPTurn is what is called a *submachine* – it has nested states but for readability reasons I show them in another diagram (see Figure 14.20). The default transition for the ProcessPTurn state in Figure 14.19 is automatically taken when the submachine in Figure 14.20 reaches the terminal pseudostate. Also note that when the turn lane processing has finished, the vehicle sensor is reset with the event evResetSensor.

```

C:\Users\Bruce Douglass\AppData\Roaming\Microsoft\Internet Explorer\Quick Launch\IBM\IBM R...
Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cygwinru...
n.bat" FCNComp3.exe
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
Primary U Thru Light is Green
Primary U Thru Light is Yellow
Primary U Thru Light is Red
Secondary U Thru Light is Green
Secondary U Thru Light is Yellow
Secondary U Thru Light is Red
Primary U Thru Light is Green

```

Figure 14.16
Output of execution of step 3.

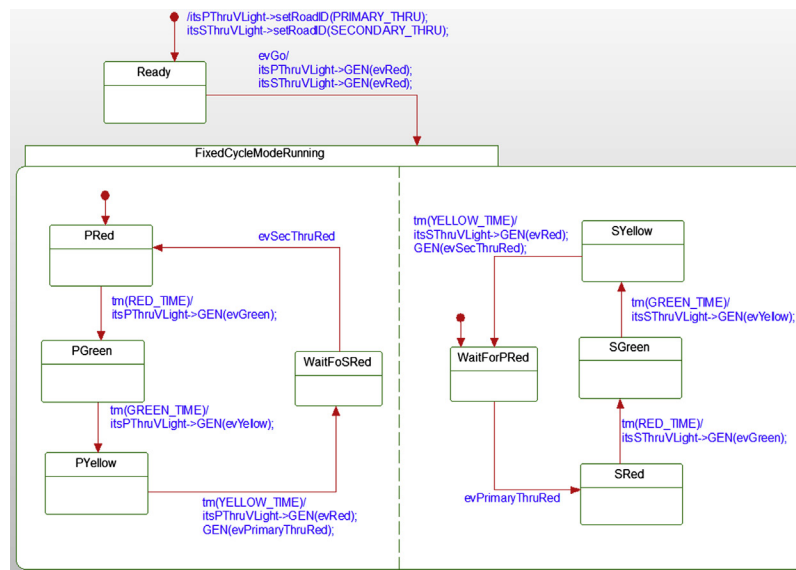


Figure 14.17
TrafficController state machine with and-states.

To round out the behavioral design for step 4, the state machines for the `VehicleSensor` and the `TCTester4` classes are shown in [Figure 14.21](#). We can run the normal mode without the turn car waiting by sending the `evGo` event to the `TrafficController`. Or we can cause a car to wait and then run the collaboration by executing `evTest4_1`. The outcome of running `evTest4_1` is shown in [Figure 14.22](#), and is exactly what we expect.

14.3.6 Step 5: Add the Second Turn Lane

This step is pretty simple. We just need to replicate the changes we made in the primary turn and-state in the secondary turn and-state and add a few test cases to make sure it all works. The only structural change is to add a fourth association from `TrafficController` to

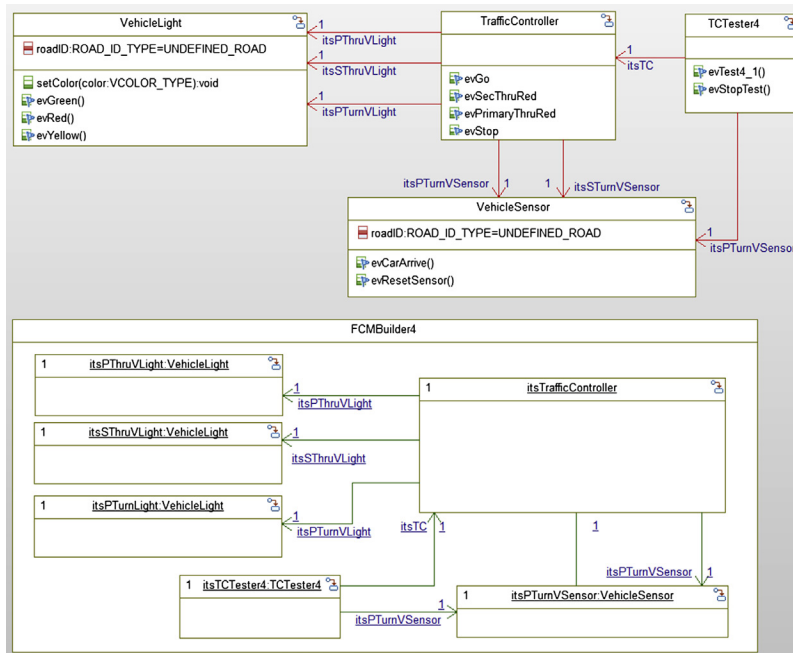


Figure 14.18
FCM Step 4.

VehicleLight named `itsSTurnVLight` (Figure 14.23). Behaviorally, we need to update the state machine for **TrafficController** to handle cars turning from the secondary road and the state machine for **TCTest4** (Figures 14.24 and 14.25). The state machine for the **TCTest5** class is provided in Figure 14.26. Finally, the output of the test run is shown in Figure 14.27.

As an aside, note that in the real system we have more vehicle sensors for through traffic detection, but they are unused in Fixed Cycle Mode, so we haven't added them yet. When we add other modes, they will be added.

14.3.7 Step 6: Add Pedestrian Buttons and Identify Where Processing of Pedestrian Requests Will Be Handled

For this next step, we want to add pedestrian buttons and handle the case where pedestrians have requested to cross the road. We won't add the actual light behavior until step 7 but we'll add a state in place to hold the behavior when it is introduced. Figure 14.28 shows the class structure for the updated collaboration (although not the object model). The state machine for the **TrafficController** is shown in Figure 14.29. In it, you can see the placeholders for the handling of the pedestrians in states `ProcessPPed` and `ProcessSPed`. If a road is in its green state (e.g., `PGreen`), then it will exit to the pedestrian handling state (e.g., `ProcessPPed`) if the sensor is in the `PedestrianWaiting` state; if not, then the timeout will occur

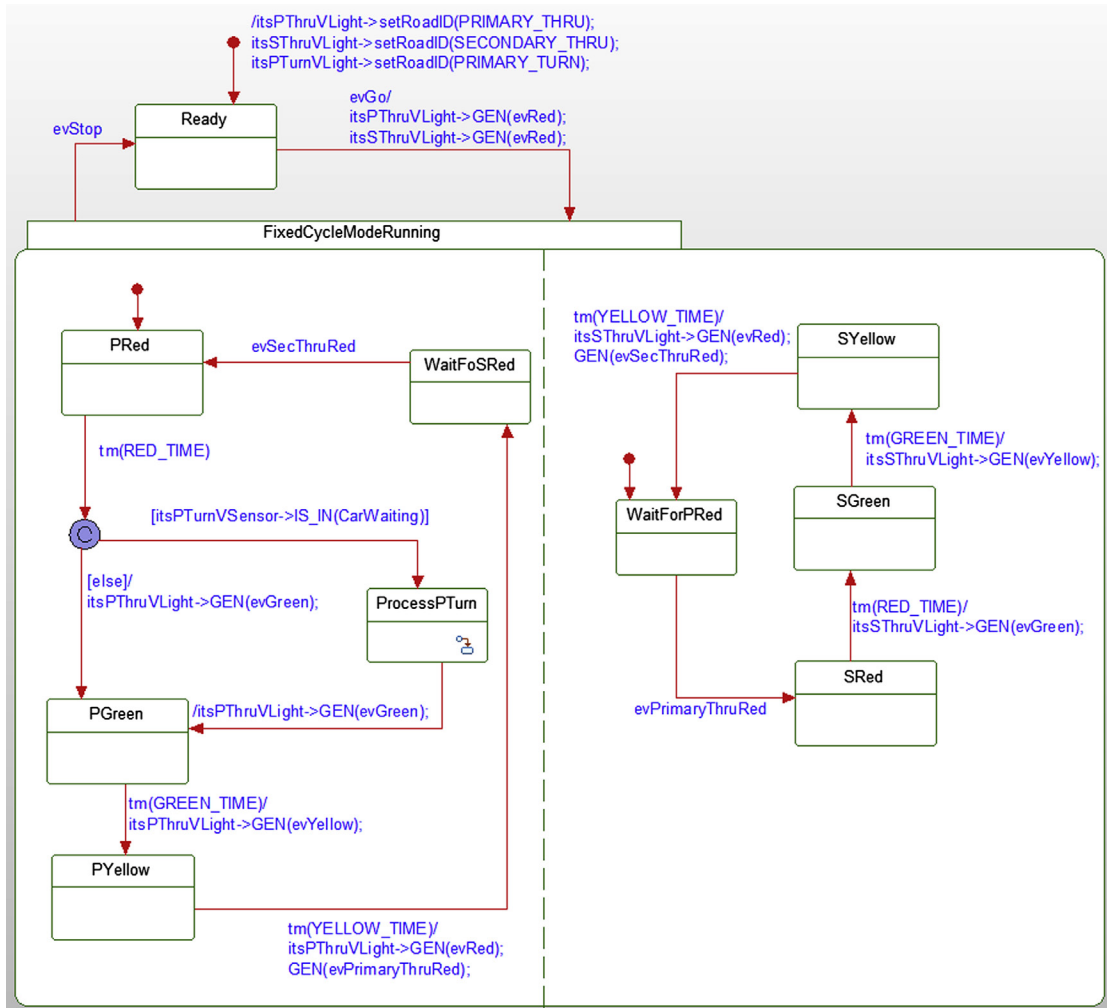


Figure 14.19

TrafficController step 4 state machine.

naturally and the light will proceed as normal. This ensures that the through light does not change to yellow while pedestrians are still getting a walk or flashing don't walk signal. Another minor change in this state machine is some transition actions were converted to state entry actions when more than one transition into that state carried the same action.

The state machine for `PedestrianSensor` is very similar to `VehicleSensor` and is shown in the next figure, [Figure 14.30](#).

We won't show the execution and test for this step but we will once we get the pedestrian lights added in step 7.

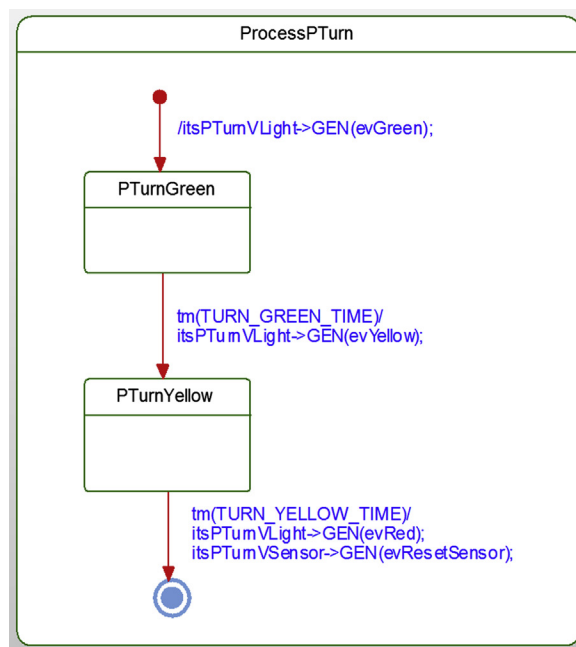


Figure 14.20
ProcessPTurn submachine.

14.3.8 Step 7: Handle Pedestrian Requests

In this step, we'll add the `PedestrianLight` class and elaborate the state machine processing in the `TrafficController` class. And, of course, we'll add more test cases to demonstrate the model's correctness.

Figure 14.31 shows the completed collaboration. Of course, this isn't the complete system – there are other use cases, other modes of operation, and other requirements to realize. Nevertheless, we've done a lot of good work in a very short time.

Figure 14.32 shows the elaborated `TrafficController` state machine. Additional actions were added to set up the pedestrian lights and we've added submachines to the `ProcessPPed` and `ProcessSPed` states. These submachines, along with the state machine for the `PedestrianLight` class, are shown in Figure 14.33. Code Listing 14.1 shows the code for the `PedestrianLight::setColor()` operation. In the real system, of course, we would drive the hardware to turn on the appropriate light sources, but for now, we'll just use standard output.

Finally, the output of running `Test7_3` – which sets up cars in both turn lanes and pedestrians on both crosswalks – is shown in Figure 14.34. Two cycles of the light are shown and this demonstrates the system correctly handles the waiting cars and pedestrians and then resets the sensors and proceeds normally thereafter.

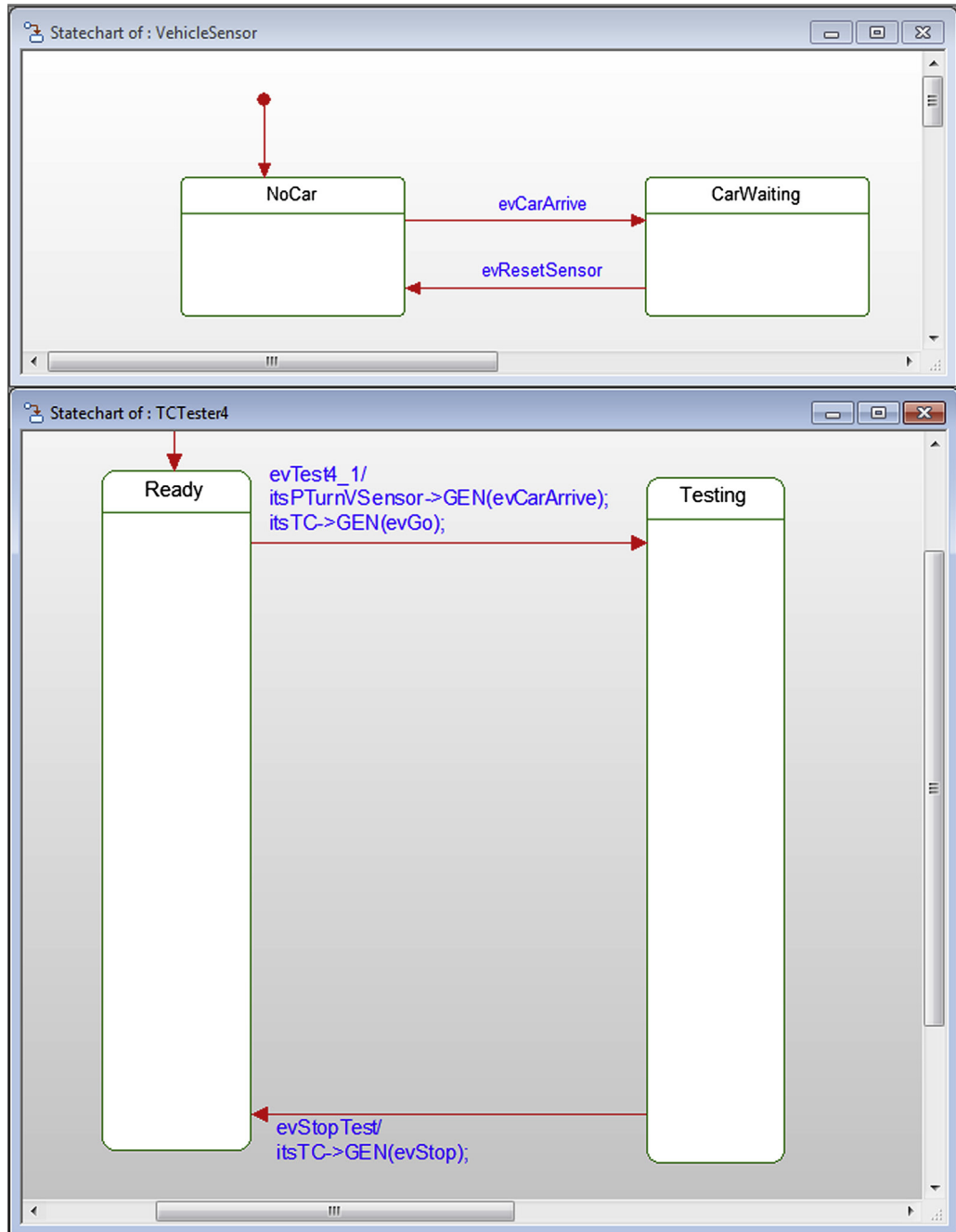


Figure 14.21
State machines for VehicleSensor and TCTester4.

```

C:\Users\Bruce Douglass\AppData\Roaming\Microsoft\Internet Explorer\Quick Launch\IBM\IBM R...
Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cygwinru...
n.bat" FCMComp4.exe
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
Primary U Turn Light is Green
Primary U Turn Light is Yellow
Primary U Turn Light is Red
Primary U Thru Light is Green
Primary U Thru Light is Yellow
Primary U Thru Light is Red
Secondary U Thru Light is Green
Secondary U Thru Light is Yellow
Secondary U Thru Light is Red
Primary U Thru Light is Green
Primary U Thru Light is Yellow

```

Figure 14.22
Output from Test4_1.

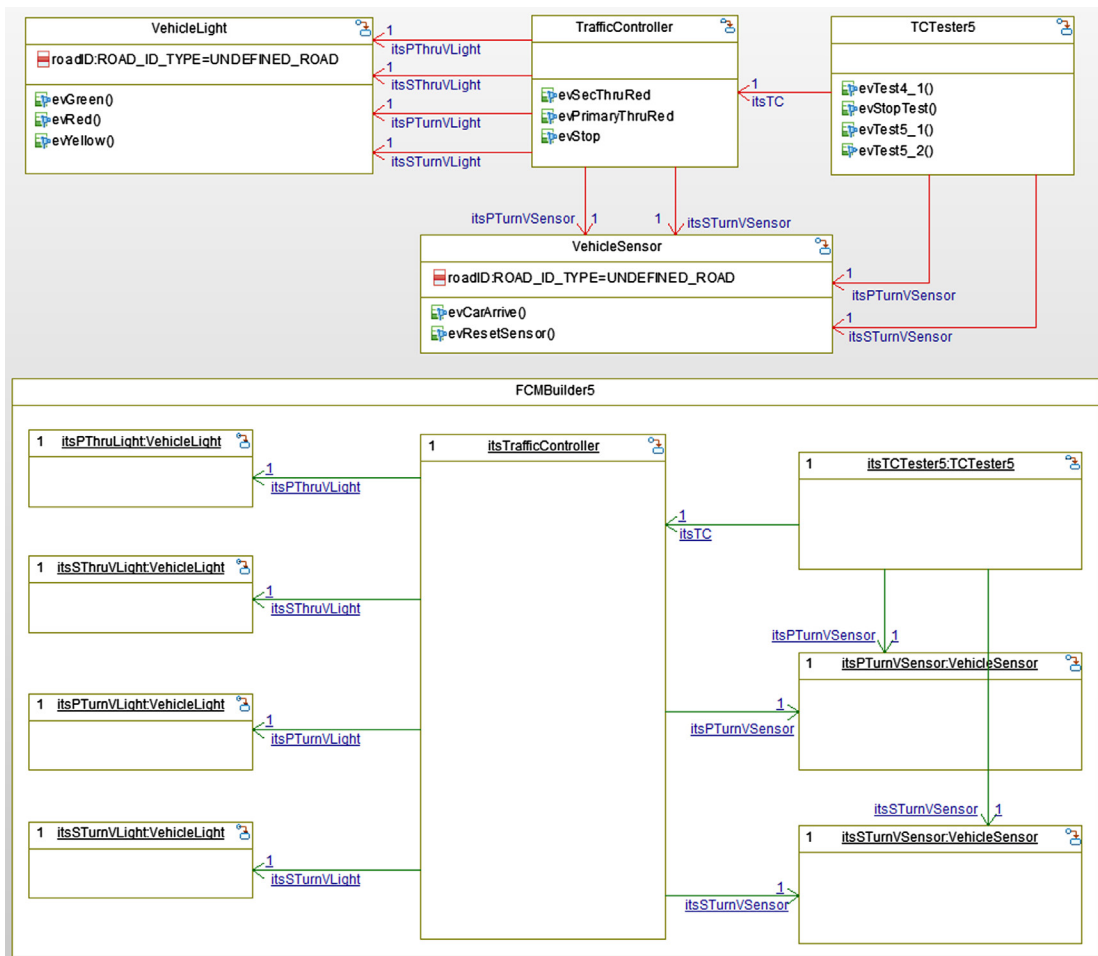


Figure 14.23
FCM step 5.

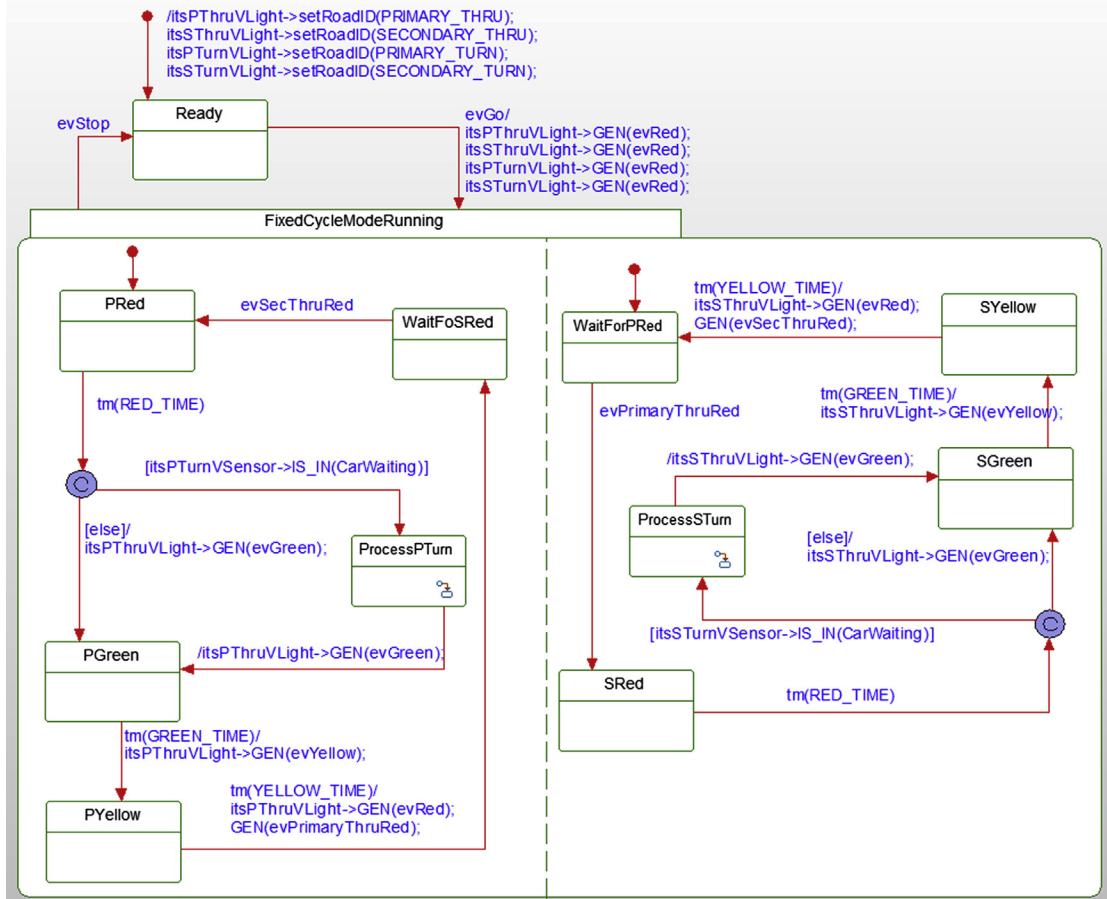


Figure 14.24

TrafficController step 5 state machine.

14.3.9 Test-Driven Development for Coyote UAV

For the UAV exercise, here is a proposed set of nanocycle iterations to produce the design collaboration and code:³

1. Get hydraulics working to move the aileron to a commanded control surface position.
2. Add limit control so as not to damage equipment (don't command less than `MIN_AILERON_POS` or more than `MAX_AILERON_POS`).
3. Add elevator control, including limit management.
4. Add rudder control, including limit management.
5. Add the ability to convert a desired aircraft roll into an aileron control position.
6. Add the ability to convert a desired aircraft pitch into an elevator control position.

³ Remember, a lot of the hard math is provided by the `KinematicModel` class.

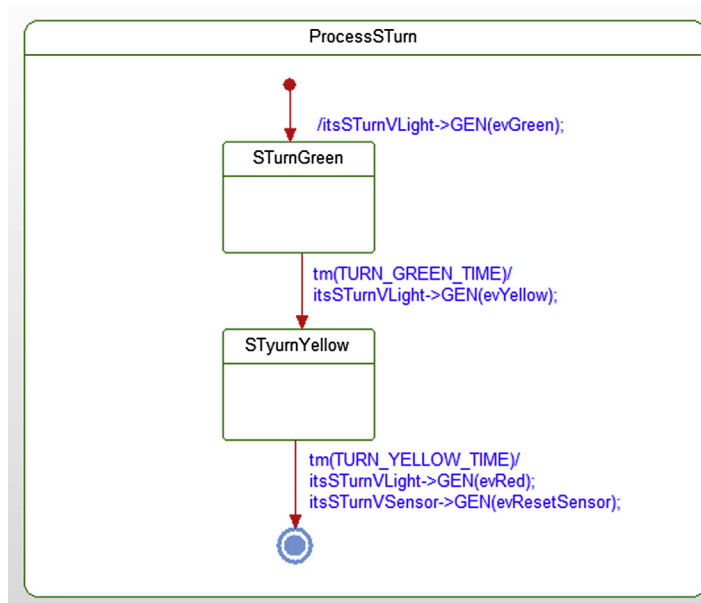


Figure 14.25
ProcessSTurn submachine.

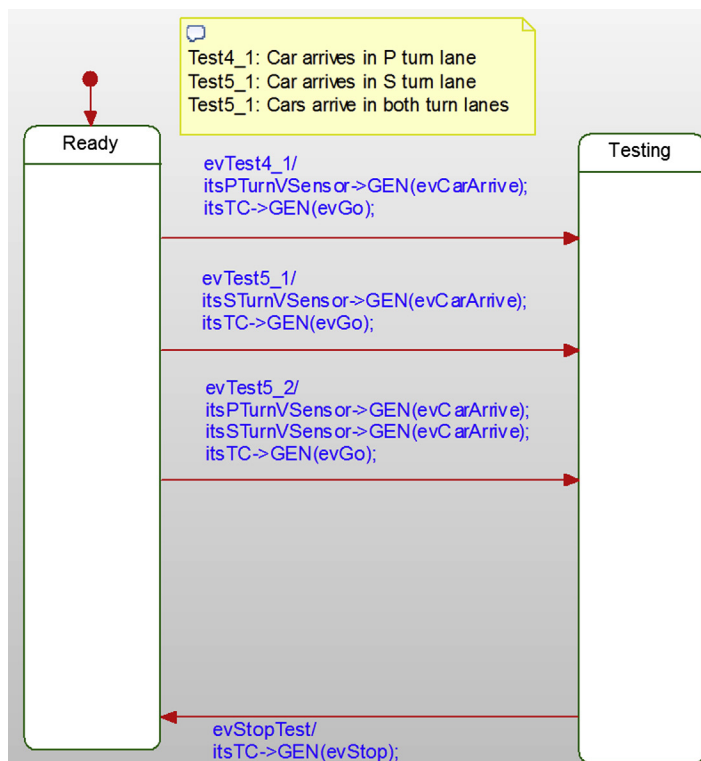


Figure 14.26
TCTest5 state machine.

```

Executing: "C:\Users\Bruce Douglass\IBM\Rational\Rhapsody\8.0\Share\etc\cygwinru
n.bat" FCMComp5.exe
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
UNKNOWN ROAD is Red
Primary U Turn Light is Green
Primary U Turn Light is Yellow
Primary U Turn Light is Red
Primary U Thru Light is Green
Primary U Thru Light is Yellow
Primary U Thru Light is Red
Secondary U Turn Light is Green
Secondary U Turn Light is Yellow
Secondary U Turn Light is Red
Secondary U Thru Light is Green
Secondary U Thru Light is Yellow
Secondary U Thru Light is Red
Primary U Thru Light is Green
Primary U Thru Light is Yellow
Primary U Thru Light is Red
Secondary U Thru Light is Green
Secondary U Thru Light is Yellow
Secondary U Thru Light is Red
Primary U Thru Light is Green

```

Figure 14.27
Test5_2 test outcome.

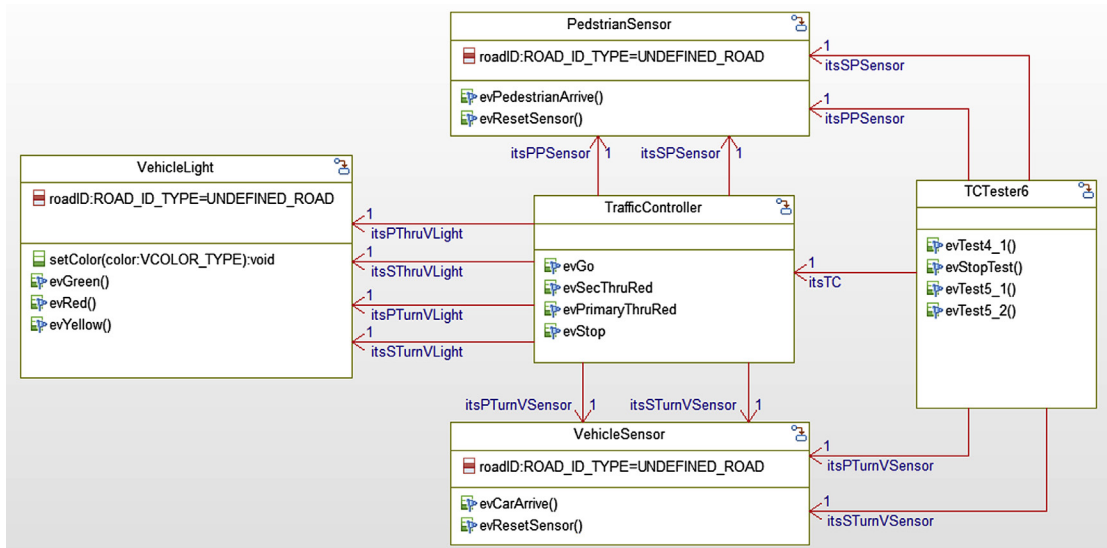


Figure 14.28
FCM step 6.

7. Add the ability to convert a desired aircraft yaw into a rudder control position.
8. Add a three-axis gyroscope to determine current aircraft orientation. Add a relationship from the `KinematicModel` class to the gyroscope assembly.
9. Add incremental adjustment of control surfaces based on the error between commanded and actual control surface position.
10. On startup, zero control surfaces and perform self-tests on the hydraulic motors and gyroscope.
11. Before going operational, execute a set of self-tests on the control surfaces. Only go operational if no unrecoverable errors are found.

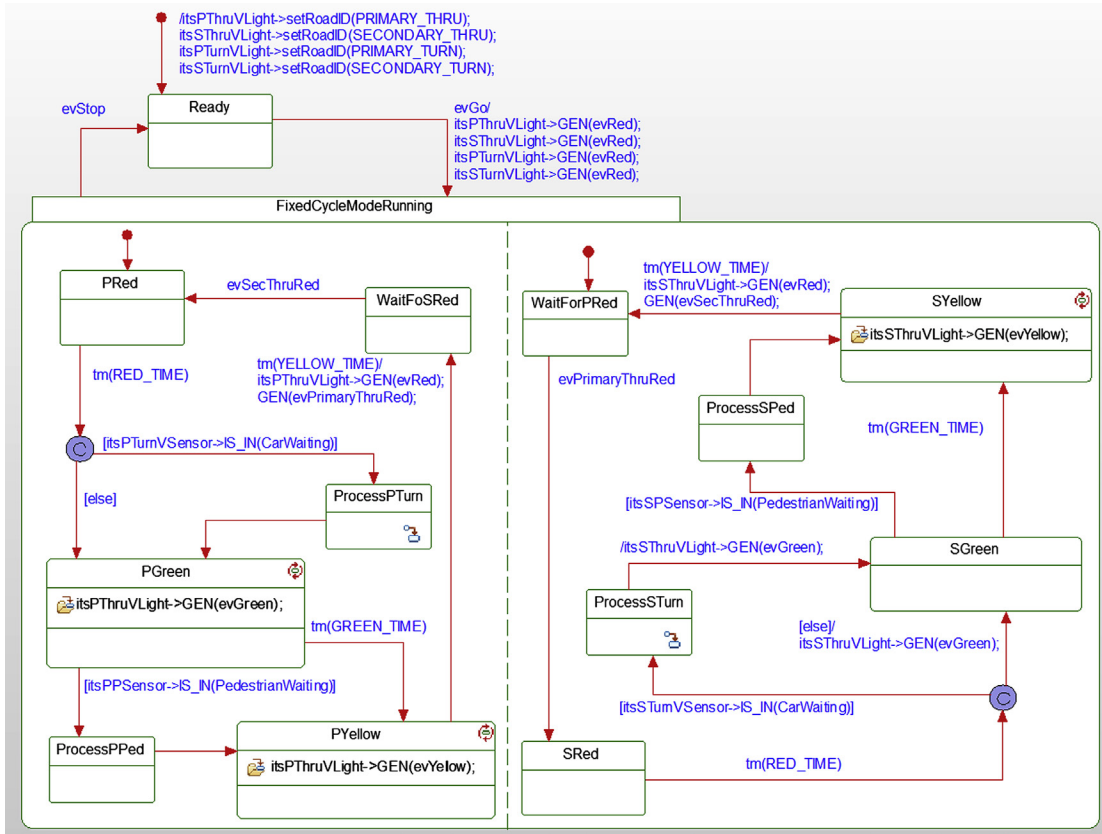


Figure 14.29
TrafficController step 6 state machine.



Figure 14.30
PedestrianSensor state machine.

Of course, this plan is subject to change as we learn things by going through the iterations.

Because of the number of pages required to show the step-by-step iteration results for this exercise, I won't provide the same level of detail as we did for the Roadrunner system. However, the final collaboration for the exercise that I came up with is shown in [Figure 14.35](#). To save space, I removed the testing classes from the diagram.

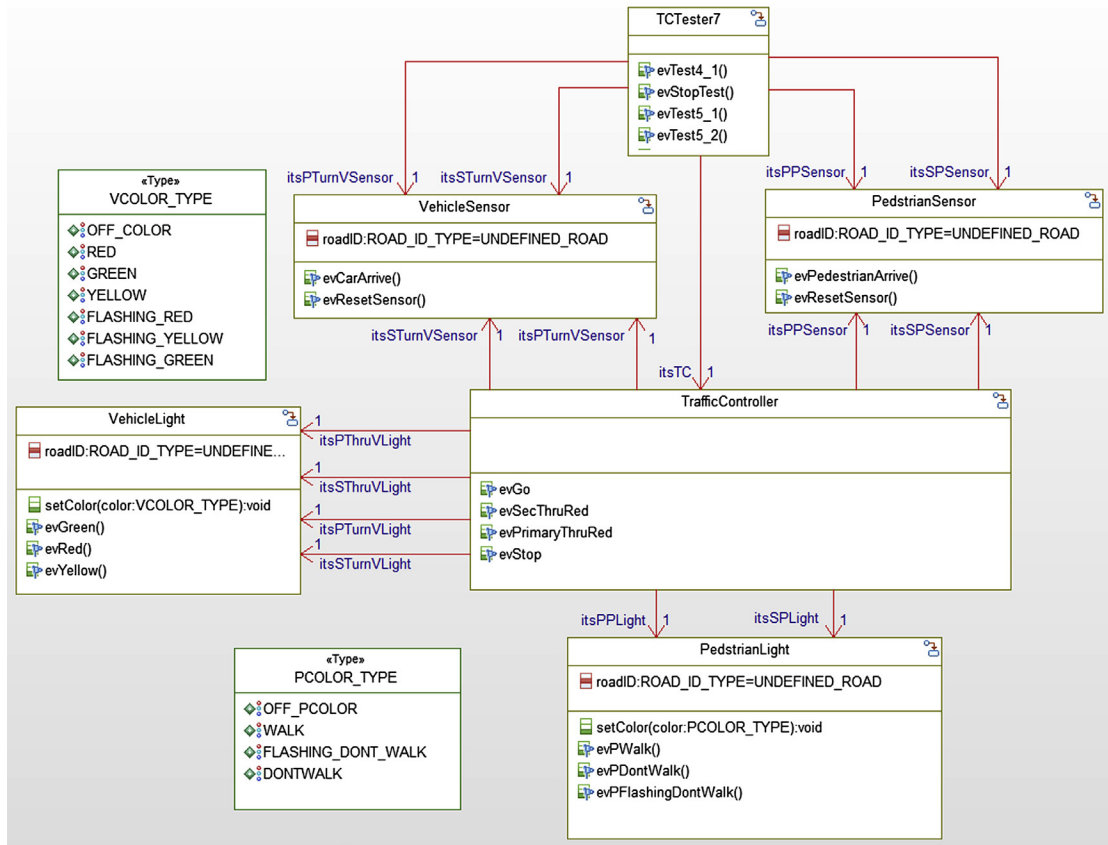


Figure 14.31
FCM step 7.

The `AttitudeControl` class aggregates a couple of instances of class `Attitude` – one for the commanded attitude and one for the measured attitude (which it gets from the `Gyroscope Assembly`). `AttitudeControl` also aggregates an instance of `ControlSurfaceSet`, which contains the desired positions of all the control surfaces.

The `AttitudeControl` class has a state machine for controlling its behavior. It is shown in [Figure 14.36](#). Notice the state machine loops for running the self-tests and zeroing. Physical movement and test execution takes time and these elements are running concurrently. Therefore, the `AttitudeControl` class uses a timeout to check for completion status before moving on.

When the `AttitudeControl` runs (based on the timeout in its and-state), it queries the `KinematicModel` for an updated orientation set. It then converts those into positions of the control surfaces and then commands the surfaces to those positions.

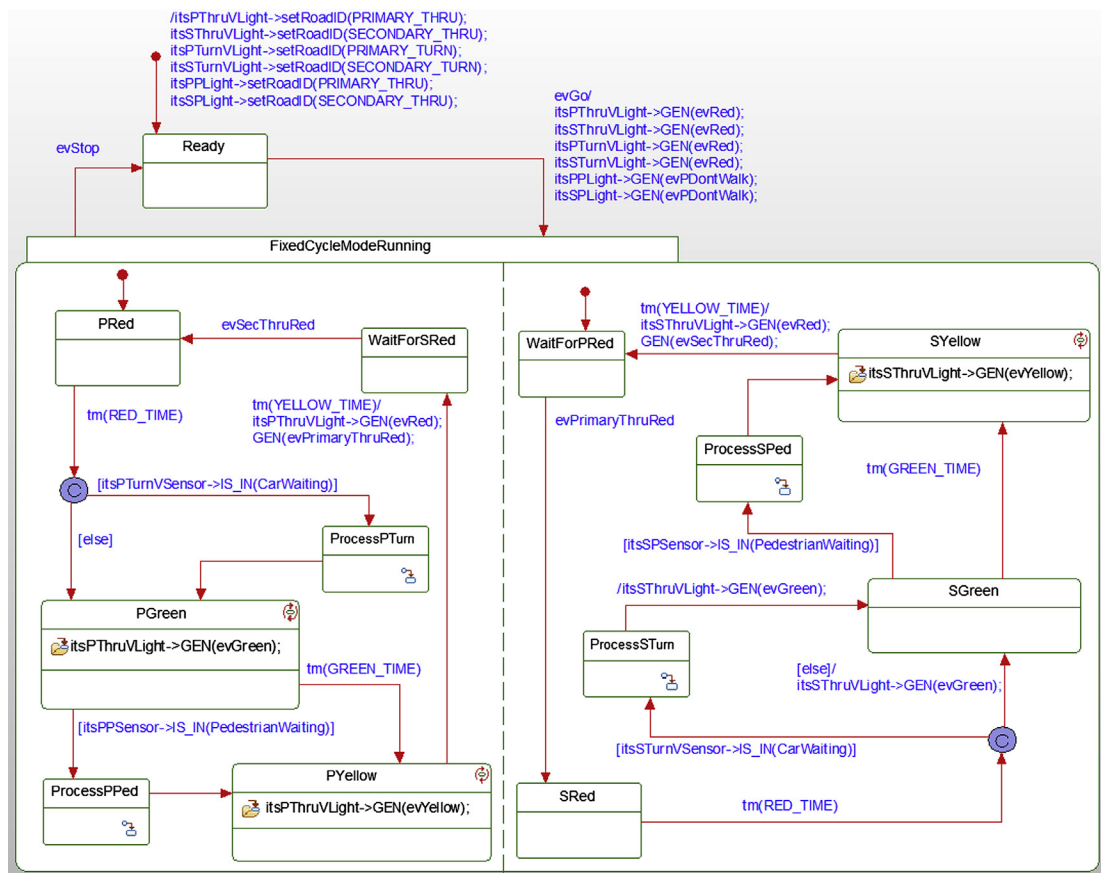


Figure 14.32
TrafficController step 7 state machine.

The last state machine is for the `ControlSurface` class (Figure 14.37). Instances of this class are used for each of the control surfaces; each instance of `ControlSurface` aggregates its `HydraulicMotor` and `SurfacePositionSensor` so that it can command a new position as well as measure its current position. If the position achieved is wrong (as determined by the absolute value of the difference between commanded and actual positions), the control surface is tweaked. If the commanded position is out of range, it is rejected.

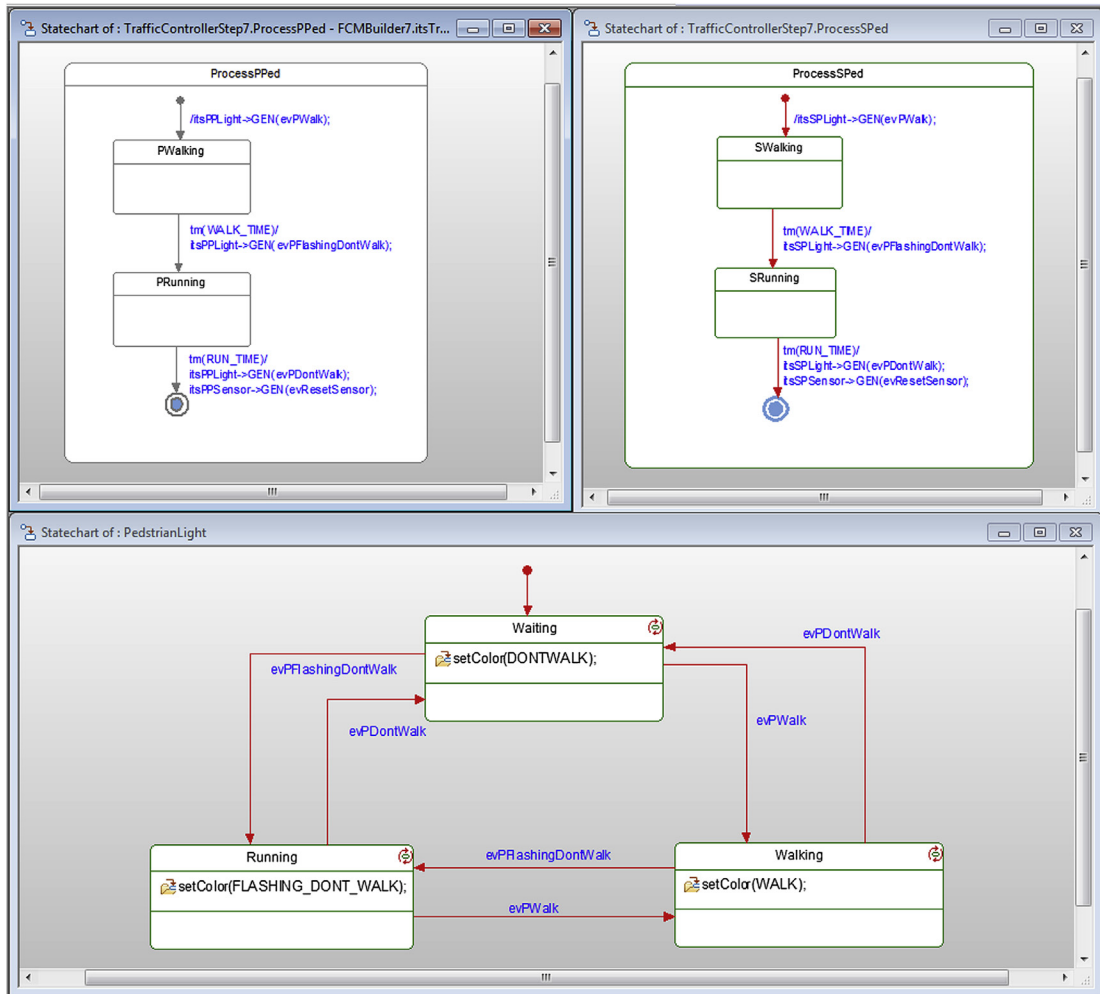


Figure 14.33
Step 7 other state machines.

```

void PedstrianLight::setColor(const PCOLOR_TYPE color) {
    switch (roadID) {
        case PRIMARY_THRU:      std::cout << "Primary PED Light is ";
                                break;
        case SECONDARY_THRU: std::cout << "Secondary PED Light is ";
                                break;
        default:                std::cout << "UNKNOWN ROAD is ";
                                };

    switch (color) {
        case WALK:              std::cout << "Walk" << std::endl;
                                break;
        case FLASHING_DONT_WALK: std::cout << "Flashing Don't Walk" << std::endl;
                                break;
        case DONTWALK:          std::cout << "Don't Walk" << std::endl;
                                break;
        default:                std::cout << "Huh??" << std::endl;
                                };
    };
};

```

Code Listing 14.1

Code for PedestrianLight::setColor.

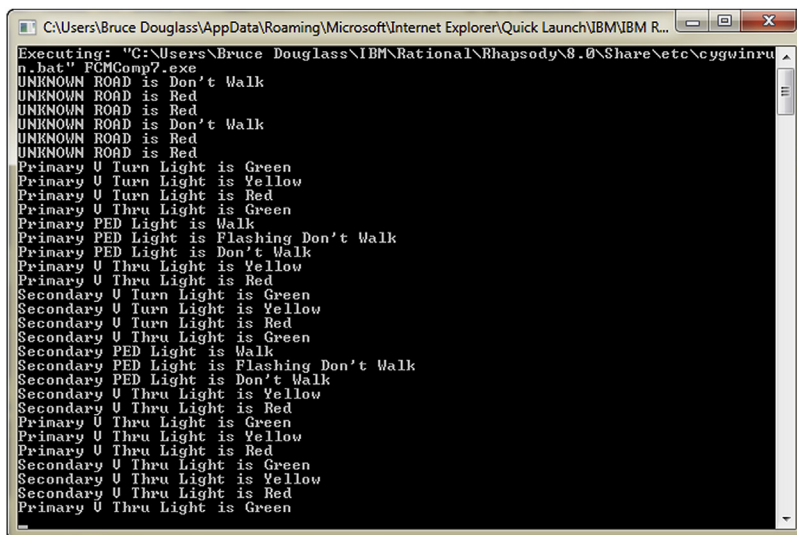


Figure 14.34
Outcome of Test7_3.



Figure 14.35
CUAV Auto Adjust Attitude collaboration.

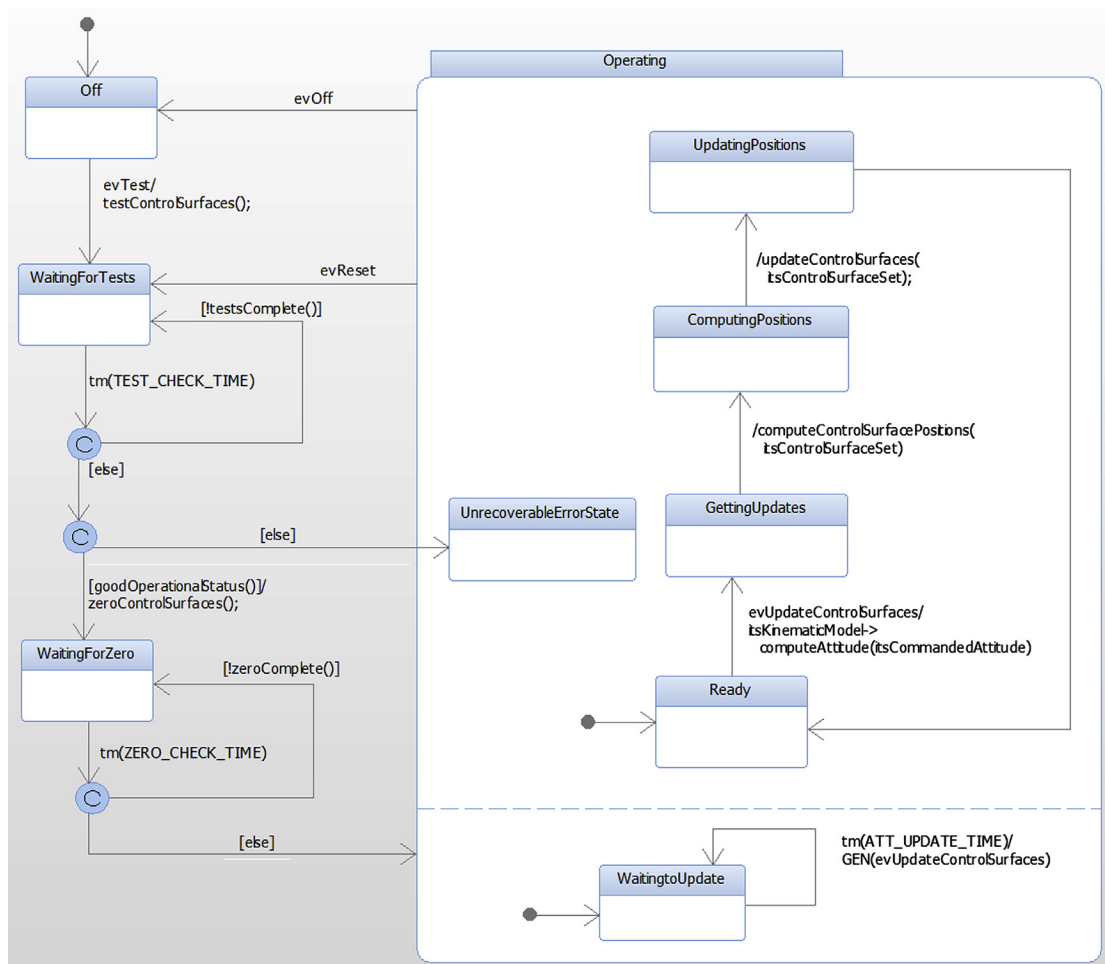
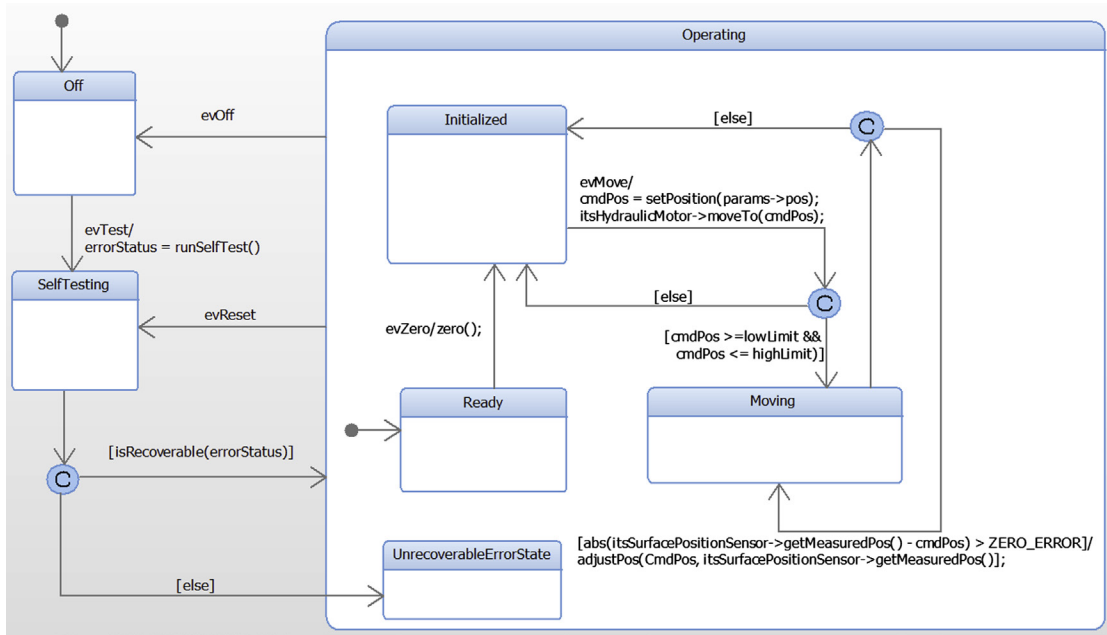


Figure 14.36
AttitudeControl state machine.

**Figure 14.37**

ControlSurface state machine.

Distribution Architecture: Answers

15.1 Answer 8.1: Roadrunner Distribution Architecture

Figure 15.1 shows the distribution architecture for the Intersection Controller subsystem. The distribution-related classes are shaded. Note that many of the previous associations between classes have been “removed” since they will be implemented via a communication protocol. The figure shows three primary communication classes:

- `WiredNetworkComms` to support Ethernet connections for the front panel display (display and keyboard) and for the remote monitoring and control station
- `WirelessNetworkComms` to support communications with above-surface vehicle detector assemblies (which include the priority and vehicle detection sensors as well)
- `SerialComms` to support the wired connections among elements local to the intersection (vehicle light assemblies, vehicle detectors, pedestrian light assemblies, pedestrian detectors, and the Intersection Controller)

In addition, there are some helper classes identified:

- `MessageQueue` – For storing messages received but not yet processed and about to be sent. There will be two instances of this class to support bidirectional communication
- `Message` – The internal format for the data to be sent or received.
- `SerialMsg` – A message to be sent or received over the serial interface.
- `Datagram` – A message to be sent or received over the wired or wireless Ethernet protocol.

We can go a bit deeper here as well. Figure 15.2 goes into more detail about the structure of the serial protocol stack. You can see that it consists of three layers, the `SerialAppLayer`, the `SerialTransportLayer`, and the `SerialDataLinkLayer`, each supporting communication semantics at its own level of abstraction.

15.2 Answer 8.2: Coyote UAV Distribution Architecture

The distribution architecture for connecting the ground system and the aircraft has been largely emplaced, so this part of the problem should create no difficulties. As mentioned in Chapter 8, there are two wireless data links that connect these two major systems – the low-speed encrypted command link and the high-speed unencrypted video link (Figure 15.3).

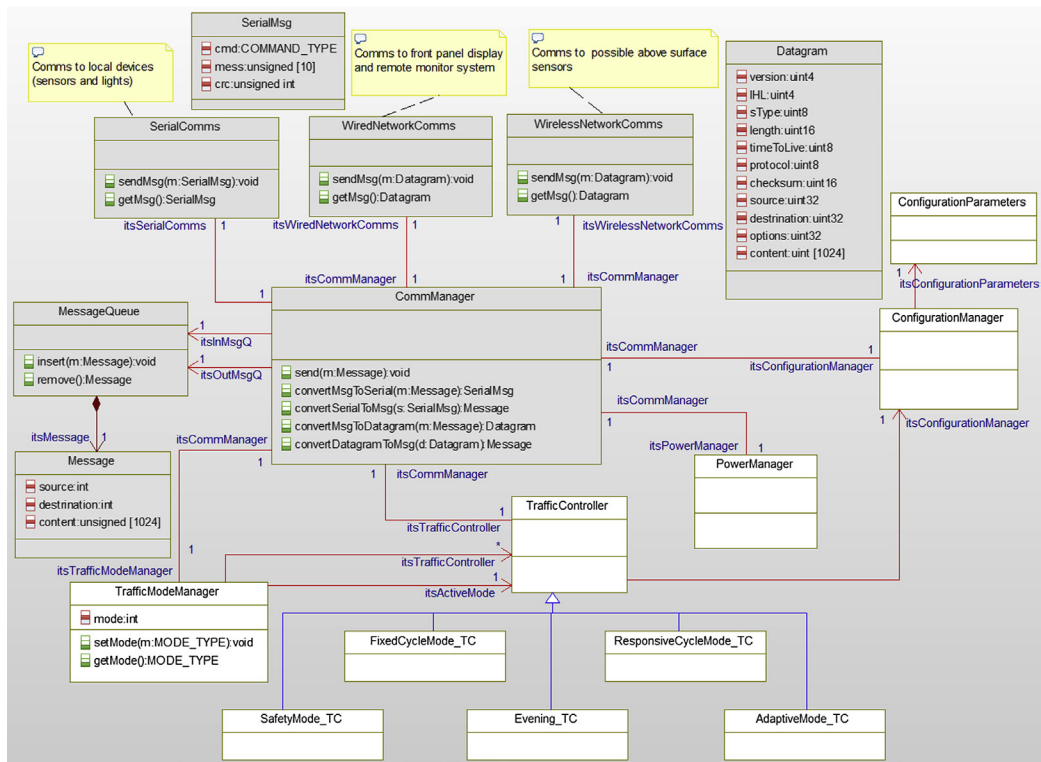


Figure 15.1
Intersection Controller distribution model.

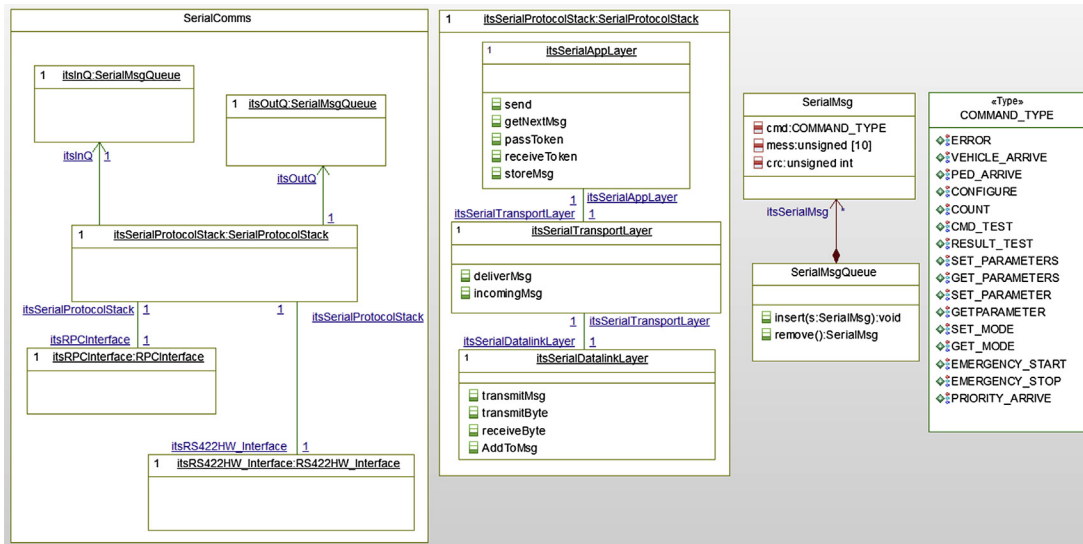


Figure 15.2
Serial interface structure.

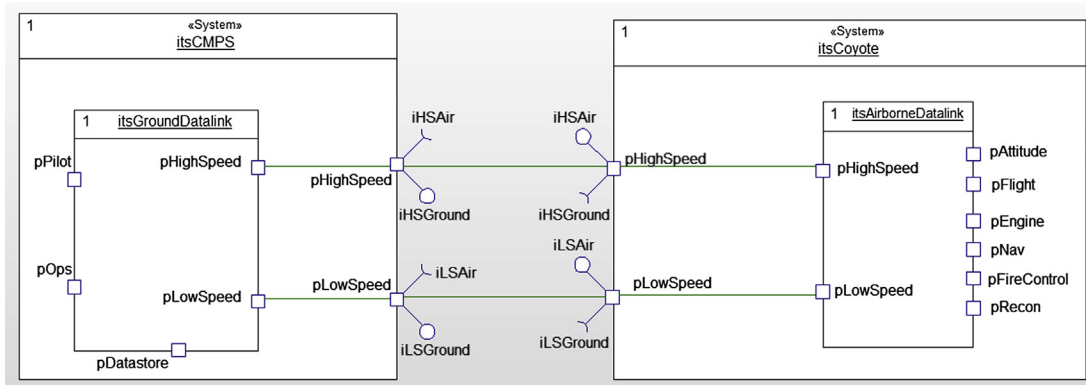


Figure 15.3
CUAV ground-aircraft distribution architecture.

The internal design of the data link is shown at a high level in [Figure 15.4](#). You see that it consists of a standard OSI-protocol stack architecture. The `AppLayer` interfaces provide application-level communication semantics (like sending messages, managing incoming and outgoing messages, and application notification when messages arrive). The `Presentation-Layer` converts messages from internal aircraft message format to the transmission format (and vice versa) and also provides encryption/decryption services. The `SessionLayer` manages sessions. Sessions are used primarily to support multimessage conversations such as uploading mission parameters and downloading ongoing reconnaissance data. The `TransportLayer` selects the channels to be used to send the network messages and manages their delivery. Finally, the `DatalinkLayer` manages the delivery and receipt of network messages through transmission or receipt of network packets (fragments of network messages). The `Transaction` class allows simultaneous communication of multiple messages on each channel; a new transaction object is created to manage the transmission of every network packet (except for packet ACKs), including the retries. When the packet is successfully delivered, the transaction object is then deleted. Since a network message is composed of a set of packets, there is a similar mechanism to ensure network messages (with all their packets) are successfully transmitted.

Internal to the UAV, we apply the port proxy pattern, as the problem dictated. We must create a port proxy pair for each subsystem pair we need to connect. The semantic interfaces between subsystems are already defined in the subsystem architecture; these interfaces are offered and provided as appropriate by the proxy back to its associated subsystem. The connection between the port proxy pair is made via the connected port conjugates.

I added a stereotype `<<PortProxy>>` to help identify the special purpose of these classes in collaborations. The instantiable port proxies all inherit their network ports and behavior from

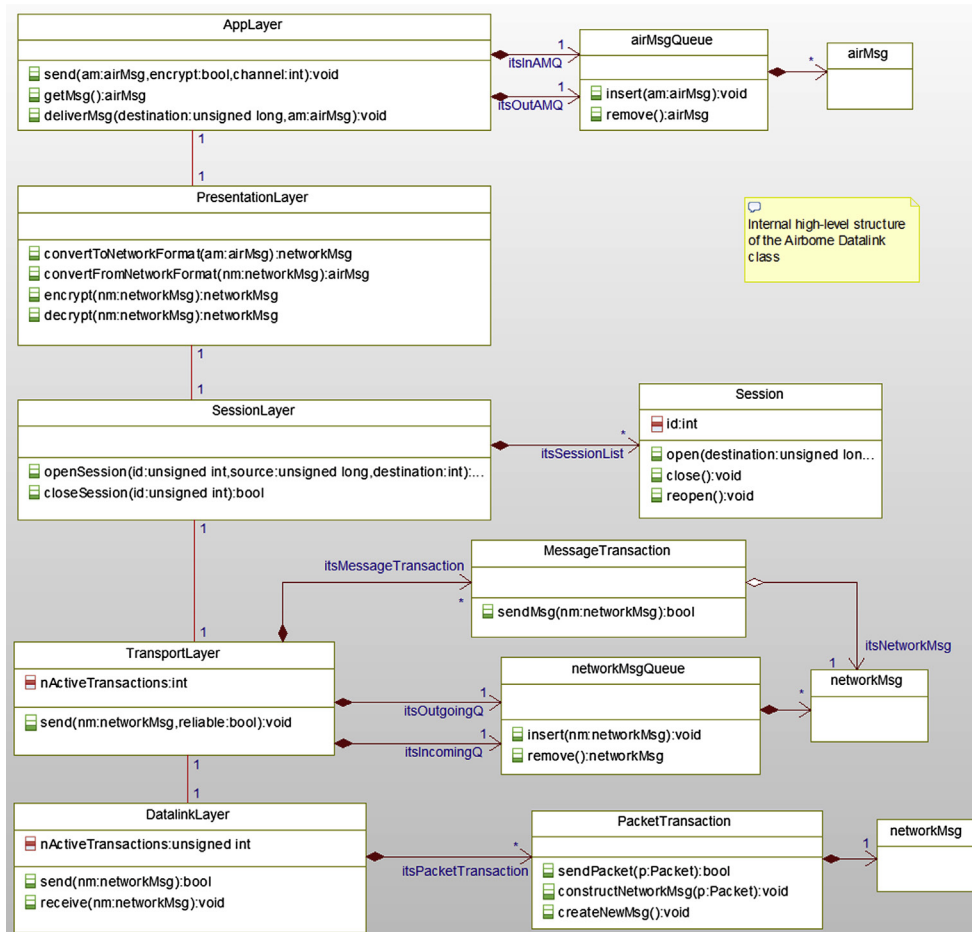


Figure 15.4
Airborne Datalink internal structure.

the base class `UAVPortProxy`: (Figure 15.5). Each subclass also inherits an untyped port `pSemantic`, which it will type by attaching appropriate semantic interfaces. Each subclass will also need specific code for marshaling the semantic messages into network messages and to perform the unmarshaling for messages coming off the network. The advantage of this pattern is that it isolates the network-specific code into the port proxies so that if the underlying network infrastructure changes, only the affected proxies must be updated.

For example, suppose we wanted to merge the content of the Fuel Management subsystem into the Flight Management subsystem. In this case, they would no longer be connected over the avionics bus but could be directly interfaced. In a more traditional pattern of intermixing the network code and the semantic code, we would have to rewrite significant portions of both

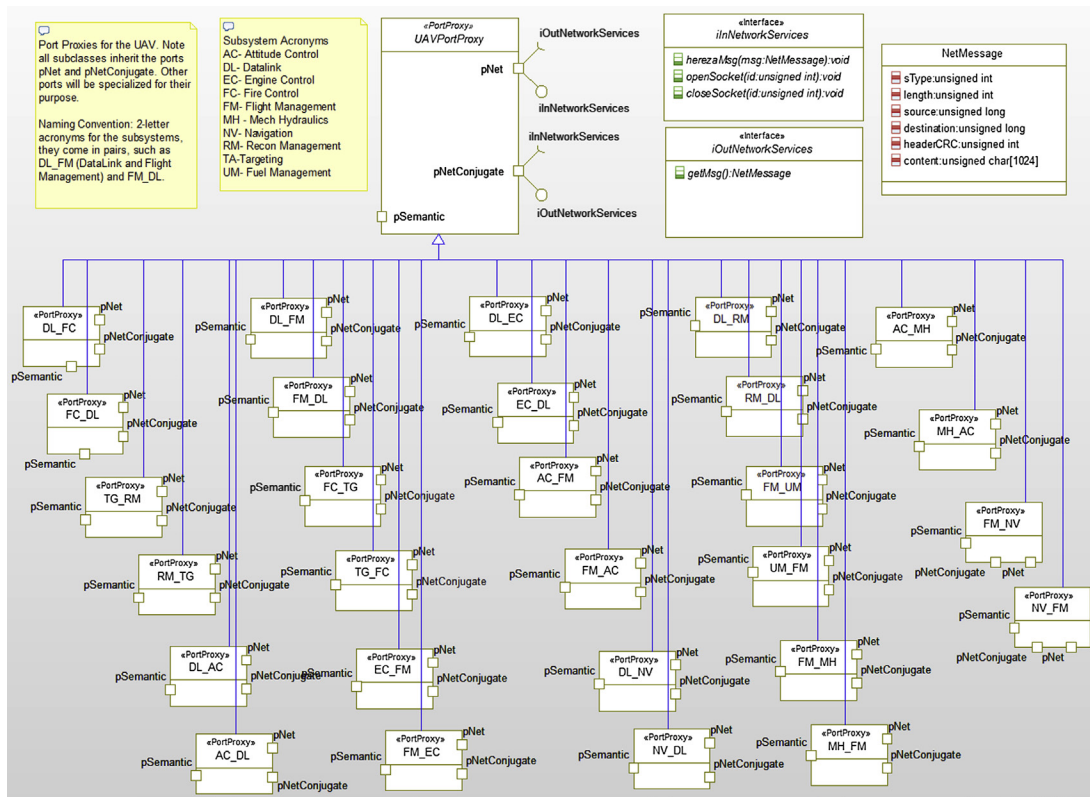


Figure 15.5
UAV port proxies.

subsystems. However, if we use this pattern, we can merely remove the proxies and directly connect the elements from the different subsystems together.

Before we add the proxies into our UAV architecture to create a distribution architecture diagram, let's recall what the architecture looked like before we added the proxies (Figure 15.6).

Figure 15.7 shows the same architecture with the port proxies connecting the elements. It is important to remember that when we instantiate this pattern *no changes are made to the existing software within the subsystems*. All the additions will be made within the proxies themselves. Interfaces are not shown to simplify the diagram and the enclosing Coyote system class is omitted for the same reason.

Figure 15.7 is a bit hard to read (it's easier on screen than in a book) so Figure 15.8 shows a closeup of the elements in the architecture that connect to the Flight Management subsystem.

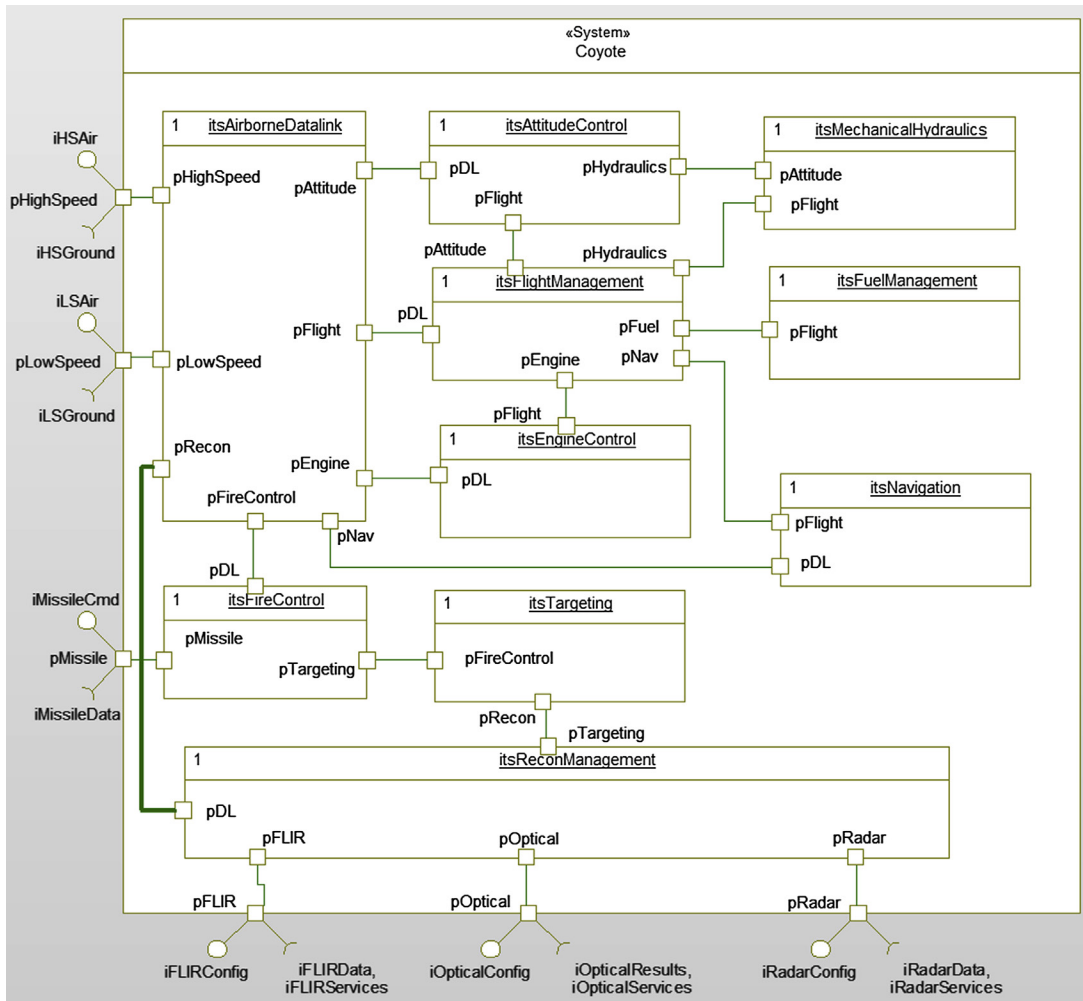


Figure 15.6
UAV subsystem architecture predistribution.

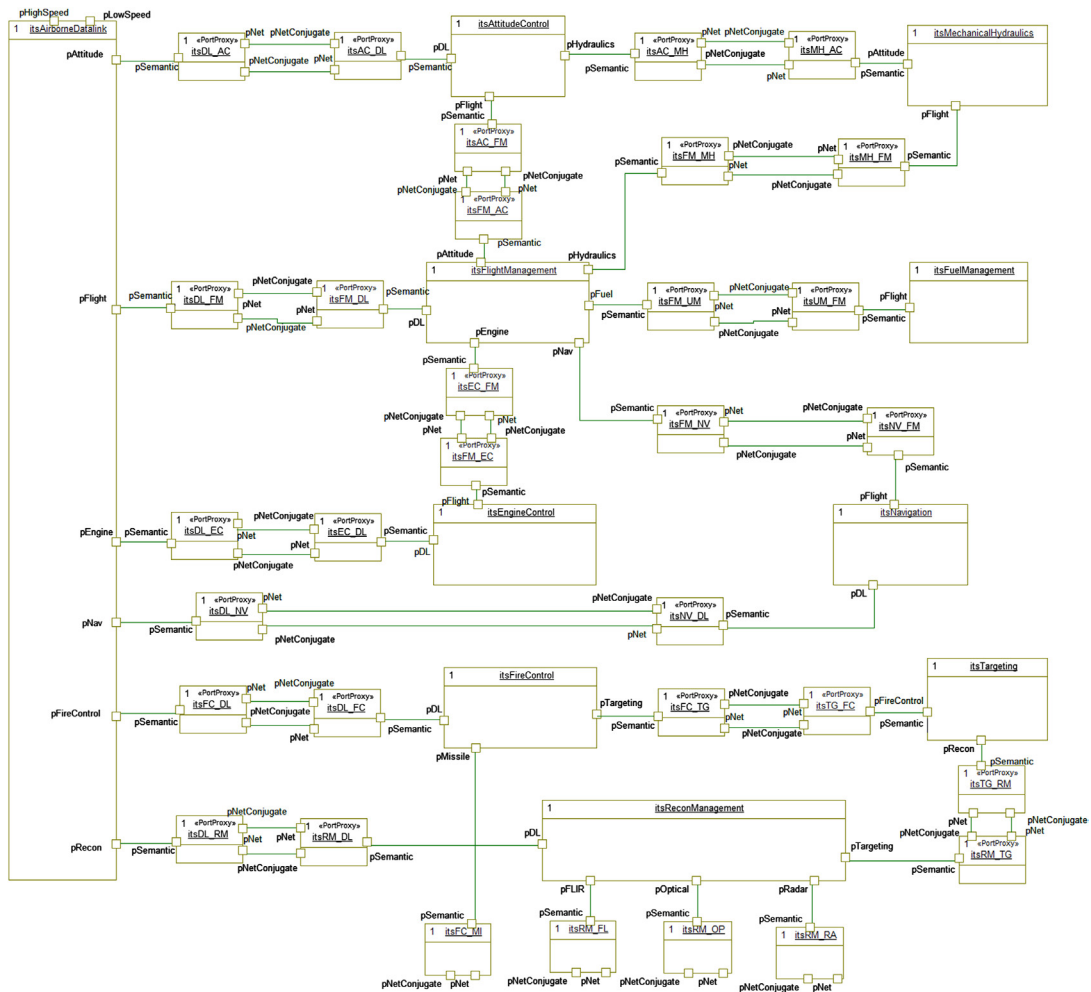


Figure 15.7
CUAV distribution with proxies.

CUAV distribution with proxies (zoomed).

Concurrency and Resource Architecture: Answers

16.1 Answer 9.1: Roadrunner Concurrency and Resource Architecture

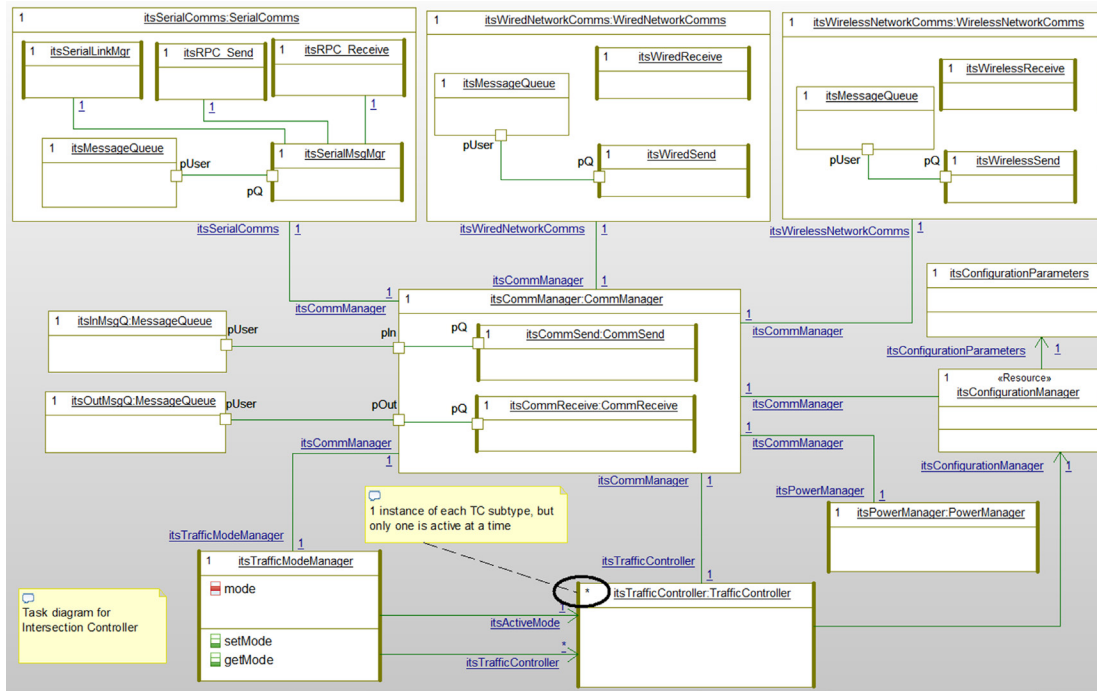
Figure 16.1 shows the concurrency architecture for the Roadrunner Intersection Controller subsystem. In general, for every link, there are two tasks, one for sending messages out and one for receiving them. This is not true for the RS-422 link in which one task does both because of its half-duplex nature. The various links all have a single message queue. The `CommManager` has two queues so it can store both incoming and outgoing messages.

The `PowerManager` runs in its own task as does each of the `TrafficController` instances. Remember that there are several subtypes of this class – one for each traffic mode – and they are all instantiated. However, all are in an inactive state except for one. This is managed by the `TrafficModeManager`, which runs in its own thread so that it can change mode when appropriate, such as when commanded or when a time-of-day event indicates it is time to change mode.

As an aside, note that the concurrency model uses both ports and direct links between instances. A port allows you to delegate directly to an internal part. A link to a composite object means that the composite object itself must delegate the service to its internal part(s) as necessary.

16.2 Answer 9.2: Reconnaissance Concurrency and Resource Architecture

The UAV side of things is only slightly more complex in this exercise. This is because we are modeling threads at the subsystem level. We also haven't considered classes from other use case collaborations. There are a couple of aspects of the collaboration that have not been considered to this point. There are ports for connecting the subsystem (and its contained collaborations) with the Airborne Datalink subsystem and with the various payloads. This is likely to be done via networks – one for the UAV itself and another for connecting to the payloads. Network communications should almost always be decoupled from the execution of the required semantics, so we need two network interfaces to operate in separate threads. Additionally, the configuration of the imaging and platforms operates independently from the

**Figure 16.1**

Roadrunner Intersection Controller concurrency architecture.

image processing, so the objects facilitating these functions should be in separate threads. Finally, targeting, a topic only lightly touched on so far, is clearly independent as well, and so will be in yet another thread. The concurrency model for the Reconnaissance Management subsystem is shown in [Figure 16.2](#).

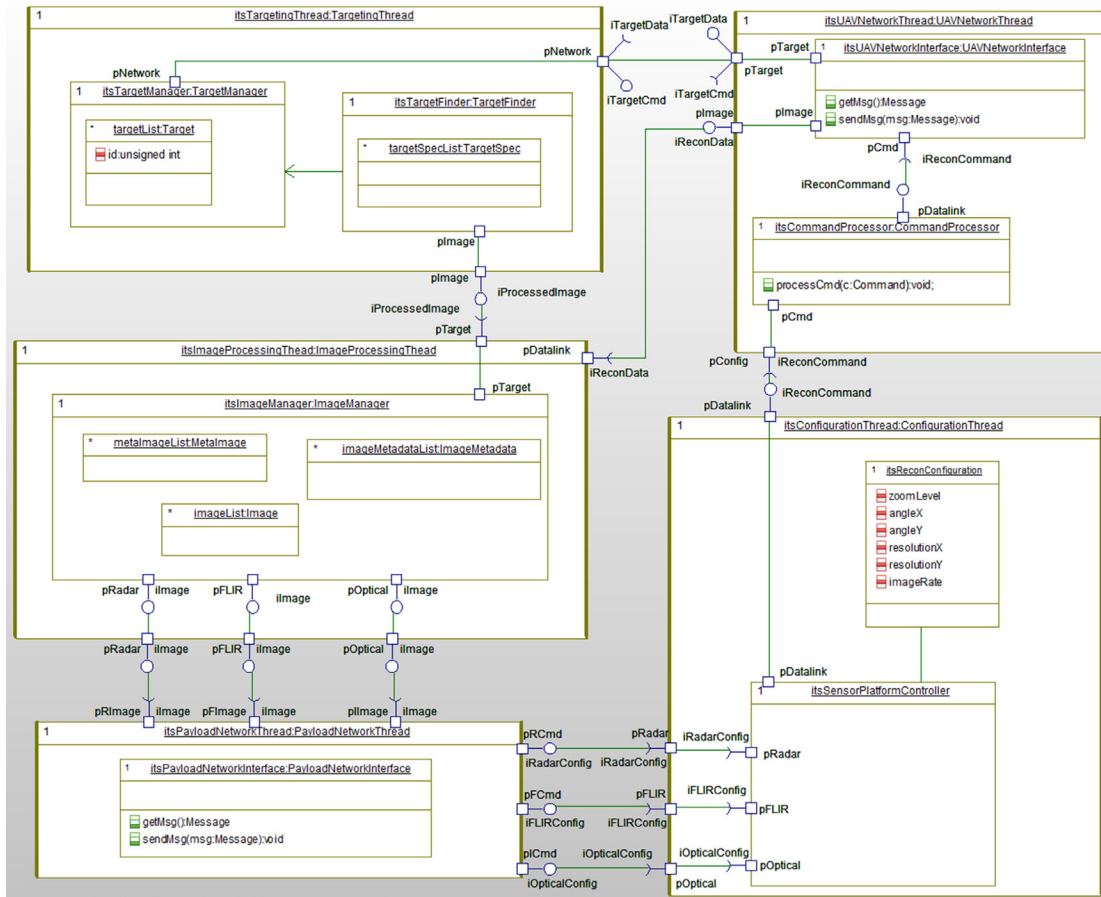


Figure 16.2
Reconnaissance subsystem concurrency model.

Collaboration and Detailed Design: Answers

17.1 Answer 10.1: Applying Collaboration Design Patterns: Part 1

For the Roadrunner portion of this exercise, the application of the adapter pattern is very straightforward. The design class diagram is shown in [Figure 17.1](#).¹ Two example physical interfaces are shown (`Acme_PLI` and `PenultimateRadarSensor`) and adapters are introduced to adapt their interfaces to the existing collaboration. Simple implementations are shown in comments.

The CUAV problem required the application of the chain of responsibility (CoR) pattern to make the use and addition of new image processing algorithms easier and more flexible. The first question is where the image processing takes place. Two obvious candidates are under the control of the `ImageManager` or under the control of the `TargetFinder`. The `ImageManager` class is responsible for early image processing, which would naturally include sharpening and enhancement of the images. On the other hand, the `TargetFinder` might employ different enhancement methods to find different kinds of targets, so an argument can be made for both. It is even possible to apply the CoR pattern twice, once for basic image enhancement and again for target-specific enhancements.

In this case, we'll apply the pattern to the `ImageManager` only. In terms of image processing, we'll include not only image enhancement, but also conversion to a compressed (JPEG) format, encryption, averaging, and center-surround sharpening as image processing steps. We also have a need to undo these steps, so we've applied the pattern twice – once to do image processing and once to undo some of the steps, such as encryption and compression. As with the Roadrunner pattern application, we only show the relevant portion of the collaboration.

You'll note in [Figure 17.2](#) that some of the operations defined for the `ImageHandler`, such as compression and encryption, have been removed since they are now handled by the image handlers in the chain of responsibility. This small reorganization of responsibilities is an example of the refactoring we referred to previously.

¹ Parts of the collaboration that weren't affected are elided from the diagram for simplification.

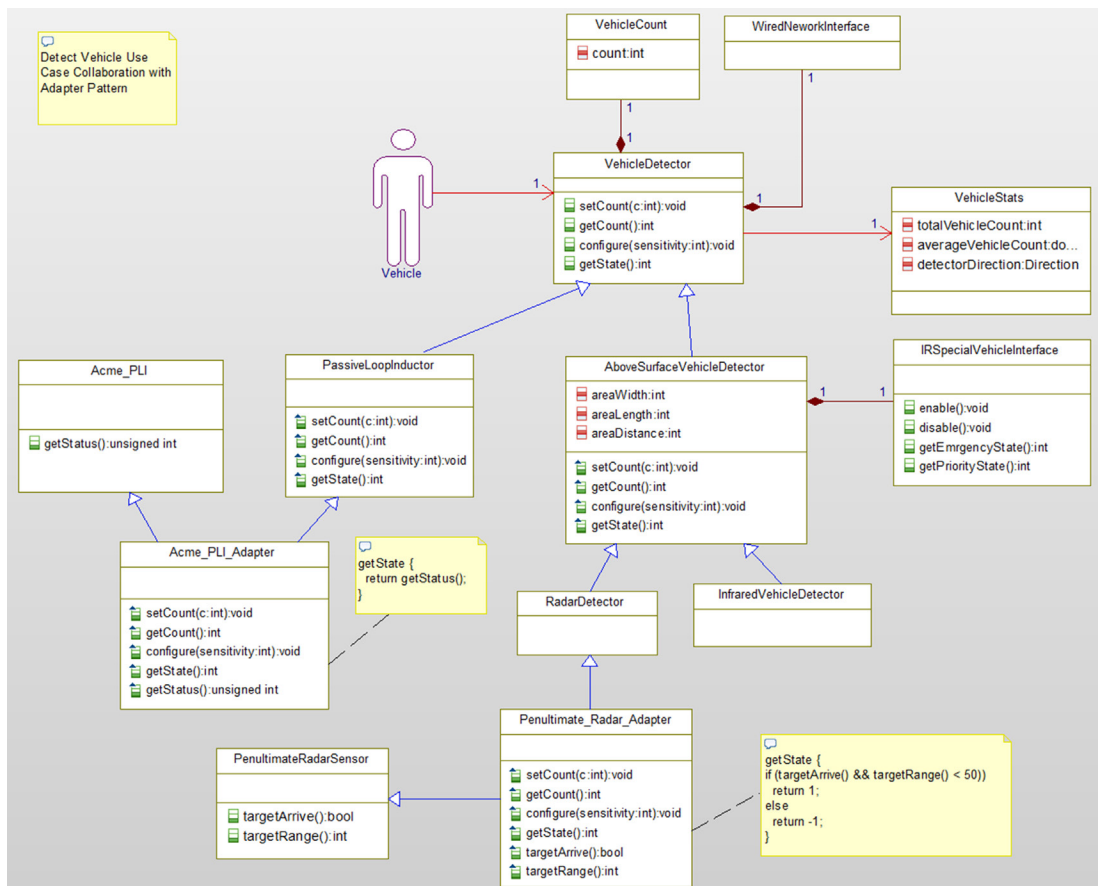


Figure 17.1
Detect Vehicle adapter pattern.

17.2 Answer 10.2: Applying Collaboration Design Patterns: Part 2

The observer pattern optimizes the interaction between client and server so that additional clients can be added, and so that data is only sent when it is appropriate to do so. Each light assembly coordinates the traffic for a single road and direction for both through and turn traffic, so there are two light assemblies per road, for a system total of four. There are also a total of eight pedestrian light assemblies (two per road per side). All these light assemblies must share information to ensure that the overall system is in a safe state at all times (specifically cross-oriented traffic does not have non-red or non-Don't Walk signals at the same time). [Figure 17.3](#) is an object diagram showing the connections among the light assembly instances. The instance naming convention assumes the primary road is North-South and the secondary road is East-West. Thus, the name for the vehicle light instance on the primary road that faces North is `vPrimaryNorthFacingLight` and the walk light for the secondary road

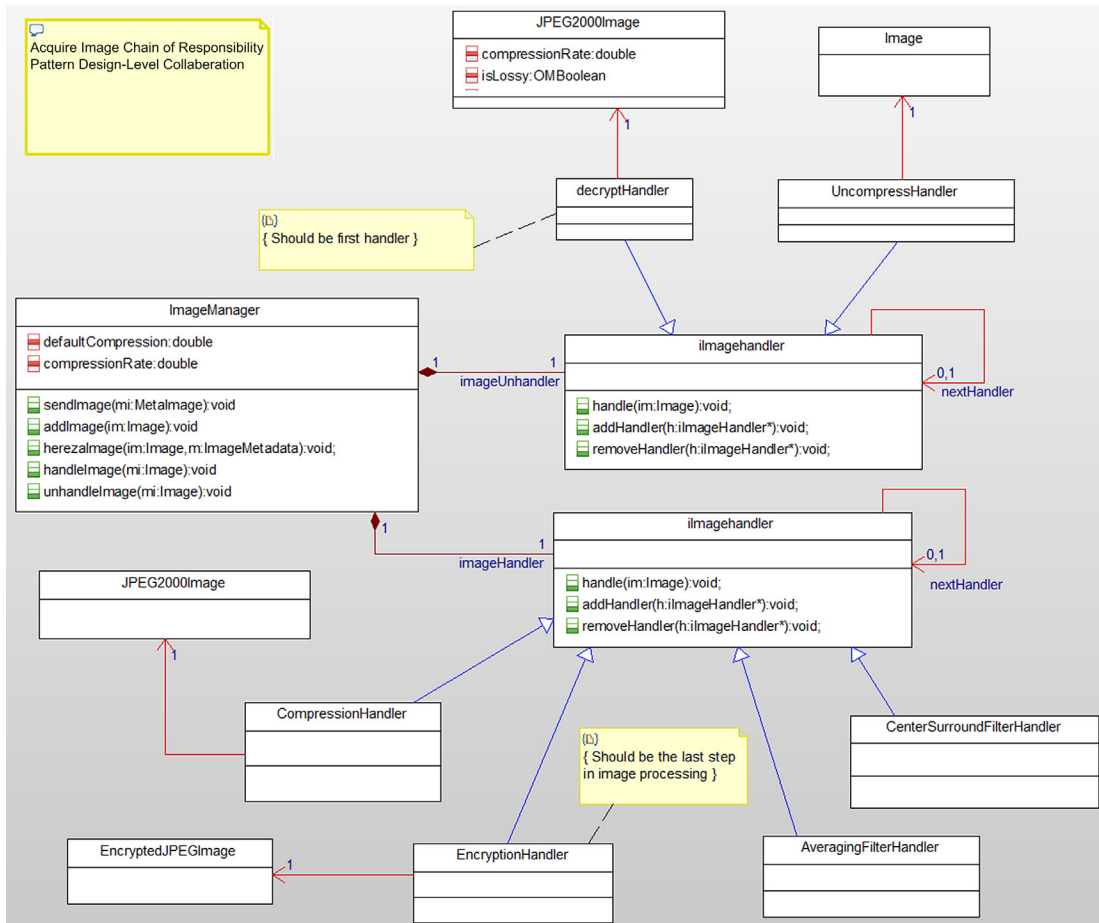


Figure 17.2
Acquire Image chain of responsibility pattern.

that faces East and is on the North side of the street is called `pSecondaryEastFacingNorthSide-Light`. It's verbose but has the advantage of being unambiguous.

Figure 17.3 shows the link topology of the system.² The Intersection Controller is in the middle, the white-filled objects are the vehicle lights, and the shaded boxes are the pedestrian lights. As you can see, the topology is complex and validating that all the lights are connected to all the correct clients would be a chore.

When we apply the data bus pattern to all the lights and clients (they are largely the same elements, after all), the topology is much more straightforward, because the lights and clients need only a single link to the data bus instance. The responsibility for the topology creation is

² Extra credit to the reader who validates all the links ☺.

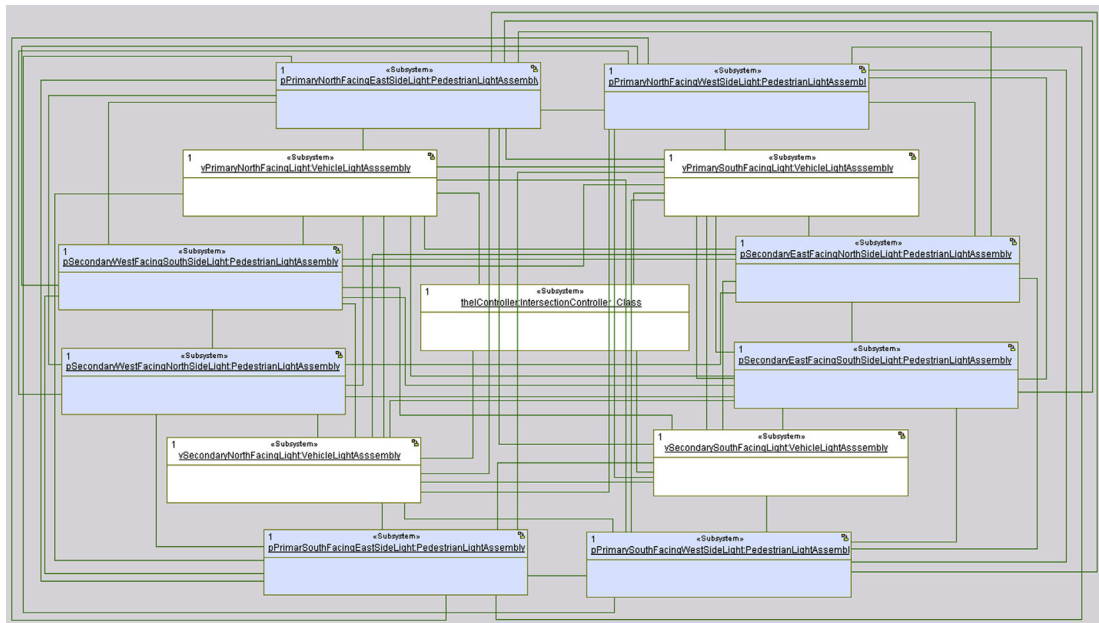


Figure 17.3
Light assembly connection topology.

distributed to the clients and servers. Each light assembly is a server, so it must link to the data bus instance and call `update()` to update its light state. Each client must subscribe to all the other lights. Note that the Intersection Controller is the special case because it is not a light but is a client to all the light assemblies.

The data structure is simple – nothing more than an enumerated value containing all of the possible states for the light. Two different enumerated types are needed – one for vehicle lights and one for pedestrian lights.

Each light assembly then maintains a (local) list of all the relevant (cross-traffic) lights. When those lights change state, the data bus updates the light assembly, which in turn updates its (local) copy. The behavior of each light then checks consistency with the other lights, something like this:

```
//Wants to change state to permit traffic to flow (i.e., yellow or green)
Set a global state change semaphore (also in the data bus, but updated by all
lights)
For all cross traffic vehicle lights
  If it is not red,
    send an error message to the Intersection Controller
    release the global semaphore
exit
```

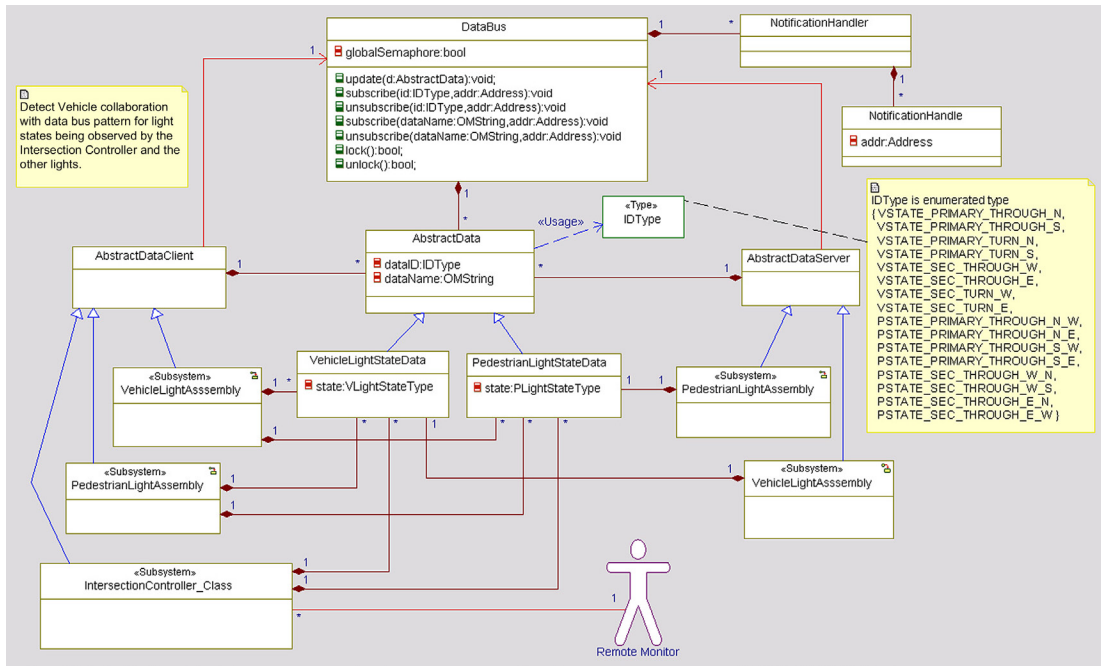


Figure 17.4
Detect Vehicle with data bus pattern.

```

For all cross-traffic pedestrian lights
    If it is not Don't Walk
        send an error message to the Intersection Controller
        release the global semaphore
        exit
    Change state
    Release the global semaphore
    
```

This can be optimized somewhat by observing that certain light sets change together; for example, the walk signals that face each other on the same road and side of the street change together. Such light sets can be controlled by a single state change. However, the light set construction may be nontrivial. For example, it is not necessarily true that through lights facing different directions on the same road change together; if the turn lanes are operated in SEQ mode, then the turn light in one direction is set when the through lanes in the same direction are on but the opposing traffic on the same road must be prevented from going at that time. So, in some cases, such light sets are mode specific.

Figure 17.4 shows the Detect Vehicle collaboration updated to include the data bus pattern. Note that the shared data are the light states – there are two data classes represented; one for the vehicle light states and one for pedestrian light states. Note that the clients and server light assemblies are replicated on the diagram – this is simply to minimize line crossing. On the

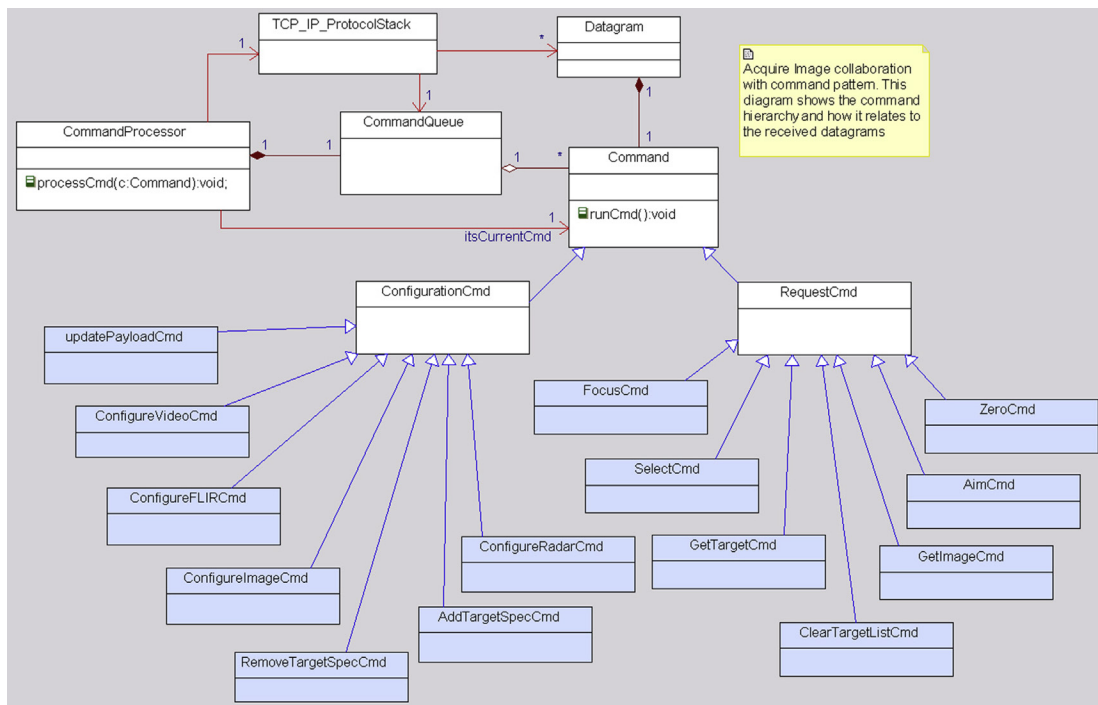


Figure 17.5
Acquire Image command structure.

server side, they have a composition with a unary multiplicity to the data they are providing, while on the client side, they have a * multiplicity to the data they get from their peers.

The `dataID` is an enumerated type that identifies the road (`PRIMARY` or `SEC` for secondary), whether it is `THROUGH` or `TURN` (vehicle lights only), whether it faces North, South, East, or West (N, S, E, W), and in the case of the pedestrian lights, which side of the road it's on (the last character N, S, E, or W). This differentiates the data from one server from all the others. Note that there will be at least one instance of light state data for each enumerated value in the `IDType`.

The global semaphore was added as an attribute of the data bus because it affects all data – a change in the state of a single light is fundamentally a change in the system state. The `lock()` operation returns `TRUE` if the attempt to lock was successful and `FALSE` otherwise; similarly, the `unlock()` operation returns `TRUE` if the attempt to unlock was successful and `FALSE` otherwise (although this should never happen unless the system contains an error).

While the topology shown in Figure 17.4 may seem complex, in reality it is an order of magnitude simpler than the topology shown in Figure 17.3 at the cost of a few additional objects and the memory that they require.

The second half of this problem requires the command pattern be applied to the Acquire Image collaboration. The command pattern is straightforward and requires that the command hierarchy be identified as well as how the commands invoke services on the existing elements in the Acquire Image collaboration. In addition, it is necessary to specify how the command objects come into existence.

The solution is shown in two different class diagrams. The first, [Figure 17.5](#), shows the generalization taxonomy for the command classes and how the commands are created. Specifically, they get passed to the command processor via the TCP/IP protocol stack encapsulated into datagrams, one command per datagram. Once the datagram is received, the command is extracted and placed into a command queue for processing. The commands themselves are specialized into two kinds – configuration commands and direct requests for actions or data.

Each of the specific commands is a specialized form of command and contains relevant data and a specialization of the `execute()` command to invoke the appropriate service of the elements of the collaboration. The command processor has the responsibility to create the link between the command instance and the appropriate collaboration object and then invoke the `Command::runCmd()` method. [Figure 17.6](#) shows the associations from the various commands (shaded for emphasis) to the collaboration classes.

17.3 Answer 10.3: Applying Detailed Design State Behavioral Patterns

The any state pattern is usually simple to apply and the situation with the `TrafficModeManager` state machine is no exception. Before we look at the solution, let's look at what the high-level state machine would look like if we don't use the pattern. [Figure 17.7](#) shows the resulting rat's nest of transitions.

With the applied pattern, the state machine is simpler and far easier to understand, as you can see in [Figure 17.8](#). The `TrafficModeManager` can transition from any mode to any other mode directly, just as in the previous figure, but this one can actually be read!

For the CUAV part of this problem, the first step is to make the `ImageManager` a composite class containing the device drivers as parts. [Figure 17.9](#) does this and links the appropriate device driver to the correct ports.

Now, we want to specify how the classes of these parts relate to each other; specifically, how they are subclasses of a base class called `SensorPayloadDeviceDriver`. [Figure 17.10](#) shows this. Note, by the way, the small state icon in the upper right-hand corner of the classes. The model includes a state machine for the `SensorPayloadDeviceDriver` class and because the specific device driver classes are subclasses of this base class, they automatically inherit its state behavior.

The next step is to elaborate the state machine for `SensorPayloadDeviceDriver`. This state machine is shown in [Figure 17.11](#). The state machine uses actions on transitions as well as

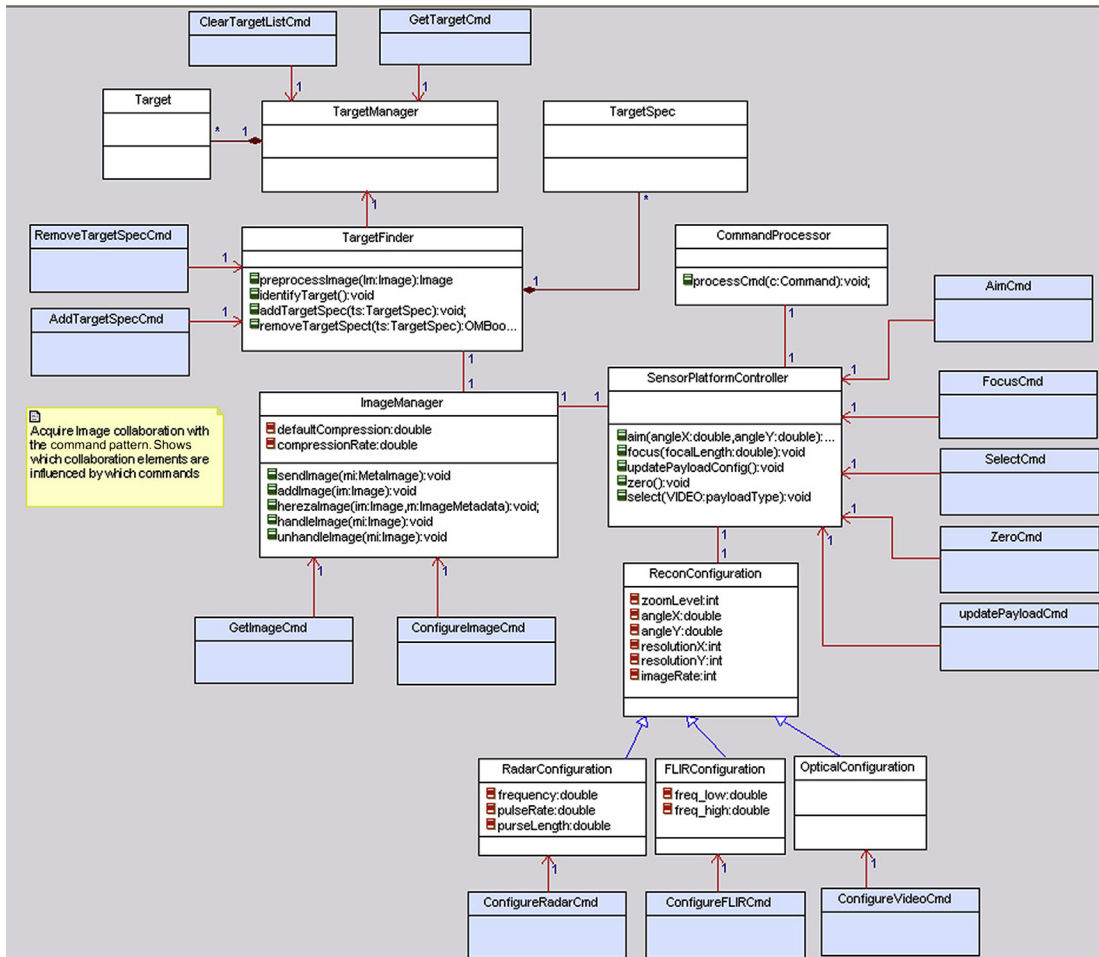


Figure 17.6
Acquire Image command pattern.

state entry actions. If desired, the state machine can be written using only actions on transitions or only state exit actions.³

The last aspect of this problem is to add the polling state pattern to this state machine to separate the semantic actions and states (shown in Figure 17.11) from the polling behavior. The result is shown in Figure 17.12.

³ Exercise for the reader ... ☺.

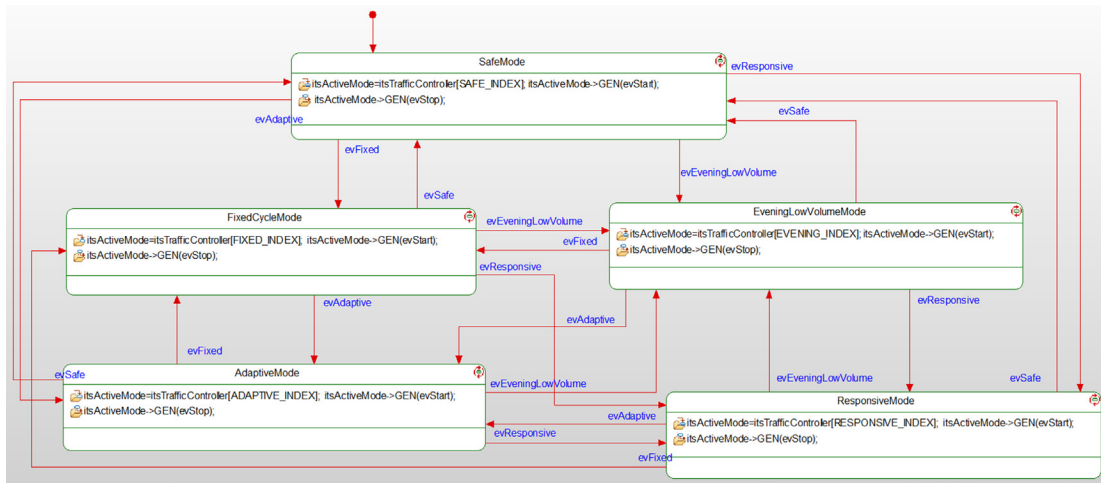


Figure 17.7

TrafficModeManager state machine before application of any state pattern.

This is a relatively straightforward application of the polling state pattern. The upper and-state replicates the semantics of the previous state machine. The lower and-state realizes the polling behavior. The use of the conditional pseudostate allows different transition branches to be taken based on the result of the condition of the `success` value returned from the `acquireData()` operation. Note the use of an anonymous state (`state_8`). Why is that there?

UML state machines provide what is called *run-to-completion* behavior. What this means is that each transition, including the guard, predecessor state exit actions, transition actions, and subsequent state entry actions are run completely before the next event is processed. However, the guards are executed *first* before any actions are executed. This means that if the conditional pseudostate is connected to the `tm()` transition, the value of `success` will be used before it is set by the execution of the `acquireData()` action. We need some way to force the execution of this action before the guards are evaluated. One solution, used in Figure 17.12, is to put a state in between the conditional pseudostate and the `tm()` transition. Because the `tm()` transition terminates on a state, all the actions are executed. The transition from `state_8` to the conditional pseudostate is null-triggered, so it fires immediately after the run-to-completion step. Since the previous step assigned `success` a value, it can be used in the guards.

Note however, a possible pitfall. If `success` returns a value other than those listed in the exiting transitions, then the state machine would be stuck in `state_8` because there would be no correct exiting transition. For this reason, we've included an `else` transition guard to handle both the case `NONE` and any unexpected returned value.

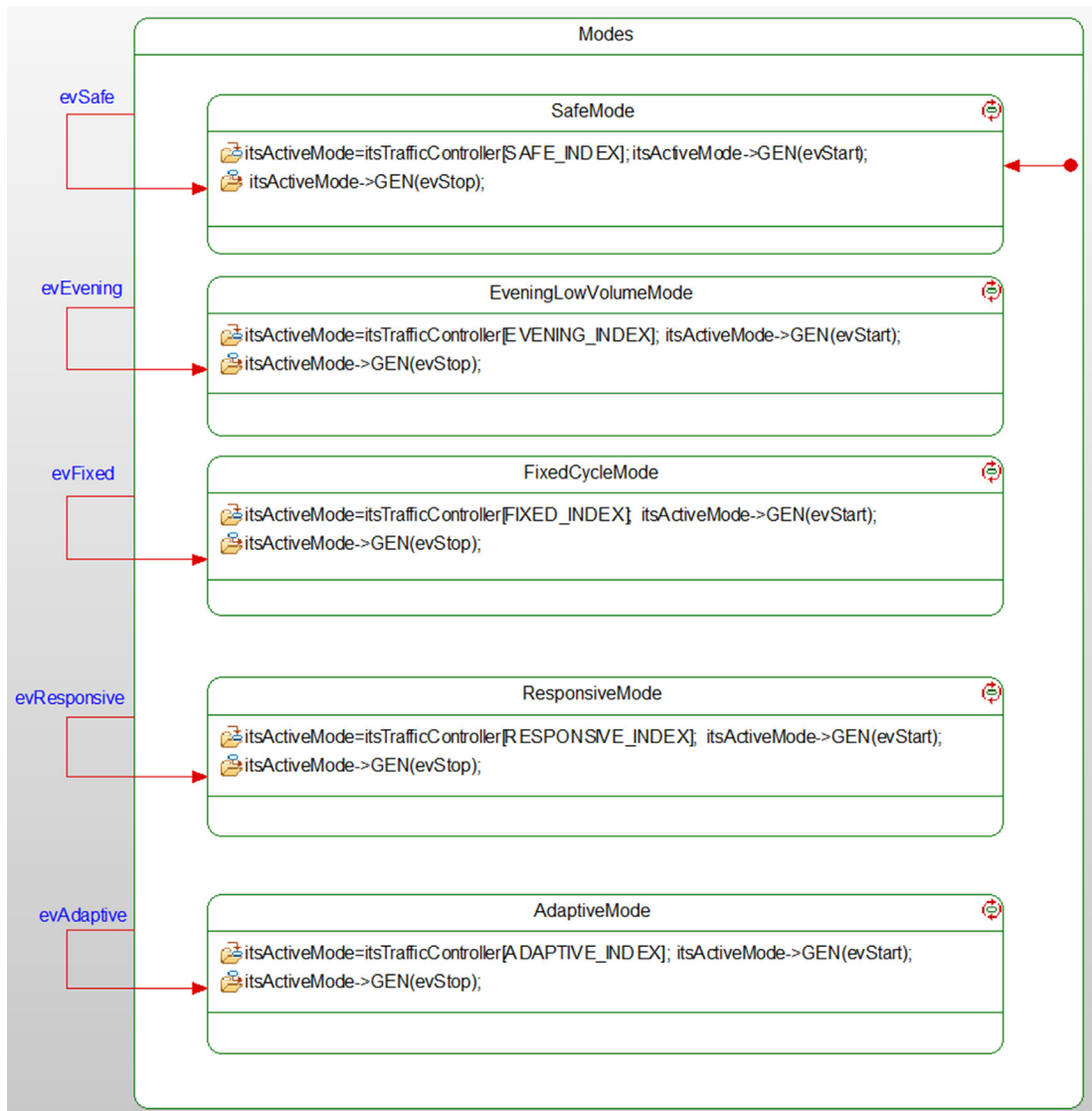


Figure 17.8
TrafficModeManager with any state pattern.

17.3.1 Additional Work

1. Fill out the remaining states for the Safe, Evening, Adaptive, and Responsive light modes by detailing the state machines of the appropriate subclasses of `TrafficController`.
2. Find other classes in the collaboration that are “reactive” (i.e., exhibit state behavior) and construct state machines for them.

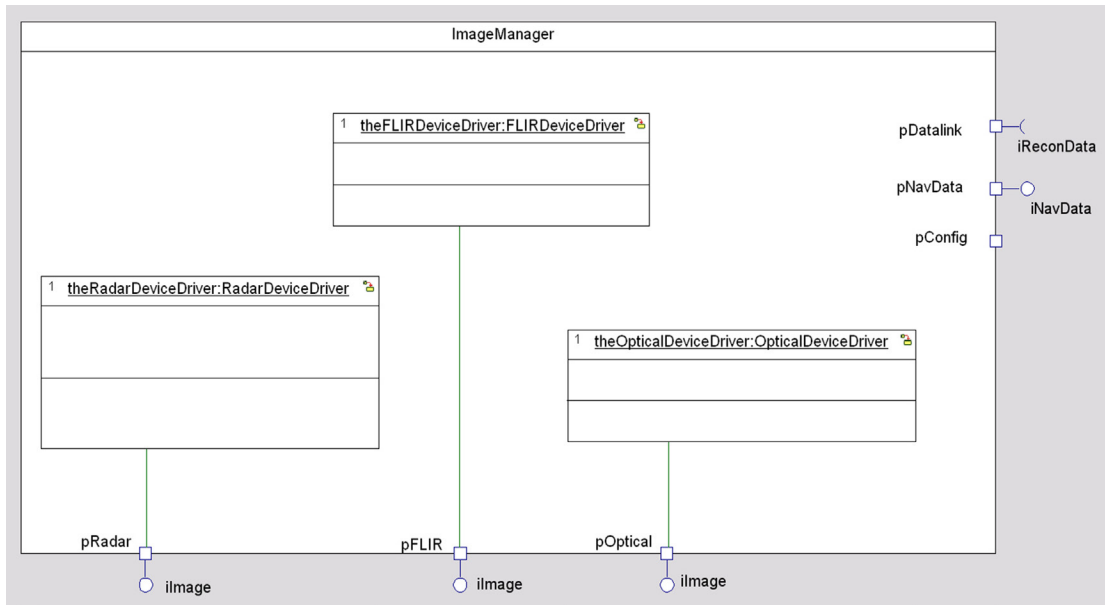


Figure 17.9
ImageManager with parts.

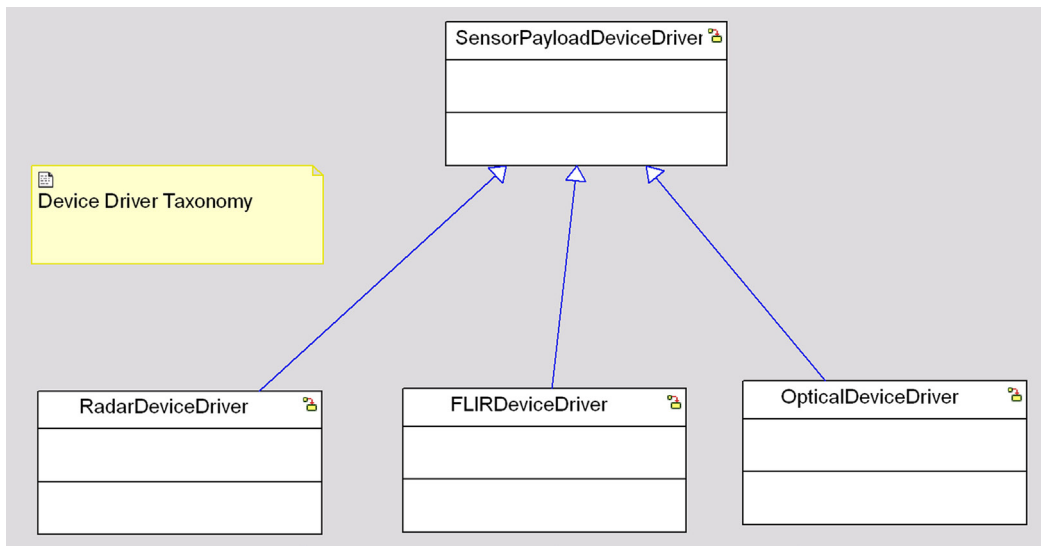


Figure 17.10
Sensor device driver taxonomy.

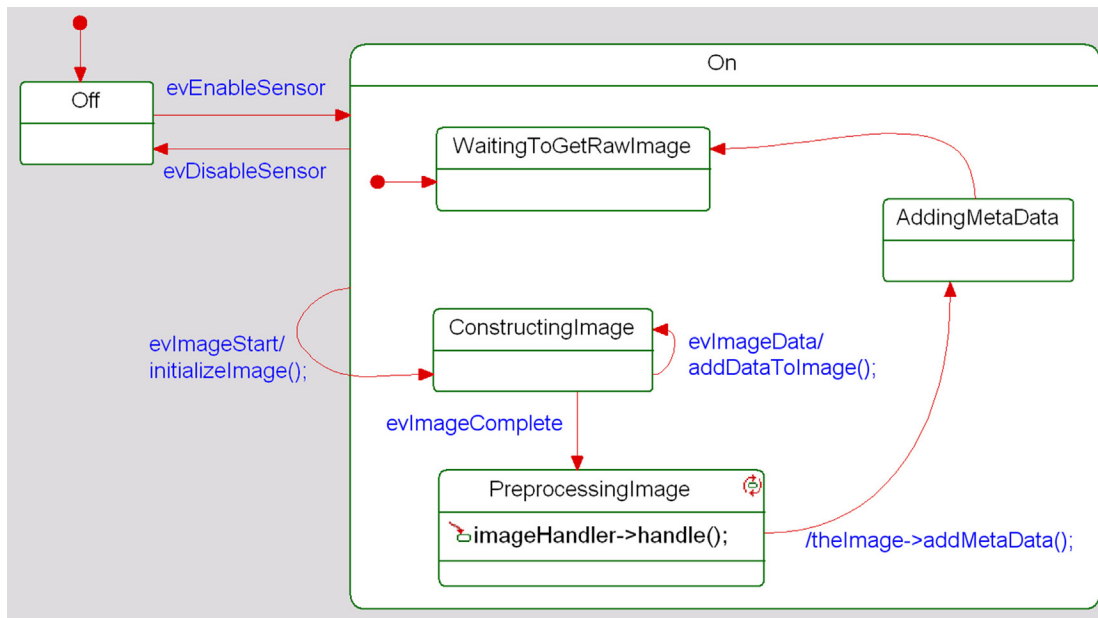


Figure 17.11
Sensor payload device driver state machine.

17.4 Answer 10.4: Applying Detailed Design Idioms

The solution to this problem lies in creating state machines for the lights that can generate or handle the appropriate exception events and go to the appropriate fail-safe state when a fault occurs. The first state machine is for the vehicle light, shown in [Figure 17.13](#). The `isOnGreen()`, `isOnYellow()`, and `isOnRed()` operations invoke the services of the appropriate photo detector and return `TRUE` if they detect that the specified light is on and `FALSE` otherwise. The `exNoGreen`, `exNoYellow`, and `exNoRed` events represent the exceptions that are raised if a particular light is out. The appropriate action in this case is to go to the `Red` state; if the red light is out, then no safe action can be taken.

As can be seen in [Figure 17.14](#), the pedestrian light state machine is very similar, if slightly simpler.

The vehicle light assembly and pedestrian light assembly, respectively, handle the events as appropriate. The first of these is represented by `VehicleLightAssembly`, shown in context in [Figure 17.15](#). Note that the system has four instances of this subsystem – one for each through direction and one for each turn direction. Each assembly controls multiple lights – a set of red, green, and yellow lights for each lane (including both directions). The assembly also has a set of photodetectors to allow determination of the success of attempts to turn on lights (via the `isOnGreen()`, `isOnYellow()`, and `isOnRed()` operations that invoke the operation `Photodetector::isOn()`). Also note that the

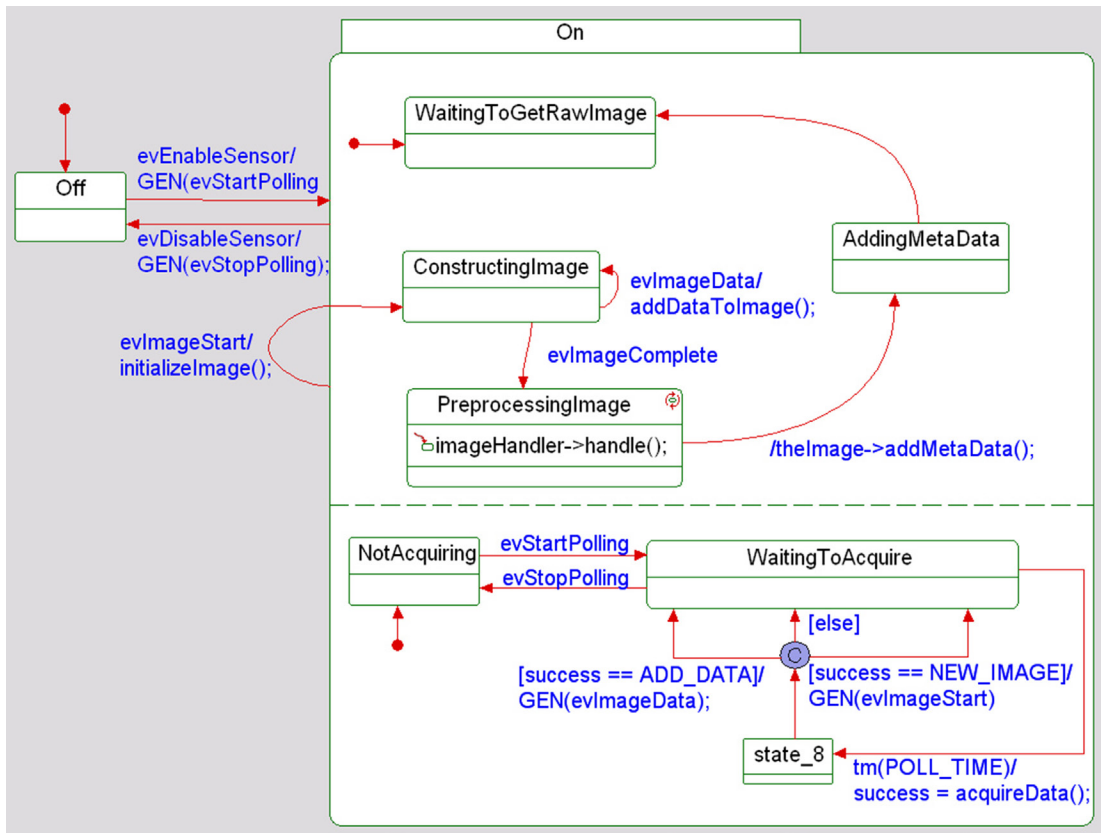


Figure 17.12

Sensor payload device driver with polling state pattern.

events `exNoRed`, `exNoYellow`, and `exNoGreen` are subclasses of event `exLightFault`. That means that if any of the subclassed events are sent by the `VehicleLight` to the `VehicleLightAssembly`, they can all trigger a transition triggered by the event `exLightFault`.

The state machine for the vehicle light assembly is shown in Figure 17.16. A fault state has been added; it is differentiated from the safe state so that the data bus can easily indicate to the clients the situation.

The last state machine, shown in Figure 17.17, is for the pedestrian light assembly. This state machine incorporates the exception state pattern in the same way as `VehicleLightAssembly`.

The CUAV problem is arguably collaboration rather than detailed design, but is nevertheless appropriate for the chapter and the application in question. The application of the pattern is very straightforward. Figure 17.18 shows the various sensor device drivers connecting to ports. These ports will be connected to an instance of the watchdog class at runtime. This is

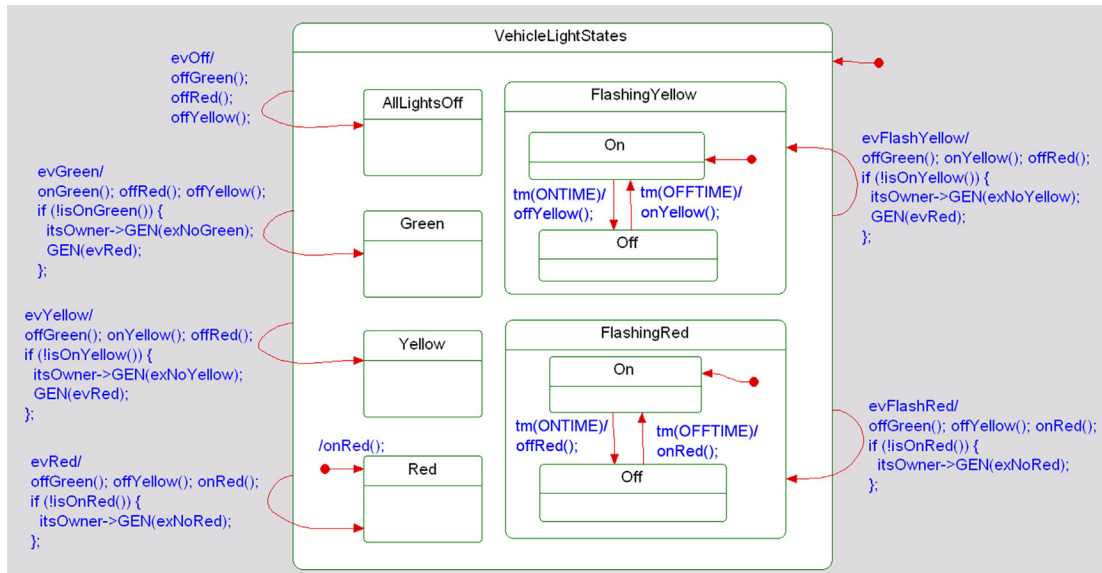


Figure 17.13
Vehicle light state machine.

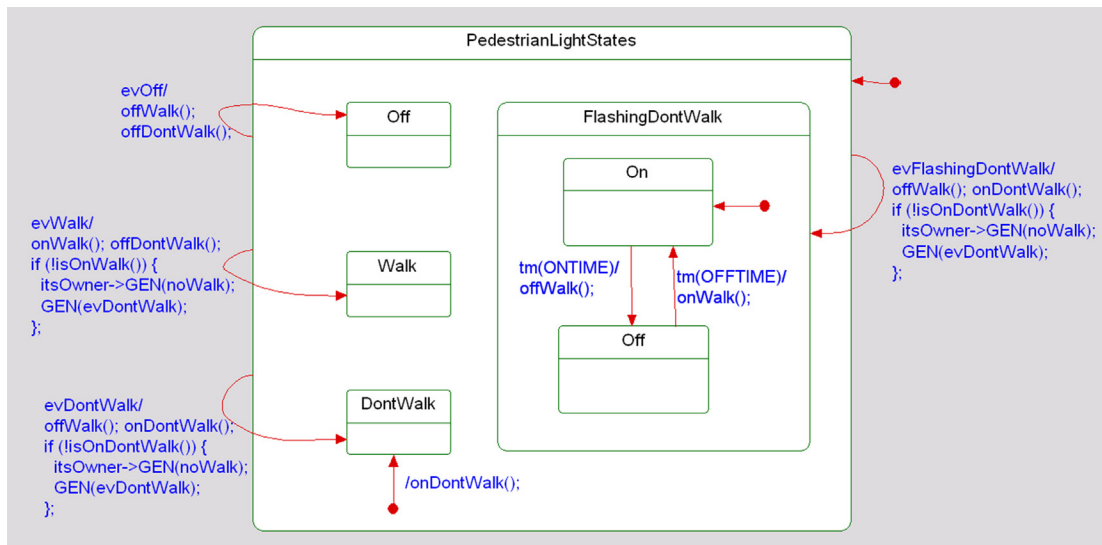


Figure 17.14
Pedestrian light state machine.

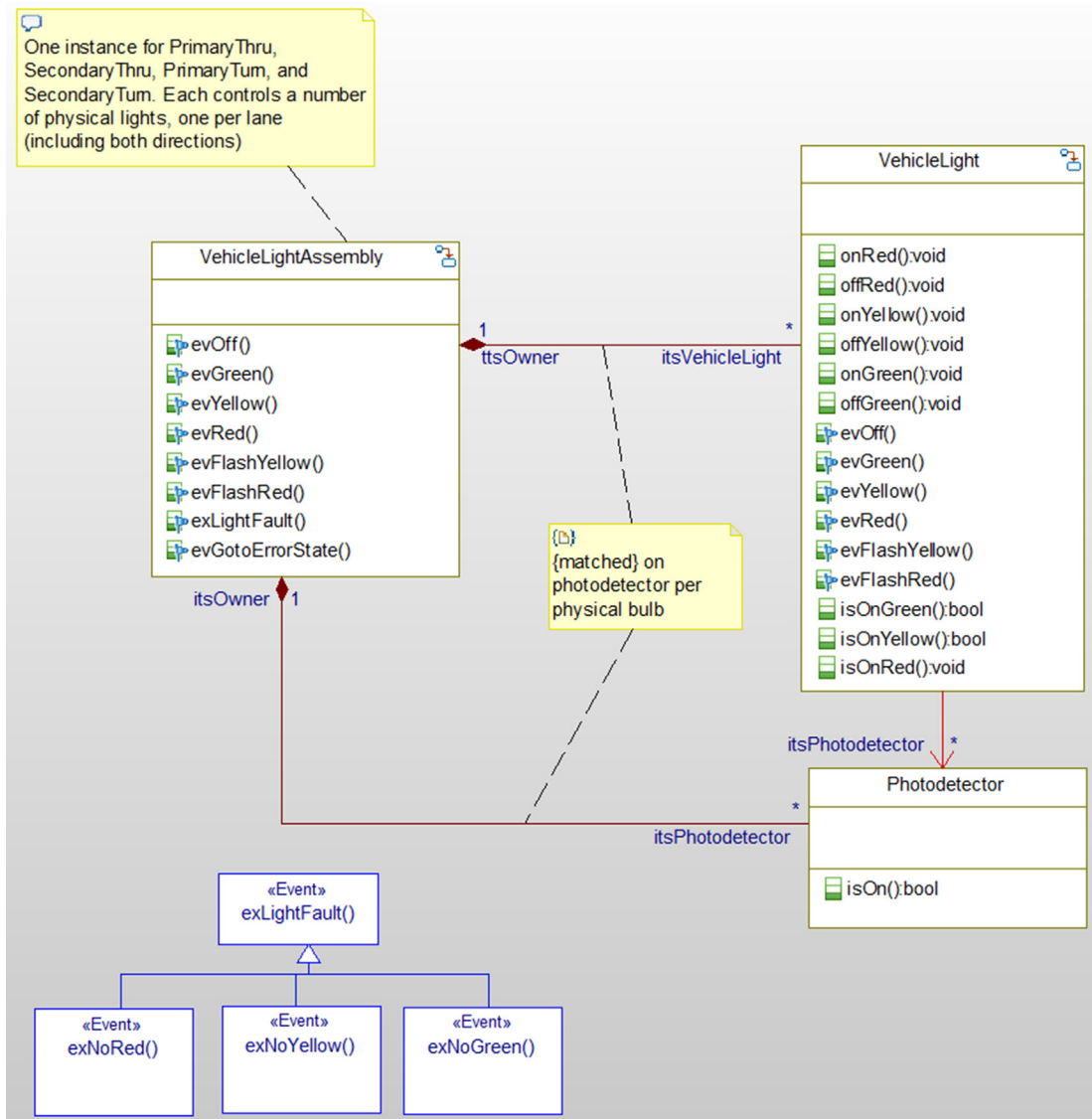


Figure 17.15
VehicleLightAssembly in context.

implied by the association from the `ImageManager` class near the port. Because the device drivers must be a subclass of `WatchdogClient`, this is shown in a separate diagram, [Figure 17.19](#). The safety executive has an association to the `ImageClient` and so may destroy it and create a new one should the watchdog fire. If desired, the safety executive could also signal the Reconnaissance Management subsystem to perform that task as well.

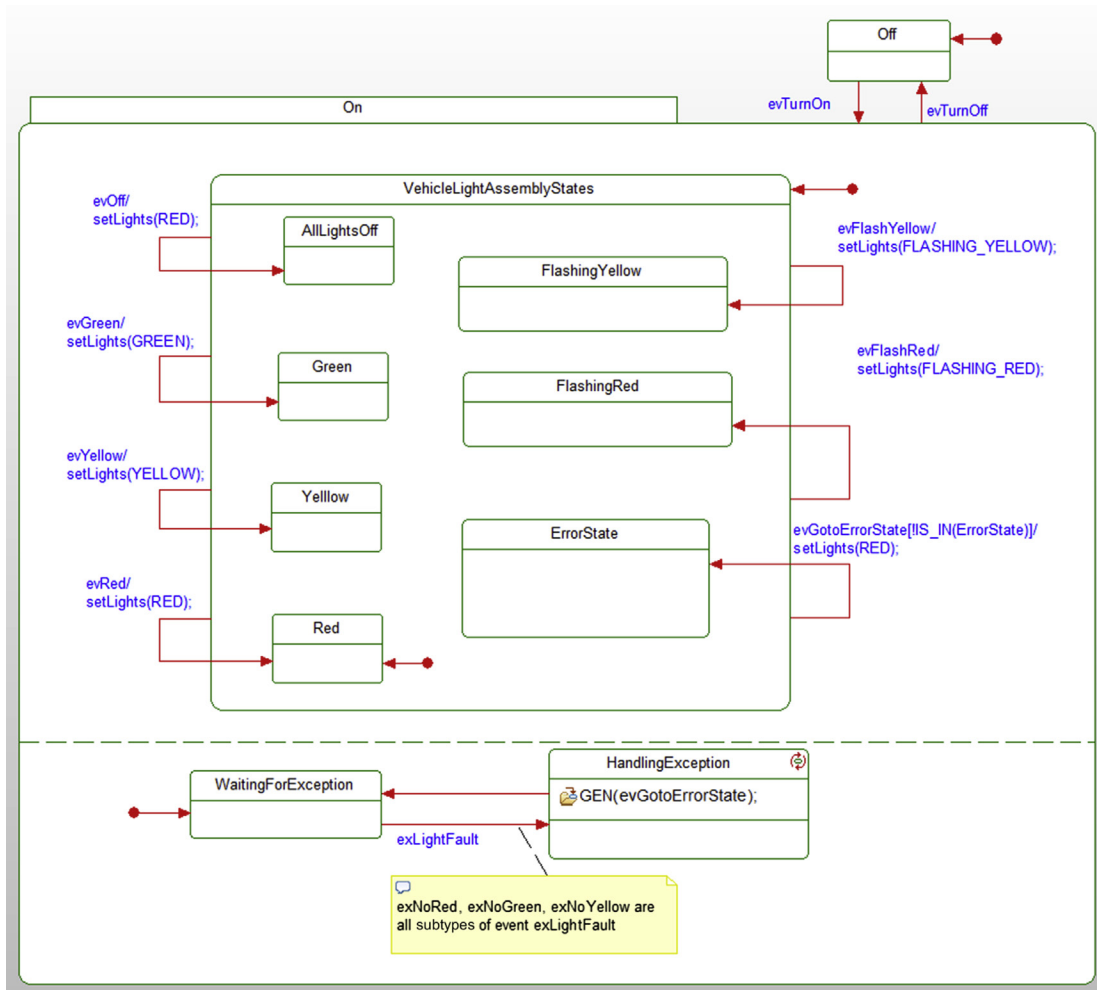


Figure 17.16
Vehicle light assembly state machine with exceptions.

The state machine for `WatchPuppy` is unchanged from the pattern specification, but is shown in [Figure 17.20](#).

Lastly, `SensorPayloadDeviceDriver`'s state machine is modified to stroke the watchdog. Because the device driver polls periodically, that is a convenient place to add the watchdog stroke behavior. The updated state machine is shown in [Figure 17.21](#).

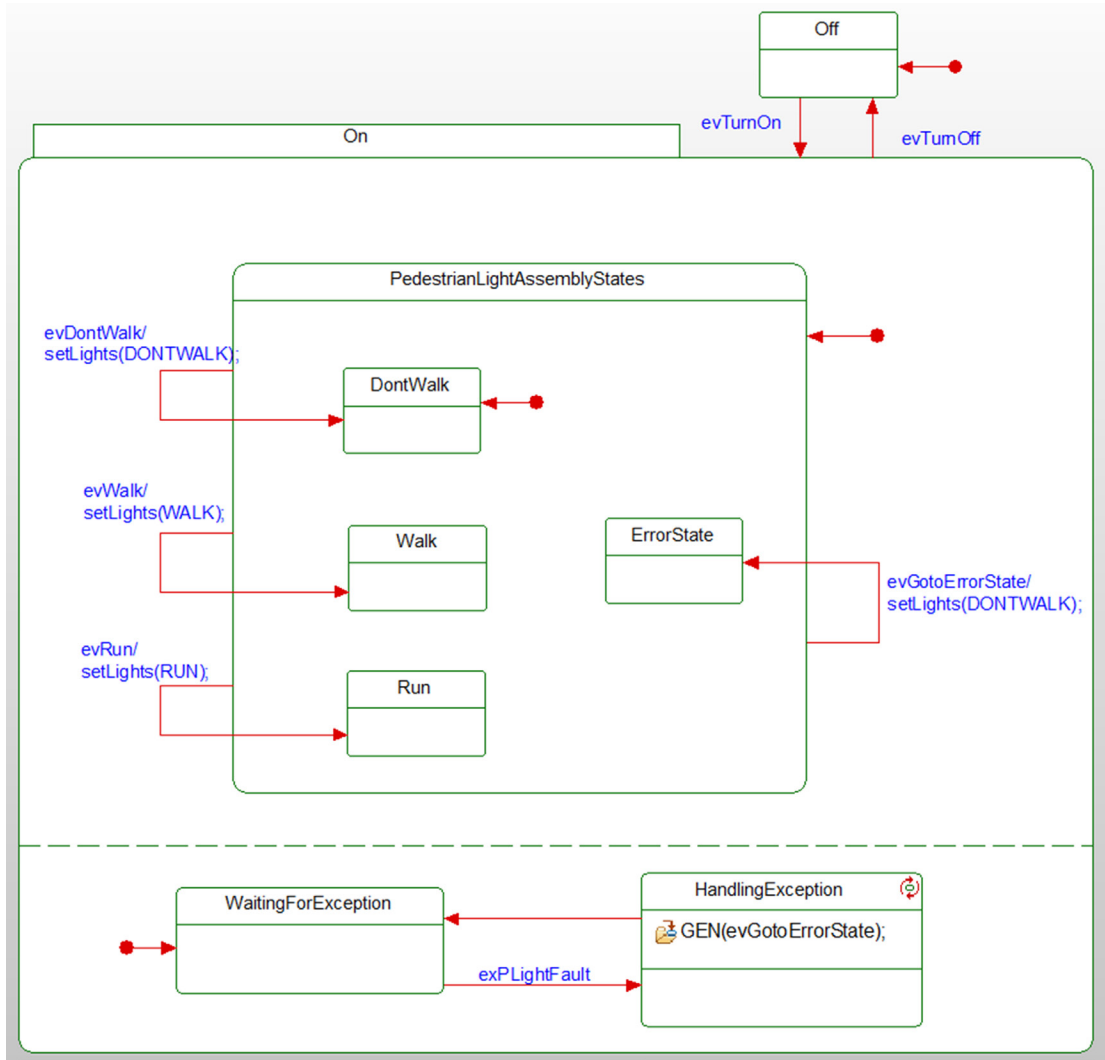
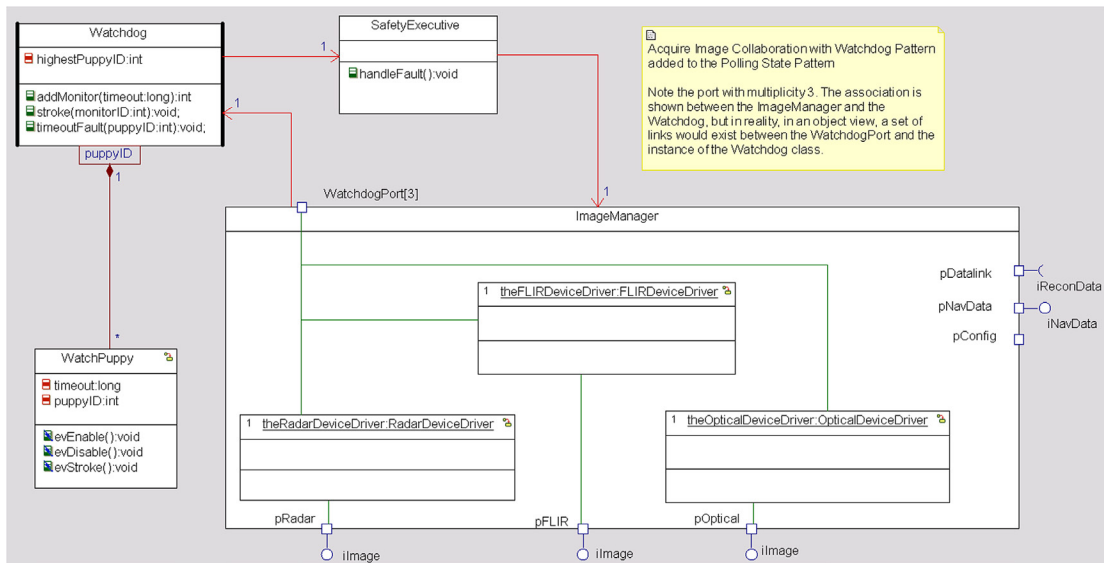
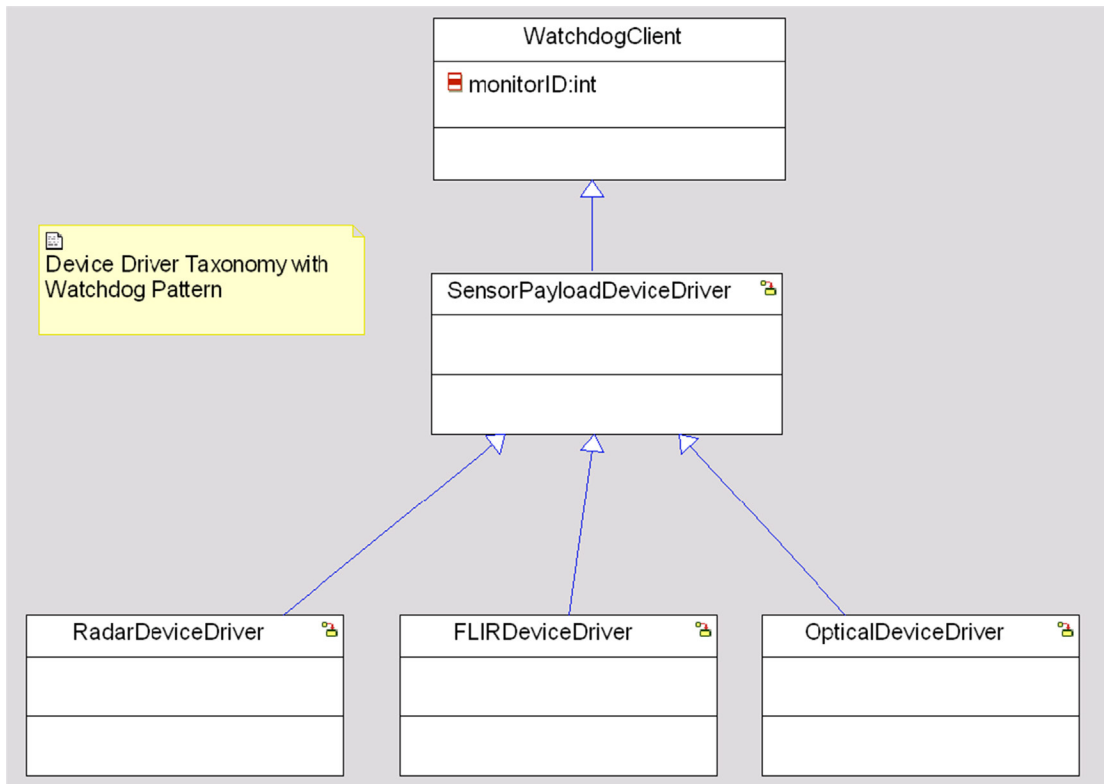


Figure 17.17
Pedestrian light assembly state machine with exceptions.

**Figure 17.18**

Acquire Image collaboration with watchdog pattern.

**Figure 17.19**

Sensor device driver taxonomy with watchdog pattern.

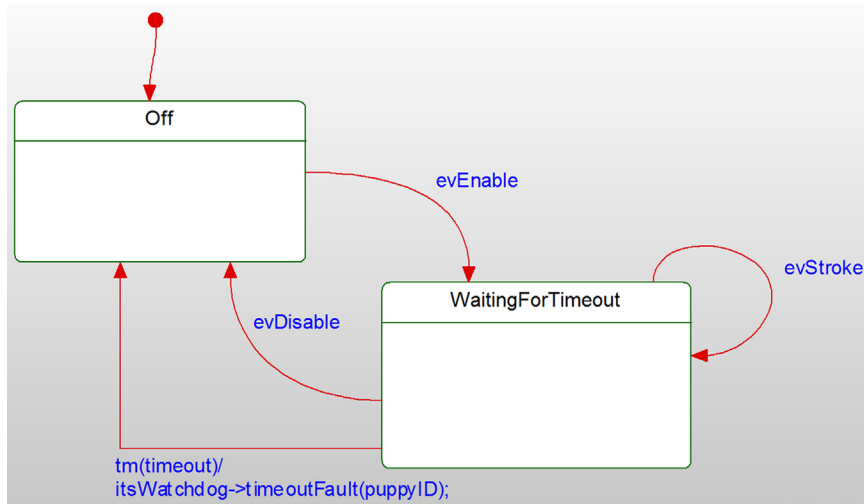


Figure 17.20
WatchPuppy state machine.

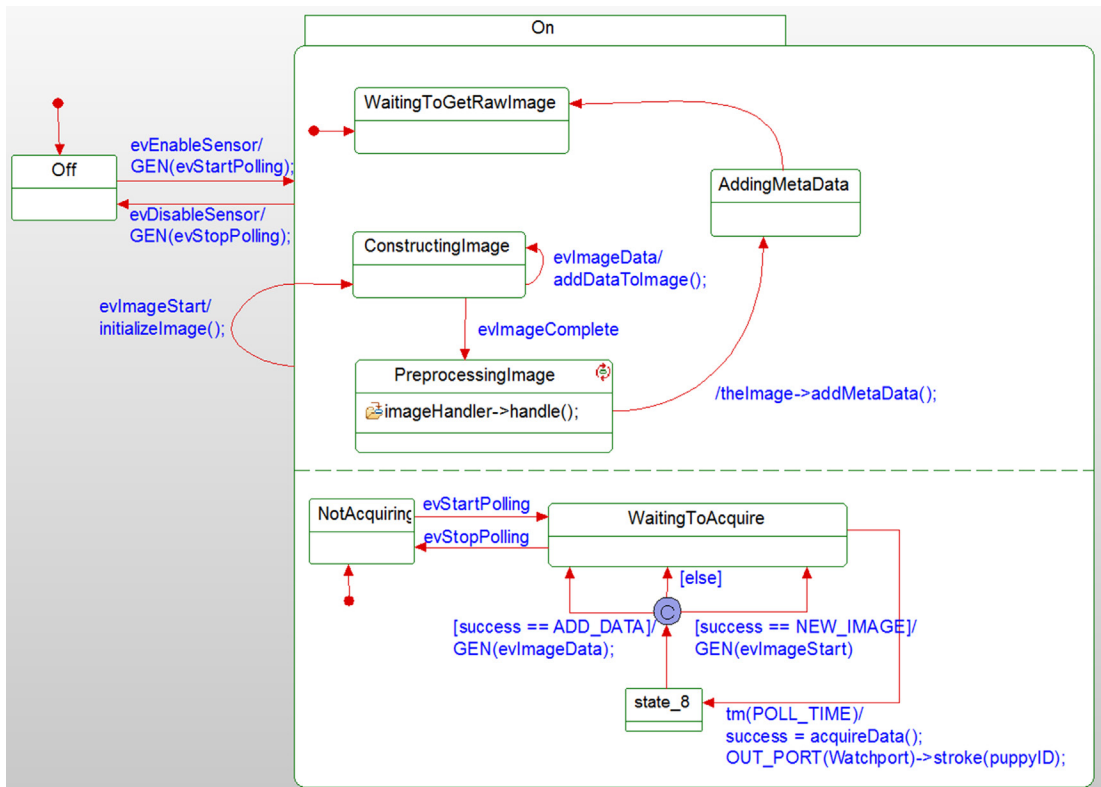


Figure 17.21
Sensor payload device driver state machine.

The Roadrunner™ Intersection Controller System Specification

Overview

The Roadrunner Intersection Controller (RIC) is an two street intersection controller that controls individual traffic lights, subsurface and above-surface vehicle detectors, and pedestrian button signal sources for a single intersection. Each intersection is limited to two intersecting roads, and supports both pedestrian and vehicular traffic in both directions along the road; options include one-way roads and turn lanes. The RIC can be programmed from a front panel display (FPD) or a remote traffic manager via a wired Ethernet Interface, such as the Tasmanian Devil Traffic Management System, available separately.

The Intersection Controller

Each intersection is controlled by a separate Intersection Controller (IC) that may be tuned manually from a secured front panel or through a remote network connection. The IC supports up to six vehicular lanes (three in each direction, including a turn lane), which may be detected by subsurface passive induction loop, or above-surface infrared or radar detectors ([Figure A.1](#)). Pedestrian lights and buttons are supported in each direction. Initial setup of the IC shall configure the number of sensor sources for the intersection.

Each IC shall have a panel control that allows direct local configuration and mode setting.

In addition to all normal operational modes, the RIC shall have a parameter to respond to both priority and emergency vehicle transponders. Priority vehicle transponders are used primarily for mass-transit vehicles (e.g., buses) and allow the optimization of their schedules. Emergency vehicle transponders indicate an approaching emergency vehicle (typically fire and police agencies). Such transmitters shall be highly directional so that it is possible to identify which road (primary or secondary) and which lane the vehicle is in. Both priority and emergency modes are only available with the above-surface infrared and radar vehicle sensor configurations.

If a priority vehicle transmitter is activated and the Priority Active parameter is set to TRUE, then when a transponder is within range of the intersection, the intersection shall either

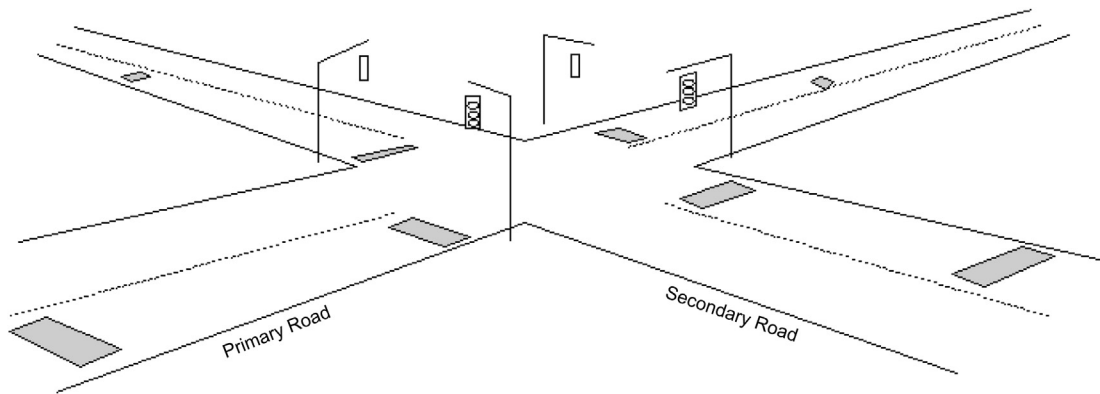


Figure A.1
Intersection.

extend the through traffic GREEN by 10 seconds or, if the through traffic light is RED, shall shorten the cross-traffic light green by 10 seconds. This is to expedite priority traffic through the intersection.

If an emergency transmitter is activated and the intersection has its Enable Emergency Traffic parameter set to TRUE, then when a transponder is within range of the intersection, the intersection shall immediately cycle the lights to RED for cross traffic and GREEN to same-direction traffic, with all turn lanes set to RED. If the same-direction traffic is already GREEN, then the GREEN time shall be extended. The light shall stay in the emergency state (GREEN for same direction, RED for cross direction) until 5 seconds after the transponder has passed the intersection or been deactivated. Any intersection traffic priority processing shall be aborted in the presence of an active emergency transmitter.

Configuration Parameters

A number of configuration parameters may be set that apply to all or many modes. Parameters specific to a particular mode are described within that mode. All parameters may be set on the front panel or set via the RIC, if present ([Table A.1](#)).

The intersection shall be able to perform vehicle counting and produce the performance statistics in [Table A.2](#):

Intersection Modes

The RIC supports a number of different operational modes. Modes may be set on the secured front panel for the intersection, or they may be set by the RIC.

Table A.1 Configuration Parameters.

Parameter	Value Range	Description
Command Mode (CMp)	0 .. 5	Sets the currently active mode (default 0).
Primary Road (PRp)	0 .. 1	Identifies which road is primary (default, first inputs).
Primary Road Directions (PRD)	SINGLE, DUAL	Identifies if the primary road is one-way (SINGLE) or two-way (DUAL) (default DUAL).
Secondary Road Directions (SRD)	SINGLE DUAL	Identifies if the secondary road is one-way (SINGLE) or two-way (DUAL) (default DUAL).
Vehicle Detector Type (VDTp)	NONE, SPLI, ASI, ASR	Identifies which kind of vehicle detector is used. Note: All active lanes within an intersection must use the same type of vehicle detector (default NONE).
Wireless Frequency (WFP)	0 .. 10	Selectable from 10 wireless frequencies (default 0 – NONE).
Primary Turn Lanes (PTLp)	FALSE, TRUE	Indicates whether primary road has separate turn lane detectors (default FALSE).
Secondary Turn Lanes (STLp)	FALSE, TRUE	Indicates whether secondary road has separate turn lane detectors (default FALSE).
Turn Lane Mode (TMp)	SIM, SEQ	If the turn mode is SIM, then turn lane lights for both directions activate simultaneously; if the mode is SEQ, then the turn lights for both directions are done sequentially, and the turn light GREEN occurs at the same time as the through light GREEN.
Primary Pedestrian (PPp)	FALSE, TRUE	Indicates whether primary road has pedestrian buttons and walk indicators (default TRUE).
Secondary Pedestrian (SPp)	FALSE, TRUE	Indicates whether secondary road has pedestrian buttons and walk indicators (default TRUE).
Priority Active (PAP)	FALSE, TRUE	When TRUE, RIC receives and responds to the presence of signals from priority transponders. Note: Only valid with above-surface infrared and radar vehicle detectors (default FALSE).
Emergency Active (EAP)	FALSE, TRUE	When TRUE, RIC receives and responds to the presence of signals from emergency priority transponders. Note: Only valid with above-surface infrared and radar vehicle detectors (default FALSE).
Current Time	Hh:mm:ss	Current time of day in 24-hr format (no default).
Current Date	M:D:Y	Current date month:date:year (no default).
Morning Start	Hh:mm	Start of morning rush mode (default 06:00).
Morning Mode	Mode	Mode of the intersection for morning rush (default 0).
Midday Start	Hh:mm	Start of midday traffic mode (default 10:00).
Midday Mode	Mode	Mode of the intersection for midday traffic (default 0).
Evening Start	Hh:mm	Start of evening rush mode (default 16:00).
Evening Mode	Mode	Mode of the intersection for evening traffic (default 0).
Night Start	Hh:mm	Start of night mode (default 21:00).
Night Mode	Mode	Mode of the intersection for night traffic (default 0).

Table A.2 Performance Statistics.

Parameter	Description
Primary Vehicle Count	Number of vehicles that have passed through the intersection since manual reset (primary road)
Primary Traffic Count Morning	Count of vehicles through the previous morning period, or the current morning period if active
Primary Traffic Count Midday	Count of vehicles through the previous midday period, or the current midday period if active
Primary Traffic Count Evening	Count of vehicles through the previous evening period, or the current evening period if active
Primary Traffic Count Night	Count of vehicles through the previous night period, or the current night period if active
Secondary Vehicle Count	Number of vehicles that have passed through the intersection since manual reset (secondary road)
Secondary Traffic Count Morning	Count of vehicles through the previous morning period, or the current morning period if active
Secondary Traffic Count Midday	Count of vehicles through the previous midday period, or the current morning midday if active
Secondary Traffic Count Evening	Count of vehicles through the previous evening period, or the current evening period if active
Secondary Traffic Count Night	Count of vehicles through the previous night period, or the current night period if active

Mode 0: Safe Mode

This mode is the default when the system has not been configured. All vehicle light outputs are set to FLASHING RED and pedestrian lights are disabled. This mode persists until another mode is selected. Flash cycle time is 75% “on” duty cycle at a rate of 0.5 Hz.

Mode 1: Evening Low Volume Mode

In this mode, the designated primary road is set to FLASHING YELLOW, the secondary road is set to FLASHING RED, and pedestrian lights are set to off. The same cycle times as in Mode 0 shall be used.

Mode 2: Fixed Cycle Time

Mode 2 is the most common operational mode. In this mode, the lanes cycle GREEN-YELLOW-RED in opposite sequences with fixed intervals. The system shall ensure that if any traffic light is non-RED, then all the lights for cross traffic shall be RED and pedestrian lights (if any) shall be set to DON’T WALK. Note that the turn lane times and/or pedestrian times are only valid in this mode if (1) the turn lane and/or pedestrian parameter is set TRUE in the RIC system parameters and (2) if a signal from the appropriate detector determines the existence of waiting traffic for the turn or pedestrian light

Table A.3 Mode 2 Parameters.

Parameter	Value Type	Description
Reset Parameters	FALSE, TRUE	(FALSE) Sets all the parameters for Mode 2 to defaults
Primary Green Time (PG2)	10 to 180 seconds	(30) Length of time the primary GREEN light is on
Primary Yellow Time (PY2)	2 to 10 seconds	(5) Length of time the primary YELLOW light is on
Primary Red Delay Time (PR2)	0 to 5 seconds	(0) Length of time between when primary RED light is turned on and the secondary GREEN light is activated
Primary Walk Time (PW2)	0 to 60 seconds	(20) Length of time the primary WALK light is on when the primary GREEN light is activated
Primary Warn Time (PA2)	0 to 30 seconds	(10) Length of time the primary FLASHING DON'T WALK light is on after the WALK light has been on
Primary Turn Green Time (PT2)	0 to 90 seconds	(20) Length of time the primary turn light is GREEN. Note: Only valid when the Primary Turn Light parameter is TRUE.
Primary Turn Yellow Time (PZ2)	0 to 10 seconds	(5) Length of time the primary turn light is YELLOW. Note: only valid when the Primary Turn Light parameter is TRUE.

Table A.4 Default Cycle Times for Mode 2.

Turn Lane	Ped Signal	Green	Yellow	Red	Walk	Don't Walk	Turn Green	Turn Yellow
F	F	30	5	0	0	0	0	0
T	F	30	5	0	0	0	15	5
F	T	50	5	0	15	5	0	0
T	T	50	5	0	15	5	15	5

The durations of the light times shall be independently adjustable by setting the appropriate parameters (see [Table A.3](#)). Note that in the table, the values in parentheses are defaults.

The default values depend on the system configuration ([Table A.4](#)).

The values in [Table A.4](#) are true for each direction, independently. Thus, if the primary road has a car waiting in its turn lane and a pedestrian walking, but the secondary road has neither, then the timing diagram in [Figure A.2](#) represents the cycle times for simultaneous turn lane mode (i.e., the turn lanes in both directions for a road turn together and the straight traffic doesn't begin until the turn lanes have cycled to RED).

Mode 3: Responsive Cycle Mode

Mode 3 provides a fixed cycle time when the secondary road is triggered by either a pedestrian or vehicle. That is, it operates exactly like Mode 2 except that in the absence of cross-traffic (vehicle or pedestrian), primary road, or primary pedestrian signals, the system shall

Scenario: Pedestrian and turn lanes enabled 001

Preconditions: Secondary light is GREEN, Primary is RED; Car waiting in primary turn lane; pedestrian waiting for primary walk signal; 5s left in Secondary cycle GREEN

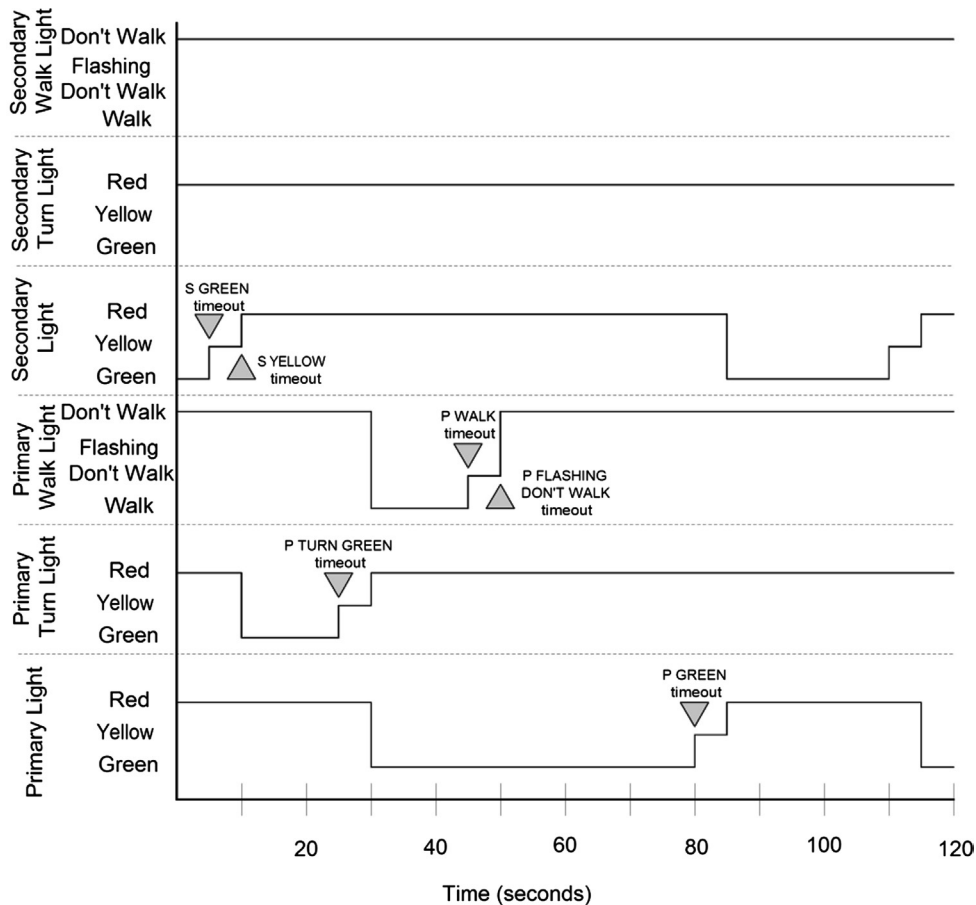


Figure A.2
Roadrunner Mode 2 timing diagram.

cycle to the default condition: primary through lights are GREEN, primary turn lights are RED, secondary through and turn lights are RED, and all pedestrian lights are DON'T WALK. As long as the signal is refreshed within the GREEN or WALK times, the currently active vehicle or walk light shall be refreshed. If the appropriate interval elapses without a fresh vehicle or pedestrian signal, then the intersection shall cycle to the default state. The default shall be maintained for at least the minimum duration of the Primary Green Time, as specified by the Mode 3 parameters. If there are both turning vehicles and pedestrians waiting along the same road, they shall be handled in the same order as in Mode 2 (turn lane first, then pedestrians). If a road's turn lane is GREEN and the pedestrian signal occurs along the

same road, then the turn lane shall cycle to RED, and the pedestrian lights shall cycle to DON'T WALK. If there is waiting traffic in both roads of the intersection, then the intersection shall cycle as if it were in Mode 2 until no signals are received for the cycle times associated with the signals.

The same parameter set is used as in Mode 2, except the system responds to them with Mode 3 behavior, when Mode 3 is the selected mode.

Mode 4: Adaptive Mode

Mode 4 is for intersections with higher density traffic. In this mode, the intersection adapts to the local history of traffic by adjusting the cycle times depending on traffic density. This requires vehicle-counting behavior from the vehicle detectors. This is similar to Mode 3 and has the same parameters, plus those in [Table A.5](#).

The Vehicle Detector

Three types of Vehicle Detectors shall be supported: subsurface passive loop inductors (SPLIs), above-surface infrared sensors (ASIs) and above-surface radars (ASRs).

Subsurface detectors shall use a wired interface to communicate with the controller, while ASIs and ASRs shall support both wired and secure wireless communication. All vehicle detectors shall be able to perform vehicle counting.

Table A.5 Mode 4 Parameters.

Parameter	Value Type	Description
Averaging Interval (Alp)	10 ... 120 minutes	Period over which traffic volume is averaged to compute the relative density between the two roads
Minimum Density (MDp)	10 ... 1000	(100) Specifies the minimum number of vehicles, from both roads, that must have traversed the intersection before adaptive extension will be employed
90%	0 ... 60 seconds	(30) Length of time that a road's GREEN time may be extended due to higher traffic volume when the road's traffic is 90% of total intersection traffic volume
80%	0 ... 60 seconds	(20) Length of time that a road's GREEN time may be extended due to higher traffic volume when the road's traffic is 80% of total intersection traffic volume
70%	0 ... 60 seconds	(10) Length of time that a road's GREEN time may be extended due to higher traffic volume when the road's traffic is 70% of total intersection traffic volume
60%	0 ... 60 seconds	(5) Length of time that a road's GREEN time may be extended due to higher traffic volume when the road's traffic is 60% of total intersection traffic volume

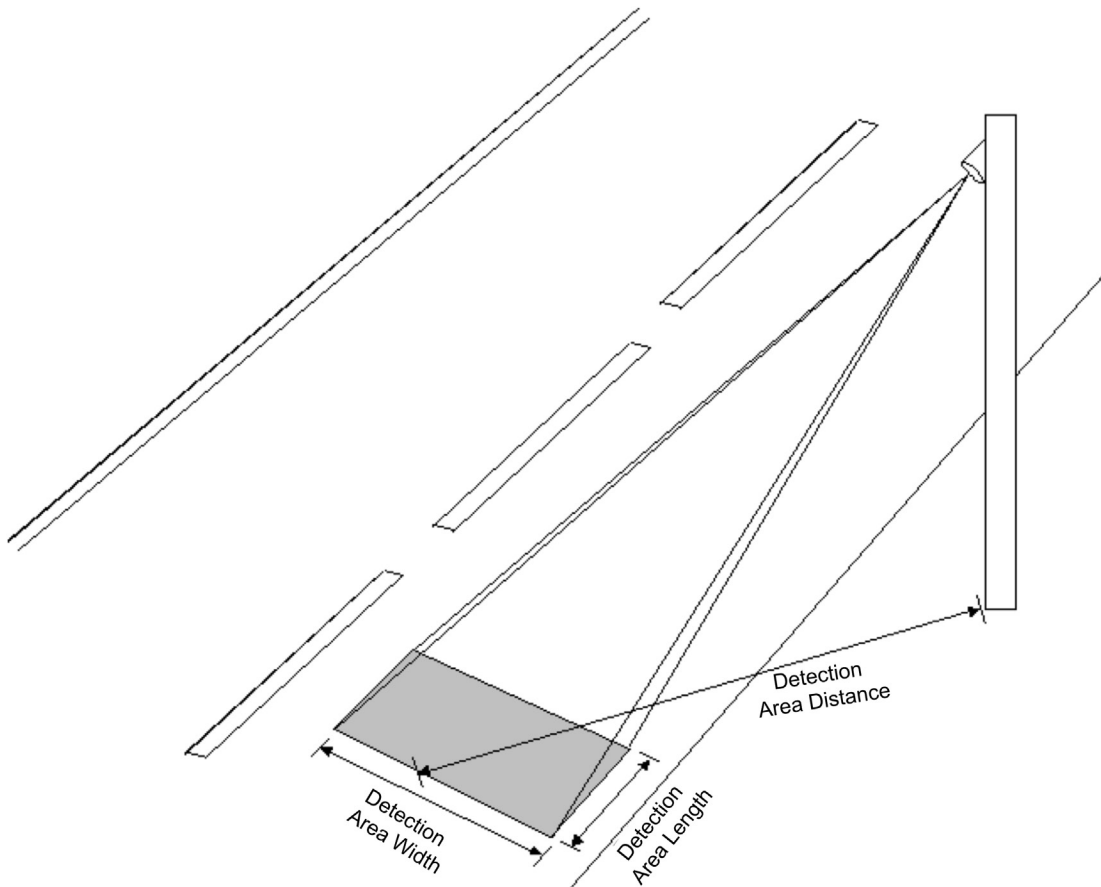


Figure A.3
Infrared and radar vehicle detector.

In addition, ASIs and ASRs shall be able to receive directional transmissions from priority vehicle and emergency vehicle transmitters. The maximum range of such reception shall be no less than 250 feet and no more than 1000 feet.

[Figure A.3](#) shows the relevant measures for both ASI and ASR detectors. When a vehicle enters the detection area (shown as the shaded area in the figure), the detector shall report the presence of a vehicle. Separate detectors are used for each lane in each direction.

Vehicular Traffic Light

Two kinds of traffic lights are supported: the standard three-light model and the four-light model, the additional light being for a green turn arrow ([Figure A.4](#)). When the intersection is allowing turn lane turns with the four-light model, then the green arrow shall be on.

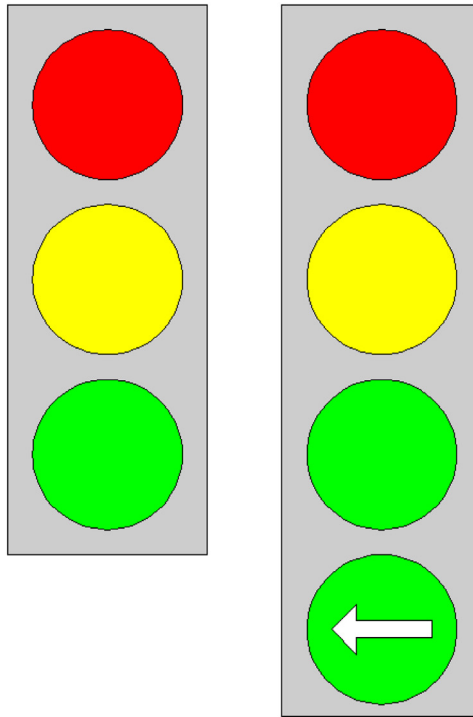


Figure A.4
Three- and four-signal traffic lights.

If the traffic light can no longer detect that it is connected to the intersection controller, it shall FLASH RED. This shall occur no longer than 10 seconds from a communications or RIC failure.

Pedestrian Light and Sensor

The pedestrian light is a two-signal light that can either be in the state of WALK, FLASHING DON'T WALK, or DON'T WALK, as shown in [Figure A.5](#). If the light no longer detects that it is in communication with the RIC, it shall go to a state of DON'T WALK within 10 seconds of the communication or RIC failure.

Front Panel Display

The front panel display (FPD) is an enclosed display secured with a mechanical key access lock. The FPD provides an LCD for viewing information and a keypad for entering information. The FPD also has a wired Ethernet interface so that a local or remote network can view or set the values known to the system.



Figure A.5
Pedestrian light.

The FPD shall support the following choices as menus.

- Intersection Configuration Setup
- Mode 2 (Fixed Cycle Time) Setup
- Mode 3 (Response Cycle) Setup
- Mode 4 (Adaptive Cycle) Setup
- Intersection Statistics Display
- Device Manufacture Display

The FPD shall have a set of front panel keys and knobs as shown in [Figure A.6](#). The turn knobs are used for menu selection and item selection. The numbers are for entering values; the arrow keys are for moving from field to field within segments of a parameter (such as mm:dd:yy for month:day:year date format). The Item Selection knob allows the selection of a parameter to change (if changeable). Before a parameter can be changed, it must be selected and then the user must press the EDIT key. Now the values can be entered with the keypad (if numeric) or selected from a list (if enumerated). When a change is made, it must be either confirmed by pressing the CONFIRM key or aborted with the ABORT key. The RESET key returns the value to the factory default if a current item is in the edit mode, or resets all parameters on the page to their defaults if no item is currently being edited.

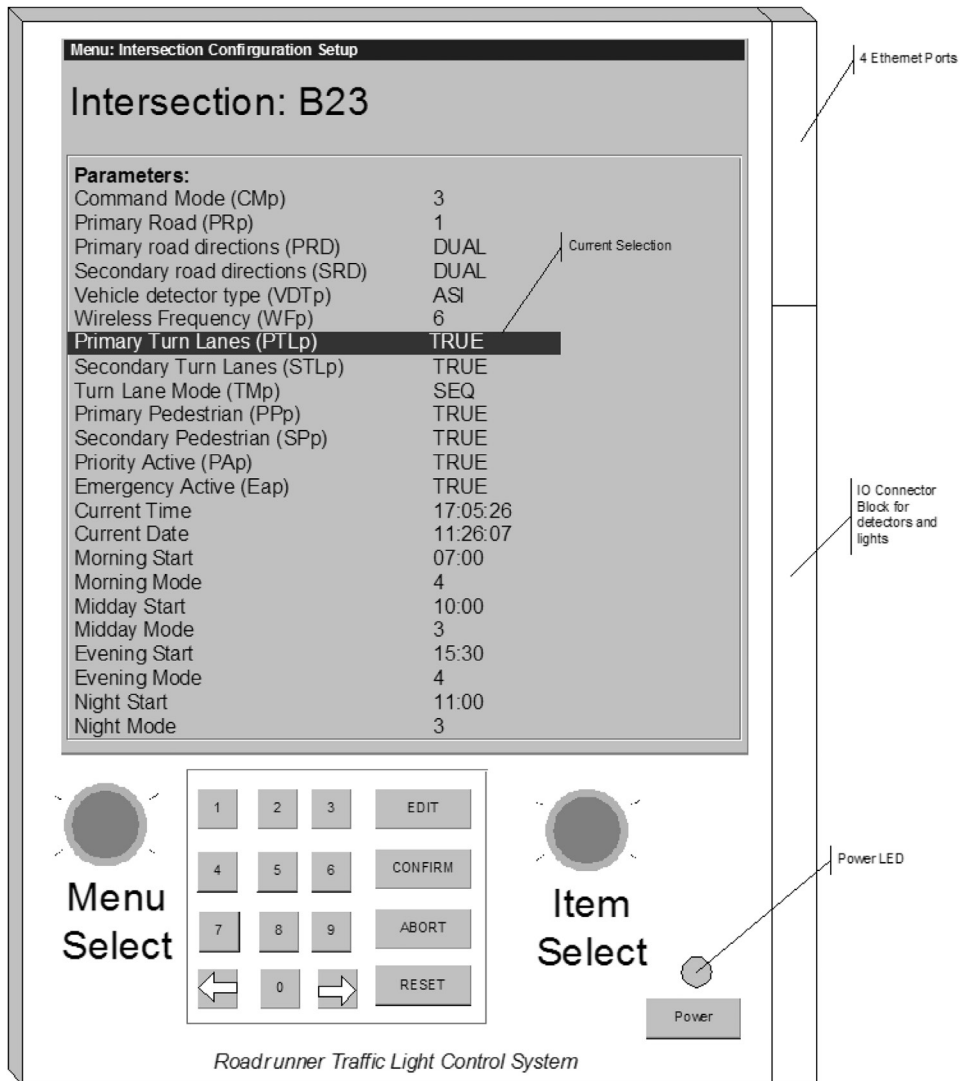


Figure A.6
Front panel display.

The front panel has a power button that must be held down for 5 seconds before the power status can be changed. A LED shows RED if the controller is off but is receiving power, YELLOW if the controller is on but running off battery (UPS), or GREEN if on and operating from mains.

The FPD supports four Ethernet ports for external interfacing and digital I/O ports for interfacing with the lights and sensors. Software may be uploaded via Ethernet from a service technician's PC.

The FPD supports secure wireless communication with up to 50 devices. Security includes 64-bit encryption of data and MAC address filtering. The wireless devices are primarily used to interact with infrared and radar vehicle sensors, but may also be used for connecting with a service technician's PC. The setup information for wireless, other than the frequency, is not available on the FPD and must be uploaded from the service technician's PC. The range of the wireless connection shall not be less than 200 feet or more than 1000 feet with an unobstructed line of sight.

Remote Communications

The RIC provides wired 10/100 Ethernet interfaces for remote monitoring and management (see the FPD). All parameters available on the front panel may be requested or set via this interface. In addition, traffic statistics may be viewed or reset via this interface.

Power

All components of the RIC that require power will accept 220-240 volts. An uninterruptible power supply (UPS) option is available to provide 1 hour or 10 hour functionality in the absence of line power. When the power is on, the UPS battery shall recharge. When running on UPS, the power light on the front panel display shall be RED. When running on main power, but charging the battery (and battery is at less than 90% capacity), the power light shall be AMBER. When on main power and battery is at 90% or more capacity, or when on main power and the UPS option is not present, the power light shall be GREEN.

The Coyote Unmanned Aerial Vehicle System (CUAVS)

Overview

The Coyote Unmanned Aerial Vehicle System (CUAVS) is a system solution to medium-range reconnaissance in hostile environments with limited attack capability. It is a medium-range, long endurance UAV system that can carry a variety of payloads to assist in ground, air, and sea operations.

Req001: Suite

A full CUAVS consists of four Coyote Unmanned Aerial Vehicles (CUAVs) and a ground Coyote Mission Planning and Control System (CMPCS).

Primary CUAV System Components

The Unmanned Aerial Vehicle (UAV)

The Coyote UAV is meant to be a multipurpose reusable UAV with multimission capability.

Req002: UAV 1

The CUAV shall operate at altitude of up to 30,000 feet with ground speeds of up to 100 knots (cruise) and 150 knots (dash) and carry a payload of up to 450 lbs for durations in excess of 24 hours.

Req003: UAV 2

The Coyote is meant to fly unimpeded in low-visibility environments while carrying either reconnaissance or attack payloads.

Req004: UAV 3

While controllable from the ground station CMPCS, the Coyote is also capable of flying complex flight plans with specific operational goals of systematic area search, ground route (road-based) search, and orbit surveillance of point targets.

Req005: UAV 4

Coupled with manned control from the ground, the Coyote shall provide sustained 24-hour flight with real-time visual, infrared or radar telemetry with target recognition preprocessing.

Req006: Comm 1

Communications shall be jam-resistant, although need not be anti-jamming in a high ECM environment.

Req007: Comm 2

Control commands shall be encrypted while telemetry data can be compressed but unprotected.

Req008: Comm 3

Telemetry rates for visual telemetry support 30 frames per second (fps) at 640x400 resolution.

Req009: Flight 1

Range of flight shall be fully supported within line of sight (LOS) range but since the Coyote also has the ability to be passed among different CMPCSs, its range shall be considerably greater than LOS.

Req010: Nav 1

For navigation, the Coyote has on-board Global Positioning System (GPS) based navigation as well as being directly controllable from the ground station.

Req011: Nav 2

For navigation, the Coyote can use an inertial navigation system if GPS is unavailable.

Req012: Takeoff and landing

Unlike many smaller UAVs, the Coyote does not require specialized launch and recovery vehicles. It can use a short runway for either automated or remote-controlled takeoff and landing.

The Coyote Mission Planning and Control System (CMPCS)**Req100: CMPCS 1**

Mobile CMPCS shall have the capability to control up to four UAVs with a manned control station per UAV that fits into a smaller towable trailer.

Req101: CMPCS 2

Each control station shall consist of two manned substations – one for controlling the CUAV and one for monitoring and controlling payloads.

Req102: CMPCS 3

If desired, both functions can be slaved together into a single control substation.

Req103: CMPCS 4

Control of the aircraft shall consist of transferring navigational commands, which may be simple (set altitude, speed, direction), operational (fly to coordinate set, orbit point, execute search pattern, etc.), planned (upload multisegment flight plan), or remote controlled with a joystick interface.

Req104: UAV 5

Stable flight mechanics shall be managed by the aircraft itself but this can be disabled for remotely controlled flight.

Req105: CMPCS 5

The CMPCS displays real-time reconnaissance data as well as maintaining continuous recording and replay capability for up to 96 hours of operation for four separate CUAVs.

Coyote Payloads

Req106: Payload 1

In addition, with attack payloads, the Coyote can carry up to four Hellfire missiles with fire-and-forget navigation systems.

Req107: Payload 2

The Coyote UAV shall be easily configured to handle any of several different surveillance and reconnaissance payloads (video, forward-looking infrared (FLIR), radar), countermeasures (electronic countermeasures (ECM) and electronic counter-countermeasures (ECCM)), and attack payloads (Hellfire missiles).

Req108: Payload 3

The payloads are primarily controlled from the ground, but certain functionality, such as searching for targets, can be assigned to the UAV itself.

Req109: Payload 4

In this latter case, the Coyote performs any of a set of systematic search patterns for a target that matches the target specifications (such as tank, ship, convoy, platoon, soldier, missile launch platform, building) upon which time it notifies the controlling CMPCS.

Req110: Payload 5

The Coyote can be configured with up to 450 pounds of payload and within that limitation can contain a mixture of surveillance, ECM/ECCM, and attack payloads.

The Coyote Datalink Subsystem (CDS)***Req200: CDS 1***

The CDS shall communicate between the CUAV and the CMPCS.

Req201: CDS 2

All commands to the CUAV shall be encrypted to prevent hostile intervention in the mission, while due to bandwidth limitations, real-time telemetry may be compressed but not encrypted.

Req202: CDS 3

The CDS is an LOS communication system that is jam-resistant, although not necessarily anti-jamming.

Req203: CDS 4

It involves both spread spectrum and high-speed frequency hopping modes to improve jam resistance.

Req204: CDS 5

While the CDS is itself LOS, the CUAV may be passed among distributed CMPCSs. It can fly preprogrammed flight plans for up to 5 hours between CMPCSs, so that it has extended range beyond LOS.

Detailed Requirements***The Unmanned Aerial Vehicle (UAV)******Req300: Airframe 1***

The UAV shall be no more than 30 feet long with a wingspan of no more than 10 feet.

Req301: Airframe 2

The UAV shall be able to take off and land using a runway of no more than 1500 feet by 50 feet, both automatically and via remote piloting.

Req302: Airframe 3

When fully loaded (450 lb payload), the system shall be able to maintain at least 24 hours of cruise flight. Gross weight shall not exceed 2100 pounds.

Req303: Airframe 4

The CUAV shall be able to fly reliably in inclement and low-visibility weather; the CUAV shall be flyable without loss of reliability with 15 knot crosswinds; 30 knot headwinds; in rain, sleet, hail, and snow; and in icy conditions.

Req304: Airframe 5

The CUAV shall have a range of up to 400 nautical miles with an empty weight of no more than 1200 pounds (2100 pounds loaded). It has a ceiling of 30,000 feet and a 100-gallon (665-pound) fuel capacity.

Req405: Airframe 6

The CUAV shall have a deployable parachute for emergency recovery.

Flight Modes

Req500: Flight Mode 1

The CUAV is capable of fully automated flight, but it is envisioned that it will normally be under navigational control of the CMPCS. The following flight modes are supported:

- Fully automated
- Operational
- Remote (CUAV maintains stable attitude)
- High-fidelity remote (remote pilot maintains stable attitude)

Req501: Flight Mode 2

In fully automated mode, the CUAV can take off, fly a preprogrammed mission, and return.

Req502: Flight Mode 3

If the CUAV experiences an unexpected loss of communications with its CMPCS in excess of 60 minutes, it can abort the mission and perform an automated return and landing. This is meant to decrease incidence of UAV loss due to ECM and communications failures.

Req503: Flight Mode 4

If unable to land automatically, the CUAV will deploy its emergency parachute.

Req504: Nav 3

Automated navigation shall be performed using either an inertial navigational system or a GPS system, or a combination of the two.

Req505: Nav 4

The CUAV shall report its position along the CUAV control data link at least every 3 seconds; this data shall include latitude, longitude, and altitude information.

Req506: Nav 5

When flying in environments where fewer than three GPS satellites will be visible, the CUAV shall have an onboard altimeter as well.

Mission Modes***Req600: Mission 1***

Beyond flight modes, the CUAV shall be designed for highly flexible mission parameters. Normal mission modes include:

- Preplanned reconnaissance
- Remote controlled reconnaissance
- Area search
- Route search
- Orbit point target
- Attack

Req601: Mission 2

A mission can consist of any number of sequential submissions each operating in a different mission mode, depending on the current payload.

The Coyote Mission Planning and Control System (CMPCS)***Req700: CMPCS 6***

The CMPCS is housed in a 30x8x8 triple-axis trailer that contains stations for pilot and payload operations, mission planning, data exploitation, communications, and SAR viewing.

Req701: CMPCS 7

The CMPCS connects to multiple directional antennae for communication with the CUAVs.

Req702: CMPCS 8

All mission data is recorded at the CMPCS since the CUAV has no onboard recording capability.

Req703: CMPCS 9

The CMPCS has a UPS that can operate at full load for up to 4 hours in addition to using commercial power or power generators.

Req704: CMPCS 10

A single CMPCS can control up to four CUAVs in flight with one station per CUAV.

Req705: CMPCS 11

Each CUAV control station provides both pilot and payload operations with separate control substations, although both functions can be slaved to a single substation for low vigilance use.

Req706: CMPCS 12

For the reconnaissance payloads, the CMPCS shall provide enhanced automated target recognition (ATR) capability for all surveillance types – optical, infrared, and radar.

Req707: CMPCS 13

While the CUAV has a rudimentary capability, the CMPCS provides much more complete support for the quick identification of high-value targets in the battlefield. This capability is specifically designed to identify mobile and time-limited targets that may only be exposed for brief periods of time before they go back into hiding.

Req708: CMPCS 14

The system is expected to provide high clutter rejection and a low false-positive error rate.

Req709: CMPCS 15

The ATR shall be able to identify and track up to 20 targets within the surveillance area, with likely identification and probability assessments for each.

Req710: CMPCS 16

In addition to the ATR, the payload operator can add targets visually identified from reconnaissance data or gathered from other sources.

Req711: CMPCS 17

The battlefield view can be transmitted over links to remote command staff for tactical and strategic assessment.

Req712: CMPCS 18

Communications shall include radio, cellular, and landline phones in addition to radio-based communication with the CUAVs.

The Coyote Reconnaissance Sensor Suite Payload (CSSP)

Req800: CSSP 1

The CUAV is meant to be a mobile operational framework that is highly customizable for various missions.

Req801: CSSP 2

To that end, the CUAV can use gimbaled optical sensors (with zoom and spotter lenses), gimbaled FLIR sensors, or synthetic aperture radar (SAR) for surveillance.

Req802: CSSP 3

In most situations, the sensors' positions are controlled by the CMPCS, but they may be automatically controlled for automated search missions.

Req803: CSSP 4

Telemetry shall be broadcast over the high-bandwidth surveillance datalink with the ability to transmit 640x400 resolution data at a rate of 30 fps.

Req804: CSSP 5

This data may be compressed but there is no requirement to encrypt the high-speed data.

Req805: CSSP 6

The CUAV is equipped with a color nose camera (generally used by the UAV operator for flight control), a day variable aperture TV camera, a variable aperture infrared camera (for low light/night), and a synthetic aperture radar for looking through smoke, clouds or haze.

Req806: CSSP 7

The cameras produce full motion video and the synthetic aperture radar produces still frame radar images.

Req807: CSSP 8

The daylight variable aperture or the infrared electro-optical sensor may be operated simultaneously with the synthetic aperture radar.

Req808: CSSP 9

To reduce weight, the CUAV need not have all subcomponents of the CSSP but will normally have at least one.

The Coyote Hellfire Attack Payload (CHAP)**Req900: CHAP 1**

The CHAP provides the ability for the CUAV to carry and deploy up to four AGM-114 Hellfire missiles with the intention of being primarily for anti-tank and fixed target operations (Figure B.1).

Req901: CHAP 2

The Hellfire missile is a solid propellant rocket with a maximum speed of 950 knots that weighs up to 100 pounds.

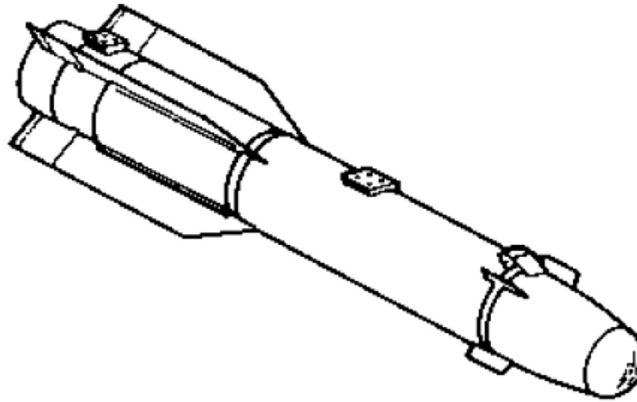


Figure B.1 Hellfire missile.

Req902: CHAP 3

The system shall be able to carry from zero to four missiles within the limitation of payload weight.

Req903: CHAP 4

The CHAP, when present, shall include a laser sight designator with backscatter immunity that targets the missile prior to release or a radar target designator.

Req904: CHAP 5

Once released, the missile shall be self-guided, so the CHAP is intended for both guided and fire-and-forget operation.

Req905: CHAP 6

The missile shall be able to defeat reactive armor and successfully deploy in adverse and hazy weather.

Req906: CHAP 7

Warheads shall be high-explosive copper-lined anti-tank and anti-bunker.

Req907: CHAP 8

The missiles attach to the CUAV on firing rails mounted on pylons, allowing two missiles to be mounted on each wing.

Req908: CHAP 9

Prior to release, target data is downloaded to the missile from the CUAV. Note that the CUAV cannot initiate Hellfire release itself; such control shall be relegated to the CMPCS.

Req909: CHAP 10

When laser targeting, the missile is fired in a Lock-On-Before-Launch (LOBL) mode, in which the target is designated prior to missile release.

Req910: CHAP 11

The CUAV shall pulse the laser designator with a particular pulse pattern, which is downloaded into the missile and used to identify it from other laser designators that may be in use.

Req911: CHAP 12

Once located, the missile homes in on the reflected laser light. This operational mode requires the CUAV to maintain the laser designator on target until the missile reaches its target, so a single target can be fired upon at a time.

Req912: CHAP 13

For fire-and-forget operations, the Hellfire missiles may be fitted with a radar seeker.

Req913: CHAP 14

Once the CUAV has located the target and the CMPCS has given the command to release, the missile maintains radar lock on the target. This allows the CUAV to fire and mask, or to seek other targets while the missile flies to its target.

Req914: CHAP 15

The CUAV shall support up to four separate targets to be fired upon in rapid succession.

The Coyote Datalink Subsystem (CDS)**Req1000: CDS 6**

The CDS allows uploading of mission parameters, such as CUAV payload configuration, flight plans, communication frequency hopping schedules, and so on; control of the vehicle operation pre-, in-, and post-flight; control of the payloads including sensor position, zoom level and selection; downloading of vehicle status, including current position, direction, speed, and fuel status; arming and releasing of missile payloads; setup and configuration of the operational flight program (OFP); and control of initiation of test modes.

Req1001: CDS 7

The CDS consists logically of two distinct data links, although these may share the same communication media.

Req1002: CDS 8

The control data link is an encrypted secure low-bandwidth data link that supports vehicle and payload commands and status information.

Req1003: CDS 9

The required data rate is no more than 100 bits per second (bps) with a reliability of 0.9994 in an ECM-free environment and 0.90 in a high-ECM environment.

Req1004: CDS 10

The system shall be able to detect and correct all single and dual bit errors and detect multiple-bit errors with a reliability of 0.9994.

Req1005: CDS 11

The high-speed data link is used for transmission from the CUAV only and is meant to include real-time telemetry, reconnaissance, and surveillance data.

Req1006: CDS 12

The data link may compress the data, but it shall be able to transmit without information loss 640x400 optical data at a rate of 30 frames per second (fps) in an ECM-free environment.

Req1007: CDS 13

In a high-ECM environment, the data rate shall be 320x200 at 15 fps for optical and FLIR imaging. SAR data shall be supported as single frame images (640x400) transmitted at a rate not exceeding 1 fps.

Req1008: CDS 14

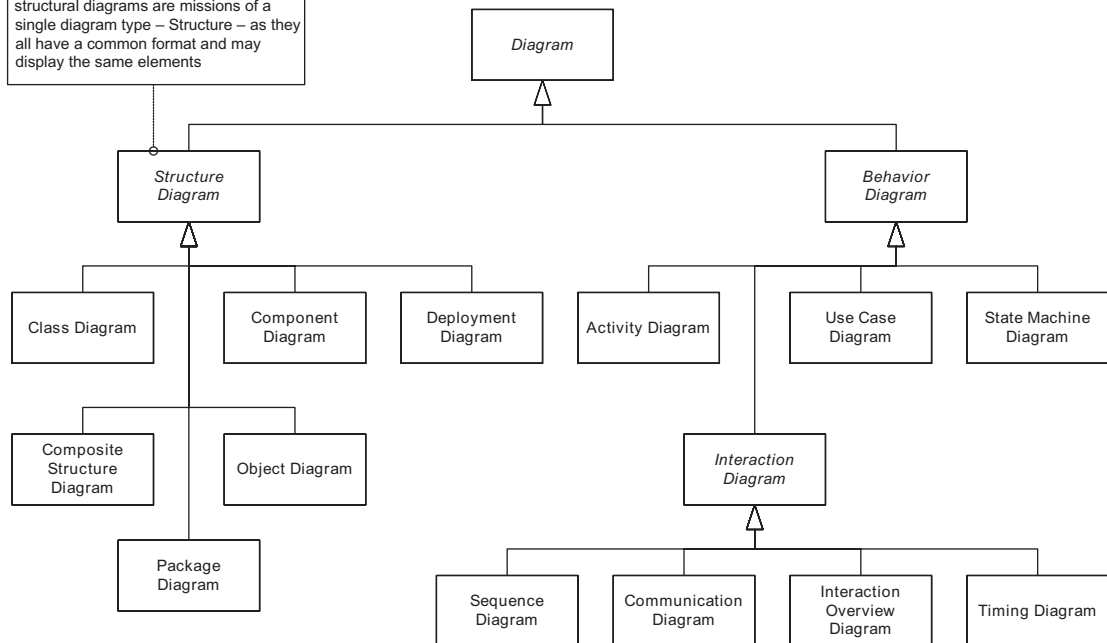
The high-speed data link need not be encrypted as its reception by the enemy cannot lead to loss of the CUAV and operational experience has shown that real-time surveillance data is of little practical use to the enemy when intercepted.

UML Notational Summary

UML Diagram Taxonomy

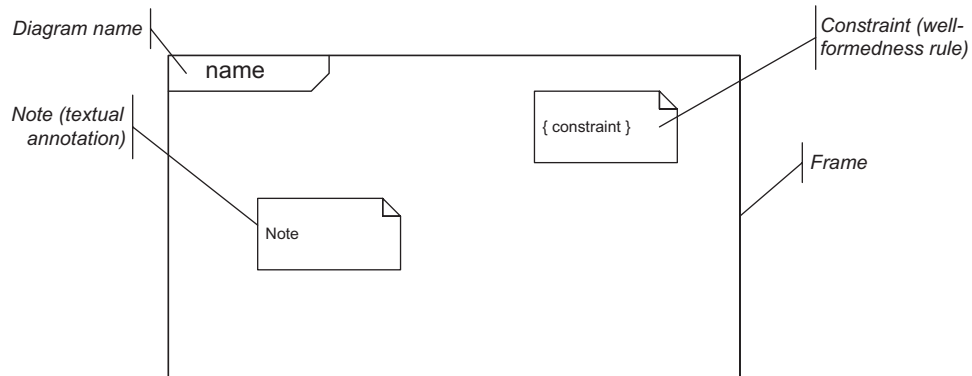
The various kinds of diagrams defined in UML 2.5.

Note: All concrete (nonabstract) structural diagrams are instances of a single diagram type – Structure – as they all have a common format and may display the same elements



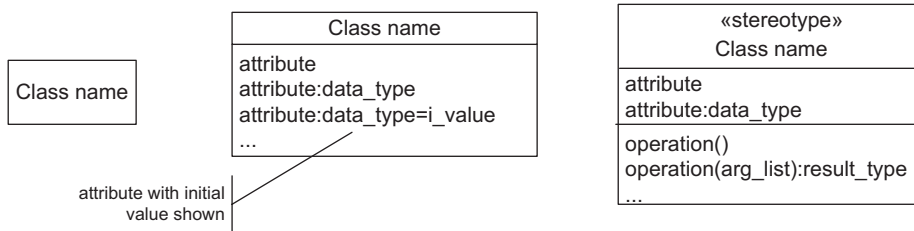
Common Diagram Elements

All diagrams may have a frame and a name, and contain notes and constraints.

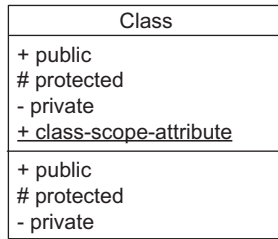


Class Diagram

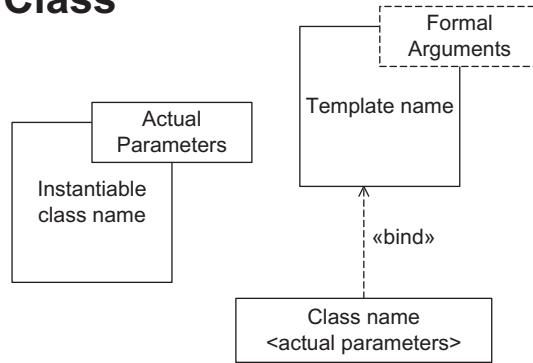
Shows the existence of classes and relationships in a logical view of a system.



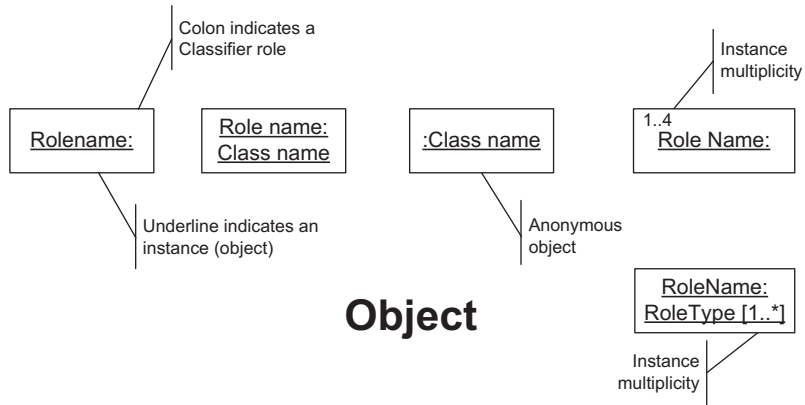
Class



Visibility



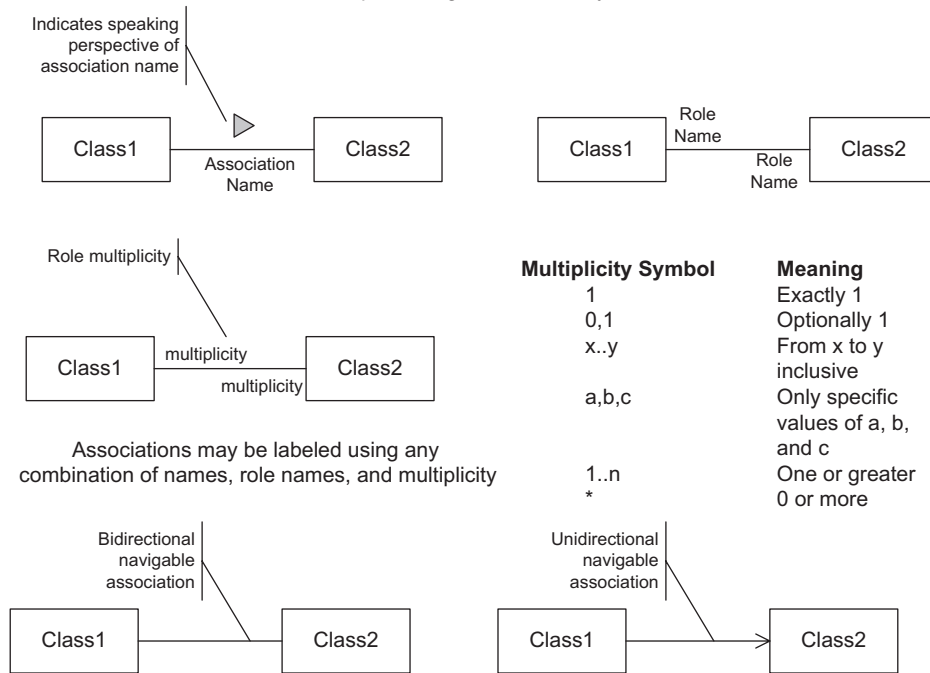
Parameterized Class



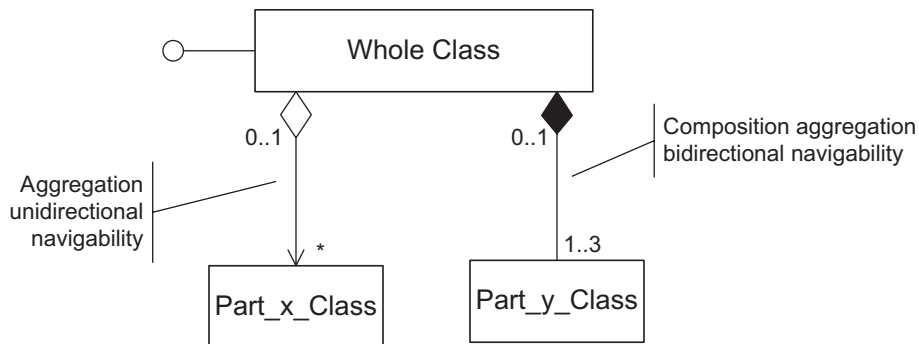
Object

Class Diagram

Shows the existence of classes and relationships in a logical view of a system.



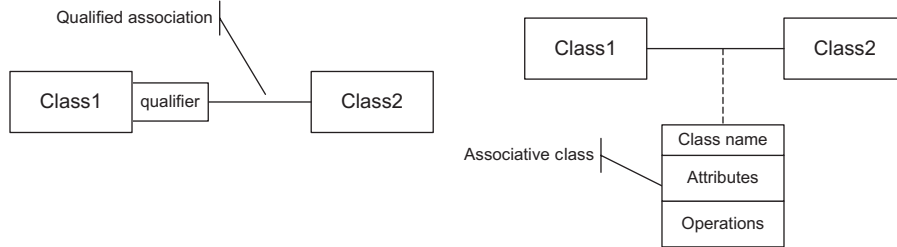
Association



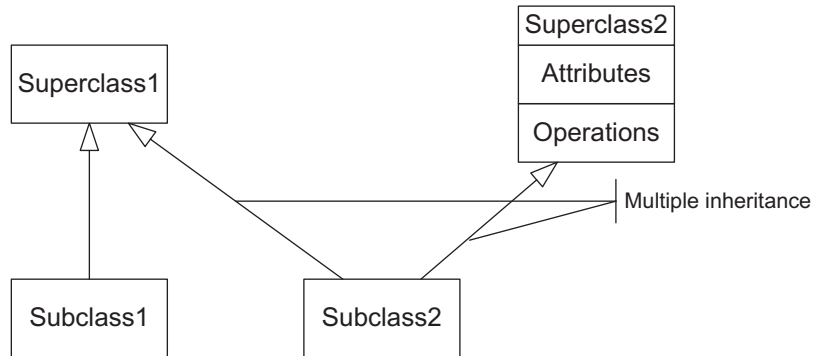
Aggregation and Composition

Class Diagram

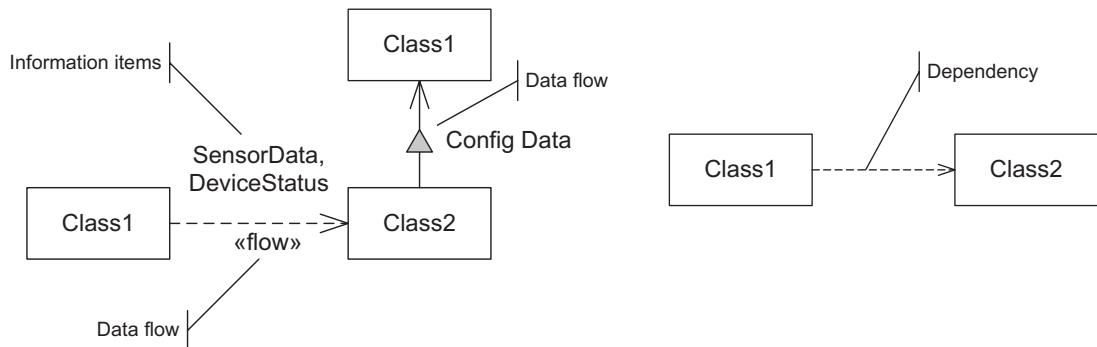
Shows the existence of classes and relationships in a logical view of a system.



Advanced Associations



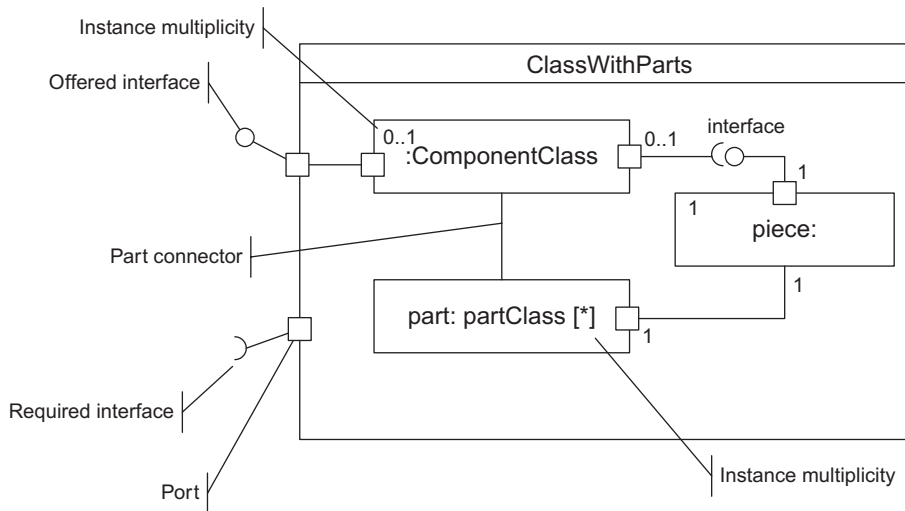
Generalization



Dependency and Data Flow

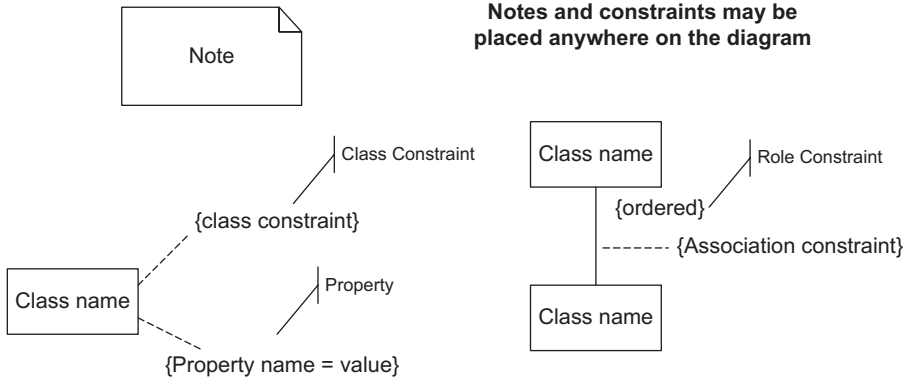
Class Diagram

Shows the existence of classes and relationships in a logical view of a system.



Structured Classes

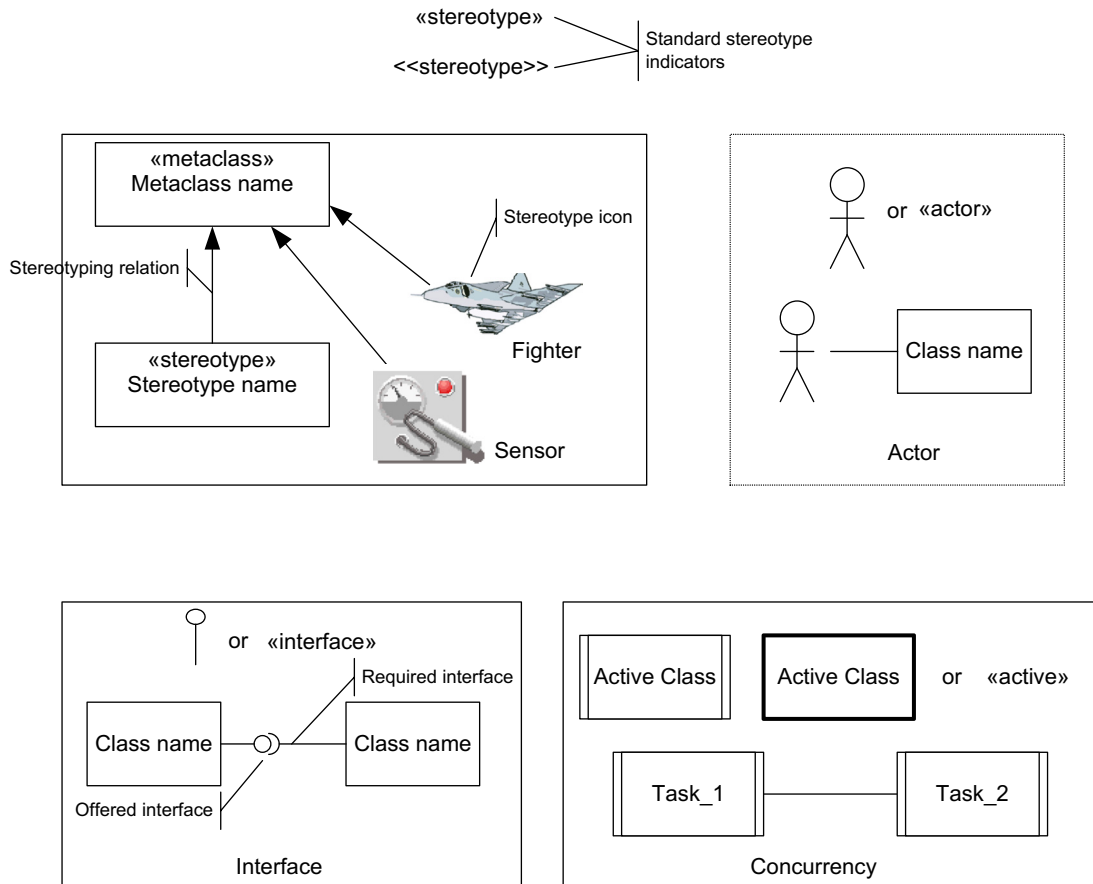
Notes and constraints may be placed anywhere on the diagram



Notes and Constraints

Class Diagram

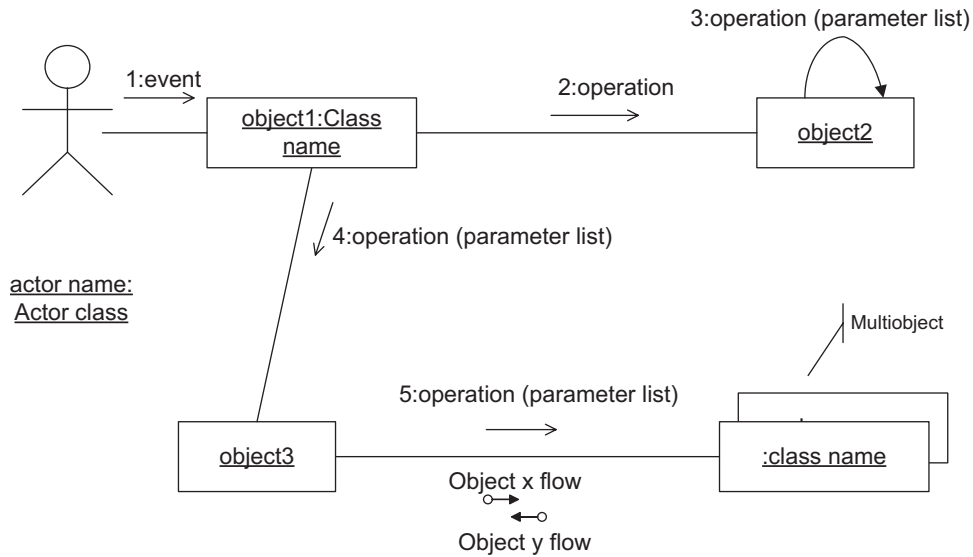
Shows the existence of classes and relationships in a logical view of a system.



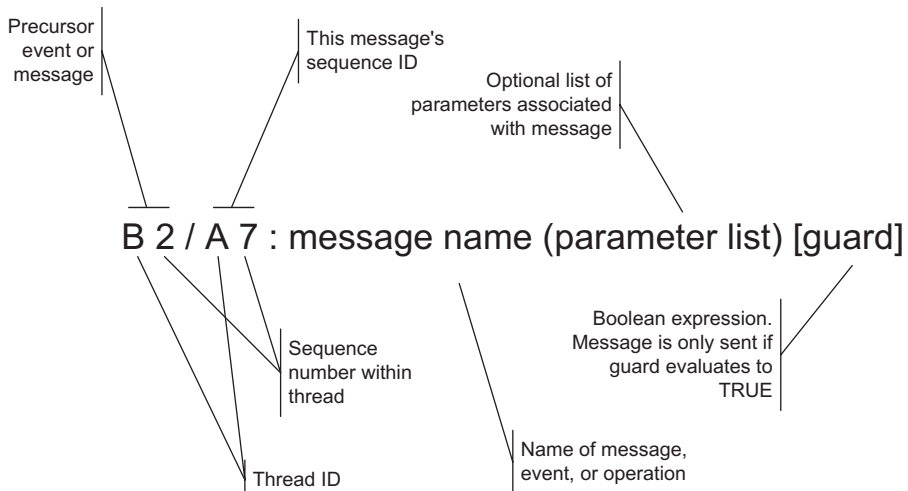
Stereotypes

Communication Diagram

Shows a sequenced set of messages illustrating a specific example of object interaction.



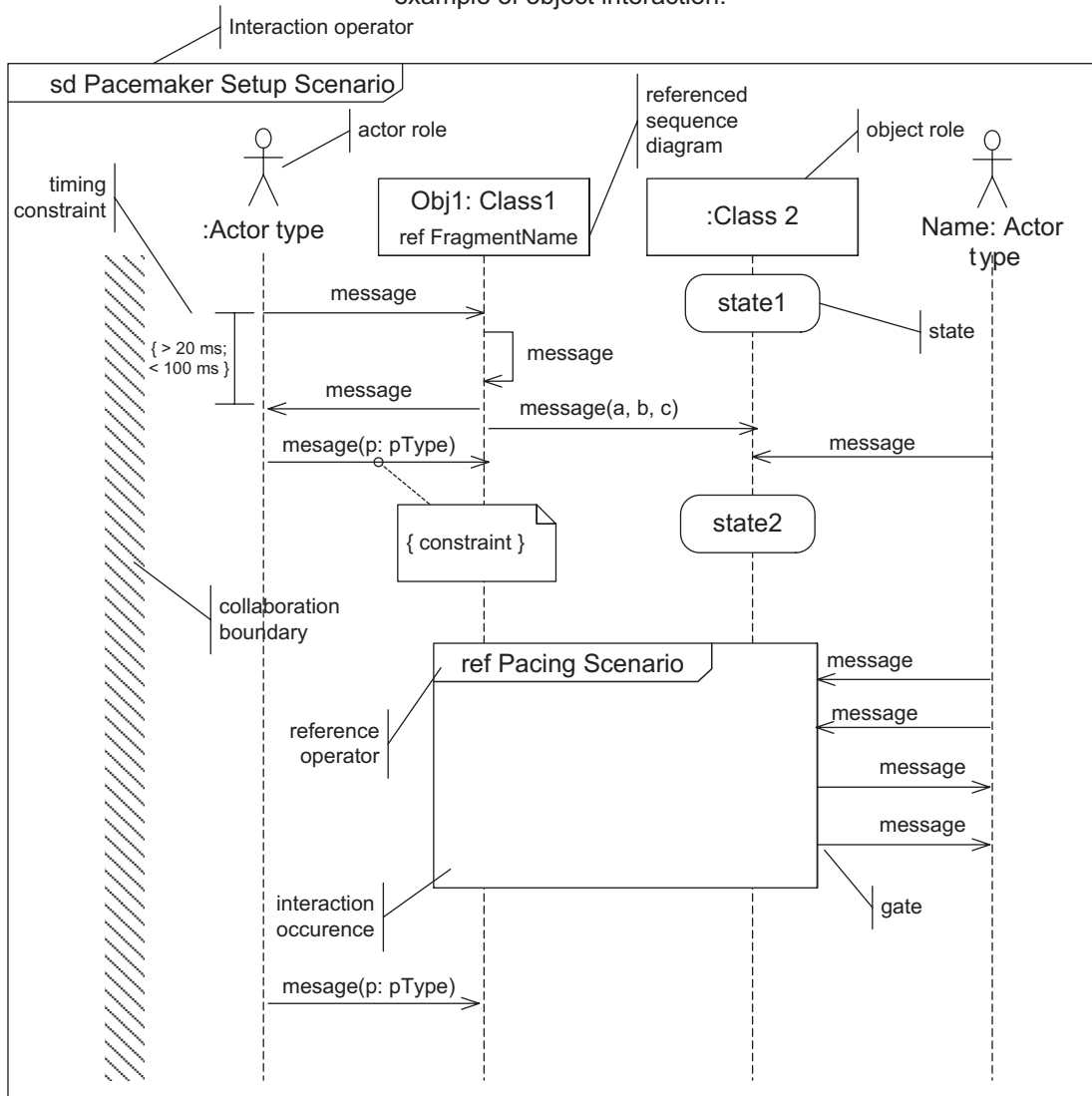
Object Collaboration



Message Syntax

Sequence Diagram

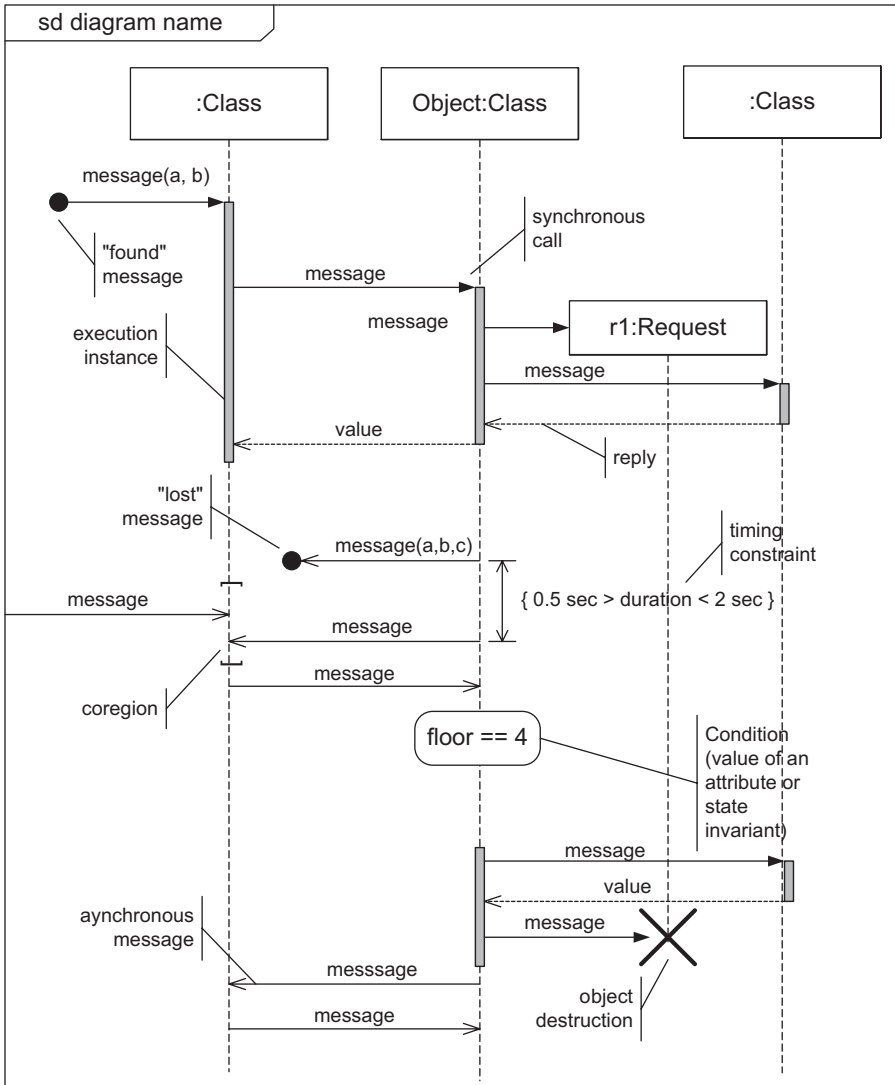
Shows a sequenced set of messages illustrating a specific example of object interaction.



Sequence Diagram Elements

Sequence Diagram

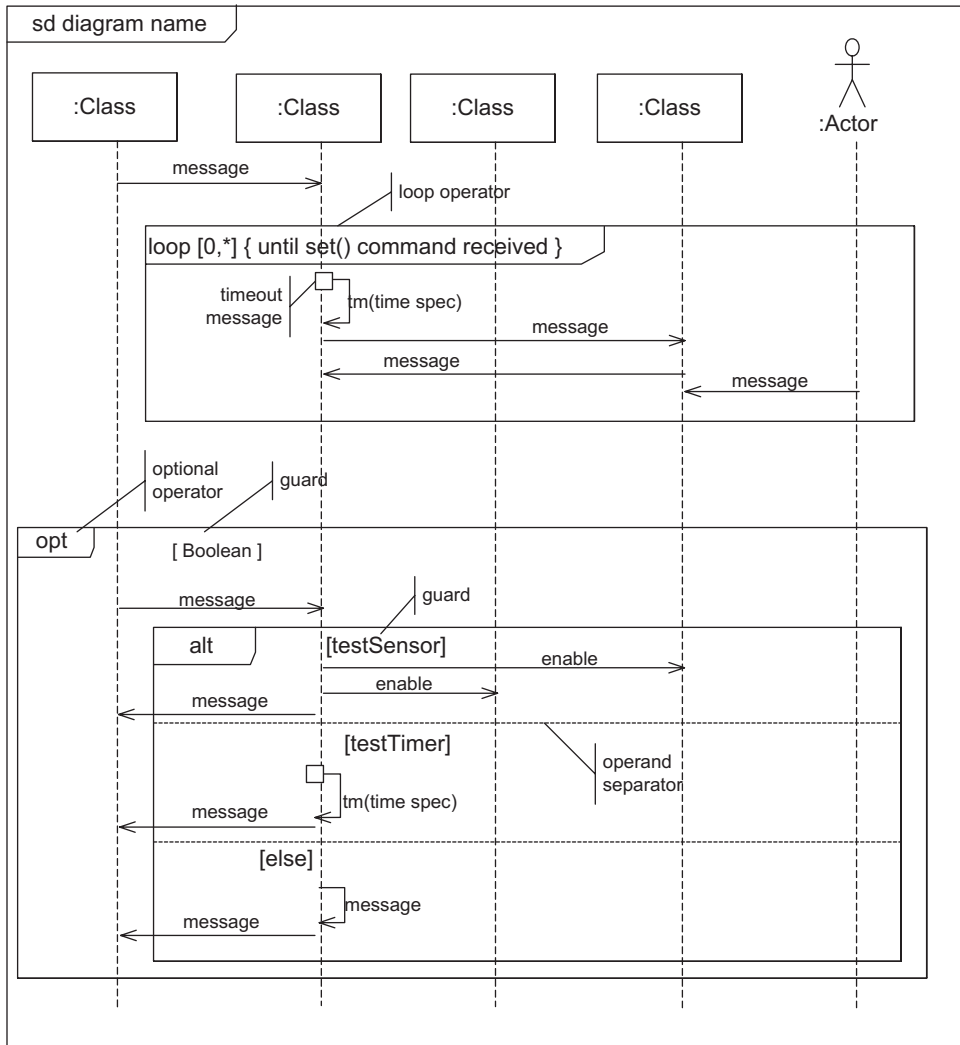
Shows a sequenced set of messages illustrating a specific example of object interaction.



Sequence Diagram Annotations

Sequence Diagram

Shows a sequenced set of messages illustrating a specific example of object interaction.



Interaction Operators

Sequence Diagram

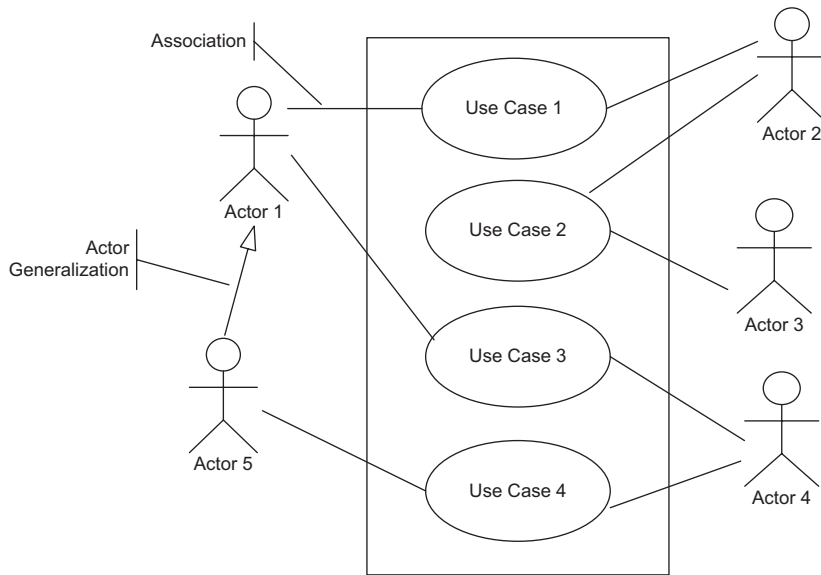
Shows a sequenced set of messages illustrating a specific example of object interaction.

Operator	Description
sd	Names an interaction fragment.
alt	The alt operator provides alternatives, only one of which will be taken. The branches are evaluated on the basis of guards, similar to statecharts. An "else" guard is provided that evaluates to TRUE if and only if all other branch alternatives evaluate to FALSE.
opt	Defines an optional interaction segment, i.e., one that may or may not occur.
break	Break is a short hand for an alt operator where one operand is given and the other is the rest of the enclosing interaction fragment. A sequence diagram analogue to C++ "break" statement.
loop	Specifies that an interaction fragment shall be repeated some number of times.
seq	Weak sequencing (default). Specifies the normal weak sequencing rules are in force in the fragment.
strict	Specifies that the messages in the interaction fragment are fully ordered - that is, only a single execution trace is consistent with the fragment.
neg	Specifies a negative, or "not" condition. Useful for capturing negative requirements.
par	Defines parallel or concurrent regions in an interaction fragment. This is similar to alt in that subfragments are identified, but differs in that ALL such subfragments execute rather than just a single one.
criticalRegion	Identifies that the interaction fragment must be treated as atomic and cannot be interleaved with other event occurrences. It is useful in combination with the par operator.
ignore/consider	The ignore operator specifies that some message types are not shown within the interaction fragment, but can be ignored for the purpose of the diagram. The consider operator specifies which messages should be considered in the fragment.
assert	Specifies that the interaction fragment represents an assertion.

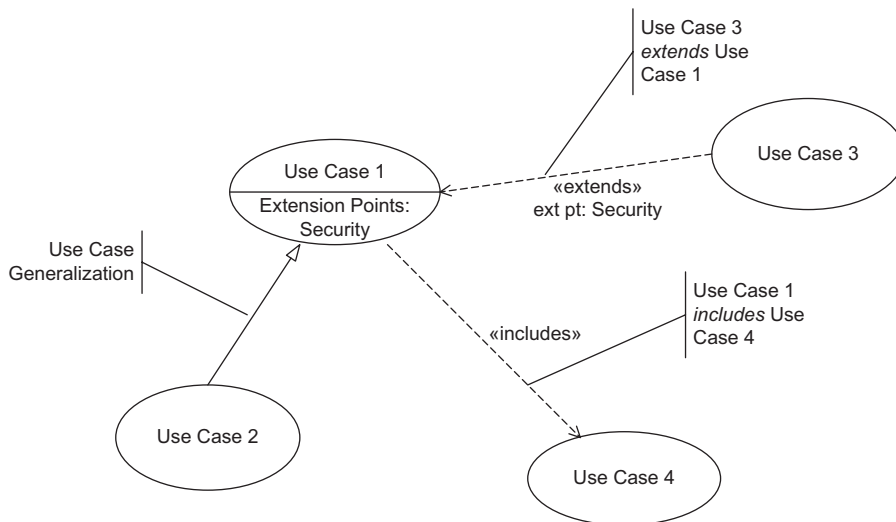
Interaction Operators

Use Cases

Use cases show primary areas of collaboration between the system and actors in its environment. Use cases are isomorphic with function points.



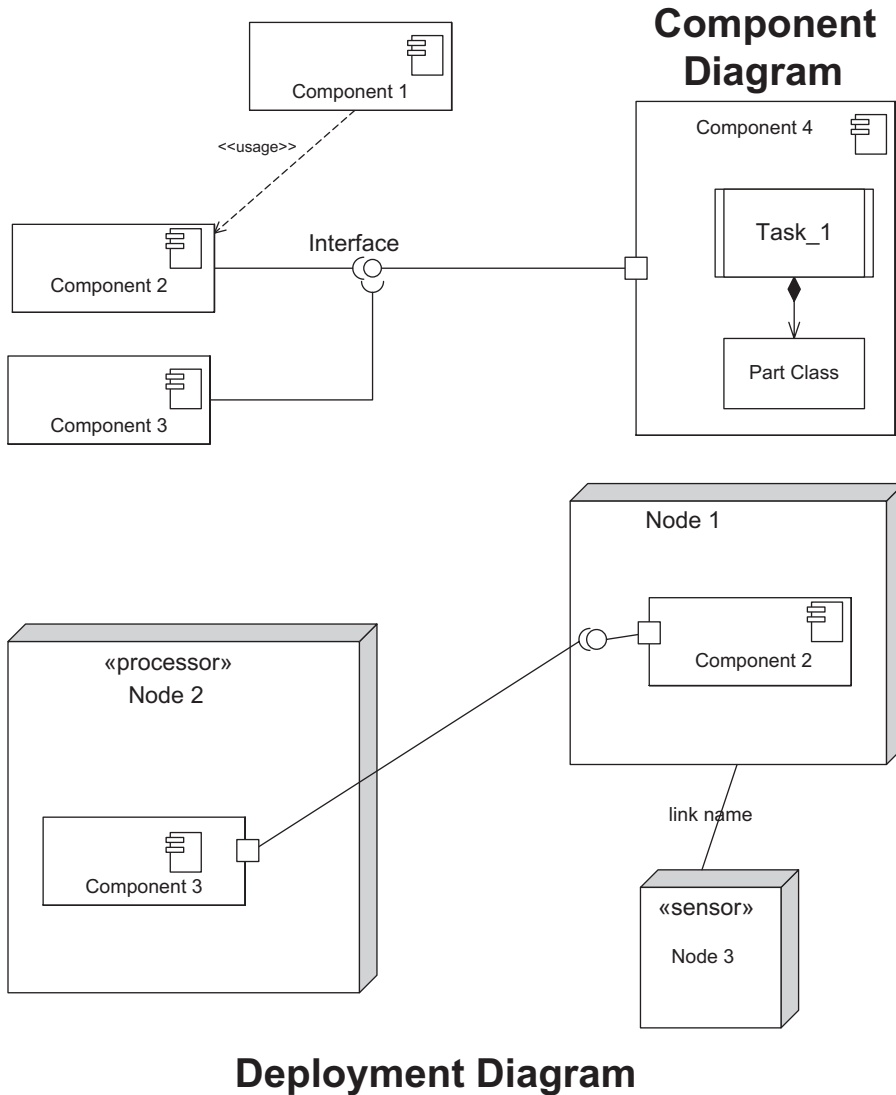
Use Case Diagram



Use Case Relationships

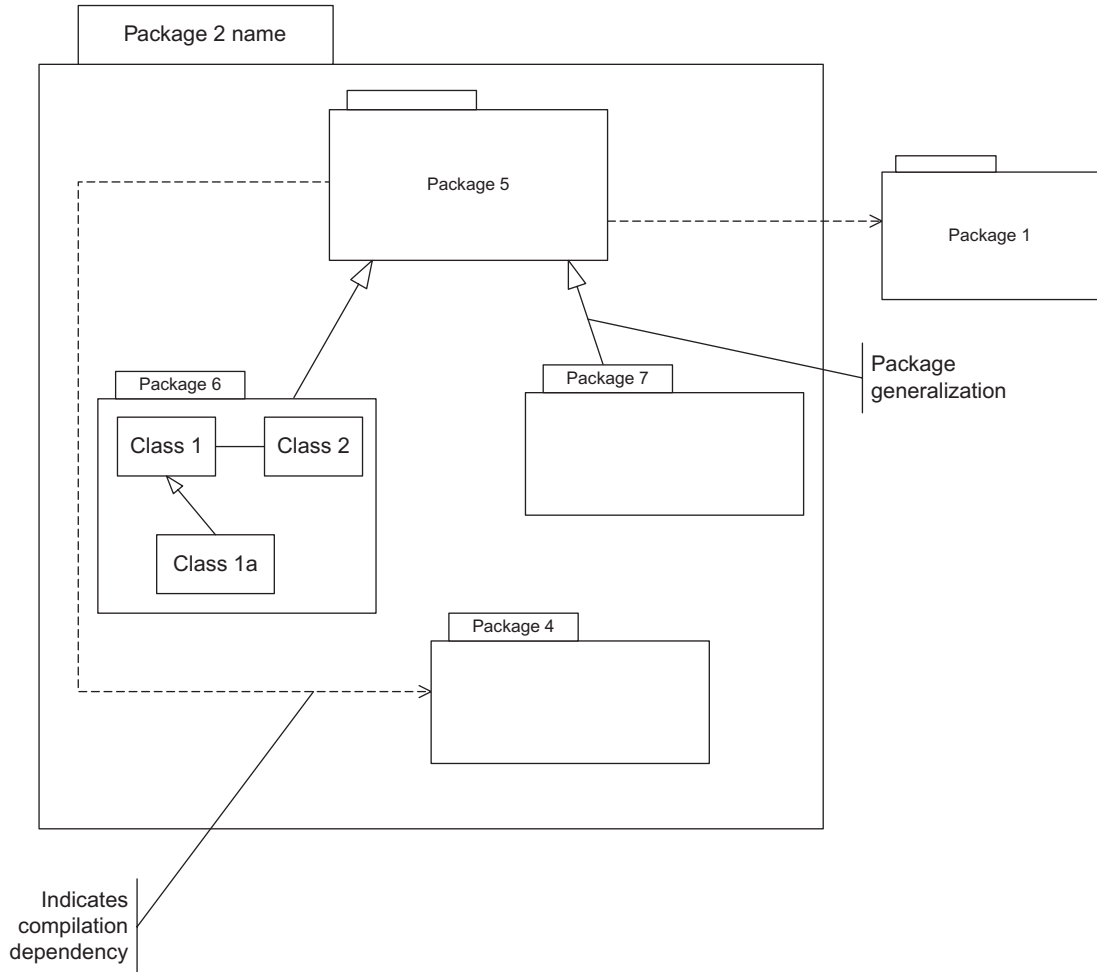
Implementation Diagrams

Implementation diagrams show the runtime dependencies and packaging structure of the deployed system.



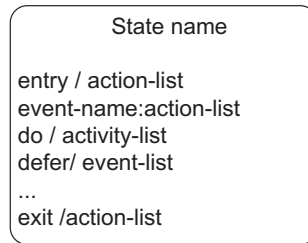
Packages

Shows a grouping of model elements. Packages may appear within class and other diagrams.

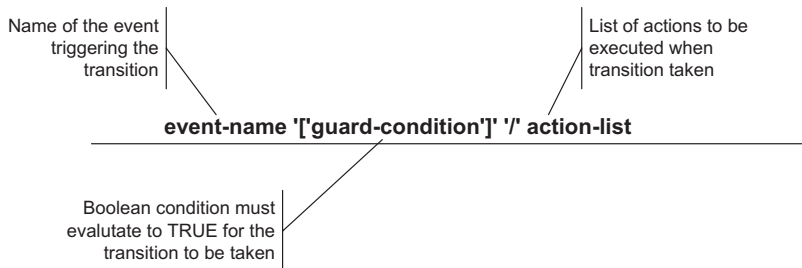


Statechart

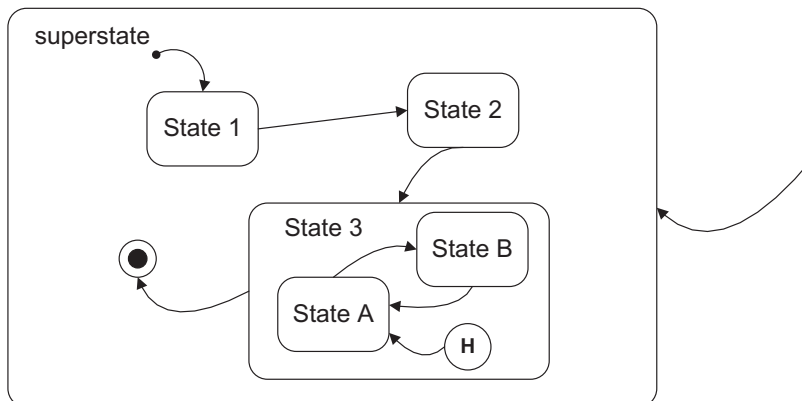
Shows the sequences of states for a reactive class or interaction during its life in response to stimuli, together with its responses and actions.



State Icon



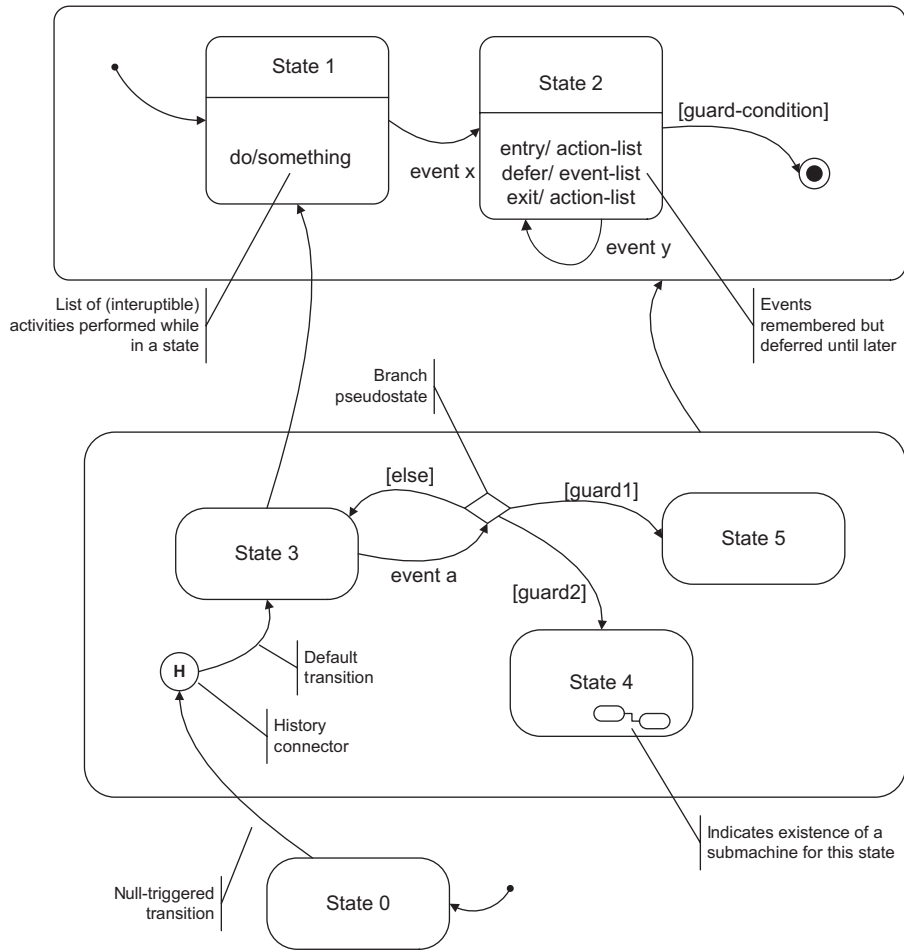
Transitions



Nested States

Statechart

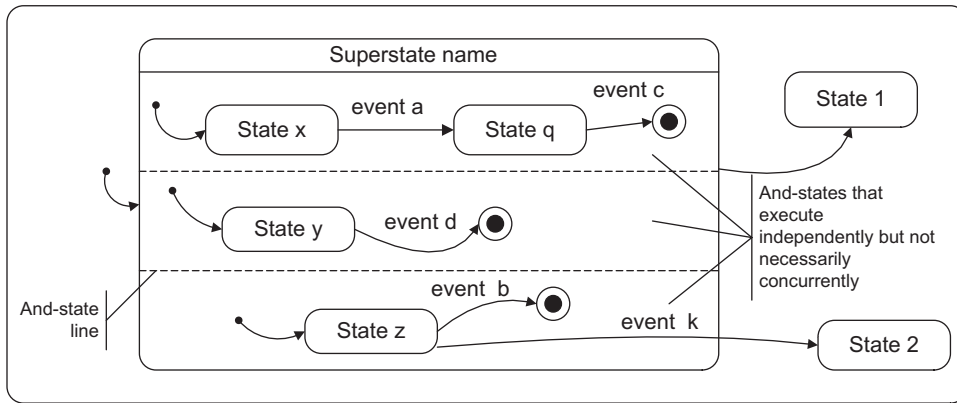
Shows the sequences of states for a reactive class or interaction during its life in response to stimuli, together with its responses and actions.



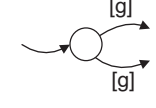
Sequential Substates

Statechart

Shows the sequences of states for a reactive class or interaction during its life in response to stimuli, together with its responses and actions.



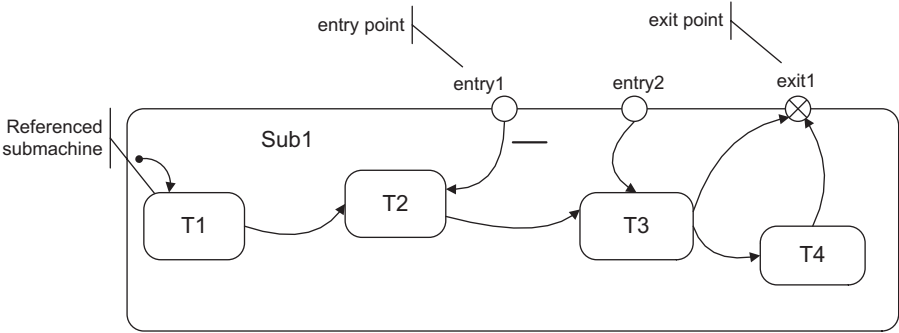
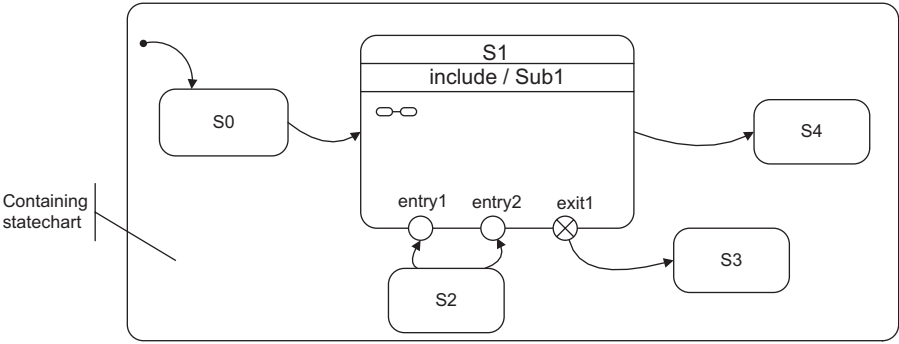
Orthogonal Substates (and-states)

Symbol	Symbol Name	Symbol	Symbol Name
 	Branch Pseudostate (<i>type of junction pseudostate</i>)		(Shallow) History Pseudostate
 	Terminal or Final Pseudostate		(Deep) History Pseudostate
	Fork Pseudostate		Initial or Default Pseudostate
	Join Pseudostate		Junction Pseudostate
	Choice Point Pseudostate		Merge Junction Pseudostate (<i>type of junction pseudostate</i>)
			Entry Point Pseudostate
			Exit Point Pseudostate

Pseudostates

Statechart

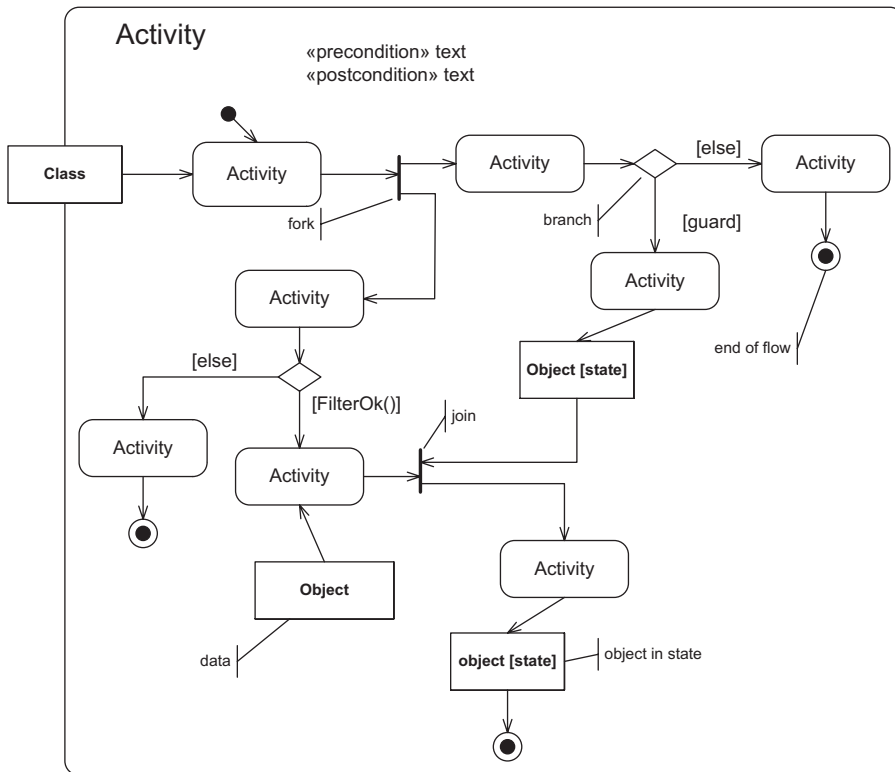
Shows the sequences of states for a reactive class or interaction during its life in response to stimuli, together with its responses and actions.



Submachines

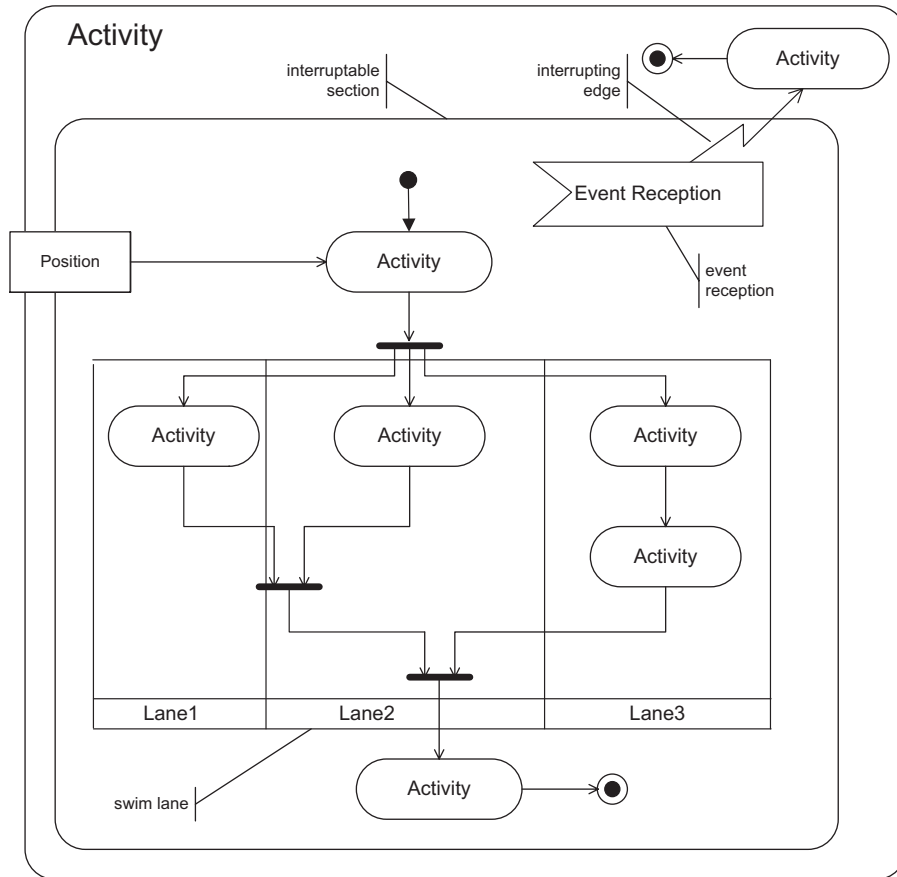
Activity Diagrams

Activity diagrams are a behavioral diagram based on token flow semantics and include branching, forks, and joins.



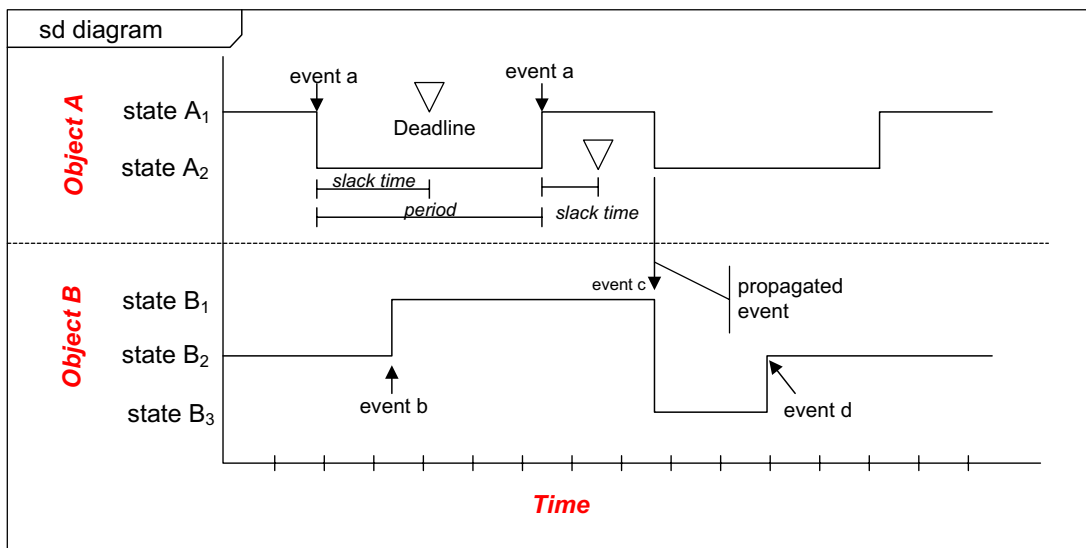
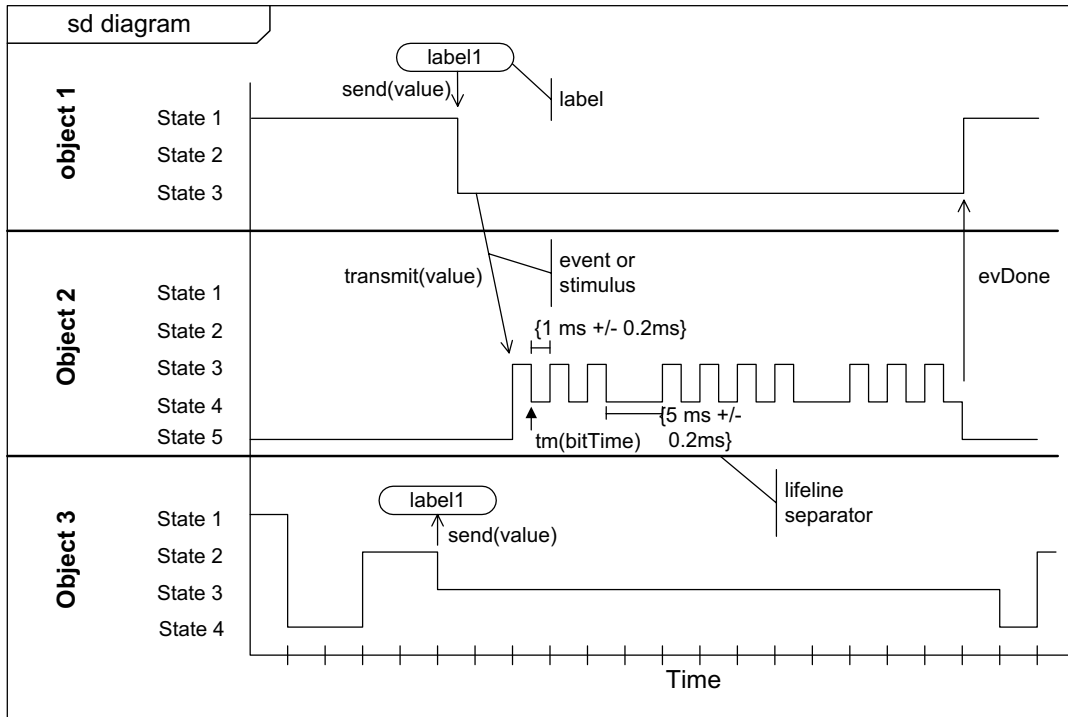
Activity Diagrams

Activity diagrams are a behavioral diagram based on token flow semantics and include branching, forks, and joins.



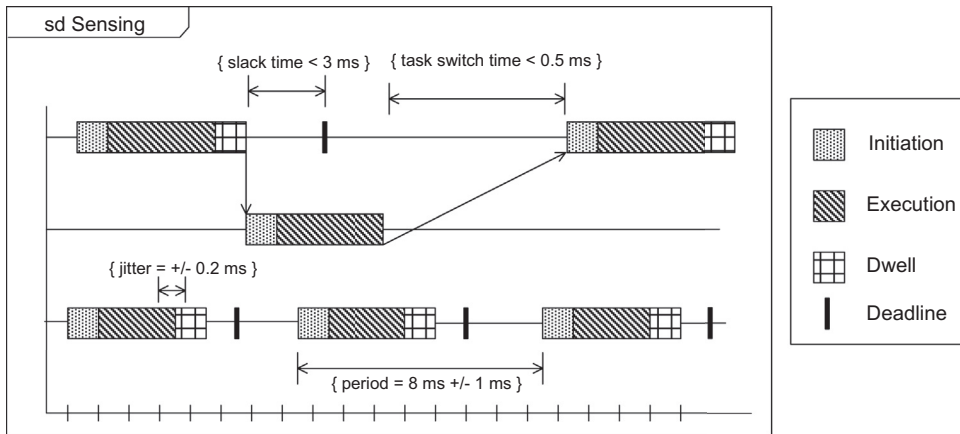
Timing Diagrams

Timing diagrams show the explicit change of state or value along a linear time axis.
(Timing diagrams are new in the UML 2.0 standard.)

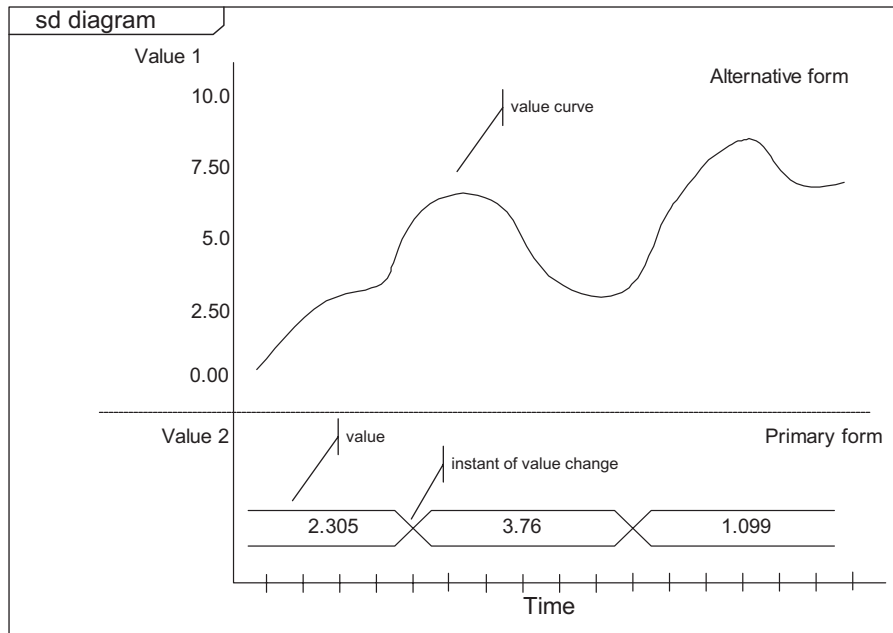


Timing Diagrams

Timing diagrams show the explicit change of state or value along a linear time axis.
(Timing diagrams are new in the UML 2.0 standard.)



With Shading



Continuous Values

Index

Note: Page numbers followed by “f” denote figures; “t” denote tables; “b” denote boxes.

A

- above-surface infrared sensors (ASIs), 187
- above-surface radars (ASRs), 187
- above-surface vehicle detector, 187, 188f
- adaptor pattern, 247–250, 249f–250f
- acquire image collaboration
 - watchdog pattern, 435–437, 440f
 - nouns and causal agents, 378–381, 381f–382f
 - services and messages, 384–385, 386f
- acquire image use case, 189, 193b–194b, 374, 378
- action semantics, 18–19
- activities, 19, 35
- activity diagrams, 20, 20f, 52, 486f–488f
 - operators, 21
 - pseudostate, 20, 484f–485f
 - token execution semantics, 20
- architectural design phase, 60f
 - concurrency and resource view, 58–59
 - dependability view, 58–59
 - distribution and deployment view, 58–59
 - Harmony architectural views, 58, 59f
 - pattern applications, 59
 - subsystem and component view, 58–59
 - UML diagrams, 58

- architectural framework (AF)
 - standards
 - capability viewpoint, 70
 - C4ISR, 69
 - data and information viewpoint, 72
 - DoD, 69
 - DoDAF, 69, 87
 - logical data model, 72
 - MODAF standard, 69
 - operational viewpoint, 70–71
 - project views, 72
 - service view, 71
 - system interoperability, 68
 - system viewpoint, 71
 - UPDM, 69, 72
- artifacts, 35
- associations, 471f
 - aggregation, 12, 470f–471f
 - association end, 10
 - association labels, 10
 - composition, 12–13, 470f–471f
 - generalization, 13–14, 471f–472f
 - multiplicity, 10
 - pointer, 10
 - role names, 10
- automated target recognition (ATR), 192
- averaging filter, 257

B

- basic class diagram, 5–6, 5f
- behavioral elements
 - action semantics, 18–19
 - activity, 19
 - activity diagrams, 20, 20f

- initial pseudostate, 20
- operators, 21
- pins, 21
- token execution semantics, 20
- asynchronous event transfer, 22–23
- call event, 22
- change event, 22
- classifiers, 18
- FSM
 - Mealy-Moore state machines, 22
 - reactive elements, 21–22
- interactions
 - communication diagrams, 26, 474f
 - decomposed lifeline, 27–28, 30f
 - “high-level” sequence diagram, 27, 29f
 - interaction fragments, 26–27, 28f
 - referenced interaction fragment, 27, 29f
 - sequence diagrams, 26, 27f
 - timing diagrams, 26
- model elements, 18
- operations and methods, 19
- reactive elements, 21–22
- “run-to-completion” semantics, 19
- signal event, 22
- state diagram
 - and-states, 484, 25–26, 25f
 - basic elements, 22, 23f
 - composite states, 24–25
 - event handler, 22–23

behavioral elements (*Continued*)
 event signature, 22
 execution order, 22
 guard, 24
 internal transitions, 24
 OffHook state, 24–25
 or-states, 24
 reactive elements, 21–22
 Rhapsody, 23
 time events, 24
 triggered operations, 22–23
 swimlanes, 21
 time event, 22

C

Capability Maturity Model
 Integration for Development (CMMI-DEV) standard, 88
 fundamental process, 85
 Harmony process mapping, 86, 86f
 maturity levels, 85, 85t
 requirements management process, 86
 CDS. *See* Coyote Datalink Subsystem (CDS)
 chain of responsibility (CoR)
 pattern, 423
 CHAP. *See* Coyote Hellfire Attack Payload (CHAP)
 CMPCS. *See* Coyote Mission Planning and Control System (CMPCS)
 collaboration design, 246f
 acquire image chain, responsibility pattern, 423, 425f
 acquire image command pattern, 429, 430f
 acquire image command structure, 428f, 429
 analysis-level collaboration, 243
 averaging filter, 257
 center-surround neural network error minimization, 257
 command pattern, 261, 261b–262b, 262f–263f
 configuration commands and direct requests, 429
 CoR pattern, 423
 data bus pattern, 259, 259b, 260f–261f

delegation pattern strategy, 247, 248f–249f
 design criteria, 245
 design patterns, 244
 Detect Vehicle
 adapter pattern, 423, 424f
 data bus pattern, 427–428, 427f
 Gaussian sharpening, 256
 global semaphore, 428
 Harmony design process, 244, 245f
 instantiated pattern, 244
 interface abstraction pattern strategy, 247–251, 249f–250f
 intersection controller, 258–259, 425
 iterative prototypes, 244
 light assembly connection topology, 424–425, 426f
 light behavior, 426
 median filter, 257
 observer pattern, 259, 424–425
 pattern classification, 246
 reconnaissance management subsystem, 256
 refactoring, 244–245
 responsibility pattern chain, 257, 257b–258b, 258f
 Roadrunner detect vehicle collaboration, 255–256, 256f
 Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance standard (C4ISR), 69
 commercial off-the-shelf (COTS) hardware, 39
 common mode failure, 149
 communication diagrams, 26
 concurrency and resource architecture
 activity diagram, coffee making, 231, 232f
 BuiltInTestTask stops, 229
 concurrency design model, 233, 234f
 concurrency metadata, 227, 228f
 concurrency unit, 225
 control system, 231, 233f

Coyote UAV concurrency architecture, 240b, 240f
 critical regions, 228
 definitions, 234, 235t
 guarded call, 228, 232
 Harmony process, 236, 236f
 activity types, 236
 cyclic executive pattern, 238
 dynamic scheduling patterns, 238
 focus of, 234
 Round Robin pattern scheduler, 238
 serialization, 237–238
 shared resource, 237–238
 Static Priority Pattern, 238
 task identification strategies, 236, 237t
 task synchronization points, 239
 Time Multiplexed Scheduler, 238
 independent design patterns and solutions, 241
 mutual exclusion problem, 227–228
 PacingTheHeartTask gets, 229
 passive element, 227–228
 PIM, 241
 priority inversion, 228–230, 229f–230f
 pseudo-concurrency, 225
 queuing requests, 228
 reconnaissance concurrency, 419–420, 421f
 recurrence property metadata, 227
 RMS, 227
 Roadrunner concurrency architecture, 239b–240b, 239f, 419, 420f
 scheduling patterns
 cyclic executive, 226
 interrupt driven, 226
 priority-based preemption, 227
 timed round Robin, 227
 task properties, 226
 unbounded priority inversion, 230–231
 urgency vs. criticality, 226, 226f

- container-iterator pattern, 247, 248f–249f
 - Coyote Datalink Subsystem (CDS), 458, 464–465
 - Coyote Hellfire Attack Payload (CHAP), 462–464, 463f
 - Coyote Mission Planning and Control System (CMPCS), 191–192, 192, 376–377, 456–457, 460–461
 - Coyote Reconnaissance Sensor Suite Payload (CSSP), 461–462
 - Coyote unmanned aerial vehicle system (CUAVS), 281–282
 - execute mission use cases, 282, 284f
 - fly UAV use cases, 282, 285f
 - high-level use cases, 282, 283f
 - optical surveillance scenario
 - area selection, centering and zooming, 294, 295f
 - automatic target tracking, 293, 294f
 - manual movement, camera, 292–293, 293f
 - Perform Optical Surveillance use case, 291–292, 292f
 - primary components
 - CDS, 458
 - CMPCS, 456–457
 - Coyote Payloads, 457
 - UAV, 455–456
 - requirements
 - CDS, 464–465
 - CHAP, 462–464, 463f
 - CMPCS, 460–461
 - CSSP, 461–462
 - flight modes, 459
 - mission modes, 460
 - UAV, 458–459
 - shared model, 316, 316f
 - subsystem models, 316, 317f
 - systems engineering (SE)
 - model
 - architecture design package, 315
 - behavioral and parametric requirements, 314
 - decomposition levels, 313, 315f
 - functional analysis model, 314
 - software and hardware specification model, 315
- D**
- Department of Defense Architectural Framework (DoDAF), 33, 68
 - dependability architecture, 175–176
 - design patterns, 151
 - analysis pattern, 155
 - architectural patterns, 156
 - definition, 155
 - design abstraction, 156
 - functional system, 156
 - pattern hatching, 157, 158f
 - quality of service, 155
 - structure of, 156–157
 - faults
 - errors, 150
 - failures, 150
 - Harmony process
 - conceptual design workflow, 152–153, 152f
 - criteria ranking, 154
 - design criteria, 153–154
 - design patterns and technologies
 - application, 155
 - design patterns and technologies
 - selection, 154–155, 154t
 - design solution validation, 155
 - high-level design, 152
 - initial model, 153
 - lightweight/heavyweight
 - patterns, 151
 - reconnaissance management
 - subsystem, 356
 - redundancy, 150
 - reliability architecture, 149–150
 - CUAV Control Aircraft
 - Attitude FMEA, 360–361, 366f
 - FMEA creation, 170–173, 171f–172f
 - Roadrunner Vehicle Detection
 - FMEA, 360, 364f–365f
 - system-level reliability, 170
 - Vehicle Sensor Assembly
 - first-cut architecture, 360, 363f
 - risk, 150
 - safety architecture, 150, 353–356, 356f
 - backward derivation checks, 158
 - channel architecture pattern, 163, 163–165, 164f
 - CUAV safety requirements,
 - safety risks, 169
 - data corruption checks, 167
 - fail-safe state, 351–352
 - failures, 168
 - FTA. *See* fault tree analysis (FTA)
 - gas mixer, 163–165
 - hazard analysis. *See* hazard analysis
 - heterogeneous redundancy
 - pattern, 358
 - homogeneous redundancy, 165
 - low-level surveillance, 357–358
 - monitor-actuator pattern, 165, 166f
 - reasonableness checks, 167–168
 - reconnaissance management
 - subsystem, 169–170, 360, 361f
 - safety/reliability analysis, 159
 - single channel protected
 - pattern, 165–167, 168, 167f–168f, 358–360
 - single point failure safe, 351–352
 - Timebase Fault, 159–160
 - traffic flow, 168
 - security and information
 - assurance architecture, 151
 - Coyote UAV GPS security
 - analysis, 362, 368f
 - cybersecurity, 173
 - key metaclasses, 173, 175f
 - patient data theft, 174, 176f
 - Roadrunner security analysis,
 - wireless communication, 361–362, 367f
 - threat analysis table, 174, 177f
 - threat modeling workflow, 173
 - UML security profile
 - metamodel, 173, 174f
 - unintended behavior, 173

detailed design, 61–62, 63f, 244
 acquire image collaboration,
 watchdog pattern, 435–437,
 440f
 any state pattern, 263
 TrafficModeManager, 429,
 432f
 behaviorally equivalent state
 machine, 264, 265f
 exceptions
 definition, 268
 error handling, 267–268
 Pedestrian light assembly state
 machine, 435, 439f
 Vehicle Light Assembly state
 machine, 435, 438f
 exception state pattern, 269, 269f
 Harmony process, 251, 252f
 ImageManager, 264, 429, 433f
 language-specific exception
 handling mechanism, 264
 latch state pattern, 253,
 253f–254f
 light assemblies and lights,
 270–271, 272f
 lower and-state replication, 431
 Pedestrian light state machine,
 434, 436f
 polling state pattern, 266,
 266–267, 266f–267f
 sensor payload device driver,
 430, 435f
 run-to-completion behavior, 431
 safety checks, 254–255, 255f
 sensor device driver taxonomy,
 429, 433f
 watchdog pattern, 435–437,
 440f
 SensorPayloadDeviceDriver
 class, 264
 sensor payload device driver
 state machine, 429–430,
 434f, 438, 441f
 StatefulClient, 270, 271f
 StatefulServer, 270, 270f
 state machine, 252
 TrafficModeManager state
 machine, 262, 263, 264f,
 429, 431f
 transitions, 265–266
 upper and-state replication, 431

VehicleLightAssembly, 434–435,
 437f
 Vehicle light state machine, 434,
 436f
 watchdog pattern, 271, 271–272,
 272b, 273f
 WatchPuppy puppyID attribute,
 273, 274f
 WatchPuppy state machine, 438,
 441f
 distribution architecture
 asymmetric architectures, 219–220
 communication protocol, 219–220
 Coyote UAV distribution
 architecture
 Airborne Datalink internal
 structure, 413, 414f
 CUAV distribution with
 proxies, 415, 417f–418f
 ground-aircraft distribution
 architecture, 411, 413f
 high-speed data link, 221
 marshal and unmarshal service
 requests and responses, 221
 port proxy pattern, 413–414,
 415f
 port proxy pattern problem,
 222, 222f
 port proxy pattern structure,
 222–223, 223f
 UAV subsystem architecture
 predistribution, 415, 416f
 embedded systems, 220
 high-speed and low-speed
 connections, 224
 Roadrunner distribution
 architecture, 220b–221b
 intersection controller
 distribution model, 411,
 412f
 serial interface structure, 411,
 412f
 symmetric architectures, 219
 DoDAF. *See* Department of
 Defense Architectural
 Framework (DoDAF)

E

earliest deadline first (EDF), 238
 evening low volume mode (ELVM),
 295–296, 298f–299f

F

fault tree analysis (FTA)
 advantages, 162
 automotive braking system, 262,
 162f
 CUAV FTA, target
 misidentification, 358, 359f
 logical operators, 159–160, 160f
 notational elements, 159, 159f
 pedal input fault, 162, 163f
 Rational Rhapsody tool, 160
 requirements and design
 elements, 160
 Roadrunner FTA
 advantages, 351–352
 All Lights Green fault, 349,
 350f
 All Lights Off fault, 349–351,
 352f
 All Lights Off with safety
 measures, 353, 355f
 Walk and Cross Green fault,
 349, 351f
 finite state machine (FSM)
 Mealy-Moore state machines, 22
 reactive elements, 21–22
 fixed cycle mode (FCM), 385, 387,
 387f
 controller and single light, 387,
 390f
 TrafficController state
 machine, 387, 391f
 nanocycle plan, 386
 pedestrian buttons, 394–395, 401f
 PedestrianSensor and
 VehicleSensor, 395, 402f
 TrafficController, 394–395,
 402f
 pedestrian requests, 396, 403f
 Test7_3 outcome, 396, 406f
 TrafficController, 396, 404f
 primary and secondary roads,
 388, 391f
 execution output, 389, 393f
 TrafficController state
 machine, 388, 392f
 running tests for, 387, 389f
 second turn lane, 393–394,
 398f–400f
 state machines for, 387, 388f

- tests output, 387, 390f
- TrafficController state machine
 - with and-states, 390–392, 393f
- turn lanes, car sensor, 394f
 - ProcessPTurn submachine, 392, 396f
 - Test4_1 output, 393, 398f
 - TrafficController, 392, 395f
 - VehicleSensor and TCTester4, 393, 397f
- fixed cycle time, mode 2
 - Coyote UAV, 188
 - Default Cycle Times, 188, 189t, 371, 372t
 - parameters, 188, 189t, 371, 372t
 - per-use-case application, 189–190
 - reconnaissance management use cases, 189, 191f
 - Roadrunner fixed cycle mode nouns, 372, 373t–374t
 - timing diagram, 188, 190f
 - functional requirements, 29–30
- G**
 - Gaussian filtering, 257
 - Gaussian sharpening, 256
 - generalization, 13–14
 - global positioning system (GPS), 190–191
- H**
 - Harmony development process
 - activities, 35
 - artifacts, 35
 - change management, 44
 - configuration management, 44
 - control project, 44
 - COTS hardware, 39
 - development environment, 43
 - development iteration, 37
 - executable modeling tools, 38–39
 - full Harmony process, 35–36
 - general Harmony process/
 - Harmony hybrid process, 39, 40f
 - hand-off activity, 39
 - Harmony/ESW, 35–36
 - delivery process, 42, 43f
 - hazard analysis, 35–36
 - high-quality systems, 34
 - iterative part of, 41, 42f
 - iterative prototype, 37
 - macrocycle, 38
 - microcycles, 35–36
 - mission statement, 37
 - nanocycle, 35–36, 38–39
 - phases, 35
 - pre-iteration planning, 42
 - process, definition, 35
 - process elements, 35, 36f
 - QA audit, 44
 - software-only Harmony process, 41–42
 - spiral lifecycle, 37
 - stakeholder requirements, 43
 - systems engineering activities, 40, 41f
 - TDD, 38–39
 - time frames, 37, 38f
 - waterfall lifecycle, 36–37
 - worker guidance and delivery, product, 34
 - worker roles, 35
 - work products, 35
 - Harmony process
 - DoDAF, 33
 - hand-off
 - IPT, 50
 - shared model, 50
 - subsystem specifications, 46, 49f
 - Harmony development process. *See* Harmony development process
 - MODFAF, 33
 - software workflow
 - abstraction levels, 58
 - abstract view, 50, 51f
 - analysis model, 57
 - architectural level, 58. *See also* architectural design phase
 - detailed design, 58, 61–62, 63f
 - “essential” properties, 50–51
 - high-fidelity modeling phase. *See* high-fidelity modeling
 - increment review workflow. *See* increment review workflow
 - interdisciplinary subsystem teams, 50
 - iteration planning, 51–52
 - MDA, 50–51
 - mechanistic/collaboration level, 58, 60–61, 61f
 - PSM, 57
 - quality of service constraints, 58
 - use case/user story analysis workflow, 479. *See also* User story analysis workflow
 - verification and validation, 62, 64f
 - SysML, 33
 - systems engineering Harmony workflow. *See* systems engineering Harmony workflow
 - Harmony requirements capture process
 - “black box” step, 105–106
 - Harmony systems engineering workflow, 105, 106f
 - V-process, 105, 105f
 - “white box” analysis, 105–106
 - hazard analysis, 352–353
 - All Lights Green hazard, 353, 353f
 - All Lights Off fault, 353, 354f
 - CUAV Target Misidentification hazard, 360, 362f
 - structural overview, 357
 - Walk and Cross Green hazard, 353, 354f
 - high-fidelity modeling, 55f
 - agile development approach, 181
 - banking system, 56
 - coherent use cases, 181
 - essential software elements, 54–55, 56t
 - execution and unit test, 57
 - Harmony nanocycle, 57
 - Harmony spiral model, 181
 - navigation system, 56
 - object collaboration, 474f–475f
 - object identification. *See* object identification
 - optimization decisions, 56
 - PIM, 54–55
 - properties, 182
 - purpose of, 179
 - sequence diagrams, 57
 - source code baseline, 57
 - UML, structured design, 180–181, 180f

I

IEC 61508
 high demand systems, 73
 low demand systems, 73
 process requirements
 ISO 26262 standard, 75
 safety lifecycle, 74, 74f
 semiformal methods, 75, 76t
 structured process, 75
 V-style lifecycle, 74–75, 75f
 safety management methods, 73
 safety risk, 72–73
 SILs, 73, 73t
 increment review workflow
 architecture scaling, 64–65
 architecture selection and
 implementation, 64–65
 hardware-software
 co-development, 65
 operational contracts, 65
 project management, 65
 refactoring, 64–65
 risk management plan, 65
 work products, Party phase, 63
 industry standards
 AF standards. *See* architectural
 framework (AF) standards
 CMMI-DEV standard. *See*
 Capability Maturity Model
 Integration for Development
 (CMMI-DEV) standard
 craft, 67–68
 DO-178B standard, 79, 87–88
 decision coverage, 78
 DO-178 objectives, 76, 77t
 Harmony process, 83–84, 83f
 lifecycle data, 79, 80t–81t
 modified condition, 78
 no certification credit, 78
 requirements-based testing, 78
 safety levels, 76, 77t
 statement coverage, 78
 structural coverage, 78
 verification, 76
 DO-178C standard
 avionics systems, 79
 common memory-related
 problems, 81–82
 DO-331 contents, 79–81
 dynamic polymorphism, 81
 Harmony process, 83–84, 83f

lifecycle data, 79, 80t–81t
 non-normative model
 elements, 79–81
 object-oriented (OO)
 programming, 81
 supplements, 79
 tool qualification, 82–83
 DoDAF, 68
 IEC, 68
 IEC 61508, 87. *See also* IEC
 61508
 IEC 62304, 84, 88
 safety-critical software, 68
 software engineering, 67–68
 Interdisciplinary Project Team
 (IPT), 50
 intersection controller. *See*
 Roadrunner™; Intersection
 Controller (RIC) system

K

key concepts strategy, 185

M

mean time between failure
 (MTBF), 149
 mechanistic design phase, 60–61,
 61f, 112
 model-driven architecture (MDA),
 50–51

N

notation
 use case diagram (class
 notation), 107, 108f
 use case diagram (standard), 107,
 107f

O

object identification
 application, 186–216
 control elements, 186
 key concepts, 185
 nouns and causal agents, 182,
 184
 acquire image collaboration,
 378–381, 381f–382f
 CMPCS, 191–192, 192,
 376–377
 collaboration elements, class
 diagram, 372–374, 375f

CUAV acquire image nouns,
 378, 379t–380t
 fixed cycle time, mode 2. *See*
 fixed cycle time, mode 2
 image acquisition and
 processing, 192–193, 377
 UAV, 190–191, 192, 374–376
 vehicle detector, 187, 188f,
 369–370, 370t, 371f
 object discovery strategies, 182,
 183t
 persistent information, 185–186
 physical devices, 185
 real-world items, 184–185
 services and messages, 184,
 193b–194b
 acquire image collaboration,
 384–385, 386f
 classes/application-specific
 data types, 382
 Detect Vehicle strategy, 383,
 384f
 fixed cycle mode, 383–384,
 385f
 FixedCycleModeController,
 383–384
 TDD approach. *See* test-driven
 development (TDD)
 approach
 transactions, 185
 visual elements, 186
 Object Management Group
 (OMG), 1

P

packages, 16–17, 17f, 481f–482f
 Perform Area Search (PAS) use
 case, 122–123, 124f–125f
 aircraft performs area search,
 127, 130f
 aircraft takes off, 126–127, 128f
 area search setup, 129, 131f
 black box system-actor
 interfaces, 124
 context diagram, 123–124, 126f
 Coyote system, 129
 high-level view, 124, 127f
 multiple sequence diagrams, 124
 “rainy day” scenarios, 129
 remote navigation, 127, 129f
 “sunny day” scenarios, 129

white box view, 326–327,
328f–329f
physical devices strategy, 185
platform-independent model (PIM),
54–55, 241
platform-specific model (PSM), 57
port proxy pattern, 413–414, 415f

Q

quality assurance (QA) audit, 44
quality of service (QoS), 29–30

R

rate monotonic scheduling (RMS),
227
reconnaissance management
 subsystem, 169–170, 256,
 356, 360, 361f
refactoring, 244–245
relations
 associations, 10
 aggregation, 12
 association end, 10
 association labels, 10
 composition, 12–13
 dependency, 14–15, 16f
 generalization, 13–14, 14f
 MissileTransaction and Missile
 classes, 10
 multiplicity, 10–11
 pointer, 10
 role names, 10
 system-wide behavior, 9–10
 Target class, 10, 11f
requirements
 formal languages, 90
 high-reliability process, 89
 operational architecture, 89–90
 state machines
 activity diagrams, 101–102
 actors initialization, code list,
 299
 advantages, 101
 architectural specification, 104
 Coyote Flight requirements,
 operational contracts, 107,
 108t
 derived use case activity
 diagram, 309, 311f
 evening low volume mode use
 case, 296–298, 298f

fixed cycle mode, 297, 299,
300f, 301, 302f
fully constructive scenario,
102
guidelines, 102
Harmony nanocycle approach,
102–109, 296–297
Harmony requirements capture
 process. *See* Harmony
 requirements capture
 process
logical use case interfaces,
308, 309f
partially constructive scenario,
102
Pedestrian_a class, 301, 301f
Perform Optical Surveillance
 use case, context diagram,
 306–307, 308f
pre- and postcondition
 interfaces, 308–309, 310t
primary traffic state machine
 management, 305, 306f
reactive behavior, 101
reflexive messages, 108–109
secondary pedestrian
 simulation, 305, 307f
statechart, 309–311,
 482f–485f
state icon, 482f
Statemate™, 101
submachines, 485f–486f
turn lane modes, 104b,
 302–304, 303f–304f
use case block statechart,
 311–312, 312f
use case diagram (class
 notation), 107, 108f
use case diagram (standard),
 107, 107f
use case state machine
 specification, 103b–104b,
 295–296, 298f–299f
Vehicle_a class, 301, 301f
UML and SysML
 abstraction levels, 98
 actor-system interactions, 286
 asynchronous event, 288–289
 condition mark, 289
Coyote UAV system, 95b–96b.
 See also Coyote unmanned

aerial vehicle system
 (CUAVS)
fault equivalence classes, 98
fixed cycle mode, 285
functional analysis, 97
mapping requirements, 95b,
 280–281, 281f–282f
minimal spanning scenario set,
 98
operational context class
 diagram, 99
operational view, Coyote UAV
 optical surveillance
 scenarios, 100b
quality of service, 97b,
 283–284, 287f
“rainy day” scenarios, 98
requirements management
 tools, 90–91
responsive cycle mode use
 case, 284, 288f, 290
Roadrunner Traffic Light
 Control System. *See*
 Roadrunner Traffic Light
 Control System
specification view, use case
 description, 100b, 294–295,
 296f–297f
table creation, 96b, 283, 286f
Traffic light scenario,
 289f–291f
use case diagram, 91, 91f
use case trace links, 92, 92f
word processing and spread-
 sheet tools, 90–91
Roadrunner™ Intersection
 Controller (RIC) system
 adaptive mode, 449, 449t
 configuration parameters, 444,
 445t
 emergency transmitter, 444
 evening low volume mode, 446
 fixed cycle time, 446–447, 447t,
 448f
 front panel display (FPD), 454
 Ethernet and digital I/O ports,
 453
 keys and knobs, 452, 453f
 menus, 452
infrared and radar vehicle
 detector, 450, 450f

- Roadrunner™ Intersection
 - Controller (RIC) system
(*Continued*)
 - intersection controller, 444f
 - operational modes, 443
 - pedestrian lights and buttons, 443
 - priority vehicle transmitter, 443–444
 - pedestrian light and sensor, 451, 452f
 - performance statistics, 444, 446t
 - remote communications, 454
 - responsive cycle mode, 447–449
 - safe mode, 446
 - subsurface detectors, 449
 - uninterruptible power supply, 454
 - vehicle detector, types, 449
 - vehicular traffic light, 450–451, 451f
- Roadrunner Traffic Light Control System
 - requirement types, 92b–94b, 94f, 277, 278f
 - responsive cycle mode use case, 121, 122f
 - Roadrunner use cases, 121, 121f, 123f
 - system-level behavior, 122
 - use case identification, 94b–95b
 - configure system and traffic management, 277, 279f
 - detect vehicle and pedestrian, 277–279
 - emergency and priority mode, 279
 - generalization/specialization relations, 280
 - intersection controller, 277
 - operational environment, 280
- S**
- safety integrity levels (SILs), 73, 73t
- sequence diagrams, 26–27, 27f, 29f, 44–46, 57, 475f–479f
- shared model
 - high-level packages, 137b, 337
 - internal suborganization, 139
 - IntersectionController_Class
 - interface diagram, 338, 342f
 - logical interface, 143f
 - physical interface, 144f
 - Roadrunner downstream
 - engineering package structure, 337–338, 340f
 - Roadrunner interfaces, 338, 341f
 - Roadrunner message format, 338–339, 342t
 - VehicleLightAssembly interface
 - diagram, 338, 341f
 - Vehicle Light Assembly
 - physical interfaces, 339–340, 343f
- spiral lifecycle, 37
- standard modeling language, 3–4
- structural elements and diagrams
 - attributes, 4–5, 7
 - class diagram, 5–6, 5f
 - configuration management, 16
 - DeliveryController class, 6
 - instance, 5
 - interface, 6, 7f
 - event receptions, 6
 - lollipop notation, 6–7
 - operations, 6
 - parameters and signature, 6
 - stereotype, 6–7, 473f–474f
 - methods, 4–5, 7
 - packages, 16–17, 17f
 - ports, 9, 9f
 - relations
 - aggregation, 12
 - associations, types, 10
 - composition, 12–13
 - dependency, 14–15, 16f
 - generalization, 13–14, 14f
 - MissileTransaction and Missile classes, 10
 - multiplicity, 10–11
 - role names, 10
 - system-wide behavior, 9–10
 - Target class, 10, 11f
 - structured classes, 8–9, 8f, 472f–473f
 - subsystems, 17–18
 - UML 2.0, components, 18
- subsurface passive loop inductors (SPLIs), 187
- systems architecture
 - architectural design, 112
 - aspects of, 112, 112f
 - concurrency and resource view, 112–113
 - definition, 111–112
 - design criteria, 111–112
 - detailed design, 112
 - distribution, dependability and deployment view, 112–113
 - hand-off, downstream engineering
 - cross-disciplinary interface, 145–146
 - deployment mode, 142–143
 - derived requirements, 144
 - internal nested packages, 141
 - navigation subsystem, 340–342, 344f
 - Navigation subsystem
 - deployment architecture, 343–344, 345f
 - nested packages, 142, 482f–483f
 - patient ventilator, deployment diagram, 143, 145f
 - Roadrunner Vehicle Light Assembly electronic interface, 343, 346f
 - shared model creation, 137b–140b, 143f–144f. *See also* shared model
 - “standard” ports, 143
 - subsystem responsibilities allocation, 136
 - SW-EE interface, metadata, 145–146, 147f
 - tasks, 140
 - text-based specifications, 136
 - Vehicle Light Assembly
 - deployment architecture, 343, 345f
 - ventilator deployment architecture, 143, 147f
 - ventilator high-level deployment architecture, 143, 146f
- Harmony process, 112
- mapping operational contracts, subsystems architecture analysis mode, 324
- Coyote vehicle service allocations, 328, 331f
- CUAV system allocated operations, 328, 330f

- Flight Management interfaces
 - diagram, 330, 332f
 - “goodness” of, 120
 - intersection controller subsystem, 324
 - nested sequence diagram, 324, 325f–326f
 - operations, Roadrunner architecture, 324–325, 327f
 - outer-most system level, 323
 - PAS use case. *See* Perform Area Search (PAS) use case
 - Roadrunner TrafficLight Control System. *See* Roadrunner TrafficLight Control System
 - system service decomposition, 323
 - UML/SysML, ICDs, 129–133, 132f, 132f
 - Video assembly service allocations, 328, 332f
 - mechanistic design, 112
 - subsystem and component view, 112–113
 - subsystem identification
 - class diagram, 318, 319f
 - CUAV system architecture diagram, 318, 319f, 320
 - CUAV vehicle subsystem, 321, 321f, 322t
 - ground station, 320–321, 320f
 - guidelines, 119
 - Harmony process, 119
 - logical architecture, 118
 - ports, 120
 - Roadrunner model, 120
 - Roadrunner system architecture diagram, 316–317, 318f
 - stereotypes, 316–317
 - structured class, 118–119
 - system-to-subsystem decomposition, 322–323
 - subsystem use cases
 - identification, 135f
 - algorithm, 134
 - Configure System use case, 134, 136f, 334, 334f
 - CUAV aircraft architecture, 135, 141f
 - CUAV ground station architecture, 135, 142f
 - CUAV Perform Area Search use case, 335, 336f
 - CUAV system architecture, 135, 140f
 - Detect Vehicle system use case, 134, 136f, 331–333, 333f
 - Flight Management subsystem use cases, 336–337, 338f–339f
 - IntersectionController subsystem, 135
 - Manual Attitude Adjust use case, 135, 138f, 334–335, 335f
 - navigation subsystem use cases, 335–336, 337f
 - operational and operational quality of service requirement, 134
 - operational context, 133
 - Perform Area Search use case, 135, 139f
 - requirements, 133
 - Roadrunner architecture, 134, 137f
 - subsystem-level use case, 333
 - “top-down” and “bottom-up” approach, 133
 - systems model organization, 313, 314f
 - architecture package, 115
 - canonical systems engineering, 115, 116, 117f–118f
 - Coyote UAV system. *See* Coyote unmanned aerial vehicle system (CUAVS)
 - downstream software engineering, 116
 - functional analysis, 115
 - high level package, 313
 - large project organization, 114, 116f
 - large/separate models, 114
 - multiple models, 114
 - nontrivial and unscheduled effort, 113
 - single model, 114
 - software analysis and design, 116–117
 - software-specific elements, 116–117
 - specification, 116–117
 - stakeholders for, 114, 115t
 - systems engineering Harmony workflow
 - build use case model, 46, 47f
 - executable model, 44
 - sequence diagrams, 44–46
 - system architecture design workflow, 46, 48f
 - system functional analysis, 45f
 - textual requirements, 44–46
 - Systems Modeling Language (SysML), 129–133, 132f, 33, 132f
- ## T
- test-driven development (TDD)
 - approach, 38–39
 - above-surface vehicle detectors, 214
 - Aircraft attitude, 214, 215f
 - Auto Adjust Attitude use case, 214–216, 216, 215f
 - balance gas mixing, 210, 210f
 - BuilderStep1 class, 197
 - Coyote UAV
 - AttitudeControl state machine, 403, 408f
 - ControlSurface state machine, 404, 409f
 - CUAV Auto Adjust Attitude collaboration, 402, 407f
 - design collaboration and code, 399
 - FCM. *See* fixed cycle mode (FCM)
 - Gas class, 197, 199f
 - gas.cpp code, 203–206
 - gas.h code, 198–203
 - gas mixing collaboration, 197, 198f
 - GasTester1, 197, 199f
 - GasTester2 state machine, 211, 212f
 - mix gases use case, 195
 - multiple gases addition, 206–209, 207f–209f
 - nanocycle iteration, 196–197, 212–214
 - nanocycle structure, 194

- O₂ delivery and N₂ balance gas, 196–197, 196f
 - setBalanceGasParams
 - implementation, 210–211
 - test cases execution, 198, 200f
 - Test3_1 execution and outcome, 211, 213f–214f
 - ventilator use cases, 195, 195f
 - timing diagrams, 26, 488f–489f
 - transactions identification, 185
- U**
- unified modeling language (UML), 467f–470f
 - behavioral diagrams, 2. *See also* behavioral elements
 - class diagram, 2, 468f–472f
 - common diagram elements, 468f
 - deployment diagram, 480f–481f
 - file-based C UML model, 180–181, 180f
 - functional diagram, 2
 - functional requirements, 29–30
 - interaction diagrams, 2
 - OMG, 1
 - production-quality code, 1
 - QoS requirements, 29–30
 - Rhapsody™, 3–4
 - sequence diagrams, 2
 - software-intensive system, 4
 - state diagrams, 2
 - structural diagrams, 2. *See also* structural elements and diagrams
 - types, 2, 2f
 - UML metamodel, 2–3, 3f
 - use case, 28
 - extends, 30
 - generalization, 30
 - Harmony process, 31
 - includes, 30
 - for medical anesthesia system, 30, 31f
 - scenario modeling, 31
 - specification, 32
 - unmanned aerial vehicle (UAV), 190–191, 192, 374–376, 455–456, 458–459. *See also* Coyote unmanned aerial vehicle system (CUAVS)
- US Department of Defense (DoD), 69**
- user story analysis workflow
 - advantages, 52
 - Harmony-SW process variant, 52, 53f
 - informal text and formal languages, 54
 - nontechnical stakeholders, 52
 - requirements, 54
 - specification, 52
 - state machines/activity diagrams, 54
 - utility function, 226
- V**
- Visio™, 349
 - visual elements identification, 186
- W**
- watchdog pattern, 271, 271–272, 272b, 273f, 435–437, 440f
 - waterfall lifecycle, 36–37