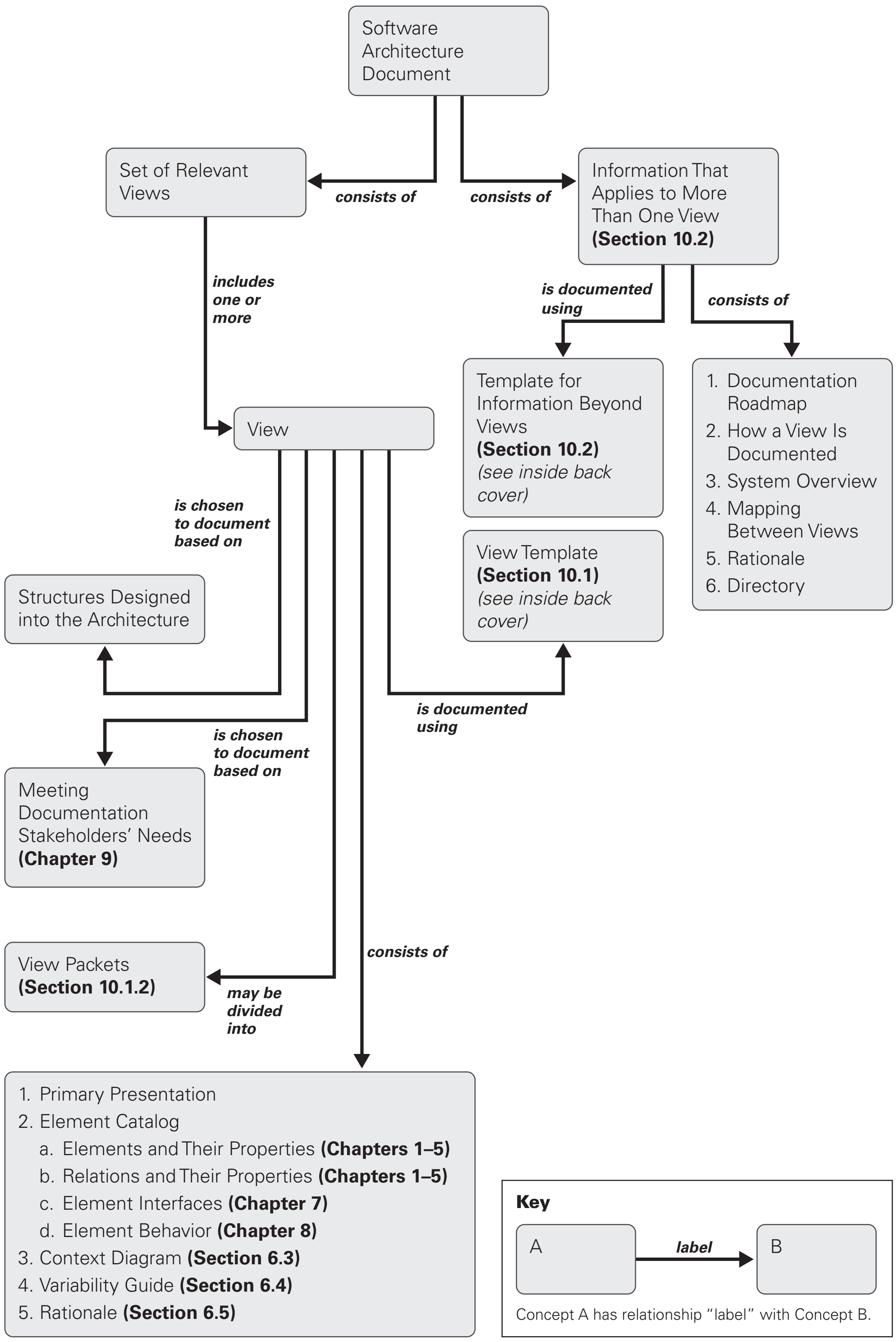
SEI SERIES IN SOFTWARE ENGINEERING

Documenting Software Architectures

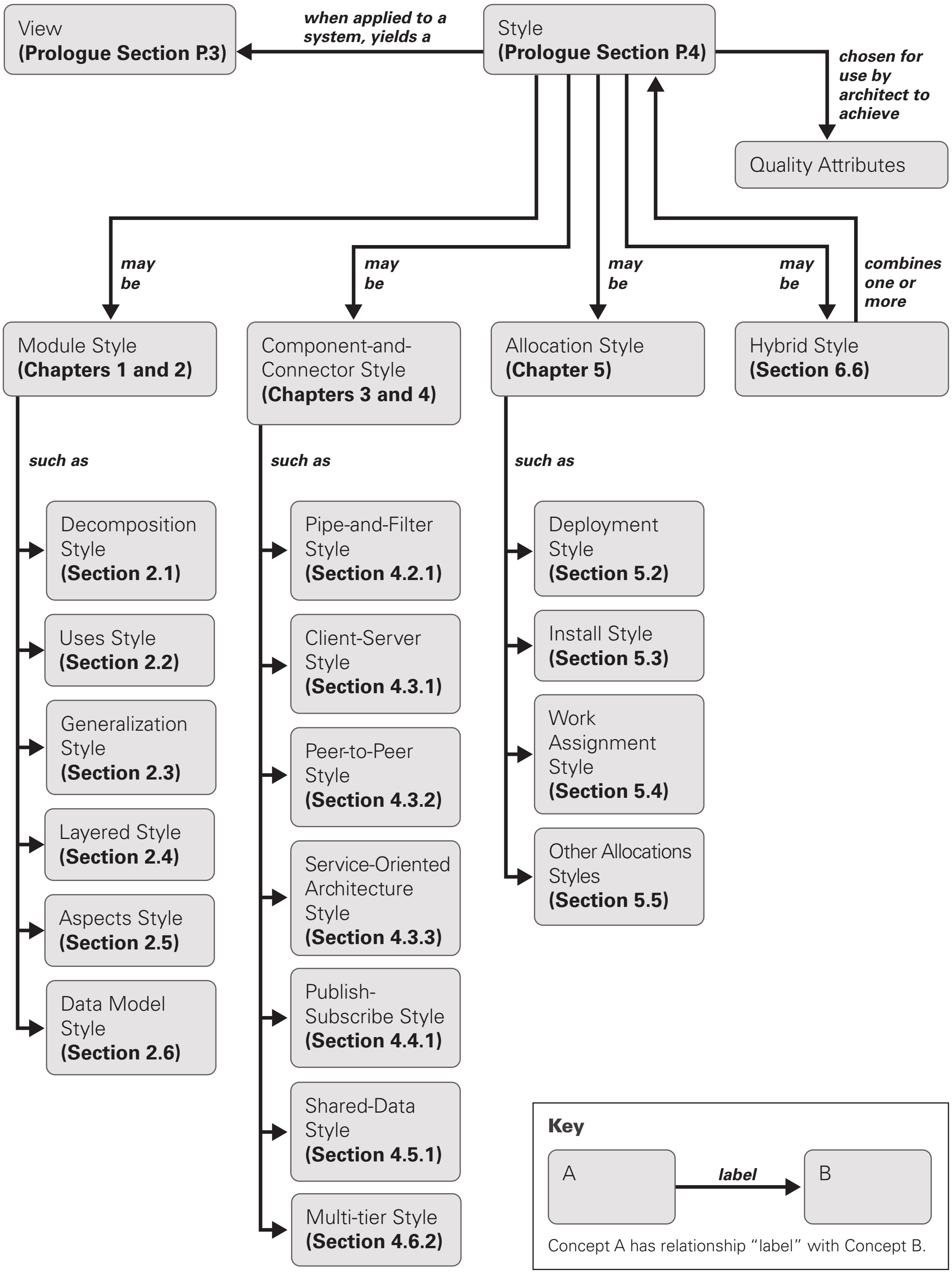
Views and Beyond

SECOND EDITION

Paul Clements • Felix Bachmann • Len Bass
David Garlan • James Ivers • Reed Little
Paulo Merson • Robert Nord • Judith Stafford



Software Architecture Document
consists of
consists of
Set of Relevant Views
Information That Applies to More Than One View (Section 10.2)
includes one or more
is documented using
consists of
Template for Information Beyond Views (Section 10.2) (see inside back cover)
1. Documentation Roadmap
2. How a View Is Documented
3. System Overview
4. Mapping Between Views
5. Rationale
6. Directory
View
is chosen to document based on
View Template (Section 10.1) (see inside back cover)
Structures Designed into the Architecture
is documented using
is chosen to document based on
Meeting Documentation Stakeholders' Needs (Chapter 9)
consists of
View Packets (Section 10.1.2)
may be divided into
1. Primary Presentation
2. Element Catalog
a. Elements and Their Properties (Chapters 1–5)
b. Relations and Their Properties (Chapters 1–5)
c. Element Interfaces (Chapter 7)
d. Element Behavior (Chapter 8)
3. Context Diagram (Section 6.3)
4. Variability Guide (Section 6.4)
5. Rationale (Section 6.5)
Key
A
label
B
Concept A has relationship "label" with Concept B.



View (Prologue Section P.3)
when applied to a system, yields a
Style (Prologue Section P.4)
chosen for use by architect to achieve
Quality Attributes
may be
may be
may be
may be
combines one or more
Module Style (Chapters 1 and 2)
Component-and-Connector Style (Chapters 3 and 4)
Allocation Style (Chapter 5)
Hybrid Style (Section 6.6)
such as
such as
such as
Decomposition Style (Section 2.1)
Uses Style (Section 2.2)
Generalization Style (Section 2.3)
Layered Style (Section 2.4)
Aspects Style (Section 2.5)
Data Model Style (Section 2.6)
Pipe-and-Filter Style (Section 4.2.1)
Client-Server Style (Section 4.3.1)
Peer-to-Peer Style (Section 4.3.2)
Service-Oriented Architecture Style (Section 4.3.3)
Publish-Subscribe Style (Section 4.4.1)
Shared-Data Style (Section 4.5.1)
Multi-tier Style (Section 4.6.2)
Deployment Style (Section 5.2)
Install Style (Section 5.3)
Work Assignment Style (Section 5.4)
Other Allocations Styles (Section 5.5)
Key
A
label
B
Concept A has relationship “label” with Concept B.

Praise for the First Edition of *Documenting Software Architectures*

"For many years, box and line diagrams have decorated the text that describes system implementations. These diagrams can be evocative, sometimes inspirational, occasionally informative, but are rarely precise and never complete. Recent years have brought appreciation for the importance of a deliberate structural design, or architecture, for a system. Now, in *Documenting Software Architectures,* we have guidance for capturing that knowledge, both to aid design and—perhaps more significantly—to inform subsequent maintainers, who hold over half the total cost of a system's software in their hands. Half of this cost goes into figuring out how the system is organized and where to make the change. A documented architecture is the essential roadmap for the system, leading the maintainer through the implementation jungle."

—*Mary Shaw, Alan J. Perlis Professor of Computer Science, Carnegie Mellon University*
Coauthor of Software Architecture: Perspectives on an Emerging Discipline

"Multiple software architecture views are essential because of the diverse set of stakeholders (users, acquirers, developers, testers, maintainers, inter-operators, and others) needing to understand and use the architecture from their viewpoint. Achieving consistency among such views is one of the most challenging and difficult problems in the software architecture field. This book is a tremendously valuable first step in defining analyzable software architecture views and frameworks for integrating them."

—*Barry Boehm, TRW Professor of Software Engineering*
Director, USC Center for Software Engineering

"There is probably no better set of authors to write this book. The material is readable. It uses humor effectively. It is nicely introspective when appropriate, and yet in the end it is forthright and decisive. The philosophical elements of the book are fascinating. The authors consider concepts that few others even are aware of, present the issues related to those concepts, and then resolve them! This is a tour de force on the subject of architectural documentation."

—*Robert Glass, Editor-in-Chief,* Journal of Systems and Software
Editor/Publisher, The Software Practitioner

"We found this book highly valuable for our work with our business units and would recommend it to anyone who wants to understand the needs for and improve their skills in describing software architectures for complex systems."

—*Steffen Thiel, Robert Bosch Corporation*

"Since our projects involve numerous stakeholders, documenting the architecture from various views is of particular importance. For this task, this book provides pragmatic and well-structured guidance and will be an important reference for industrial practice."

—*Martin Simons, Daimler Chrysler Research and Technology*

"Software architecture is an abstract representation of the most essential design decisions. It is expressed using concepts that are not directly visible in software implementation. How to identify these decisions? How to represent them? How to find the concepts that make complex software understandable? This excellent book is written by a group of expert architects sharing their experience and understanding of useful architectural concepts, essential design decisions, and practical ways to represent architectural views of complex software."

—*Alexander Ran, Principal Scientist of Software Architecture, Nokia*

"I particularly appreciate the major theme of the book: that a software architecture consists of a variety of different structures, each defined by a set of elements and a relationship among those elements. I further appreciate the authors pointing out why the diagrams that seem so beloved by today's software designers are often deceptive and of little value. (I frequently say that in software engineering every diagram takes a thousand words to explain it.) It was also refreshing to see an explanation of why 'levels of abstraction,' a favorite term of many software designers, is an empty phrase. These are just a few of the elements that made me impatient to see this book published."

—*David Weiss, Director of Software Technology Research, Avaya Laboratories*

"The authors have written a solid book that discusses many of the most important issues facing software designers. They point out many decisions that can be considered, discussed, and made before coding begins to provide guidance for the programmers. These issues are far more important than most of the decisions that programmers focus on. Properly made and documented, the decisions discussed in this book will guide programmers throughout the remainder of the software development process."

—*David Parnas, Director of the Software Engineering Programme, McMaster University*

Documenting Software Architectures

Second Edition

Documenting Software Architectures
Views and Beyond

Second Edition

Paul Clements
Felix Bachmann
Len Bass
David Garlan
James Ivers
Reed Little
Paulo Merson
Robert Nord
Judith Stafford

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Software Engineering Institute | Carnegie Mellon

Library of Congress Cataloging-in-Publication Data

Documenting software architectures : views and beyond / Paul Clements
... [et al.]. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-55268-6 (hardcover : alk. paper)
1. Computer architecture. 2. Software documentation. I. Clements,
Paul, 1955– II. Title.
QA76.9.A73D63 2010
005.1'5—dc22
2010024318

ISBN-13: 978-0-321-55268-6
ISBN-10: 0-321-55268-7

Text printed in the United States on Recycled paper at Courier in Westford, Massachusetts.
First printing, October 2010

These pictures are meant to entertain you. There is
no significant meaning to the arrows between the boxes.

—A speaker at a recent software architecture conference, coming to a complex but ultimately inadequate boxes-and-lines-everywhere viewgraph of her system's architecture and deciding that trying to explain it in front of a crowd would not be a good idea

I'd like to start with a diagram. It's a bunch of shapes
connected by lines. Now I will say some impressive words:
synchronized digital integrated dynamic e-commerce space.
Any questions?

—Dilbert, making a viewgraph presentation

At the end of the day, I want my artifacts to be enduring.
My goal is to create a prescriptive, semi-formal architectural
description that can be used as a basis for setting
department priorities, parallelizing development, [managing]
legacy migration, etc.

—A software architect for a major financial services firm

This page intentionally left blank

Contents

About the Cover

The cover shows a bird's wing, a motif chosen because it has much in common with software architecture. Rather than appeal to the overused analogy of house architectures, we find physiological systems to be a richer metaphor for software and system architectures. Among such systems, a bird's wing is one of the most compelling examples.

How would you "document" a bird's wing for someone who did not know what it was? A bird's wing, like a software system, can be shown by emphasizing any of a number of structures—nerves, feathers, bones, blood vessels, muscles; each structure must be compatible with the others and must work toward fulfilling a common purpose. Feathers are elements that, at a glance, appear to be replicated countless times across the wing; on closer inspection, however, the feathers reveal a rich substructure of their own and small but systematic variations. All feathers are almost alike, but no two are identical.

The wing exhibits strong quality attributes: lightness in weight, aerodynamic sophistication, outstanding thermal protection. The wing's reliability, cycling through millions of beats, is unparalleled. Unlike a house, which mostly just sits there, the essence of a wing is in its dynamic behavior. In coarse terms, the wing extends, flaps, and retracts; in finer terms, the bird commands movements almost too subtle to see, controlling pitch, roll, and yaw with exquisite finesse. For millennia, humans have tried to comprehend the wing by examining its parts and from different points of view. But the whole wing is much more than the sum of its elements and structures: It is in the whole that beauty and grace emerge alongside breathtaking performance. Falcon wings deliver so much speed that falcons have evolved thick tears that won't evaporate during a

200-mph dive, and they have developed a special structure just inside their nostrils to keep the slipstream from ramming into their lungs. Insect eaters such as swallows routinely endure 14 times the pull of gravity, and they do it dozens of times a day. The common starling, merely an average flier, can slip through the air at 120 body lengths per second; by comparison, the fastest known aircraft, the SR-71 "Blackbird," can manage only about 32 (Wright 2003).

Structure, substructure, replication with variation, dynamic behavior, critical quality attributes, and emergent properties of the entire system: All these aspects are important to capture when documenting a software architecture. We haven't learned how to document beauty and grace yet, but for that we substitute the documentation of what the designer had in mind. For software, we can do this. For the wing of a bird, we can only admire the result.

Foreword to the Second Edition

A colleague of mine, in the market for a home, fell in love with an older property that had been designed by a student of Frank Lloyd Wright himself. Curious about its history, its structure, its evolution, he contacted the local planning office, which happily and quickly provided him with a copy of the original blueprints.

Why, my friend asked me, can we get the drawings for a house that's several decades old, but we are unable to see the architecture of software written last year?

In this book, the authors offer some pragmatic wisdom that helps attend to my friend's lament.

The theory and the practice of the architecture of software-intensive systems are in a very vibrant phase. The early work of Mary Shaw and David Garlan in particular gave rise to software architecture as an identifiable domain of study, and in the years since, we've seen the emergence of architecture-as-an-artifact as a mainstream concern for the development and evolution of systems. This has manifest itself in notations such as the Unified Modeling Language (which was explicitly influenced by Philippe Kruchten's 4+1 model view of software architecture) as well as a panoply of architectural frameworks, such as The Open Group Architecture Framework and the Department of Defense Architecture Framework. Add to these methods such as IBM's Unified Process and, at another extreme, the Federal Segment Architecture Methodology, and it is clear that architecture-as-an-artifact has found an important role in the reasoning about and governing of software-intensive systems.

There are some things we can say with confidence. Every system has an architecture. All complex systems are hierarchical in nature, but also exhibit other patterns of regularity. There's an intimate dance that occurs between the processes of architecting and of implementation. And, to understand and reason about the architecture of a software-intensive system, one has to consider multiple views from the perspectives of specific concerns from multiple classes of stakeholders.

The most commonly used notation and tool for describing a system's architecture is a boxes-and-lines sketch created on a whiteboard. Such documentation is both expeditious and useful, but it is neither enduring nor rigorous nor complete. In this book the authors offer the definitive reference on the documenting of the architecture of software-intensive systems, in ways that are enduring and rigorous and complete. And useful, by the way!

I remember reading the first edition of this book, and e-mailing my compliments to the authors for producing such a comprehensive reference. Well, they've outdone themselves. This new edition is brighter, shinier, more complete, more pragmatic, more focused than the previous one, and I wouldn't have thought it possible to improve on the original. As the field of software architecture has grown over these past decades, there is much more to be said, much more that we know, and much more that we can reflect upon of what's worked and what hasn't—and the authors here do all that, and more.

So, my hope for you, dear reader, is this: May the software you write today have an architecture that your children's children may discern and celebrate.

—Grady Booch
IBM Fellow

Foreword to the First Edition

Ten years ago, I was brought in to lead the architecture team of a new and rather ambitious command-and-control system. After some rocky beginnings, the architectural design work started to proceed full speed, and the architects were finally forging ahead, inventing and resolving and designing and trying, almost in a euphoric state. We had many brainstorming sessions, filling whiteboards with design fragments and notebooks with scribblings; various prototypes validated or invalidated our reasoning. As the development team grew in size, the architects had to explain the principles of the nascent architecture to a wider and wider audience, consisting of not only new developers but also many parties external to the development group. Some were intrigued by this new concept of a software architecture. Some wanted to know how this architecture would impact them: for planning, for organizing the teams and the contractors, for delivery of the system, for acquisition of some of the system parts. Some parties wanted to influence the design of this architecture. Further removed from development, customers and prospects wanted a peek, too. So the architects had to spend hours and days describing the architecture in various forms and levels and tones to varied audiences, so that each party could better understand it.

Becoming this center of communication slowly stretched our capacity. On the one hand, we were busy designing the architecture and validating it; on the other hand, and at the same time, we were communicating to a large audience what it was and why it was that way and why we did not choose some other solution. A few months into the project, overwhelmed, we even began having a difficult time agreeing among ourselves about what it was we had actually decided.

This led me to the conclusion that "if it is not written down, it does not exist." This became sort of a leitmotiv in the architecture team for the following two years. As the ancient Chinese poet Lao-Tsu says in the *Tao Te Ching:*

> Let your workings remain a mystery.
> Just show people the results.
> (Tablet #36)

The architecture *could be* whatever we had talked about, argued, imagined, or even drafted on a board, and so on. But the architecture of this system *was* only what was described in one major document: the *Software Architecture Document (SAD).* Architectural elements and architectural decisions not captured in this document simply did not exist. This one rule—*"If it is not in the SAD, it does not exist."*—became our incentive to evolve and to keep the document up-to-date, almost to the week; there was also an incentive to not include anything and everything and untried ideas, as this was the project's definite arbiter.

The SAD rapidly became a central element in the life of the project. It became our best display window for showing off our stuff, our comfort when we were down, and our shield when attacked.

The key problem we faced at the time was: What do we document for a software architecture? How do we document it? What outline do we use? What notation? How much or how little? There were few exemplars of architectural description for systems as ambitious as ours. Driven by necessity, we improvised. We made some mistakes and corrected some. We discovered rapidly that architecture is not flat but rather a multidimensional reality, with several intertwined facets, and some facets—or views—of interest to only a few parties. We found out that many readers would not even open a document that weighed more than a pound, and we would have a difficult time updating it anyhow. We realized that without capturing the reasons for our choices, we were doomed to reconstruct them again and again, every time a new stakeholder with a sharp mind came around. We picked a visual notation, not too vague and fuzzy but not too esoteric and convoluted, either, in order to not discourage most parties.

Today, software architects have a great starting point for deciding how to document their software architectures. You have it in your hands. The authors went through many experiences similar to mine and extracted the important lessons learned. They read many software architecture documents.

They reviewed the academic literature, studied all the published books, checked the standards, and synthesized all this wisdom in this handbook: the essential things you need to know to define your own software architecture document. You will find guidance for the scope of software architecture; its organization; the techniques, tools, and notation to use or not to use; and comparisons, advice, and rules of thumb. In here, you'll find the templates to get you started and the continuing guidance for when you get lost or despairing on the way.

This book is of immense value. The description and communication of software architecture is quite crucial to its many stakeholders, and this handbook should save you months of trials and errors, lots of undeserved hassle, and many costly mistakes that could potentially jeopardize the whole endeavor. It will become an important reference on the shelf of the software architect.

—Philippe Kruchten
Director of Process Development
Rational Software Canada, Vancouver

This page intentionally left blank

Preface

The purpose of this book is to answer the following question:

How do you document an architecture so that others can successfully use it, maintain it, and build a system from it?

The audience for this book includes all the people involved in the production and consumption of architecture documentation. The goal of this book is to help you decide what information about an architecture is important to capture and to provide guidelines, notations, and examples for capturing it. We intend this book to be a practitioner-oriented guide to the various kinds of information that constitute an architecture. We give practical guidance for choosing what information should be documented and show—with examples in various notations, including but not limited to the Unified Modeling Language (UML)—how to describe that information in writing so that others can use it to carry out their architecture-based work: implementation, analysis, and recovery. We also show how to create a comprehensive software architecture document that others can use.

Although piles of books exist about how to use a particular notation (UML comes to mind), we believe what an architect really needs is guidance in which architecture and its stakeholders are the first-class citizens, and language is relegated more appropriately to a supporting role. That's what we've tried to provide with this book.

Languages and Tools for Architecture

Commercial languages and tool suites are available for capturing design information, especially in the realm of object-oriented systems. Some of these tools are bound up with associated design methods, notations, and commercial products. Some tools are aimed at points in the design space other than architecture. If you have decided to adopt one of these tools and/ or notations, will this book relate to you?

Very few things become obsolete faster than references to specific tools, so we've avoided those. Instead, we have concentrated on the information you should capture about an architecture. We believe that is the approach you should take, too: Concentrate on the information you need to capture, and then figure out how to capture it using an available tool. Almost all tools provide ways to add free-form annotations to the building blocks they provide; if all else fails, these annotations will let you capture and record information in ways you see fit. Remember that not all the people for whom architecture documentation is prepared will be able to use the tool environment you've chosen or understand the commercial notation you've adopted.

Having said that, however, we acknowledge that a few standard languages and notations have come to dominate, chief among them UML. And so this book provides a plethora of examples showing UML 2 representing the architecture views we cover, as well as other concepts such as refinement and behavior. If you have chosen UML as your modeling language, you'll feel at home.

Appendix A contains a summary of UML's visual notation and its applicability to document the concepts in this book. Appendices B and C summarize the Systems Modeling Language (SysML) and the Architecture Analysis and Design Language (AADL), respectively. Our purpose is not to teach these languages, but to offer a quick refresher for those familiar with them and a flavor-providing introduction for everyone else.

What's New in the Second Edition

- A number of new architecture styles have entered the mainstream, and this edition talks about documenting those. These include service-oriented architectures, multi-tier architectures, and architectures for aspect-oriented systems. We also treat the architecture-level documentation of a software system's data model, as well as its installation and production environment, as first-class styles.

- This edition is much more Agile-friendly, orienting its advice to be consistent with the Agile Manifesto's entreaty to value working software over comprehensive documentation.
- We treat the systematic documentation of rationale with much greater depth, reflecting best industrial practices. We've added a new chapter about reviewing an architecture document to make sure it's serving its stakeholders as intended.
- The suggested templates for architecture documentation have several improvements, reflecting years of use and feedback. They are also more flexible, and we lay out different options for arranging your documentation.
- We have replaced the comprehensive example of a documented software architecture with a new one. The architecture is for a Web-based service-oriented system, more in today's industrial mainstream. To make the book smaller and allow us to maintain the example over time, we put the example online. And many of our in-line examples have been replaced or updated.
- Since the first edition was published, the Unified Modeling Language has graduated to version 2.0 and beyond. That opened up new possibilities for more straightforwardly documenting various architecture constructs, especially components and connectors. Where necessary, our figures are updated to reflect the new constructs.
- This edition has concise appendices summarizing three important languages and notations useful for documenting architectures: UML, AADL, and SysML. Each appendix constitutes a mini-reference guide on the language.
- Finally, this edition reflects the experience we've gained with Views and Beyond in the intervening years since the first edition was published. This experience has come from creating documented architectures for very challenging systems, and helping other people do so. It also comes from using architecture documentation in practice, such as when we evaluate other organizations' software architectures. Finally, it has come from interacting with more than a thousand participants in our two-day industrial course based on the book. These interactions with practicing software architects have let us make our advice more prescriptive and crisp and reflect the problems and situations that architects face daily.

Complete Example of a Software Architecture Document Online

You can see a fully worked-out example of a software architecture document using the approaches and templates described in this book at **wiki.sei.cmu.edu/sad**.

—P.C.
Austin, Texas

—F.B., L.B., D.G., J.I., R.L., R.N.
Pittsburgh, Pennsylvania

—P.M.
Brasilia, Brazil

—J.S.
Boston, Massachusetts

Acknowledgments

We would like to thank a number of people for making this book a reality.

There wouldn't be a second edition without a first edition, and all of the people whose help and support we acknowledged there deserve a thank you here as well.

At the Software Engineering Institute, Linda Northrop provided unstinting support (for the second time) for this effort. Mark Klein, head of the SEI's Architecture-Centric Engineering initiative, made this book part of that initiative's transition efforts. Many thanks to Barbara White, who was invaluable in helping to deal with thorny word-processing issues. Thanks to Kurt Hess for creating the cover, producing many of the figures in the book, and keeping all of the figures and graphics organized. Rob Wojcik reviewed a complete early draft and made many helpful suggestions, especially in the "patterns versus styles" sidebar in the prologue. Thanks to John Morley for his help in editing the book.

At Addison Wesley, Peter Gordon did his usual wonderful job of nudging this book along. Thanks to Kim Boedigheimer for always having the right answers to our many questions. Our thanks also go the production professionals who contributed their superb talents to produce the result you see, especially Anna Popick, Christopher Keane, and John Fuller.

Special thanks to Grady Booch for writing a splendid foreword for this edition. Grady also provided helpful and directed comments about the first edition that guided our thinking for the second edition. Our continued thanks go to Philippe Kruchten for writing the first edition's foreword, which we've proudly retained.

Thanks to the distinguished experts in the architecture community who participated in our online discussion about the difference between an architecture pattern and an architecture style. These include Alexander Wolf, Frank Buschmann, Celso Gonzalez, David Emery, Eoin Woods, Grady Booch, Hans van Vliet, Kurt Wallnau, Maarten Boasson, Mary Shaw, Morven Gentleman, Neno Medvidovic, Rich Hilliard, Rick Kazman, and Ruth Malan.

Thanks go to T.V. Prabhakar, who had the idea for the concept map that adorns the inside cover. The concept map was rendered using Cmap, available from cmap.ihmc.us.

Thanks to Larry Jones for summarizing a tutorial by Philippe Kruchten on capturing architecture decisions, and thanks to Philippe for giving us the notes from that tutorial. Thanks to Peter Eirich for bringing the IEEE draft standard 1175-4 on specifying behavior to our attention. Thanks to Phil Taylor for suggesting the title "Beyond the Basics" for Chapter 6 to replace an unsatisfactory chapter name from the first edition. Thanks to Joe Batman for his discussion of event-driven versus enterprise systems. Several quotes in the book pertain to advertising, and we co-opted those for architecture. We found them on zagstudios.com; thanks go to Kimberley Freeman, the site's proprietor.

Heartfelt thanks go to Jan Bosch, Stefan Ferber, Robert Glass, and Eoin Woods for thorough and very helpful reviews of an early draft.

Special thanks to the thousand-plus attendees of the SEI's "Documenting Software Architectures" course who over the years have provided insightful comments that helped us to improve the course material and gave us motivation to produce a revised edition of the book.

The rules for sound documentation cited in the prologue are based on those in "A Rational Design Process: How and Why to Fake It" (Parnas and Clements 1986). Figures 2.36 and 2.37 are adapted from a 2004 presentation by Ramnivas Laddad.

Finally, we were privileged to have a select group of architecture experts contribute their experience and energy to various sections of this book. We acknowledge their contributions in the chapters or sections in which they contributed material by giving them a "With" byline. Our distinguished list of contributing authors can be found in "About the Contributors," on page 513.

Reader's Guide

Audience

There are three primary audiences for this book.

1. Software architects who are charged with producing architecture documentation for software projects. For these people we tried to answer the question "What information do I need to capture about my architecture, and what notations and techniques are available for communicating it clearly and usefully in a timely fashion?"
2. Stakeholders of an architecture who must digest and use the documentation they receive from the architect or architecture team. A software architect can provide this book as a companion to his or her documentation, pointing consumers to specific sections that explain documentation-organizing principles, notations, concepts, or conventions.
3. People who wish to learn introductory concepts about software architecture. By establishing the purposes and uses of software architecture (and hence, its documentation), and by establishing a basic set of concepts important in the creation and communication of architecture, this book serves as an introduction to the subject.

We assume basic familiarity with the concepts of software engineering. In many cases, we will sharpen and solidify basic concepts that you already know, such as *architecture views*, *architecture styles*, and *interfaces*.

Stylistic Conventions

The book's core message is contained in the main flow of the text. But we also provide extra information in the margins, including

- Definitions: Where we introduce a term such as **<u>view</u>**, we make it bold and underlined; a margin note adjacent to that line gives the definition. These terms are also listed in the glossary at the end.

A **view** is a representation of a set of system elements and relationships among them.

- Nuggets of practical advice.

Every graphical presentation should include a key that explains the notation used.

- Pointers to sources of additional information, either within this book or outside.

The prologue contains an introduction to the basic architecture concepts used in this book.

- Illuminating quotes that we hope will add to the fullness of the message.

A good notation should embody characteristics familiar to any user of mathematical notation: Ease of expressing constructs arising in problems, suggestivity, ability to subordinate detail, economy, amenability to formal proofs.

—Ken Iverson (1987, p. 341)

Advice that won't fit into a margin note will be called out in the body of the text. Longer diversions occur as sidebars, which are visually distinguished passages that appear at the end of a section. "Coming to Terms" sidebars tackle issues of terminology, while "Perspectives" sidebars are observations or background information written and signed by one or more of the authors.

At the end of each chapter, you can find

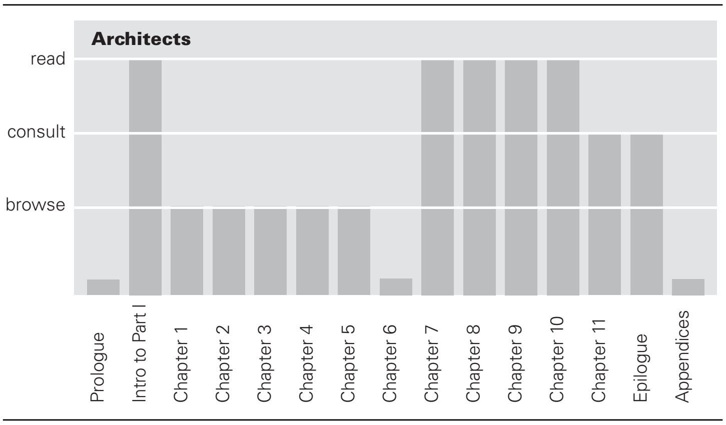
- A summary checklist that highlights the main points and prescriptive guidance of the chapter
- A set of discussion questions that can serve as the basis for classroom or brown-bag-lunch-group conversation
- "For Further Reading," a section that offers references for more in-depth treatment of related topics

A glossary appears at the end of the book.

How to Read and Use This Book

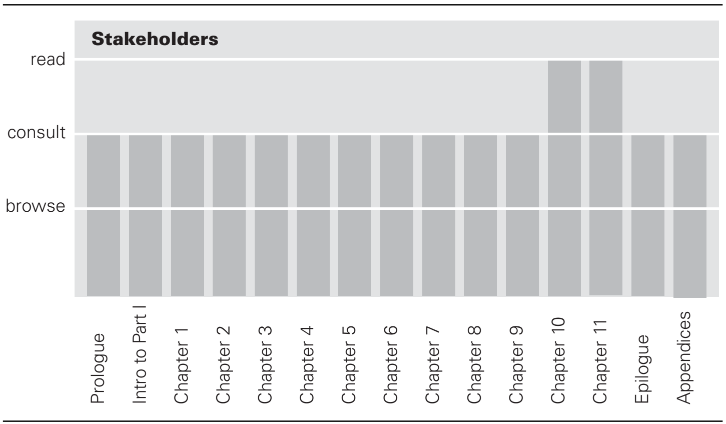
All architects should

- Read the introduction to Part I, to gain an understanding of styles and views, and to get a glimpse of the collection of styles discussed in this book.
- Browse Chapters 1–5 to gain a deeper understanding of the views that might be used in your documentation. Later, once you've chosen a set of views to document, you can read about them in more depth as needed.
- Read Chapter 10, to learn the organizational scheme for a documentation package.
- Read Chapter 9, to learn how to choose the important views for a particular system. This will let you plan your documentation package, matching your stakeholders and the uses your documentation will support with the kind of information you need to provide.
- Browse the sections in Chapter 6 to learn about documenting variability, context diagrams, and other helpful concepts. Come back and concentrate on these as needed.
- Read Chapters 7 and 8 to learn about documenting software interfaces and documenting behavior of a system.
- Consult Chapter 11 to see how your architecture document should be reviewed, so that you can better position it for a successful review by giving reviewers the information they need.
- If you are interested in making your documentation compliant with other prescriptive methods, such as IBM Rational's 4+1 approach or ISO/IEC 42010, consult the epilogue.



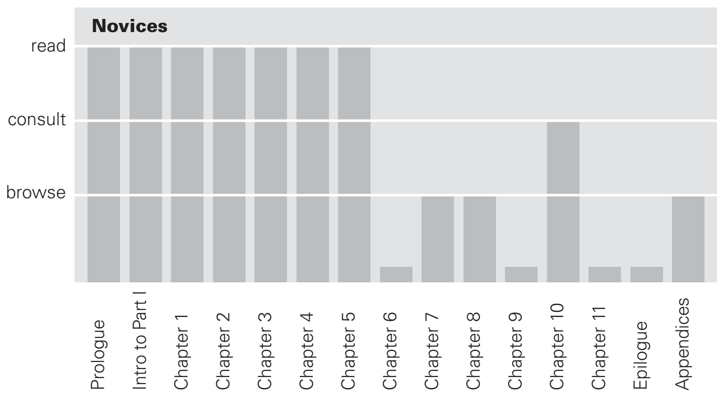
An architecture stakeholder using an architecture document written with the precepts of this book may wish to consult this book to gain a deeper understanding. You should

- Read Chapter 10 to gain a better understanding of the layout of the document, and how the layout achieves coverage of the architectural information being conveyed.
- Consult other chapters as necessary to provide more insight into specific parts of the architecture document. For example, you may wish to read the introduction to Part I to learn about module, component-and-connector, and allocation styles, and then consult the chapter on a specific style.
- Read Chapter 11 if your job is to conduct or participate in a review of the architecture document.



Readers who wish to learn introductory concepts about software architecture should

- Read the prologue to learn what software architecture is, why it is important, and the critical role of documentation in a development project.
- Read the introduction to Part I, to gain an understanding of styles and views, and to get a glimpse of the collection of styles discussed in this book.
- Read Chapters 1–5 to become familiar with some architecture styles that are widely used in modern software systems.
- Browse Chapters 7 and 8 to learn about the important architecture concepts of interfaces and behavior.
- Consult Chapter 10 to see a format for an architecture document.
- Browse the appendices to help you understand the examples in the book if you're not familiar with the notations.



This page intentionally left blank

Prologue: Software Architectures and Documentation

The prologue establishes a small but fundamental set of concepts that will be used throughout the book. We begin with short overviews of software architecture (Section P.1) and architecture documentation (Section P.2), and then we go on to discuss the following topics:

- Section P.3: Architecture views
- Section P.4: Architecture styles (and their relation to architecture patterns) and the classification of styles into three categories: module styles, component-and-connector styles, and allocation styles
- Section P.5: Rules for sound documentation

P.1 A Short Overview of Software Architecture

P.1.1 Overview

Software architecture has emerged as an important subdiscipline of software engineering. Architecture is roughly the prudent partitioning of a whole into parts, with specific relations among the parts. This partitioning is what allows groups of people—often separated by organizational, geographical, and even time-zone boundaries—to work cooperatively and productively together to solve a much larger problem than any of them could solve individually. Each group writes software that interacts with the other groups' software through carefully crafted interfaces that reveal the minimal and most stable information necessary for interaction. From that interaction emerges the functionality and quality attributes—security, modifiability, performance, and so forth—that the system's stakeholders demand. The larger and more complex the sys-

The **software architecture** of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

tem, the more critical is this partitioning—and hence, architecture. And as we will see, the more demanding those quality attributes are, the more critical the architecture is.

A single system is almost inevitably partitioned simultaneously in a number of different ways. Each partitioning results in the creation of an architectural structure: different sets of parts and different relations among the parts. Each is the result of careful design, carried out to satisfy the driving quality attribute requirements and the most important business goals behind the system.

Many projects make the mistake of trying to impose a single partition in multiple component domains, such as equating threads with objects, which are equated with modules, which in turn are equated with files. Such an approach never succeeds fully, and adjustments eventually must be made, but the damage of the initial intent is often hard to repair. This invariably leads to problems in development and occasionally in final products.

—Jazayeri, Ran, and van der Linden (2000, pp. 16–17)

Architecture is what makes the sets of parts work together as a coherent and successful whole. Architecture documentation help architects make the right decisions; it tells developers how to carry them out; and it records those decisions to give a system's future caretakers insight into the architect's solution.

P.1.2 Architecture and Quality Attributes

For nearly all systems, quality attributes such as performance, reliability, security, and modifiability are every bit as important as making sure that the software computes the correct answer. A software system's ability to produce correct results isn't helpful if it takes too long doing it, or the system doesn't stay up long enough to deliver it, or the system reveals the results to your competition or your enemy. Architecture is where these concerns are addressed. For example:

- If you require high performance, you need to
  - Exploit potential parallelism by decomposing the work into cooperating or synchronizing processes.
  - Manage the interprocess and network communication volume and data access frequencies.
  - Be able to estimate expected latencies and throughputs.
  - Identify potential performance bottlenecks.
- If your system needs high accuracy, you must pay attention to how the data elements are defined and used and how their values flow throughout the system.
- If security is important, you need to
  - Legislate usage relationships and communication restrictions among the parts.
  - Identify parts of the system where an unauthorized intrusion will do the most damage.
  - Possibly introduce special elements that have earned a high degree of trust.

- If you need to support modifiability and portability, you must carefully separate concerns among the parts of the system, so that when a change affects one element, that change does not ripple across the system.
- If you want to deploy the system incrementally, by releasing successively larger subsets, you have to keep the dependency relationships among the pieces untangled, to avoid the "nothing works until everything works" syndrome.

The solutions to these concerns are purely architectural in nature. It is up to architects to find those solutions and communicate them effectively to those who will carry them out. Architecture documentation has three obligations related to quality attributes. First, it should indicate which quality attribute requirements drove the design. Second, it should capture the solutions chosen to satisfy the quality attribute requirements. Finally, it should capture a convincing argument why the solutions provide the necessary quality attributes. The goal is to capture enough information so that the architecture can be analyzed to see if, in fact, the system(s) derived from it will possess the necessary quality attributes.

Chapter 10 will show where in the documentation to record the driving quality attribute requirements, the solutions chosen, and the rationale for those solutions.

COMING TO TERMS

What Is Software Architecture?

If we are to agree on what it means to document a software architecture, we should establish a common basis for what it is we're documenting. No universal definition of software architecture exists. The Software Engineering Institute's Web site collects definitions from the literature and from practitioners around the world; so far, more than 150 definitions have been collected.

It seems that new fields try to nail down standard definitions or their key terms as soon as they can. As the field matures, basic concepts become more important than ironclad definitions, and this urge seems to fade. When object-oriented development was in its infancy, you could bring any OO meeting to a screeching halt by putting on your best innocent face and asking, "What exactly is an object?" This largely ended when people realized that the scatter plot of definitions had an apparent (if unarticulated) centroid, from which very useful progress could be made. Sometimes "close enough" is, well, close enough.

Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.

—Eoin Woods (SEI 2010)

You can read the SEI collection of definitions, or contribute your own, at www.sei.cmu.edu/architecture.

This seems to be the case with software architecture. Looking at the major attempts to nail down its definition gives us a good glimpse at our own centroid. With that in mind, here are a few influential definitions:

> By analogy to building architecture, we propose the following model of software architecture: Software Architecture = {Elements, Form, Rationale}. That is, a software architecture is a set of architectural (or, if you will, design) elements that have a particular form. We distinguish three different classes of architectural elements: processing elements; data elements; and connecting elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together. (Perry and Wolf 1992, p. 44)

> . . . beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. (Garlan and Shaw 1993, p. 1)

> The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. (Garlan and Perry 1995, p. 269)

> An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architecture style that guides this organization—these elements and their interfaces, their collaborations, and their composition. (Booch, Rumbaugh, and Jacobson 1999, p. 31)

> The fundamental organization of a system embodied in its components, their relations to each other, and to the environment, and the principles guiding its design and evolution. (IEEE 1471 2000, p. 9)

> The software architecture of a program or computing system is the structure or structures of the system, which

comprise software elements, the externally visible properties of those elements, and the relations among them. By "externally visible properties," we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. (Bass, Clements, and Kazman 2003, p. 27)

The set of principal design decisions governing a system. (Taylor, Medvidovic, and Dashofy 2009, p. xv)

A few other "mainstream" definitions have emerged since then, but they are largely restatements and recombinations of the ones we just listed. The centroid seems to have stabilized.

That centroid takes a largely structural perspective on software architecture: Software architecture is composed of elements, connections or relations among them, and, usually, some other aspect or aspects, such as (take your pick) configuration; constraints or semantics; analyses or properties; or rationale, requirements, or stakeholders' needs.

These perspectives do not preclude one another, nor do they represent a fundamental conflict about what software architecture is. Instead, they represent a spectrum in the software architecture community about the emphasis that should be placed on architecture: its constituent parts, the whole entity, the way it behaves once built, or the building of it. Taken together, they form a consensus view of software architecture.

In this book we use a definition similar to the one from Bass, Clements, and Kazman (2003). We chose it because it helps us know what to document about an architecture. The definition emphasizes the plurality of structures present in every software system. These structures, carefully chosen and designed by the architect, are the key to achieving and reasoning about the system's design goals. And those structures are the key to understanding the architecture. Therefore, they are the focus of our approach to documenting a software architecture. Structures consist of elements, relations among the elements, and the important properties of both. So documenting a structure entails documenting those things.

PERSPECTIVES

What's the Difference Between Architecture and Design?

The question of how architecture is different from design has nipped at the heels of the software development community for years. It is a question I often hear when teaching an introductory course on architecture. It matters here because the question deals with what we should put in an architecture document and what we should put somewhere else.

The first thing we can say is that clearly architecture *is* design, but not all design is architecture. That is, many design decisions are left unbound by the architecture and are happily left to the discretion and good judgment of downstream designers and even implementers. The architecture establishes constraints on downstream activities, and those activities must produce artifacts—finer-grained designs and code—that comply with the architecture.

It's tempting to stop there, but if you're paying attention you've seen that we've just translated the question: Architecture consists of architectural design decisions, and all others are nonarchitectural. So what decisions are nonarchitectural? That is, what design decisions does the architect leave to the discretion of others?

To answer this question, we return to the primary purpose of architecture, which is to assure the satisfaction of the system's quality and behavioral requirements and business goals. The architect does this by making design decisions that manifest themselves in the system's architectural structures.

Thus, architectural decisions are ones that permit a system to meet its quality attribute and behavioral requirements. All other decisions are nonarchitectural.

Clearly any design decisions resulting in element properties that are *not* visible—that is, make no difference outside the element—are nonarchitectural. A typical example is the selection of a data structure, along with the algorithms to manage and access that data structure.

You may have been hoping for a more concrete answer, such as "the first three levels of module decomposition are architectural, but any subsequent decomposition is not." Or, "the classes, packages and their relations in a UML class diagram are architectural, but sequence diagrams are not." Or "defining the services of an SOA system is architectural, but designing the internal structure of each service provider component is not."

But those don't work because they draw arbitrary and artificial boundaries. Attempts like that to be practical end up being impractical because true architecture bleeds across those boundaries.

Here are some more sometimes-heard artificial definitions.

First, "architecture is the small set of big design decisions." Some people define "small set" by insisting that an architecture document should be no more than 50 pages. Or 80. Or 30. Their feeling, apparently, is that architecture is the set of design decisions that you can squeeze into a given page quota, and everything beyond that is not. This is, of course, utter nonsense.

Another oft-heard nonanswer is "architecture is what you get before you start adding detail to the design." Terminology often directs our thinking, rather than serves it. A pernicious example that puts us in the wrong mind set is "detailed design." Detailed design is what many people say follows architecture. The term is everywhere, and needs to be stamped out. It implies that the difference between architectural and nonarchitectural design is something called "detail." Architecture is apparently not allowed to be detailed, because if it is, well, you're doing detailed design then, aren't you? Never mind that we have no idea how to measure "detail" nor to set a threshold for when there is too much of it to be architectural. If your design starts to look "detailed" then you aren't doing architecture and you'll be reported to the Detailed Design Police for overstepping your authority. More utter nonsense.

Don't use the term "detailed design"! Use "nonarchitectural design" instead.

It's true that some architectural design decisions may lack much specificity; that is, they preserve freedom of choice for downstream designers. Some architectural design decisions may not be "decisions" at all, but broad constraints. Plug-ins that populate your Web browser are an example. No architecture nails down the complete set, but the architecture does constrain new ones to meet certain standards and interfaces. Or the architect might describe an element by saying, "The element delivers its computational result through this published interface, is thread-safe, puts no more than three messages on the network per invocation, and returns its answer in less than 20 ms." The team implementing that element is free to make whatever design decisions they wish as long as they satisfy the architect's prescription for it.

On the other hand, some architectural decisions can be quite "detailed," such as the adoption of specific protocols, an XML schema, or communication or technology standards. Such decisions are usually made for purposes of interoperability or various flavors of modifiability (such as scalability or extensibility).

Even interfaces of elements, which some decry as "obviously" outside the realm of architecture, can be supremely architectural. For instance, in a service-oriented architecture (SOA), components interact through published interfaces. Important design decisions made when defining these interfaces include the granularity of the operations, the data format, and the type of interaction (synchronous or asynchronous) for each operation. Or consider an element that processes data from a real-time sensor. Making this element's interface process a stream as opposed to individual data elements will make an enormous difference in the ability of the element (and hence the system) to meet real-time performance requirements. This decision cannot be left up to the element's development team; everything depends on it.

A **hierarchical element** is any kind of element that can consist of like-kind elements. A module is a hierarchical element because modules consist of submodules, which are themselves modules. A task or a process is not a hierarchical element.

A legitimate question about detail does arise when considering modules and other **hierarchical elements**: When do you stop? When have you designed enough levels in the hierarchy? Are submodules enough, or does the architect need to design sub-sub-sub-submodules? Here's a good test of our claim for when architecture stops. Module decomposition is about achieving independent development and modifiability. Both are achieved by carefully assigning coherent responsibilities to each module. When the modules you've designed are fine-grained enough to satisfy the system's modifiability and independent development requirements, you've discharged your obligation as an architect.

Finally, what is architectural is sensitive to context. Suppose the architect identifies an element but is content to sketch the element's interface and behavior in broad terms. If the element being prescribed is very large and complex, the team developing it may choose to give it an internal substructure of its own, which for all the world looks like an architecture. And within the context of that element, it is. But in the context of the overall system, the substructure is not architectural but merely an internal design decision made by the development team for that element.

To summarize, architecture is design, but not all design is architectural. The architect draws the boundary between architectural and nonarchitectural design by making those decisions that need to be bound in order for the system to meet its development, behavioral, and quality goals. All other decisions can be left to downstream designers and implementers. Decisions are architectural or not, according to context. If structure is important to achieve your system's goals, that structure is architectural. But designers of elements, or subsystems, that you assign may have to introduce structure of their own to meet their goals, in which case such structures are architectural: *to them* but not to you.

And (repeat after me) we all promise to stop using the phrase "detailed design." Try "nonarchitectural design" instead.

—P.C.

P.2 A Short Overview of Architecture Documentation

P.2.1 Why Document Software Architecture?

Doing business without advertising [or designing an architecture without documenting it] is like winking at a girl in the dark. You know what you're doing, but nobody else does.

—Steuart Henderson Britt

Even the best architecture, most perfectly suited for the job, will be essentially useless if the people who need to use it do not know what it is, cannot understand it well enough to apply it, or (worst of all) misunderstand it and apply it incorrectly. All of the effort, analysis, hard work, and insightful design on the part of the architecture team will have been wasted. They might as well have gone on vacation for all the good their architecture will do.

Creating an architecture isn't enough. It has to be communicated in a way to let its stakeholders use it properly to do their jobs. If you go to the trouble of creating a strong architecture, you *must* go to the trouble of describing it in enough detail, without ambiguity, and organized so that others can quickly find needed information.

Documentation speaks for the architect. It speaks for the architect today, when the architect should be doing other things besides answering a hundred questions about the architecture.

And it speaks for the architect tomorrow, when he or she has left the project and now someone else is in charge of its evolution and maintenance.

Documentation is often treated as an afterthought, something people do because they have to. Maybe a contract requires it. Maybe a customer demands it. Maybe a company's standard process calls for it. In fact, these may be legitimate reasons. But none of them are compelling enough to produce high-quality documentation. Why should the architect spend valuable time and energy just so a manager can check off a deliverable?

The best architects produce the best documentation not because it's "required," but because they see that it is essential to the matter at hand: producing a high-quality product, predictably and with as little rework as possible. They see their immediate stakeholders as the people most intimately involved in this undertaking: developers, deployers, testers, and analysts.

But the best architects also see documentation as delivering value to themselves. Documentation serves as the receptacle to hold the results of design decisions as they are made. A well-thought-out documentation scheme can make the process of design go much more smoothly and systematically. Documentation helps the architect while the architecting is in progress, whether in a six-month design phase or a six-day Agile sprint.

COMING TO TERMS

Specification, Representation, Description, Documentation

What shall we call the activity of writing down a software architecture for the benefit of others or for our own benefit at a later time? Leading contenders are *documentation*, *representation*, *description*, and *specification*. None of these terms has a standardized meaning in our field: the difference between them is unclear. For the most part, we use *documentation* throughout this book, and we want to explain why.

Specification tends to connote an architecture rendered in a formal language. Now, we are all for formal specs. But formal specs are not always practical, nor are they always necessary. Sometimes, they aren't even useful: How, for example, do you capture in a formal language the rationale behind your architectural decisions, and why would you try?

Representation connotes a model, an abstraction, a rendition of a thing that is separate or different from the thing itself. Is architecture something more than what someone writes down about it? Arguably yes, but it's certainly pretty intangible in any case. We felt that raising the issue of a model versus the thing being modeled would only elicit needlessly diverting questions best left to those whose hobby, or calling, is philosophy: Does an abstraction of a tree falling in a model of a forest make a representation of a sound? This does not seem like the start of a productive conversation.

Description has been staked out by the architecture description language (ADL) community, and more recently by the standards community coming up with mandates for how to write down an architecture. It's curious that the people you'd think would be the most formal snagged the least rigorous sounding term of the bunch. (The next time you board a jet, sit in front of a computer-controlled X-ray machine, or watch the launch of a billion-dollar space vehicle your tax dollars paid for, ask yourself whether you hope the control software has been specified to the implementers, or merely described.) We eschewed *description*, then, because it all at once sounds too formal—we didn't want people to think that writing down an architecture requires an architecture description language—and too informal. Descriptions can be notoriously vague, such as when your friends *describe* the blind date they set you up with. Sometimes we need a little more specificity in our lives, and certainly we need it in our architectures.

ADLs are discussed in Section 3.4.2 and in the For Further Reading section of Chapter 8. For an overview of ADLs, see the work by Stafford and Wolf (2001).

That leaves *documentation*. *Documentation* connotes the creation of an artifact: namely, a document, which may of course consist of electronic files, Web pages, a snapshot of a whiteboard, or paper. Thus, documenting a software architecture becomes a concrete task: producing a software architecture document. Viewing the activity as creating a tangible product has advantages. We can describe good architecture documents and bad ones. We can use completeness criteria to judge how much work is left in producing this artifact and determining when the task is done. Planning or tracking a project's progress around the creation of artifacts, or documents, is an excellent way to manage. Making the architecture information available to its consumers and keeping it up to date reduces to a solved problem of configuration control. Documentation can be formal or not, as appropriate, and may contain models or not, as appropriate. Documents may describe, or they may specify. Hence, the term is appropriately general.

No matter what you call it, the essence of the activity is writing down—and keeping current—the results of architectural decisions so that the stakeholders of the architecture—people who need to know what it is to do their job—have the information they need in an accessible, nonambiguous form.

Section 6.1.3 ("Spectrum of Design") discusses how architecture documentation captures the very abstract to the very detailed.

P.2.2 Uses and Audiences for Architecture Documentation

In Chapter 9, the documentation's expected uses, along with the documentation obligations each use imparts, become the basis for helping an architect plan the documentation package.

Architecture documentation must serve varied purposes. It should be sufficiently abstract to be quickly understood by new employees. It should be sufficiently concrete to serve as a blueprint for construction. It should have enough information to serve as a basis for analysis.

Architecture documentation is both prescriptive and descriptive. For some audiences, it prescribes what *should* be true, placing constraints on decisions yet to be made. For other audiences, it describes what *is* true, recounting decisions already made about a system's design.

The best architecture documentation for, say, performance analysis may well be different from the best architecture documentation we would wish to hand to an implementer. And both of these will be different from what we put in a new hire's "welcome aboard" package or a briefing we put together for an executive. The process of documentation planning and review needs to ensure support for all the relevant needs.

Chapter 9 discusses planning the contents of a documentation package. Chapter 11 discusses reviewing documentation.

We can see that many different kinds of people are going to have a vested interest in an architecture document. They hope and expect that the architecture document will help them do their respective jobs. Understanding their uses of architecture documentation is essential, as those uses determine the important forms.

Fundamentally, architecture documentation has three uses.

1. *Architecture serves as a means of education.* The educational use consists of introducing people to the system. The people may be new members of the team, external analysts, or even a new architect. In many cases, the "new" person is the customer to whom you're showing your solution for the first time, a presentation you hope will result in funding or go-ahead approval.

2. *Architecture serves as a primary vehicle for communication among **stakeholders***. An architecture's precise use as a communication vehicle depends on which stakeholders are doing the communicating. Some examples are described in Table P.1.

   Perhaps one of the most avid consumers of architecture documentation is none other than the architect in the project's future. The future architect may be the same person as the present one, or he or she may be a replacement, but in either case he or she is guaranteed to have an enormous stake in the documentation. New architects are interested in learning how their predecessors tackled the difficult issues of the system and why particular decisions were made. Even if the future architect is the same person, he or she will use the documentation as a repository of thought, a storehouse of design decisions too numerous and hopelessly intertwined ever to be reproducible from memory alone.

   Even in the short term, documenting an architecture helps in the process of designing the architecture. First, the documentation provides dedicated compartments for recording various kinds of design decisions as soon as they are made. Second, the documentation gives you a rough but helpful way to gauge progress and the work remaining: As "TBD"s disappear from the document, completion draws near. Finally, documentation provides a framework for systematic attack on designing the architecture. Key design decisions, usually made early, should be written down so that the shadow they cast on subsequent design decisions is explicit and remembered.

A **stakeholder** of an architecture is someone who has a vested interest in it. (Many of an architecture's stakeholders are listed in Table P.1.)

Chapter 9 is about how stakeholders' needs will help determine the contents of the architecture documentation.

Stakeholders (explicitly or implicitly) drive the whole shape and direction of the architecture, which is developed solely for their benefit and to serve their needs. . . . Without stakeholders, there would be no point in developing the architecture because there would be no need for the system it will turn into, nor would there be anyone to build it, deploy it, run it, or pay for it. . . . Architectures are created solely to meet stakeholder needs.

—Rozanski and Woods (2005, p. 21)

QUOTE

In our organization, a development group writes design documents to communicate with other developers, external test organizations, performance analysts, the technical writers of manuals and product helps, the separate installation package developers, the usability team, and the people who manage translation testing for internationalization. Each of these groups has specific questions in mind that are very different from the ones that other groups ask:

- What test cases will be needed to flush out functional errors?
- Where is this design likely to break down?
- Can the design be made easier to test?

- How will this design affect the response of the system to heavy loads?
- Are there aspects of this design that will affect its performance or ability to scale to many users?
- What information will users or administrators need to use this system, and can I imagine writing it from the information in this design?
- Does this design require users to answer configuration questions that they won't know how to answer?
- Does it create restrictions that users will find onerous?
- How much translatable text will this design require?
- Does the design account for the problems of dealing with double-byte character sets or bi-directional presentation?

—Kathryn Heninger Britton (Hoffman and Weiss 2001, pp. 337–338)

Get the habit of analysis—analysis will in time enable synthesis to become your habit of mind.

—Frank Lloyd Wright

3. *Architecture serves as the basis for system analysis and construction.*
   - Architecture tells implementers what to implement.
   - For those interested in the ability of the design to meet the system's quality objectives, the architecture documentation serves as the fodder for evaluation. The architecture documentation must contain the information necessary to evaluate a variety of attributes, such as security, performance, usability, availability, and modifiability. Analyses of each one of these attributes have their own information needs.
   - For system builders who use automatic code-generation tools, the documentation may incorporate the models used for generation.

Table P.1 Some of the stakeholders of architecture documentation, their roles, and how they might use it

Name	Description	Use for Architecture Documentation
Analyst	Responsible for analyzing the architecture to make sure it meets certain critical quality attribute requirements. Analysts are often specialized; for instance, performance analysts, safety analysts, and security analysts may have well-defined positions in a project.	Analyzing satisfaction of quality attribute requirements of the system based on its architecture.

Table P.1 Some of the stakeholders of architecture documentation, their roles, and how they might use it (*continued*)

Name	Description	Use for Architecture Documentation
Architect	Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system.	Negotiating and making trade-offs among competing requirements and design approaches. A vessel for recording design decisions. Providing evidence that the architecture satisfies its requirements.
Business manager	Responsible for the functioning of the business/organizational entity that owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, and more.	Understanding the ability of the architecture to meet business goals.
Conformance checker	Responsible for assuring conformance to standards and processes to provide confidence in a product's suitability.	Basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions.
Customer	Pays for the system and ensures its delivery. The customer often speaks for or represents the end user, especially in a government acquisition context.	Assuring required functionality and quality will be delivered, gauging progress, estimating cost, and setting expectations for what will be delivered, when, and for how much.
Database administrator	Involved in many aspects of the data stores, including database design, data analysis, data modeling and optimization, installation of database software, and monitoring and administration of database security.	Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.
Deployer	Responsible for accepting the completed system from the development effort and deploying it, making it operational, and fulfilling its allocated business function.	Understanding the architectural elements that are delivered and to be installed at the customer's or end user's site, and their overall responsibility toward system function.
Designer	Responsible for systems and/or software design downstream of the architecture, applying the architecture to meet specific requirements of the parts for which they are responsible.	Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understanding how their part will communicate and interact with other parts of the system.
Evaluator	Responsible for conducting a formal evaluation of the architecture (and its documentation) against some clearly defined criteria.	Evaluating the architecture's ability to deliver required behavior and quality attributes.
Implementer	Responsible for the development of specific elements according to designs, requirements, and the architecture.	Understanding inviolable constraints and exploitable freedoms on development activities.

continues

Table P.1 Some of the stakeholders of architecture documentation, their roles, and how they might use it (*continued*)

Name	Description	Use for Architecture Documentation
Integrator	Responsible for taking individual components and integrating them, according to the architecture and system designs.	Producing integration plans and procedures, and locating the source of integration failures.
Maintainer	Responsible for fixing bugs and providing enhancements to the system throughout its life (including adaptation of the system for uses not originally envisioned).	Understanding the ramifications of a change.
Network administrator	Responsible for the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment, configuration, maintenance, and monitoring of network components.	Determining network loads during various use profiles and understanding uses of the network.
Product line manager	Responsible for development of an entire family of products, all built using the same core assets (including the architecture).	Determining whether a potential new member of a product family is in or out of scope and, if out, by how much.
Project manager	Responsible for planning, sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities.	Helping to set budget and schedule, gauging progress against established budget and schedule, and identifying and resolving development-time resource contention.
Representative of external systems	Responsible for managing a system with which this one must interoperate, and its interface with our system.	Defining the set of agreement between the systems.
System engineer	Responsible for design and development of systems or system components in which software plays a role.	Assuring that the system environment provided for the software is sufficient.
Tester	Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the architecture.	Creating tests based on the behavior and interaction of the software elements.
User	The actual end users of the system. There may be distinct kinds of users, such as administrators, superusers, and so on.	Users, in the role of reviewers, might rely on architecture documentation to check whether desired functionality is being delivered. Users might also refer to the documentation to understand what the major system elements are, which can aid them in emergency field maintenance.

P.2.3 Architecture Documentation and Quality Attributes

If architecture is largely about the achievement of quality attributes, and if one of the main uses of architecture documentation is to serve as a basis for analysis (to make sure the architecture will achieve its required quality attributes), where do quality attributes show up in the documentation? There are five major ways:

1. Any major design approach (such as an architecture pattern or style) chosen by the architect will have quality attribute properties associated with it. Client-server is good for scalability, layering is good for portability, an information-hiding-based decomposition is good for modifiability, services are good for interoperability, and so forth. Explaining the choice of approach is likely to include a discussion about the satisfaction of quality attribute requirements and trade-offs incurred. Look for the place in the documentation where such an explanation occurs. In our approach, we call that *rationale*.

For more on styles and patterns, see "Coming to Terms: 'Architecture Style' and 'Architecture Pattern' " on page 32, in this chapter.

Documenting rationale is covered in Section 6.5.

2. Individual architectural elements that provide a service often have quality attribute bounds assigned to them. Consumers of the services need to know how fast, secure, or reliable those services are. These quality attribute bounds are defined in the interface documentation for the elements, sometimes in the form of a Quality of Service contract. Or they may simply be recorded as *properties* that the elements exhibit.

Interface documentation is covered in Chapter 7.

Properties are discussed in Section I.3, in the introduction to Part I.

3. Quality attributes often impart a "language" of things that you would look for. Security involves things like security levels, authenticated users, audit trails, firewalls, and the like. Performance brings to mind buffer capacities, deadlines, periods, event rates and distributions, clocks and timers, and so on. Availability conjures up mean time between failure, failover mechanisms, primary and secondary functionality, critical and noncritical processes, and redundant elements. Someone fluent in the "language" of a quality attribute can search for the kinds of architectural elements (and properties of those elements) that were put in place precisely to satisfy that quality attribute requirement.
4. Architecture documentation often contains a *mapping to requirements* that shows how requirements (including quality attribute requirements) are satisfied. If your requirements document establishes a requirement for availability, for instance, then you should be able to look up that requirement by name or reference in your architecture document to see the place(s) where that requirement is satisfied.

Documenting a mapping to requirements is covered in Section 10.3.

5. Every quality attribute requirement will have a constituency of stakeholders who want to know that that quality attribute requirement is going to be satisfied. For these stakeholders, the architect should provide a special place in the documentation's introduction that either provides what the stakeholder is looking for or tells the stakeholder where in the document to find it. It would say something like "If you are a performance analyst, you should pay attention to the processes and threads and their properties (defined [here]), and their deployment on the underlying hardware platform (defined [here])." In our documentation approach, we put this here's-what-you're-looking-for information in a section called the documentation roadmap.

The documentation roadmap is described in Section 10.2.

P.2.4 Economics of Architecture Documentation

We'd all like to make our stakeholders happy, of course. Giddy, in fact. So why is producing high-quality architecture documentation often relegated to the "I'll do it if I have time" category of an architect's many tasks? Why do project managers often fail to insist that architecture documentation accompany the other archival artifacts produced during development? The answer, of course, is that an architecture document, let alone one that induces giddiness, costs time and money.

The man who stops advertising to save money is like the man who stops the clock to save time. [The same could be said for the architect who stops documenting.]

—Thomas Jefferson

Project managers are, by and large, rational people. (No, seriously, they are.) They are willing to invest resources in activities that yield demonstrable benefit, and not so much otherwise. As architects, we should be able to make a business case for producing and maintaining architecture documentation. And here it is: Activities that the project manager is going to have to fund will be less costly in the presence of high-quality, up-to-date documentation than they would otherwise.

A formula to show the savings looks like this:

$$\sum_{\text{over all activities A}}(\text{Cost of A without AD} - \text{Cost of A with AD}) > \text{Cost of AD},$$

where "Cost of A without AD" and "Cost of A with AD" are the cost of performing activity A without and with (respectively) an architecture document. "Cost of AD" is the cost of producing and maintaining the architecture documentation. In other words, the payback from good architecture documentation should exceed the effort to create it. Payback is measured in terms of effort saved.

This formula gives us a way to think about documentation, its effort, and its payoff. When deciding whether you should produce a particular piece of documentation, ask yourself how much effort it will take to do so, and what activities will be cheaper as a result. By choosing even a small number of key activities that will benefit from the presence of documentation, you should be able to make a convincing back-of-the-envelope argument that the effort invested will more than pay for itself.

And if you can't—that is, if the effort doesn't pay for itself—then you shouldn't expend it. Put your resources elsewhere.

The formula is nicely general; it does not require that you actually enumerate all the activities involved. The ones that are not affected by the presence or absence of architecture documentation at all simply wash out of the formula. But other activities such as coding, re-engineering, launching a change effort, and so on should have significant cost savings.

P.2.5 The Views and Beyond "Method"

We call our approach to documentation Views and Beyond. This is to emphasize that we use the concept of a view—explained in the next section—as the fundamental organizing principle for architecture documentation, but also because we go beyond views to include additional information that belongs in an architecture document.

Views and Beyond is not actually a method. It does not have a sequence of steps, with entry and exit criteria for each. Rather, it is more a collection of techniques that carry out an underlying philosophy. The philosophy is that an architecture document should be helpful to the people who depend on it to do their work (far from least of which is the architect). The techniques can be bundled into a few categories:

Chapter 9 covers a way to use stakeholder needs to determine the views you include in your architecture document.

Chapter 11 covers reviewing documentation.

Chapter 10 covers packaging and organization of documentation.

1. Finding out what stakeholders need. If you don't do this, you're going to end up with documentation that may serve no one.
2. Providing the information to satisfy those needs by recording design decisions according to a variety of views, plus the beyond-view information.
3. Checking the resulting documentation to see if it satisfied the needs.
4. Packaging the information in a useful form to its stakeholders.

Don't consider architecture documentation as a task separate from design; rather, make it an essential part of the architecture design process, serving as a ready vessel for holding the output of architectural decisions as soon as those decisions are made.

While items 3 and 4 denote document-centric activities, items 1 and 2 denote activities that should be carried out in conjunction with performing the architecture design. That is, we *don't want* Views and Beyond to be an architecture *documentation* method; rather, we want it to help the architect identify and record the necessary design decisions as they are made. Documentation should be the helpful result of making an architecture decision, not a separate step in the architecture process. The more that documentation is treated like a follow-on to design, with its own separate method, the less likely it is to be done at all.

P.2.6 Views and Beyond in an Agile Environment

[W]e have come to value . . . working software over comprehensive documentation.

—The Agile Manifesto (Agile Alliance 2002)

Section E.4 in the epilogue elaborates on architecture documentation in an Agile environment.

It is an unfortunate myth that Agile development and documentation (particularly architecture documentation) are at odds with each other. They aren't, and there are many examples of Agile leaders saying exactly that. Nevertheless, it is possible to interpret the advice in this book as prescribing a heavyweight and cumbersome approach to documentation. You can imagine an architect lagging hopelessly behind the project, which has gone on to deliver the product while he or she is still struggling to complete a Views-and-Beyond-style documentation package from six iterations ago. Neither the architect (nor this book) would likely be invited back to the next project.

Here is some advice that applies to *all* projects but especially to Agile projects: The Views and Beyond approach provides guidance for documenting many kinds of architecture information: structures, elements, relations, behavior, interfaces, rationale, traces to requirements, style guides, system context, and a whole lot more. But nowhere is it written that you have to do all of that. Decide what is useful (you can use the formula in Section P.2.4 to help you decide). Then, for example, if you decide that documenting the rationale behind a certain design decision is going to pay off in the future, then you can use the available guidance to help you do it. If you decide that documenting certain views is useful, then you can use the available guidance to help you do it. And so forth.

Choose what's useful and cost-effective to document. Document that. Period.

P.2.7 Architectures That Change Faster Than You Can Document Them

When your Web browser encounters a file type it's never seen before, odds are that it will go to the Internet, download the appropriate plug-in to handle the file, install it, and reconfigure itself to use it. Without even needing to shut down, let alone go through the code-integrate-test development cycle, the browser is able to change its own architecture by adding a new component.

Service-oriented systems that utilize dynamic service discovery and binding also exhibit these properties. More challenging systems that are highly dynamic, self-organizing, and reflective (meaning self-aware) are on the horizon. In these cases, the identities of the components interacting with each other cannot be pinned down, let alone their interactions, in any static architecture document.

Another kind of architectural dynamism, equally challenging from a documentation perspective, is found in systems that are rebuilt and redeployed with great rapidity. Some development shops, such as those responsible for commercial Web sites, build and "go live" with their system many dozens of times every single day.

Whether an architecture changes at runtime, or as a result of a high-frequency release-and-deploy cycle, both share something in common with respect to documentation: They change much faster than the documentation cycle. In either case, nobody is going to hold up things until a new architecture document is produced, reviewed, and released.

But knowing the architecture of these systems is every bit as important, and arguably more so, than for systems in the world of more traditional life cycles. Here's what you can do if you're an architect in a highly dynamic environment:

1. **Document what is true about all versions of your system.** Your Web browser doesn't go out and grab just any piece of software when it needs a new plug-in; a plug-in must have specific properties and a specific interface. And it doesn't just plug in anywhere, but in a predetermined location in the architecture. Record those invariants as you would for any architecture. This may make your documented architecture more a description of constraints or guidelines that any compliant version of the system must follow. That's fine.
2. **Document the ways the architecture is allowed to change.** In the previous examples, this will usually mean adding new components and/or replacing components with new implementations. In the Views and Beyond approach, the place to do this is called the variability guide.
3. **Make your system capture its own architecture-of-the-moment automatically.** When your Web browser or SOA system crashes, your recovery team is going to want to know exactly what configuration was running when the problem occurred. This ability can run the spectrum from primitive (write changes in a log file) to sophisticated (drive a real-time display of the components and their interactions, much like what is found in network service centers).

Using a variability guide to document an architecture's variation points is covered in Section 6.4.

P.3 Architecture Views

A **view** is a representation of a set of system elements and the relationships associated with them.

Perhaps the most important concept associated with software architecture documentation is that of the **view**. A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion. Our analogy with the bird wing proves illuminating. If you are interested in any but the most superficial understanding, then no single rendition of a bird wing will do. Instead, you need many: feathers, skeleton, circulation, muscular views, and many others. Which of these views *is* the "architecture" of the wing? None of them. Which views *convey* the architecture? All of them.

For more information about the bird wing analogy, see "About the Cover" on page xxi.

In this book, we use the concept of views to give us the most fundamental principle of architecture documentation, illustrated in Figure P.1:

Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.

Chapter 9 shows how to choose the relevant views. Section 10.1 shows how to document a view, and Section 10.2 shows how to document the information that applies to more than one view.

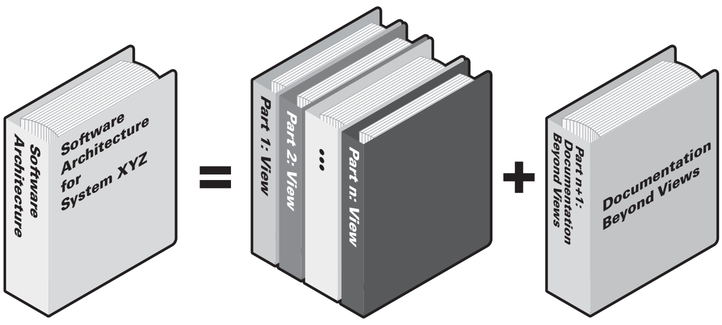
What are the relevant views? It depends on your goals. As we saw previously, architecture documentation can serve many purposes: a mission statement for implementers, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning.

Different views also expose different quality attributes to different degrees. Therefore, the quality attributes that are of most concern to you and the other stakeholders in the system's development will affect the choice of what views to document. For instance, a *layered view* will tell you about your system's portability, a *deployment view* will let you reason about your system's performance and reliability, and so forth.

Layered views are covered in Section 2.4. Deployment views are covered in Section 5.2.

Different views support different goals and uses. This is fundamentally why we do not advocate a particular view or collection

Figure P.1
A documentation package for a software architecture can be composed of one or more view documents and documentation that explains how the views relate to one another, introduces the package to its readers, and guides them through it.


Software Architecture
Software Architecture for System XYZ
=
Part 1: View
Part 2: View
...
Part n: View
+
Part n+1: Documentation Beyond Views
Documentation Beyond Views

of views. The views you should document depend on the uses you expect to make of the documentation. Different views will highlight different system elements and/or relations.

It may be disconcerting that no single view can fully represent an architecture. Additionally, it feels somehow inadequate to see the system only through discrete, multiple views that may or may not relate to one another in any straightforward way. The essence of architecture is the suppression of information not necessary to the task at hand, and so it is somehow fitting that the very nature of architecture is such that it never presents its whole self to us but only a facet or two at a time. This is its strength: Each view emphasizes certain aspects of the system while deemphasizing or ignoring other aspects, all in the interest of making the problem at hand tractable. Nevertheless, no one of these individual views adequately documents the software architecture for the system. That is accomplished by the complete set of views along with information that transcends them.

An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

—Gamma et al. (1995, p. 22)

The documentation for a view contains

- A primary presentation, usually graphical, that depicts the primary elements and relations of the view
- An element catalog that explains and defines the elements shown in the view and lists their properties
- A specification of the elements' interfaces and behavior
- A variability guide explaining any built-in mechanisms available for tailoring the architecture
- Rationale and design information

Section 10.1 substantially elaborates this outline.

The documentation that applies to all of the views contains

- An introduction to the entire package, including a reader's guide that helps a stakeholder find a desired piece of information quickly
- Information describing how the views relate to one another, and to the system as a whole
- Constraints and rationale for the overall architecture
- Such management information as may be required to effectively maintain the whole package

Section 10.2 substantially elaborates this outline.

COMING TO TERMS

A Short History of Architecture Views

Nearly all modern approaches to designing and documenting architectures rely on the concept of an architectural view. Where did this concept come from?

More than three decades ago, David Parnas (1974) observed that software consists of many structures, which he defined as partial descriptions showing a system as a collection of parts and showing some relations among the parts. This definition largely survives in architecture papers today. Parnas identified several structures prevalent in software. A few were fairly specific to operating systems, such as the structure that defines what process owns what memory segment, but others are more generic and broadly applicable. These include the *module structure*, in which the units are work assignments and the relation is *is-a-part-of* or *shares-part-of-the-same-secret-as*; the *uses structure*, in which the units are programs, and the relation is *depends on the correctness of*; and the *process structure*, in which the units are processes, and the relation is *gives computational work to*.

Quite a bit later, DeWayne Perry and Alexander Wolf recognized that, similar to building architecture, a variety of views of a system are required. Each view emphasizes certain architectural aspects that are useful to different stakeholders or for different purposes (Perry and Wolf 1992).

Later, Philippe Kruchten (1995) of the Rational Software Corporation wrote an influential paper describing four main views of software architecture (logical, process, development, physical) that can be used to great advantage in system building, along with a distinguished fifth view that ties the other four together by showing how they satisfy key use cases: the "4+1" approach to architecture. The 4+1 approach has since been embraced as a foundation piece of the Rational Unified Process.

To see how the 4+1 views correspond to views described in this book, see Section E.2 of the epilogue.

At about the same time, Dilip Soni, Robert Nord, and Christine Hofmeister of Siemens Corporate Research made a similar observation about views of architecture they found in use in industrial practice (Soni, Nord, and Hofmeister 1995). They wrote about the conceptual view, module interconnection view, execution view, and code view. These views, which correspond more or less to Kruchten's four views, have become known as the Siemens Four View model for architecture.

The Siemens Four View model is explained in the book by Hofmeister, Nord, and Soni (2000).

Other "view sets" have emerged since these. In their book *Software Systems Architecture*, Rozanski and Woods (2005) advocate using functional, information, concurrency, development, deployment, and operational views. Philips Research, the R&D arm of the giant Dutch electronics company, has created the "CAFCR" model of architecture, which calls for five views: the ***c***ustomer, ***a***pplication, ***f***unctional, ***c***onceptual, and ***r***ealization views.

In the year 2000, the IEEE adopted a standard (IEEE 1471-2000) for architecture descriptions. Unlike approaches that prescribe a fixed set of views, this standard advocates creating your own views that best serve the stakeholders and their concerns associated with your system. (The Views and Beyond approach also advises flexibility in choosing your view set.)

IEEE 1471-2000 is now known as ISO/IEC 42010:2007. We describe this standard in Section E.1 of the epilogue.

P.4 Architecture Styles

Recurring forms have been widely observed, even if written for completely different systems. These forms occur often enough that they are worth writing and learning about in their own right. We call these forms **architecture styles**. (In this book, we usually just say *styles*.) Styles have implications for architecture documentation and deserve definition and discussion in their own right.

An **architecture style** is a specialization of element and relation types, together with a set of constraints on how they can be used.

Styles allow one to apply specialized design knowledge to a particular class of systems and to support that class of system design with style-specific tools, analysis, and implementations. The literature is replete with a number of styles, and most architects have a wide selection in their repertoires.

For example, we'll see that modules can be arranged into a useful configuration by restricting what each one is allowed to use. The result is a layered style that imparts to systems that use it qualities of modifiability and portability. Different systems will have a different number of layers, different contents in each layer, and different rules for what each layer is allowed to use. However, the layered style is abstract with respect to these options and can be studied and analyzed without binding them.

For another example, we'll see that client-server is a common architecture style. The elements in this style are clients, servers, and the protocol connectors that depict their interaction. When used in a system, the client-server style imparts desirable

In all processes of life people imitate, and so must artists. They are influenced by their peers as by their antecedents because this is the way of organic development. Late Beethoven and early Schubert, for instance, are almost indistinguishable; while Brahms took certain themes, note for note, from Beethoven; and Shakespeare stole nearly all of his plots—all the good ones certainly.

—Agnes de Mille, American dancer and choreographer (Atlantic 1956) properties to the system, such as the ability to add clients with little effort. Different systems will have different protocols, different numbers of servers, and different numbers of clients each can support. However, the client-server style is abstract with respect to these options and can be studied and analyzed without binding them.

The layered style is described in Section 2.4.

The client-server style is described in Section 4.3.1.

Some styles are applicable in every software system. For example, every system is decomposed into modules to divide the work; hence, the decomposition style applies everywhere. Other examples of "universal styles" are uses, deployment, and work assignment. Some styles occur only in systems in which they were explicitly chosen and designed in by the architect: layered, service oriented, and multi-tier, for example.

Choosing a style, whether it's one covered in this book or somewhere else, imparts a documentation obligation to record the specializations and constraints that the style imposes and the characteristics that the style imparts to the system. We call this piece of documentation a **style guide**. The obligation to document a style can usually be discharged by citing a description of the style in the literature: this book, for example. If you invent your own style, however, you should write a style guide for it because it will help you and your peers to apply that style in other systems.

A **style guide** is the description of an architecture style that specifies the vocabulary of design (sets of element and relationship types) and the rules (sets of topological and semantic constraints) for how that vocabulary can be used.

The contents of a style guide are given in Section I.2, in the introduction to Part I. Section 6.1.4 discusses how to create and document a new style.

No system is built exclusively from a single style. On the contrary, every system can be seen to be an amalgamation of many different styles. Some (such as decomposition and work assignment) occur in every system, but in addition to these, systems can exhibit a combination of one or more "chosen" styles as well.

Combining views is an important concept covered in Section 6.6.

Even restricting our attention to component-and-connector styles, it's possible for one system to exhibit several styles in the following ways:

A **bridging element** is an element that is common to two views and is used to provide the continuity of understanding from one view to the other. A bridging element appears in both views and has supporting documentation, usually a mapping between views, that makes the correspondence clear, perhaps by showing the combined picture.

- Different "areas" of the system might exhibit different styles. For example, a system might use a pipe-and-filter style to process input data but route the result to a database that is accessed by many elements. This system would be a blend of pipe-and-filter and shared-data styles. Documentation for this system would include (1) a pipe-and-filter view that showed one part of the system and (2) a shared-data view that showed the other part. In a case like this, one or more elements must occur in both views and have properties of both kinds of elements. (Otherwise, the two parts of the system could not communicate with each other.) These **bridging elements** provide the continuity of understanding from one view to the next. They likely have multiple interfaces, each providing the mechanisms for letting the element work with other elements in each of the views to which it belongs. The filter/database connector in Figure P.2 is an example.

- An element playing a part in one style may itself be composed of elements arranged in another style. For example, a service provider in an SOA system might, unknown to other service providers or its own service users, be implemented using a multi-tier style. Documentation for this system would include an SOA view showing the overall system, as well as a multi-tier view documenting that server, as illustrated in Figure P.3.
- Finally, the same system might simply be seen in different lights, as though you were looking at it through filtered glasses. For example, a system featuring a database repository, as in Figure P.4, may be seen as embodying either a shared-data style or a client-server style. The glasses you choose will determine the style that you "see."

In the last case, your choice of style-filtered glasses depends, once again, on the uses to which you and your stakeholders intend to put the documentation. For instance, if the shared-data style is more easily understood by the stakeholders that will consume that view, you might choose it. If you need the perspective afforded by more than one style, however, you have a choice. You can document the corresponding views separately, or you can combine them into a single view that is, roughly speaking, the union of what the separate views would be.

This combined view is called an overlay. Overlays are discussed in Section 6.6.

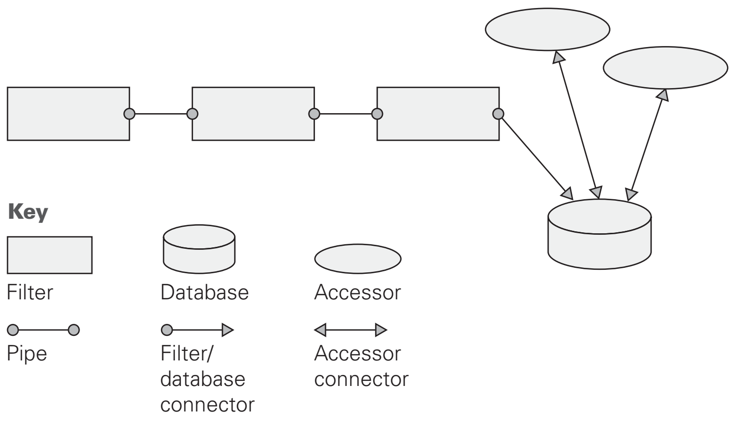


Figure P.2
A system combining a pipe-and-filter style with a shared-data style. The "filter/database connector" is a bridging element.

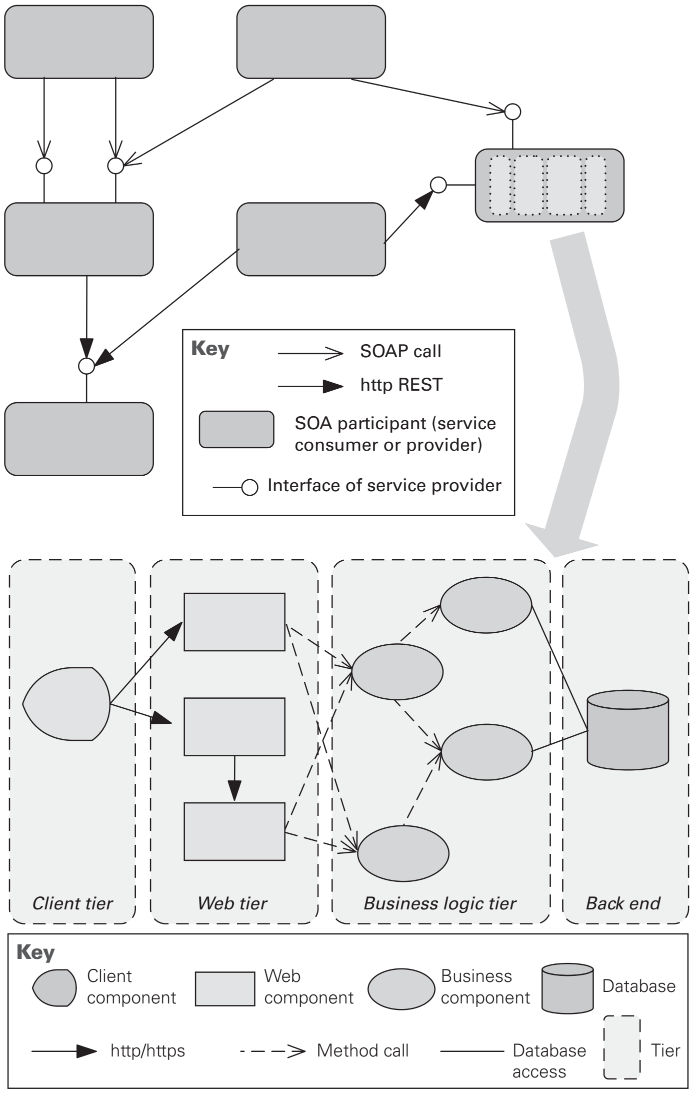


Figure P.3
A system combining two styles. Here a service provider is composed internally in a multi-tier style.

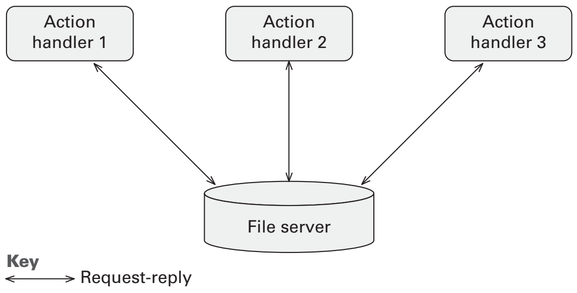


Figure P.4
This system could be in the shared-data style, or the client-server style, depending on your perspective.

P.4.1 Three Categories of Styles

Although no fixed set of views is appropriate for every system, broad guidelines can help us gain a footing. Architects need to think about their software in three ways simultaneously:

1. How it is structured as a set of implementation units
2. How it is structured as a set of elements that have runtime behavior and interactions
3. How it relates to nonsoftware structures in its environment

Each style we present in this book falls into one of these three categories:

1. *Module* styles
2. *Component-and-connector (C&C)* styles
3. *Allocation* styles

A selection of module styles is presented in Chapter 2. A selection of C&C styles is presented in Chapter 4. A selection of allocation styles is presented in Chapter 5.

When we apply a style to a system, the result is a view. Module views document a system's principal units of implementation. C&C views document the system's units of execution. And allocation views document the relations between a system's software and nonsoftware resources of the development and execution environments.

One of the best ways to avoid confusion in your architecture is to be meticulous about making it clear whether each architecture element is a module or a component.

COMING TO TERMS

Module, Component

In this book, we rely on three categories of styles: module, component-and-connector, and allocation. This three-way distinction allows us to structure the information we're presenting in an orderly way and, we hope, allows you to recall it and access it in an orderly way, so that you can write an architecture document that presents *its* information in an orderly way. But for this strategy to succeed, the distinctions have to be meaningful. Two of the categories rely on words for which we give precise meanings, but which are not historically well differentiated: *module* and *component*.

Like many words in computing, these two have meanings outside our field. Furthermore, both terms have come to be associated with movements in software engineering that have overlapping goals.

During the 1960s and 1970s, software systems increased in size and were no longer able to be produced by one person. It became clear that new techniques were needed to manage software complexity and to partition work among programmers. To address such issues of "programming in the large," various criteria were introduced to help programmers decide how to partition their software. Encapsulation, information hiding, and abstract data types became the dominant design paradigms of the day. Until this movement, computer programs were largely about calculating the correct answer, but thought leaders were now saying that how you structure your code determines other important properties of the system. *Module* became the carrier of their meaning. The 1970s and 1980s saw the advent of "module interconnection languages" and features of new programming languages such as Modula modules, Smalltalk classes, and Ada packages. Today's dominant design paradigm—object-oriented programming—has these module concepts at its heart. Components, by contrast, are in the limelight with component-based software engineering and the component-and-connector perspective in the software architecture field.

Both movements aspire to achieve rapid system construction and evolution through the selection, assembly, and wholesale replacement of independent subpieces. Both modules and components are about the decomposition of a whole software system into constituent parts. But beyond that, the two terms take on different shades of meaning.

- A module refers first and foremost to a unit of implementation. Parnas's foundational work in module design (Parnas 1972) used information hiding as the criterion for allocating responsibility to a module. Information that was likely to change over the lifetime of a system, such as the choice of data structures or algorithms, was assigned to a module, which had an interface through which its facilities were accessed. Modules have long been associated with source code, but information models, XML files, config files, BNF files for parsers, and other implementation artifacts are all perfectly fine modules.
- A component refers to a runtime entity. Szyperski says that a component "can be deployed independently and is subject to composition by third parties" (Szyperski 1998, p. 30). The emphasis is clearly on the finished product and not on the implementation considerations that went into it. Indeed, the operative

model is that a component is delivered in the form of an executable binary only: Nothing upstream from that is available to the system builder.

In short, a module suggests implementation units and artifacts, with less emphasis on the delivery medium and what goes on at runtime. A component is about units of software active at runtime with no visibility into the implementation structure.

Who cares? If every module turned into exactly one component at runtime, it would be easy to sweep the difference under the rug. But this is often far from reality! In many systems, a single module might turn into many components, or it might take many modules to turn into a single component. An easy way to see this is to imagine a trivially simple client-server system. Suppose our system has a single server, which at runtime serves up some interesting piece of data to ten interested clients, all of which do the same thing. This system has *eleven components* but only *two modules*. The server module maps 1:1 onto the server component S1. The client module maps 1:10 to the client components C1–C10. Failing to distinguish between modules and components makes it too easy to blithely assume that every unit of implementation turns into exactly one unit of execution. It isn't so.

Our use of the terms in this book reflects their pedigrees. Module styles described in this book reflect implementation artifact considerations: decompositions that assign parts of the problem to units of design and implementation,

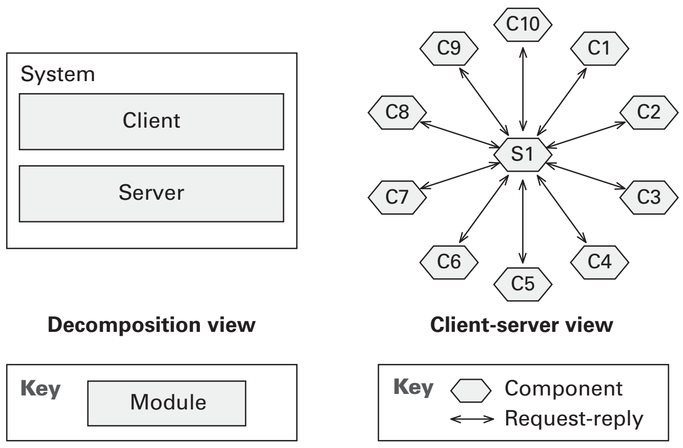


Figure P.5
A client-server system might consist of two modules but eleven components.

layers that reflect what uses are allowed when software is being written, and classes that factor out commonality from a set of instances. Modules in these styles are often units of source code, but there's also the data model style, where the module is a model of the data that the system manipulates. Of course, all these module styles have runtime implications; that's the end game of software design, after all. C&C styles described in this book focus on how processes interact and data travels around the system during execution.

Section 10.2 describes how to document the mapping between a system's modules and its components. Sections 1.5 and 3.5 discuss how modules and components relate to each other.

In many architectures, there is a one-to-one mapping between modules and components. Further, the module and its component counterpart are usually given the same name in this case. This makes it tempting to believe that the modules and components are the same, which in turn makes it tempting to believe there is no difference. Don't be tempted. Although a one-to-one mapping does no harm, the truth is that the module and component are different elements sharing the same name. In such an architecture, the module will show up in a module view, and a component with the same name will show up in one or more component-and-connector views.

Modules and components represent the current bedrock of the software engineering approach to rapidly constructed, easily changeable software systems. As such, modules and components serve as fundamental building blocks for creating and documenting software architectures.

COMING TO TERMS

"Architecture Style" and "Architecture Pattern"

What do the two terms mean?

In this book we use "architecture style" as the term for a package of design decisions that explains a generic design approach for a software system. Another term for a similar concept, used by many architects and authors, is "architecture pattern." What is the difference between these two concepts and why did we choose style over pattern?

An **architecture style** is a "specialization of element and relation types, together with a set of constraints on how they can be used" (Bass, Clements, and Kazman 2003).

An **architecture pattern** "expresses a fundamental structural organization schema for software systems" (Buschmann et al. 1996, p. 12). It is, above all, a *pattern*, which in the context of architecture "describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relations, and the ways in which they collaborate" (Buschmann et al. 1996, p. 8).

An essential part of an architecture pattern is its focus on the problem and context as well as how to solve the problem in that context. That last part we'll call the architecture approach. An architecture style focuses on the architecture approach, with more lightweight guidance on when a particular style may or may not be useful. Very informally, we can put it this way (where the arrow means "suggests"):

- Architecture pattern: {problem, context} → architecture approach
- Architecture style: architecture approach

How did these two terms come about?

"Architecture style" as we use it today traces to some early writing from the formative days of software architecture study.

In 1990 and 1991, Mary Shaw was noticing and describing recurring architecture concepts she found in many systems. She called these "elements of a design language for software architecture" or "design idioms" (Shaw 1990, 1991). In 1992 Dewayne Perry and Alexander Wolf wanted to "build an intuition" about the still-new field of software architecture (Perry and Wolf 1992). Looking around at other kinds of architecture—network architecture, computer architecture, and others—they hit upon building architecture as rich in fertile (and borrowable) concepts. One of those concepts was **architecture style**. Like Shaw before them, they were also noticing recurring design forms in software architectures, and they saw that this would be a useful term to appropriate to describe those forms. Styles, then, were observed phenomena, approaches (manifest in the kinds of elements and relations employed) that the authors noticed were being

Thus, we find in building architecture some fundamental insights about software architecture: multiple views are needed to emphasize and to understand different aspects of the architecture; styles are a cogent and important form of codification that can be used both descriptively and prescriptively; and, engineering principles and material properties are of fundamental importance in the development and support of a particular architecture and architectural style.

—Perry and Wolf (1992)

[In building architecture,] architectural styles classify architecture in terms of form, techniques, materials, time period, region, etc. . . . leading to a terminology such as Gothic "style."

—Wikipedia (2010a)

used over and over. The emphasis was on discovery and categorization of utilized forms.

In 1996 Frank Buschmann and his colleagues at Siemens made the inevitable connection between two powerful concepts: software architecture and design patterns (the latter having electrified software engineering the previous year). Their book, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* (Buschmann et al. 1996; PoSA, for short), is where the term **architectural pattern** was first used. Followed over the years by (at this writing) four sequels, the PoSA series does for architects what *Design Patterns* (Gamma et al. 1995) did for designers and programmers.

"An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." (Buschmann et al. 1996, p. 12)

Both design patterns and (software) architecture patterns owe their meaning to the building architect Christopher Alexander, who in the 1970s wrote several books detailing architecture approaches to solve common building design problems. People love to sit next to windows, he wrote, so make every room have a place where they can comfortably do so. People love balconies, he wrote, but observations show they won't spend time on a balcony less than 10 feet wide. So make your balconies at least 10 feet wide. People love outdoor spaces, he wrote, but not if they're in the shadow of a building. So in the northern hemisphere put your courtyards on the south side. He called these design nuggets patterns: "a three-part rule, which expresses a relation between a certain context, a problem, and a solution" (Alexander 1979, p. 247). The patterns community (of whatever flavor) has tried to remain faithful to his meaning.

We must not forget that the wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once.

—D. L. Parnas (1996)

Why do patterns seem more specific?

It has turned out, not as a matter of the intrinsic nature of these things but rather as a matter of practice, that the published architecture patterns tend to be more constraining—that is, they embed more design decisions—than the published architecture styles. Patterns often look "more detailed" or "less abstract" than styles. Styles tend to tell people what the element and relation types of interest are, and give topological constraints: Put layers on top of layers; pipes connect to filters, not pipes; and so on. Patterns tend to be more specific, showing instances of the element type interacting with each other.

That's because the collectors of styles were motivated to find commonality where none had been observed before. Broad categories are more inclusive. Pattern writers have tended to record very specific and context-dependent problems; hence their solutions are correspondingly specific.

Architects can use this *de facto* distinction to their advantage. For instance, if you're handling a lot of data in your system, you might want to consider a style (the shared-data style is a good candidate) and ask yourself if the element and relation types are what you need: That is, do you *really* need a database? Yes? OK, now go look for a more constrained architecture approach (which might very well be given as a pattern).

The shared-data style is described in Section 4.5.1.

Why did we use "architecture style" in this book?

In this book, which is about documenting software architectures and not so much about designing them, we concentrate on presenting a variety of solution approaches—architecture styles—so that we can show how to document systems built using them. In a software architecture document, one doesn't document a pattern, one documents an application of it—that is, the instantiated solution approach.

How do I document the use of a style or pattern in a software architecture document?

Architects can use either patterns or styles as a starting point for their design. They might be published in existing catalogs, stored in an organization's proprietary repository of standard designs, or created specifically for the problem at hand by the architect. In either case, they provide a generic (that is, incomplete) solution approach that the architect will have to refine and instantiate.

First, record the fact that the given style or pattern is being used. Then say why this solution approach was chosen—why it is a good fit to the problem at hand. If the chosen approach comes from a pattern, show that the problem at hand fits the problem and context of the pattern. If the chosen approach comes from a style, explain why the style does the needed job.

The software architecture document templates in Chapter 10 will provide a place for all of this information.

Using a pattern or a style means making successive design decisions that eventually result in an architecture. These design decisions manifest themselves as newly instantiated elements and relations among them. The architect can document a snapshot of the architecture at

The concept of making successively more constrained design decisions is called a "spectrum of design" and is discussed in Section 6.1.3.

each stage. How many stages there are depends on many things, not the least of which is the ability of readers to follow the design process in case they have to revisit it in the future.

Summary

Architecture styles represent observed architecture approaches. A style description does not generally include detailed problem/context information. Architecture patterns do. An architecture approach might be documented (and several are) as an architecture style *and* an architecture pattern. Both styles and patterns are a set of prepackaged design decisions involving the choice of element types, relation types, properties, and constraints on the topology and interaction among the elements via the relations. Both provide vocabularies that shortcut explanation and allow greatly facilitated communication ("My system is layered." "Ah, I understand. What are the layers?"), and help chart a course to the satisfaction of specific quality attribute requirements. Both can be used in combination—it is a rare system that uses only one style or one architecture pattern. And both represent essential elements of an architect's vocabulary.

Styles are described using a common set of information; this layout is called a *style guide*. The style guide we use to describe the styles covered in this book is explained in the introduction to Part I.

P.5 Seven Rules for Sound Documentation

These are the rules for any technical documentation, including software architecture documentation:

1. Write documentation from the reader's point of view.
2. Avoid unnecessary repetition.
3. Avoid ambiguity.
4. Use a standard organization.
5. Record rationale.
6. Keep documentation current but not too current.
7. Review documentation for fitness of purpose.

Architecture documentation is much like the documentation we write in other facets of our software development projects. As such, it obeys the same fundamental rules for what distinguishes good, usable documentation from poor, ignored documentation. We close the prologue with seven rules for sound software documentation. Use this checklist when you write technical documentation. (You can also use it when you read technical documentation: the rules provide objective criteria for judging a document's quality, and they let you say something constructive in a critical review.)

Rule 1: Write Documentation from the Reader's Point of View

The consumer isn't a moron. She is your wife.

—David Ogilvy, writing about advertising

This rule simply reminds us to keep the end game in mind as we produce our documentation: Make your document serve its stakeholders and their intended uses of it. It is surprisingly easy to forget that rule in the midst of looming deadlines, an overflowing e-mail queue, and a cell phone that won't shut up.

The great computing scientist Edsger Dijkstra (1930–2002), the inventor of many of the software engineering principles we now take for granted, once said that he would happily spend two hours pondering how to make a single sentence clearer. He reasoned that if the paper were read by a couple of hundred people—a decidedly modest estimate for someone of Dijkstra's caliber—and he could save each reader a minute or two of confusion, it was well worth the effort. Professor Dijkstra's consideration for the reader reflects his classic manners, but it also gives us a new and useful concept of the effort associated with a document. Usually we just count how long it takes to write. Dijkstra taught us to be concerned with how long it takes to *use*. Writing a document that a reader finds easy to use will help tilt the economics of documentation in our favor, as defined in the formula in Section P.2.4.

I have made this letter rather long only because I have not had time to make it shorter.

—Blaise Pascal, French mathematician, physicist, and moralist

Writing for the reader is just plain polite, but it has a practical advantage as well. A reader who feels that the document was written with him or her in mind appreciates the effort but, more to the point, will come back to the document again and again in the future. Documents written for the reader will be read; documents written for the convenience of the writer will not. All of us like to shop at stores that seem to want our business, and we avoid stores that do not. This is no different.

Tips on how to write for the reader include:

- Find out who your readers are, what they know, and what they expect of the document. Have an informal chat with some representatives of various kinds of readers and see what their expectations are. Don't make uninformed assumptions about what your readers know.
- Avoid stream of consciousness writing. If you find yourself writing things down in the order they occur to you, without an overall organizational plan, stop. Work out where specific kinds of information should go and put them where they belong. Make sure that you know what question(s) are being answered by each section of a document.
- Avoid unnecessary insider jargon. The documentation may be read by someone new to the field or from a company that does not share the same jargon. Add a glossary to define specialized terms.
- Avoid overuse of acronyms. Resist using an acronym when the spelled-out phrase is short or it appears only a few times. Always provide a dictionary that decodes whatever acronyms you do use.

The true measure of a man is how he treats someone who can do him absolutely no good.

—Attributed to Samuel Johnson

Rozanski and Woods's book *Software Systems Architecture* (2005) lists the following properties of an "effective architectural description": correctness, sufficiency, conciseness, clarity, currency, and precision.

Rule 2: Avoid Unnecessary Repetition

Each kind of information should be recorded in exactly one place. This makes documentation easier to use and *much* easier to change as it evolves. It also avoids confusion: information that is repeated is likely to be in a slightly different form, and now the reader must wonder "Was the difference intentional? If so, what is the meaning of the difference? Did the author change one place and forget to update the other?"

It should be a goal that information never be repeated. However, at times the cost to the reader of not repeating information in the other places where it's needed is high. Readers don't like to flip pages or click hyperlinks unnecessarily. The information may be repeated in two or more different places for clarity or to make different points. Also, expressing the same idea in different forms is often useful for achieving a thorough understanding. If keeping the information separate comes at too high a cost to the reader, repeat the information.

In a document maintained and viewed online, hyperlinks make this rule easier to follow. For example, each term can be hyperlinked to its definition; a concept can be hyperlinked to an explanation or elaboration.

The data flow diagrams . . . don't seem to be much use. They're just vague pictures suggesting what someone thinks might be the shape of a system to solve a problem, and no one's saying what the problem is. [T]he big picture isn't much use if it doesn't say anything you can understand. You're all just guessing what Fred's diagram means. It wouldn't mean anything at all to you if you didn't already have a pretty good idea of what the problem is and how to solve it.

—A character in a parable about data flow diagrams written by Michael Jackson (1995)

PERSPECTIVES

Beware Notations Everyone "Just Knows"

Rule 3 admonishes us to avoid ambiguity. "A well-defined notation with precise semantics," we say, "goes a long way toward eliminating whole classes of linguistic ambiguity from a document." Here we want to emphasize the part about "precise semantics." Just having a well-defined notation is not enough.

Consider data flow diagrams. Years ago Michael Jackson wrote a wonderful Socratic dialogue that showed how a data flow diagram is largely incapable of conveying useful information about a software design unless you already have a pretty good idea what the design is by the time you start looking at it (Jackson 1995, pp. 42–47; we reprinted the dialogue in Chapter 11 of the first edition of this book [Clements et al. 2003]). Data flow diagrams, for heaven's sake! They've been around for decades. Can it really be that nobody understands what they mean? Jackson was able to show convincingly how easily they can be misinterpreted.

Consider layer diagrams. Layered systems were first described more than four decades ago. We've all seen them; we've all written them. Yet how many times have we stopped to ask exactly what they mean? A layer diagram is about the only graphical representation of architecture in which position is significant. Box 1 on top of Box 2 is quite a different system than Box 2 on top of Box 1. What does it mean, exactly, that some rectangles are stacked up on top of each other? “Oh, the programs on top can call programs below” is an answer I often get when I ask this question in class. Well, can programs at the top call *any* programs below, or just the programs in the next lower layer? Ask this question in a room full of professional software engineers, and (if my experience teaching to these groups is any measure) you’ll usually get one-third nods, one-third head shakes, and one-third looking as though you just told them the sun is made of really shiny cheese. Can programs in a layer call other programs in the same layer? Generally the same response. And everyone, absolutely everyone, forgets to tell me that programs below are *not* allowed to call programs above, which is a rather important thing to remember about layers.

So, surprise: Simple layer diagrams are inherently ambiguous. Common variants, such as what I call “layers with a sidecar,” where a vertical box is smooshed up against the stack on one side, are even more ambiguous. (The good news is that they can be easily disambiguated.)

A well-defined notation is one in which you can look at an example and tell whether it’s a legal example of using the notation or not. Layers and data flow diagrams both have this property. But neither, traditionally presented, have precise enough semantics to be unambiguous.

Notations like this, where software engineers “just know” what they mean, are the most dangerous. We all might “know” what a layer diagram means. The problem is that what I “know” it means will be different from what you “know” it means, and different still from what the architect meant. So we’ll all go merrily along with no hint of a problem until late in the project when our errors in understanding may cause us to miss a deadline or suffer an operating failure.

—P.C.

It is far better to be explicit and wrong than to be vague.

—Frederick Brooks, Jr. (1995, p. 259)

Rule 3: Avoid Ambiguity

Ambiguity occurs when documentation can be interpreted in more than one way and at least one of those ways is incorrect. The most dangerous kind of ambiguity is undetected ambiguity. Here, each reader will think he or she understands the document, but unwittingly each reader will come to different conclusions about what it is saying.

Following two of the other rules will help you avoid ambiguity:

- By avoiding needless repetition (rule 2), you avoid the "almost but not quite alike" form of ambiguity.
- Reviewing the document with members of its intended audience (rule 7) will help spot and weed out ambiguities.

Clarity is our only defense against the embarrassment felt on completion of a large project when it is discovered that the wrong problem has been solved.

—C. A. R. Hoare (1985, p. 85)

A well-defined notation with precise semantics goes a long way toward eliminating whole classes of linguistic ambiguity from a document. This is one area where standard languages and notations help a great deal, but using a formal language isn't always necessary. Simply adopting a set of notational conventions and then using them consistently and rigorously will help eliminate many sources of ambiguity. But if you do adopt a notation, then the following corollary applies:

ADVICE

We have several things to say about box-and-line diagrams masquerading as architecture documentation.

- Don't be guilty of drawing one and claiming that it's anything more than a start at an architecture description.
- If you draw one yourself, make sure that you explain precisely what the boxes and lines mean.
- If you see one, ask its author what the boxes mean and what, precisely, the arrows connote. The result is usually illuminating, even if the only thing illuminated is the author's confusion.

Rule 3a: Explain Your Notation

The ubiquitous box-and-line diagrams that people always draw on whiteboards are one of the greatest sources of ambiguity in architecture documentation. Although not a bad starting point, these diagrams are certainly not good architecture documentation. First, most such diagrams suffer from ambiguity.

Are the boxes supposed to be modules, objects, classes, services, clients, servers, databases, processes, functions, tiers, procedures, processors, or something else? Do the arrows mean calls, uses, data flow, I/O, inheritance, communication, processor migration, or something else?

Make it as easy as possible for your reader to determine the meaning of the notation. The best way to do this is always to include a key in your diagrams. If you're using a standard visual language defined elsewhere, the key can simply name it or refer readers to the source of the language's semantics. Even if the language is standard or widely used, different versions often exist. Let your reader know, by citation, which one you're using. For example, "Key: UML 2.0" is a perfectly fine key, and it puts readers and authors on the same page. For a homegrown informal notation, include a key to the symbology. This is good practice because it compels you to understand what the pieces of your system are and how they relate to one another; it's also courteous to your readers.

Every diagram in the architecture documentation should include a key that explains the meaning of every symbol used. The key should identify the notation. If a predefined notation is being used (such as UML), the key should name it and if necessary cite the document that defines the version being used. Otherwise, the key should define the symbology and the meaning, if any, of colors, shapes, position, and other information-carrying aspects of the diagram. If your diagram uses color but the color has no particular meaning or is only there to enhance readability, say so in the key.

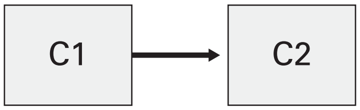
If you define an informal notation for your diagrams, try to use the same notation consistently across diagrams of the same type. Use different symbols for different types of elements and relations. For example, if you used a rounded rectangle for Web components in a diagram, avoid using a different shape for Web components in other diagrams.

PERSPECTIVES

Quivering at Arrows

Many architecture diagrams with an informal notation use arrows to indicate a directional relationship among architecture elements. Although this might seem like a good and innocuous way to indicate that two elements interact, it creates a great source of confusion in many cases. What do the arrows mean?

Consider the following architecture snippet:



What does the arrow mean? Here are some possibilities:

- C1 calls C2.
- Data flows from C1 to C2.
- C1 instantiates C2.
- C1 sends a message to C2.
- C1 is a subtype of C2. (Usually C2 would be positioned above C1, but that is not mandatory.)

- C2 is a data repository and C1 is writing data to C2.
- Conversely, C1 is a repository and C2 is reading data from C1.

Any of these might make sense, and people use arrows to mean all these things and more, often using multiple interpretations in the same diagram.

Suppose we know the arrow indicates that component C1 calls component C2. If your system uses different kinds of calls, it's a good idea to differentiate them in the diagrams. In particular, it is important to distinguish synchronous from asynchronous calls, and local from remote calls. Both aspects may have implications for behavior, performance, modifiability, and reliability of the interaction. It may also be useful to differentiate the technology used to implement the call when the solution will accommodate different ones. For example, a synchronous remote call can be implemented via a Web service such as SOAP, REST, Java RMI, or .NET remoting, among other options. To differentiate the types of interaction in the diagram, use distinct arrowheads (open, closed, solid, hollow) and lines (solid, dotted, dashed, double).

SOAP and REST are defined in Section 4.3.3. In previous versions of the SOAP specification, SOAP was an acronym, but this is no longer the case. See www.w3.org/TR/soap12-part1/#intro.

Suppose that we know that C1 calls C2. Sometimes we feel tempted to also show a data flow between the two. We could use the preceding figure and assume the arrow indicates data flow (instead of "calls"), but if C2 returns a value to C1, shouldn't an arrow go both ways? Or should a single arrow have two arrowheads? These two options are not interchangeable. A double-headed arrow typically denotes a symmetric relationship between two elements, whereas two single-headed arrows suggest two asymmetric relationships at work. In either case, the diagram will lose the information that C1 initiated the interaction. Suppose that C2 also invokes C1. Would we need to put *two* double-headed arrows between C1 and C2? When a component C1 calls a component C2, C1 may pass data as arguments to C2 and C2 may return data back to C1. Therefore, it's often a better idea to use the arrow to indicate the call's relation rather than data flow; otherwise the diagram may easily end up full of double-headed arrows that don't tell much.

Although arrows are often used to indicate interactions, often one can avoid confusion by not using them where they are likely to be misinterpreted. For example, one can use lines without arrowheads. Sometimes physical placement, rather than lines, can convey the same information. For example, a layer A on top of a layer B indicates that modules in A can use modules in B. Nesting one element inside another often means "is part of."

Finally, a good key is essential for understanding the meaning of arrows, even ones that represent "simple" interactions such as "calls." A useful arrow, suitably explained in the key, will leave no doubt as to which is the calling end and which is the called end of a call-return connector, and which way the data flows.

—D.G. and P.M.

Rule 4: Use a Standard Organization

Establish a standard, planned organization scheme, make your documents adhere to it, and ensure that readers know about it. A standard organization, also called a template, offers many benefits.

- It helps the reader navigate the document and find specific information quickly. Thus, this benefit is also related to the write-for-the-reader rule.
- It also helps the document writer plan and organize the contents. The writer doesn't have to start with a blank page when answering the question "What topics and in what order should I have in this document?" The template already provides an outline of the important topics to cover.
- It allows the writer to record information as soon as it's known. For example, pieces of section 4 may be written before sections 1–3 are there.
- It reveals what work remains to be done by the number of sections labeled "TBD" (to be determined) or "To Do."
- It embodies completeness rules for the information; the sections of the document constitute the set of important aspects that need to be conveyed. Hence, the standard organization can form the basis for a first-order validation check of the document at review time.

Corollaries to this rule are these:

1. *Organize documentation for ease of reference.* Software documentation may be read from cover to cover at most once, probably never. But a document is likely to be referenced hundreds or thousands of times. Do what you can to make it easy to find information quickly. Adding a table of contents,

Section I.2, in the introduction to Part I, contains a standard organization for a style guide. Sections 10.1 and 10.2 contain a standard organization that we recommend for documenting views and information beyond views. Chapter 7 contains a standard organization for the documentation of a software interface.

Take any long explanations of figures that are in the main text and move these to the figures' captions. In-text explanations would serve first-time readers well, but putting explanations in captions will serve second-time readers better: When they see a figure they're looking for they won't have to go search the text for its explanation.

—Instructions to the editors of this book, explaining one way in which we tried to organize the book for ease of reference an index, a glossary, and an acronym list are all good ways to help readers look up specific information.

Don't leave sections blank. Mark them as "not applicable" or "to be determined," as appropriate. Better: "Not applicable because [reason]" and "To be determined by [date or milestone]."

2. *Don't leave any section blank; mark as "TBD" what you don't yet know or "NA" what you know is not applicable.* Many times, we can't fill in a document completely because we don't yet know the information, or because decisions have not been made, or because we didn't yet have time to do it. In that case, mark the document accordingly (for example, "TBD" or "To Do"). Templates are by nature generic and hence comprehensive. If a given section of the template does not apply for the document you're creating, mark it as "NA." If the section is blank, the reader will wonder whether the information is coming later or whether it is indeed supposed to be blank. Thus this advice is related to the rule about avoiding ambiguity.

Rule 5: Record Rationale

"Well, it's an idea, and even a bad idea is better than none," said Master Li. "Error can point the way to truth, while empty-headedness can only lead to more empty-headedness or to a career in politics."

—Barry Hughart, *Bridge of Birds* (1984)

Architecture is the result of making a set of important design decisions, and architecture documentation records the outcomes of those decisions. For the most important decisions, you should record why you made them the way you did. You should also record the important or most likely alternatives you rejected and state why. Later, when those decisions come under scrutiny or pressure to change, you will find yourself revisiting the same arguments and wondering why you didn't take another path. Recording your rationale will save you enormous time in the long run, although it requires discipline to record your rationale in the heat of the moment.

Section 6.5 discusses the documentation of rationale.

Of course, not every single design decision should have the rationale captured in the architecture documentation. If a design decision is key to achieve a quality requirement of the system, its rationale is probably worth capturing. If a design decision required a long meeting with stakeholders, that's a good decision to capture. If you conducted technical experiments and studies or created prototypes to evaluate design alternatives, the conclusions of this effort should be captured as rationale for the chosen alternative. Keep in mind that one week, one month, or one year from now, you may not remember why you did things that way, and other people will not know either.

Rule 6: Keep Documentation Current but Not Too Current

Documentation that is incomplete or out of date does not reflect truth, does not obey its own rules for form and internal consistency, and is not used. Documentation that is kept current and accurate is used. Why? Because questions about the software can be most easily and most efficiently answered by referring to the appropriate document. Documentation that is somehow inadequate to answer the question needs to be fixed. Updating it and *then* referring the questioner to it will deliver a strong message that the documentation is the final, authoritative source for information.

During the design process, on the other hand, decisions are made and reconsidered with great frequency. Revising documentation to reflect decisions that will not persist is an unnecessary expense.

Your development plan should specify particular points at which the documentation is brought up to date or the process for keeping the documentation current. For example, the end of each iteration or sprint, or each incremental release, could be associated with providing revised documentation. Every design decision should not be recorded and distributed the instant it is made; rather, the document should be subject to version control and have a release strategy, just as every other artifact does.

Even with the best intentions, sometimes budget and schedule preclude conscientious updating of an architecture document as the system undergoes change. In that case, as happens all too often, the code becomes the final source of authority. Try to use the formula in Section P.2.4 to justify maintaining the document by making a case that doing so is worth the investment. If that fails, then at least mark the sections of the document that are out of date so that readers can still have confidence in the remainder.

Rule 7: Review Documentation for Fitness of Purpose

Only the intended users of a document will be able to tell you whether it contains the right information presented in the right way. Enlist their aid. Before a document is released, have it reviewed by representatives of the community or communities for which it was written.

Chapter 11 covers reviewing architecture documents.

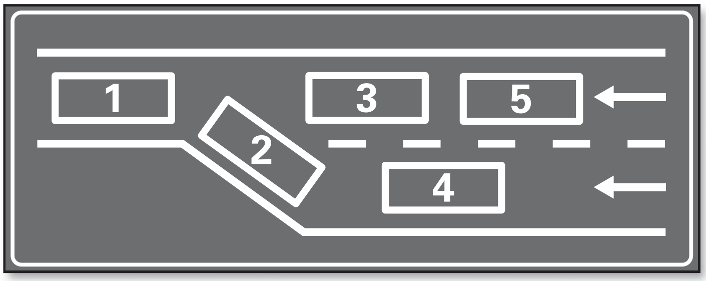
P.6 Summary Checklist

- The goal of documenting an architecture is to write it down so that others can successfully use it, maintain it, and build a system from it.
- Documentation exists to further architecture's uses as a means of education, as a vehicle for communication among stakeholders, and as the basis for analysis.
- Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.
- Documentation should pay for itself by making development activities less costly.
- Module styles help architects think about their software as a set of implementation units. C&C views help architects think about their software as a set of elements that have runtime behavior and interactions. Allocation views help architects think about how their software relates to the nonsoftware structures in its environment.

- An *architecture style* is a specialization of elements and relations, together with a set of constraints on how they can be used. A style defines a family of architectures that satisfy the constraints.
- Some styles are applicable in every software system. Other styles occur only in systems in which they were explicitly chosen and designed in by the architect.
- Follow the seven rules for sound documentation.
  1. Write documentation from the point of view of the reader, not the writer.
  2. Avoid unnecessary repetition.
  3. Avoid ambiguity. Always explain your notation.
  4. Use a standard organization.
  5. Record rationale.
  6. Keep documentation current but not too current.
  7. Review documentation for fitness of purpose.

P.7 Discussion Questions

1. Think of a technical document that you remember as being exceptionally useful. What made it so?
2. Think of a technical document that you remember as being dreadful. What made it so?
3. List several architectural aspects of a system you're familiar with, and state why they are. List several aspects that are not architectural, and state why they are not. List several aspects that are "on the cusp," and make a compelling argument for putting each into "architectural" or "nonarchitectural" categories.
4. If you visit Seoul, Korea, you might see the following sign presiding over one of the busy downtown thoroughfares:



What does it mean? Is the information this sign conveys structural, behavioral, or both? What are the elements in this system? Are they more like modules or like components? What qualities about the notation make this sign understandable or not understandable? Does the sign convey a dynamic architecture, or dynamic behavior within a static architecture? Who are the stakeholders of this sign? What quality attributes is it attempting to achieve? How would you validate it, to assure yourself that it was satisfying its requirements?

5. How much of a project's budget would you devote to software architecture documentation? Why? How would you measure the cost and the benefit?

P.8 For Further Reading

The full treatment of software architecture—how to build one, how to evaluate one to make sure it's a good one, how to recover one from a jumble of legacy code, and how to drive a development effort once you have one—is beyond the scope of this book. However, general books on software architecture are plentiful. Several authors provide good coverage: Bass, Clements, and Kazman (2003); Hofmeister, Nord, and Soni (2000); Shaw and Garlan (1996); Bosch (2000); and Gorton (2006). Also, Jeff Garland and Richard Anthony's *Large-Scale Software Architecture: A Practical Guide Using UML* is a good resource (Garland and Anthony 2003).

The Software Engineering Institute's software architecture Web page—at www.sei.cmu.edu/architecture—provides a wide variety of software architecture resources and links, including a broad collection of definitions of the term (SEI 2010).

One of the goals of documentation is to provide sufficient information so that an architecture can be analyzed for fitness of purpose. For more about analysis and evaluation of software architectures, see the book by Clements, Kazman, and Klein (2002).

The seven rules of sound documentation are adapted from a paper by Parnas and Clements (1986), which also espouses a philosophy directly relevant to this book. That paper holds that although system design is almost always subject to errors, false starts, and resource-constrained compromises, systems should be documented as though they were the product of an idealized, step-by-step, smoothly executed design process. That is the documentation that will be the most helpful in the long run. This book is consistent with that philosophy, in that it lays out what the end state of your documentation should be.

If you want a deeper appreciation of the field of architecture and its roots, then diving into some of the early papers will be worth your time:

David Parnas (1974) first made the observation that software can be described by many structures, not just one. This insight led directly to the concept of views that we use today. Architecture views in general, and "4+1 views" in particular, are a fundamental aspect of the Rational (now IBM Rational) Unified Process for object-oriented software (Kruchten 1995).

An early paper on software architecture that tied us to building architecture and our "architecture styles" to the architecture styles of buildings is by Perry and Wolf (1992).

A tour de force in style comparison is found in the paper by Shaw (1995), in which the author examines 11 different previously published solutions to the automobile cruise-control problem and compares each solution through the lens of architecture style. Chapter 3 of the book by Shaw and Garlan (1996) continues the theme. A number of example problems are presented. For each one, several architecture solutions are presented, each based on the choice of a different style. These side-by-side comparisons not only reveal qualities of the styles themselves, but also richly illustrate the overall concept.

For encyclopedic catalogs of architecture patterns, see the *Pattern-Oriented Software Architecture* series of books by the following authors: Buschmann et al. (1996); Schmidt et al. (2000); Kircher and Jain (2004); and Buschmann, Henney, and Schmidt (2007a and 2007b). Also see Martin Fowler's book *Patterns of Enterprise Application Architecture* (2002).

Smith and Williams (2002) include three chapters of principles and guidance for architecting systems in which performance is an overriding concern.

PART I

A Collection of Software Architecture Styles

The starting point of architecture design is most often a preexisting package of design decisions. Very few architects design systems completely by closing their eyes, thinking hard, and conjuring up a brand-new design.

A most useful package of design decisions is the *architecture style*. Chapters 1–5 present a range of important and widely used architecture styles. The emphasis here is on how to document a view that results from the use of a style.

I.1 Three Categories of Styles

Chapters 1–5 are organized along the lines of the three categories of styles we discussed in the prologue: module styles (Chapters 1 and 2), component-and-connector (C&C) styles (Chapters 3 and 4), and allocation styles (Chapter 5). Plan for your documentation package to include at least one module view, at least one component-and-connector view, and at least one allocation view.

Modules are the primary elements of **module styles**. A module is an implementation unit that provides a coherent set of responsibilities. A module might take the form of a class, a collection of classes, a layer, an aspect, or any decomposition of the implementation unit. Every module has a collection of properties assigned to it. These properties are intended to express the important information associated with the module, as well as constraints on the module. Sample properties are responsibilities, visibility information, and author or owner. The relations that modules have to one another include *is part of*, *depends on*, and *is a*.

A **module style** is a kind of style that introduces a specific set of module types and specifies rules about how elements of those types can be combined.

Module styles are described in Chapters 1 and 2.

A **component-and-connector style** is a kind of style that introduces a specific set of component and connector types and specifies rules about how elements of those types can be combined. Additionally, given that C&C views capture runtime aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style.

C&C styles are described in Chapters 3 and 4.

Component-and-connector styles express runtime behavior. They are described in terms of components and connectors. A component is one of the principal processing units of the executing system. Components might be services, processes, threads, filters, repositories, peers, or clients and servers, to name a few. A connector is the interaction mechanism among components. Connectors include pipes, queues, request/reply protocols, direct invocation, event-driven invocation, and so forth. Components and connectors can be decomposed into other components and connectors. The decomposition of a component may include connectors and vice versa.

Allocation styles describe the mapping of software units to elements of an environment in which the software is developed or executes. The environment might be the hardware, the file systems supporting development or deployment, or the development organization(s).

An **allocation style** is a kind of style that describes the mapping of software units to elements of an environment in which the software is developed or executes.

Allocation styles are described in Chapter 5.

I.2 Style Guides: A Standard Organization for Explaining a Style

Styles presented together for comparison and selection should be described consistently with each other. In this way, an architect can better make an informed decision about which one(s) to use. This is an application of the fourth rule for sound documentation: Use a standard organization. The outline used for describing a style is called a **style guide**.

The styles in Chapters 1–5 are presented using the form of a style guide. Below is the outline for that style guide.

A **style guide** is the description of an architecture style that specifies the vocabulary of design (sets of element and relationship types) and rules (sets of topological and semantic constraints) for how that vocabulary can be used.

OUTLINE FOR A STYLE GUIDE

1. *Overview.* The overview in a style guide explains why this style is useful. It discusses what it is about a system that the style addresses and how it supports reasoning about systems.
2. *Element types, relation types, and properties.*
   a. **Elements** are the architecture building blocks native to the style. A style guide defines one or more element types, instances of which will populate an architecture that uses that style.
   b. **Relations** determine how the elements work together to accomplish the work of the system. A style guide defines one or more relation types that

apply to the style's element types. An architecture using the style will describe the relations (instances of the relation type) that determine how the elements can work together, and any important properties of those relations. The style guide provides rules on how elements can and cannot be related.

3. *Constraints.* This section of the style guide lists the rules for putting the elements and relations together to form a valid instance of the style. For example, in a pipe-and-filter style, a pipe is allowed to attach to a filter, but not to another pipe. In a layered style, the layers are laid out adjacently in a stack, not scattered about randomly. In a work-assignment style, every software unit has to be allocated to at least one organizational element.
4. *What it's for.* This section of the style guide describes the kind of reasoning supported by views in the style. The intent is to help the architect understand to what purpose(s) a view in this style may be put. This might be how using the style helps in the development process (for example, the "uses" style is good for reasoning about modifiability). Or it might be about how the style helps the product (for instance, pipe-and-filter yields good performance when processing a series of data elements).
5. *Notations.* This section of the style guide will give descriptions of graphical and/or textual representations that are available and useful to document views in the style. Different notations will also support the conveyance of different kinds of information in the view.
6. *Relation to other styles.* This section of the style guide describes how views derived from this style might be related to views derived from different styles. For example, views from two different styles might convey different but related information about a system, and the architect would like a way to choose which one to use. This section might also include warnings about other views with which a particular view is often confused, to the detriment of the system and its stakeholders. (Layers and tiers are a good example of this. They are fundamentally different, but are often [mis]used interchangeably.)
7. *Examples.* This section provides or points to an example of a documented view derived from the given style.

An **element** is an architecture building block native to the style. An element can be a module, a component or connector, or an element in the environment of the system whose architecture we are documenting. The description of an element tells what role it plays in an architecture, lists its important **properties**, and furnishes guidelines for effective documentation of the element in a view.

A **relation** defines how elements cooperate to accomplish the work of the system. The description of a relation names the relations among elements and provides rules on how elements can and cannot be related.

A **property** contains additional information about elements and relations. A style definition includes the property name and description. When an architect documents a view based on that style, the properties will be given values. Property values are often used to analyze an architecture for its ability to meet quality attribute requirements.

I.3 Choosing Which Element and Relation Properties to Document

The discussion in Chapters 1–5 heavily emphasizes styles, which are documented in published style guides. But as you read about the styles in Part I, remember that the end game is to produce views based on the chosen style. Recall that a view is a representation of a style applied to a particular system—in this case the system whose architecture is being documented.

When documenting a view, decide on the list of properties to document about the elements in that view. Choose properties that will aid the analysis you wish the documentation to support. Documenting a view, then, includes documenting the values for the properties you chose.

One of the tasks in documenting a view is deciding which properties of elements to document. Recall from our preceding discussion of style guides that properties are additional information about the elements and their relations that are useful to document. The styles of Chapters 1–5 are each described with a set of properties likely to be useful; consider them suggestions.

Properties almost always include the name of the element as well as some description of its role or responsibility in the architecture. For example, properties of a layer—an element of the layered style, which is one of the module styles—should include the layer's name, the units of software the layer contains, and the nature of the capabilities that the layer provides. A layered view will then, for each layer, specify its name, the units of software it contains, and the capabilities it provides.

Beyond these basic properties, however, are properties that will support architecture-based analysis. If you want to analyze an architecture for performance, then properties in some views probably should include an element's best- and worst-case response times, or the maximum number of events an element can service per time unit. If you want to analyze an architecture for security, then you probably want to document properties that explain levels of encryption and authorization rules for different elements and relations.

So: If you care about quality attribute *x*, then define properties that will let you analyze for *x* in the views that are related to achieving *x*.

Also as you read Chapters 1–5, remember that a view may represent more than one style. In fact, this is the norm. Since all nontrivial software systems employ many styles at once, mandating that each view come from just one style would result in a plethora of views and a very thick architecture document. Some styles can be fruitfully combined, and that combination used to create a view. Component-and-connector styles in particular tend to combine well, and many architects produce a single component-and-connector view for their system that reflects all of the C&C styles they used.

Section 6.6 discusses which styles go together well to produce combined views.

By learning the "pure" (uncombined) styles, however, you can make more informed choices about which ones to combine. Each comes with its own vocabulary (of element and relation types); you can use these vocabularies to build meaningful combined views that carry forward the pedigree of each of their constituent styles.

I.4 Notations for Architecture Views

Notations for documenting views differ considerably in their degree of formality. Roughly speaking, there are three main categories of notation:

1. *Informal notations.* Views are depicted (often graphically) using general-purpose diagramming and editing tools and visual conventions chosen for the system at hand. The semantics of the description are characterized in natural language and cannot be formally analyzed.
2. *Semiformal notations.* Views are expressed in a standardized notation that prescribes graphical elements and rules of construction, but does not provide a complete semantic treatment of the meaning of those elements. Rudimentary analysis can be applied to determine if a description satisfies syntactic properties. Unified Modeling Language (UML) is a semiformal notation in this sense.
3. *Formal notations.* Views are described in a notation that has a precise (usually mathematically based) semantics. Formal analysis of both syntax and semantics is possible. There are a variety of formal notations for software architecture available, although none of them can be said to be in widespread use. Generally referred to as **architecture description languages** (ADLs), they typically provide both a graphical vocabulary and an underlying semantics for architecture representation. In some cases these notations are specialized to particular styles. In others they allow many styles, or even provide the ability to formally define new styles. The usefulness of ADLs lies in their ability to support automation through associated tools—automation to provide useful analysis of the architecture, or automation to assist in code generation.

Determining which form of notation to use involves making several trade-offs. Typically more-formal notations take more time and effort to create, but they repay this effort in reduced ambiguity and better opportunities for analysis. Conversely, more-informal notations are easier to create, but they provide fewer guarantees.

Think carefully about the choice of design notation for each diagram in your architecture documentation. Consider available tool support, the knowledge and the needs of the documentation stakeholders, and the purpose of the diagrams (for example, implementation guidance, analysis, or model and code generation). Some architecture information can be documented more effectively with other notations.

An **architecture description language** is a language for representing a software and/or system architecture. ADLs are usually graphical languages that provide semantics that enable analysis and reasoning about architectures, often using associated tools.

Appendix C describes one particular architecture description language, called AADL, in depth. The "For Further Reading" section of Chapter 3 provides resources for learning about other ADLs.

We'll see examples of views rendered in these different kinds of notations throughout Part I.

There is no greater impediment to the advancement of knowledge than the ambiguity of words.

—Thomas Reid, Scottish philosopher

I.5 Examples

Throughout this book, but especially in Part I, we will present many examples of architecture documentation fragments extracted from real systems. When you look at these examples, please keep in mind the following notes:

- The goal is for you to understand the kinds of information the example conveys and how the chosen notation is used to depict different types of elements and relations.
- The goal is usually not for you to understand the meaning of the specific elements and relations, that is, the responsibilities they satisfy. Any software system uses acronyms and internal jargon that become part of the vocabulary of the stakeholders familiar with that system. The examples in the book should allow you to recognize what information the architect wanted to capture without knowing the meaning of these terms.
- For each example, the piece extracted from the original architecture documentation is typically just a diagram. To that diagram we add a brief description with information that can't really be inferred by the diagram alone. This information comes from other parts of each system's architecture documentation that are not reproduced in the book. A diagram is not enough to document a view!
- We chose diagrams that we think are good examples of different styles and notations. However, they may not be perfect with respect to notation choice and usage, diagramming aesthetics, and quality of the design itself.
- Very often architecture diagrams do not show a single style in its pure form. In many of our examples, you will be able to find vestiges of styles other than the one the diagram is illustrating. That's normal.
- The example does not necessarily show the latest version of the design.

1 Module Views

In this chapter, we look at these aspects of module views:

- Elements, relations, and properties
- Purpose
- Notation
- Relation to other views

1.1 Overview

In this chapter and the next, we look at ways to document the module structures of a system's software. Such documentation enumerates the principal implementation units, or modules, of a system, together with the relations among these units. We refer to these descriptions as *module views.* As we will see, these views can be used for each of the purposes outlined in the prologue: education, communication among stakeholders, and the basis for construction and analysis.

The way in which a system's software is decomposed into manageable units remains one of the important forms of system structure. At a minimum, it determines how a system's source code is decomposed into units, what kinds of assumptions each unit can make about services provided by other units, and how those units are aggregated into larger ensembles. It also includes global data structures that impact and are impacted by multiple units. Module structures often determine how changes to one part of a system might affect other parts and hence the ability of a system to support modifiability, portability, and reuse.

It is unlikely that the documentation of any software architecture can be complete without at least one module view.

The architect must be a prophet . . . a prophet in the true sense of the term . . . if he can't see at least ten years ahead don't call him an architect.

—Frank Lloyd Wright

Table 1.1 Summary of the module views

Elements	Modules, which are implementation units of software that provide a coherent set of responsibilities.
Relations	• *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole. • *Depends on*, which defines a dependency relationship between two modules. Specific module styles elaborate what dependency is meant. • *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.
Constraints	Different module views may impose specific topological constraints.
What It's For	• Providing a blueprint for construction of the code • Facilitating impact analysis • Planning incremental development • Supporting requirements traceability analysis • Explaining the functionality of the system and the structure of the code base • Supporting the definition of work assignments, implementation schedules, and budget information • Showing the structure of information to be persisted

We begin by considering module views in the general form. Table 1.1 summarizes the discussion in the following sections about the elements, relations, constraints, and purpose of the module views. In Chapter 2 we provide this information specific to each of a number of often used module styles.

1.2 Elements, Relations, and Properties of Module Views

1.2.1 Elements

A **module** is an implementation unit of software that provides a coherent set of responsibilities.

A **responsibility** is a general statement about an architecture element and what it is expected to contribute to the architecture. This includes the actions that it performs, the knowledge it maintains, the decisions it makes, or the role it plays in achieving the system's overall quality attributes or functionality.

System designers use the term **module** to refer to a wide variety of software structures, including programming language units—such as C programs, Java or C# classes, Delphi units, and PL/SQL stored procedures—or simply general groupings of source code units—such as Java packages or C# namespaces. In this book, we adopt a much broader definition.

We characterize a module by enumerating its set of **responsibilities**, which are foremost among a module's properties. This broad notion of responsibilities is meant to encompass the kinds of features that a unit of software might provide: that is, its functionality and the knowledge it maintains.

Modules can be aggregated and decomposed. Each of the various module styles identifies a different set of modules and relations, and then aggregates or decomposes these modules based on relevant style criteria. For example, the layered style identifies modules and aggregates them based on an *allowed-to-use* relation, whereas the generalization style identifies and aggregates modules based on what they have in common.

1.2.2 Relations

Module views have the following types of relations:

- *Is part of.* The *is-part-of* relation defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole. In its most general form, the *is-part-of* relation simply indicates aggregation, with little implied semantics.

In Chapter 2, the *is-part-of* relation is refined to a decomposition relation in the decomposition style.

- *Depends on.* A *depends on* B defines a dependency relation between A and B. Many different specific forms of dependency can be used in module views. Later, we look at four in particular: *uses, allowed to use, crosscuts,* and data entity *relationships,* in the module uses, layered, aspect, and data model styles, respectively. The logical association between classes (in a UML class diagram, for example) also depicts a dependency between the classes.

In Chapter 2, the *depends-on* relation is refined to "uses" in the uses style, "allowed to use" in the layered style, and "crosscut" in the aspect style.

- *Is a.* The *is-a* relation defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent. The child is able to be used in contexts in which the parent is used. Later, we look at this relation in more detail in the generalization style. Object-oriented inheritance and interface realization are special cases of the *is-a* relation.

In Chapter 2, the *is-a* relation is refined to generalization in the generalization style.

1.2.3 Properties

Properties of modules that help to guide implementation or are input to analysis should be recorded as part of the supporting documentation for a module view. The list of properties may vary but is likely to include the following:

- *Name.* A module's name is, of course, the primary means to refer to it. A module's name often suggests something about its role in the system: a module called "account_mgr," for instance, probably has little to do with numeric simulations of chemical reactions. In addition, a module's name may reflect its position in a decomposition hierarchy; the name "A.B.C," for example, refers to a module C that is a submodule of a module B, itself a submodule of A.
- *Responsibility.* The responsibility property of a module is a way to identify its role in the overall system and establishes an identity for it beyond the name. Whereas a module's name may suggest its role, a statement of responsibility

establishes it with much more certainty. Responsibilities should be described in sufficient detail to make clear to the reader what each module does.

Documenting software interfaces is discussed in Chapter 7.

- *Visibility of interface(s).* When a module has submodules, some interfaces of the submodules may have internal purposes; that is, the interfaces are used only by the submodules within the enclosing parent module. These interfaces are not visible outside that context and therefore do not have a direct relationship to the parent interfaces. Different strategies can be used for those interfaces that have a direct relationship to the parent interfaces. The strategy shown in Figure 1.1(a) is encapsulation. The parent module provides its own interfaces and maps all requests to the capabilities provided by the submodules. The facilities of the enclosed modules are not available outside the parent. Alternatively, the interfaces of an aggregate module can be a subset of the interfaces of its submodules. The aggregate module selectively exposes some of the interfaces of the submodules. Layers and subsystems are often defined in this way. For example, if module C is an aggregate of modules A and B, C's implicit interface will be a subset of the interfaces of modules A and B (see Figure 1.1(b)).
- *Implementation information.* Because modules are units of implementation, it is useful to record information related to their implementation from the point of view of managing their development and building the system that contains them. Although this information is not, strictly speaking, architectural, it may be useful to record it in the architec-

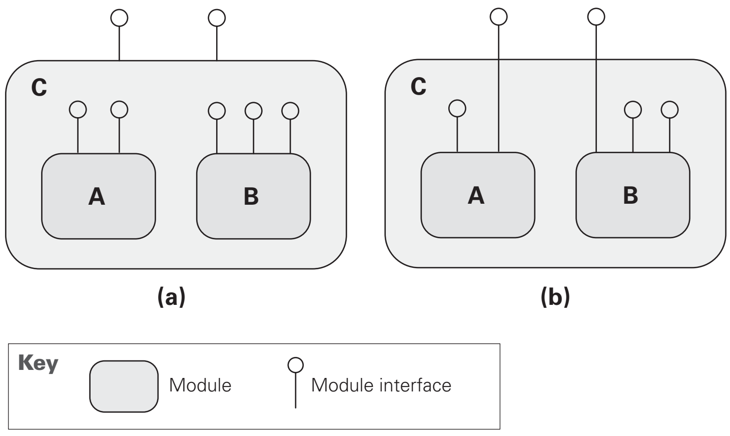


Figure 1.1
(a) Module C provides its own interface, hiding the interfaces of modules A and B. (b) Module C exposes a subset of the interfaces of modules A and B as its interface.

ture documentation where the module is defined. Implementation information might include

– *Mapping to source code units.* This identifies the files that constitute the implementation of a module. For example, a module named Account, if implemented in Java, might have several files that constitute its implementation: IAccount.java (an interface), AccountImpl.java (an implementation of Account functionality), AccountBean.java (a class to hold the state of an account in memory), AccountOrmMapping.xml (a file that defines the mapping between AccountBean and a database table—object-relational mapping), and perhaps even a unit test AccountTest.java.
– *Test information.* The module's test plan, test cases, test scaffolding, and test data are important to store.
– *Management information.* A manager may need information about the module's predicted schedule and budget.
– *Implementation constraints.* In many cases, the architect will have a certain implementation strategy in mind for a module or may know of constraints that the implementation must follow. This information is private to the module and hence will not appear, for example, in the module's interface.

In addition to identifying the implementation units, one also needs to identify where they reside in a project's filing scheme: a directory or folder in a file system, a URL in an intranet, or a location in a configuration management system's storage space. This information is in the purview of the implementation style, discussed in Section 5.5.

Module styles may have properties of their own in addition to these. Also, you may find other properties useful that are not listed.

1.3 What Module Views Are For

Expect to use module views for

- *Construction.* A module view can provide a blueprint for the source code and the data store. In this case, the modules and physical structures, such as source code files and directories, often have a close mapping.
- *Analysis.* Two important analysis techniques are requirements traceability and impact analysis. Because modules partition the system, it should be possible to determine how the functional requirements of a system are supported by module responsibilities. Some functional requirements will be met by a sequence of invocations among modules. Documenting such sequences shows how the system is meeting its requirements and identifies any unaddressed requirements. Impact analysis, by contrast, helps to predict the effect of modifying the system. Module views that show dependencies among modules or layers provide a good basis for

Section 10.3 discusses documenting the mapping between requirements and architecture.

impact analysis. Modules are modified as a result of problem reports or change requests. Impact analysis requires a certain degree of design completeness and integrity of the module description. In particular, dependency information has to be available and correct in order to create useful results.

- *Communication.* A module view can be used to explain the system's functionality to someone not familiar with the system. The various levels of granularity of the module decomposition provide a top-down presentation of the system's responsibilities and therefore can guide the learning process. For a system whose implementation is already in place, module views, if kept up to date, are very helpful, as they explain the structure of the code base to a new developer on the team—much more effective than providing the URL to the version management system repository and asking him or her to browse the source files and read the code. Thus, up-to-date module views are very useful during system maintenance.

Expect to use component-and-connector and allocation views, not module views, to analyze performance, reliability, and other runtime qualities.

On the other hand, it is difficult to use the module views to make inferences about runtime behavior, because these views are just a static partition of the functions of the software. Thus, a module view is not typically used for analysis of performance, reliability, or many other runtime qualities. For those, we typically rely on component-and-connector and allocation views.

1.4 Notations for Module Views

1.4.1 Informal Notations

A number of notations can be used to present a module view. One common informal notation uses boxes to represent the modules, with different kinds of lines between them representing the relations. Nesting is used to depict aggregation, and arrows typically represent a *depends-on* relation. In Figure 1.1 (in Section 1.2.3), for example, nesting represents aggregation, and lollipops indicate interfaces.

Figure 2.4 is an example of a textual notation for modules, using indentation to indicate *is part of*.

A second common form of informal notation is a simple textual listing of the modules with descriptions of the responsibilities. Various textual schemes can be used to represent the *is-part-of* relation, such as indentation, outline numbering, and parenthetical nesting. Other relations may be indicated by keywords. For example, the description of module A might include the line "Imports modules B, C," indicating a dependency between module A and modules B and C.

1.4.2 Unified Modeling Language

Software modeling notations, such as UML, provide a variety of constructs that can be used to represent modules. Figure 1.2 shows some examples for modules using UML notation. Figure 1.3 shows how the three basic relations native to module views are denoted using UML.

UML has a class construct, which is the object-oriented specialization of a module as described here. UML packages are used to represent an aggregation of modules. UML packages

Appendix A describes how UML can be used to show different module views, as well as C&C and allocation styles.

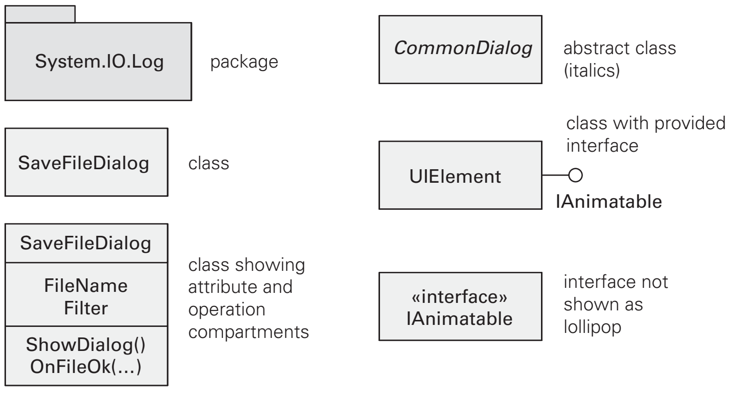


Figure 1.2
Examples of module notation in UML. A module may be represented as a class or a package. More specific types of modules can be indicated with stereotypes (as in Figure 1.4).

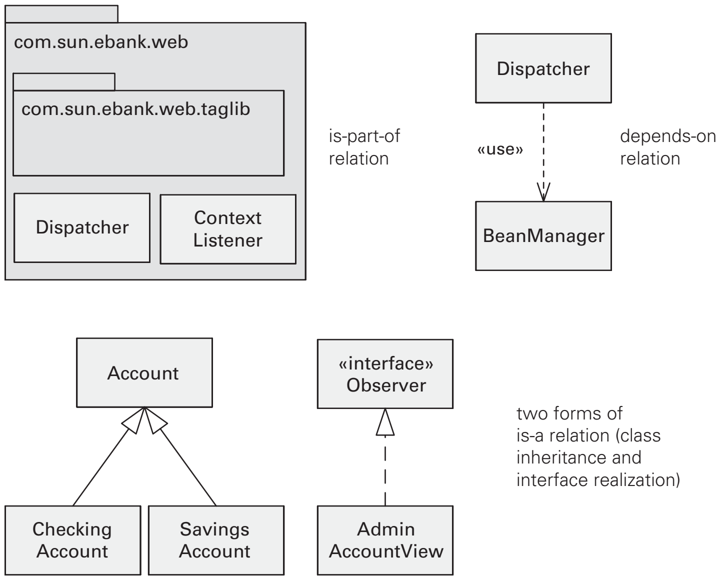


Figure 1.3
Examples of module relations in UML

can represent, for example, layers, subsystems, and collections of implementation units that live together in the implementation namespace.

UML was originally created to model object-oriented systems. It is now considered a general-purpose modeling language. As a result, UML elements and relations are generic; that is, they are not specific to implementation technologies or platforms. But you can define stereotypes to specialize the UML symbols. A **stereotype** is a UML extension mechanism and is represented in diagrams as a label in guillemets («stereotype label»). Figure 1.4 shows some examples. If used correctly, stereotypes make your UML diagrams more expressive. The UML specification provides a number of standard stereotypes, but you can also create your own.

Stereotype is a UML extension mechanism that allows the definition of a new type of modeling element or relation based on an existing UML element or relation.

Try to become familiar with UML standard stereotypes, as well as other stereotypes commonly used in your organization.

1.4.3 Dependency Structure Matrix

A **dependency structure matrix** (DSM) is a table that shows modules as the column and row headings and dependencies as the table cells. The DSM is built as a square matrix (that is, a matrix with same number of rows and columns) where element *ij* is nonzero if there is a dependency between module *i* and module *j* in the architecture.

Some tools that create DSMs can automatically interchange between class diagrams or box-and-line diagrams and DSMs. DSM-based tools are more commonly used for architecture management and enforcement for systems that are already implemented—the DSM is obtained by reverse-engineering the code.

A **dependency structure matrix** is a table that shows modules as the row and column headers; a cell is nonzero if and only if there is a dependency between the row's module and the column's module.

Section 2.2.4 has examples and more information about DSMs.

1.4.4 Entity-Relationship Diagram

An entity-relationship diagram (ERD) is a notation specifically used for data modeling. It shows data entities that require a representation in the system and their relationships. These relationships can be one-to-one, one-to-many, or many-to-many.

Section 2.6.4 has examples and more information about ERDs.

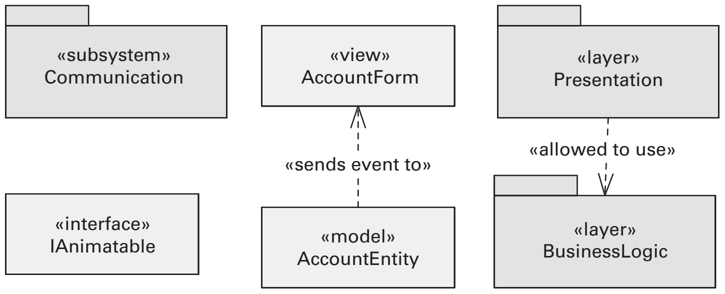


Figure 1.4 Examples of UML elements and relations with stereotypes

1.5 Relation to Other Views

Module views are commonly mapped to component-and-connector views. The implementation units shown in module views have a mapping to components that execute at runtime. Sometimes, the mapping is quite straightforward, even one-to-one. More often, a single module will be replicated as part of many runtime components and a given component could map to several modules.

Components are discussed at length in Section 3.2.

Module views also provide the software elements that are mapped to the diverse nonsoftware elements of the system environment in the various allocation views.

Allocation views are described in Chapter 5.

A common problem is the overloading of module views with information pertaining to other views. This can be quite useful when done in a disciplined fashion but can also lead to confusion. For example, showing a remote procedure call connection in a module view is implicitly introducing the "connector" concept from a component-and-connector view. The module views are often confused with views that demonstrate runtime relations. A module view represents a static partitioning of the software implementation units; therefore, multiple instances of objects—data repositories and networks, for example—are not shown in this view.

1.6 Summary Checklist

- Modules pertain to the way in which a system's software is decomposed into manageable units of responsibilities, which is one of the important forms of system structure.
- Modules are related to one another by forms of *is-part-of*, *depends-on*, and *is-a* relations.
- A module view provides a blueprint for the source code and the data model.
- Expect to have at least one module view in your documentation package.
- You should not depend on a module name to define the functional duties of the module: use the responsibility property.
- Document module interface(s) to establish a module's role in the system.
- Module views are commonly mapped to component-and-connector views. In general, a module may participate in many runtime components.

1.7 Discussion Questions

1. What is it possible and not possible to say about data flow by looking at a module view? What about control flow? What can you say about which modules interact with which other modules?
2. Which properties of a module might you think of as worthy of having special notational conventions to express them, and why? For example, you might want to color a commercial-off-the-shelf module differently from modules developed in-house.
3. The *depends-on* relation among modules is very general. What specific types of dependencies might be reflected in a module view?
4. A primary property of a module is its set of responsibilities. How do a module's responsibilities differ from the requirements that it must satisfy?
5. When documenting a particular system, you might wish to combine modules into an aggregate, to market them as a combined package, for example. Would this package itself be a module? That is, are all aggregates of modules themselves modules?
6. Would you show libraries or frameworks on which your system depends as modules in your module views?

1.8 For Further Reading

DeRemer and Kron (1976) describe programming-in-the-small languages for writing modules and a "module interconnection language" for knitting those modules together. Prieto-Diaz and Neighbors (1986) present a survey of module interconnection languages that are specifically designed to support module interconnection, and they include brief descriptions of some software development systems that support module interconnection.

The chapter on the Module Architecture View in the book by Hofmeister, Nord, and Soni (2000) describes a view of a system in terms of modules and layers and how to represent them in UML.

2 A Tour of Some Module Styles

In this chapter, we look at six important module styles:

- The *decomposition style*, used to show the structure of modules and submodules (that is, containment relations among modules)
- The *uses style*, used to indicate functional dependency relations among modules
- The *generalization style*, used to indicate specialization relations among modules
- The *layered style*, used to describe the *allowed-to-use* relation in a restricted fashion between groups of modules called layers
- The *aspects style*, used to describe particular modules called aspects that are responsible for crosscutting concerns
- The *data model style*, used to show the relations among data entities

2.1 Decomposition Style

2.1.1 Overview

By taking the elements and the properties of module views and focusing on the *is-part-of* relation, we get the decomposition style. A decomposition view describes the organization of the code as modules and submodules and shows how system responsibilities are partitioned across them. Almost all architects begin with the decomposition style. Architects tend to attack a problem with divide-and-conquer techniques, and a decomposition view records their campaign.

The criteria used for decomposing a module into smaller modules include:

- *Achievement of certain quality attributes.* For example, to support modifiability, the information-hiding design principle calls for encapsulating changeable aspects of a system in separate modules, so that the impact of any one change is localized.
- *Build-versus-buy decisions.* Some modules may be bought in the commercial marketplace, reused intact from a previous project, or obtained as open-source software. These modules already have a set of responsibilities implemented. The remaining responsibilities then must be decomposed around those established modules.
- *Product line implementation.* To support the efficient implementation of products of a product family, it is essential to distinguish between common modules, used in every or most products, and variable modules, which differ across products.
- *Team allocation.* To allow implementation of different responsibilities in parallel, separate modules that can be allocated to different teams should be defined. The skills of developers also influence the decomposition. For example, if specialized Web developers are available, modules that handle the Web UI should be kept separate.

A useful design heuristic holds that a module is small enough if it could be discarded and begun again if the programmer(s) assigned to implement it left the project.

A decomposition view may represent the first pass at a detailed architecture design; the architect may subsequently introduce other types of relations and module specializations. The decomposition view defines the modules that may appear in uses, layered, generalization, and other module-based views.

2.1.2 Elements, Relations, and Properties

See "Coming to Terms: Subsystem" on page 73, in this chapter.

Table 2.1 summarizes the characteristics of the decomposition style. Elements of the decomposition style are modules, as described in Section 1.2. Some modules that aggregate other modules can be called *subsystems*. The principal relation, the *decomposition* relation, is a form of the *is-part-of* relation and has as its primary constraint the guarantee that an element can be a part of at most one aggregate.

The element catalog of an architecture view provides various information about the elements in that view. Element catalogs are described in Section 10.1.

The module decomposition may define whether the submodules are visible within only the aggregate module—the parent—or also to other modules. The visibility of submodules can be described in the view's element catalog or conveyed graphically, for example by showing interface lollipops inside or outside the aggregate module, as in Figure 1.1.

Table 2.1 Summary of the decomposition style

Overview	The decomposition style is used for decomposing a system into units of implementation. A decomposition view describes the organization of the code as modules and submodules and shows how system responsibilities are partitioned across them.
Elements	*Module*
Relations	*Decomposition* relation, which is a form of the *is-part-of* relation. The documentation should specify the criteria used to define the decomposition.
Constraints	• No loops are allowed in the *decomposition* graph. • A module can have only one parent.
What It's For	• To reason about and communicate to newcomers the structure of software in digestible chunks • To provide input for work assignment • To reason about localization of changes

2.1.3 What the Decomposition Style Is For

A decomposition view presents the responsibilities of a system in intellectually manageable pieces that are refined to convey more and more details. Therefore, this style is well suited to support the learning process about a system. Besides the obvious benefit for the architect to support the design work, this style is an excellent learning and navigation tool for newcomers to the project and other people who do not necessarily have the whole functional structure of the system memorized. The grouping of responsibilities shown in this style also builds a useful basis for defining configuration items within a configuration management framework.

Refinement is covered in Section 6.1.

A decomposition view most often serves as the input for the work assignment view of a system, which maps parts of a software system onto the organizational units, or teams, that will be implementing and testing them. A decomposition view also provides some support for analyzing effects of changes, but because this view does not show all the dependencies among modules, you cannot expect to do a complete impact analysis. Here, views that elaborate the dependency relations more thoroughly, such as the uses style described later, are required.

The work assignment style is presented in Section 5.4.

2.1.4 Notations for the Decomposition Style

Informal Notations

In informal notations, modules in the decomposition style are usually depicted as named boxes that contain other named boxes. Decomposition may also be shown by listing the module names and using indentation to indicate *is part of*, as in Figure 2.4 (in Section 2.1.6).

The nesting notation can use a thick border suggesting opaqueness—and explained in the key—indicating that children are not visible outside the parent. If a visual notation is not available for indicating visibility, it can be defined textually, as is done for other properties.

UML

In UML, the package construct can be used to represent modules that contain other modules. A package can contain classes and other packages; the class box is normally used for the leaves of the decomposition.

In UML, decomposition is depicted in one of two ways:

1. Modules may be nested, as in Figure 2.1.
2. A succession of two diagrams can be shown, with the second a depiction of the contents of a module shown in the first. Figures 2.2 and 2.3 (in Section 2.1.6) illustrate this approach.

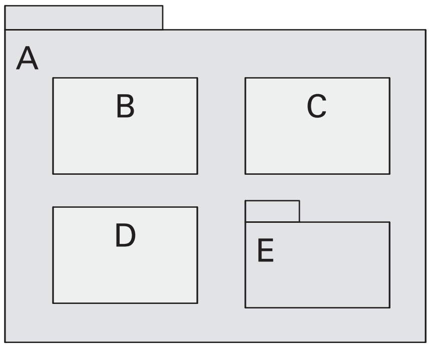
Other properties, such as the modules' responsibilities, are given textually, perhaps using an annotation. Stereotypes can provide additional information for the type of the module.

2.1.5 Relation to Other Styles

Section 3.5 also discusses the mapping between modules and components. Documenting the mapping is described in Section 10.2.

It is possible, and often desirable, to map between a decomposition view and one or more component-and-connector views. For now, it is sufficient to say that the point of providing such a mapping is to indicate how the software implementation structures map onto runtime structures: generally, a many-to-many relationship. The same module might implement all or parts of several components or connectors. Conversely, one component might require several modules for its implementation.

Figure 2.1
In UML, module decomposition is shown by nesting, with the aggregate module shown as a package.



The decomposition style is closely related to the work assignment style, a kind of allocation style. The work assignment style maps modules resulting from a decomposition to a set of teams responsible for implementing and testing those modules.

The work assignment style is described in Section 5.4.

2.1.6 Examples Using the Decomposition Style

Adventure Builder

The example software architecture document that accompanies this book online contains an example of a decomposition view for the Adventure Builder (2010) system. See wiki.sei.cmu.edu/sad.

The ATIA-M System

Army Training Information Architecture-Migrated (ATIA-M) is a large Web-based, Java EE application that supports training in the U.S. Army. It has "thick clients": Windows desktop applications developed using .NET (C#) that communicate with the server-side Java EE components using Web services technology.

Figure 2.2 shows the top-level module decomposition for the entire ATIA-M system, itself a module. The code is divided into three large modules:

Figure 2.2 is the first of many examples of architecture documentation fragments from real systems. When examining these examples, keep in mind the considerations stated in Section I.5, in the introduction to Part I. The descriptions of the elements we provide cannot be derived from the figures; rather, they rely on additional documentation that would accompany the diagrams in an architecture document.

- *Windowsapps* contains the code of the thick clients. The three submodules correspond to Training and Doctrine Development Tool (TDDT), Unit Training Management Configuration (UTMC), and a separate submodule with common code used by the different Windows applications. TDDT and UTMC were the two Windows applications originally planned, but others could be added.
- *ATIA server-side Web modules* contains all non-Java modules that would be deployed to server machines. The Web modules include JavaServer Pages (JSP) files, JavaScript and HTML code, and applets.
- *ATIA server-side Java modules* contains all Java source code in ATIA that would run on application servers. This module does not include JSP, JavaScript, HTML, applet, or thick-client code.

The decomposition of Windowsapps into three submodules is shown in Figure 2.2. The decomposition of ATIA server-side Java modules, on the other hand, was captured in another module view diagram, shown in Figure 2.3.

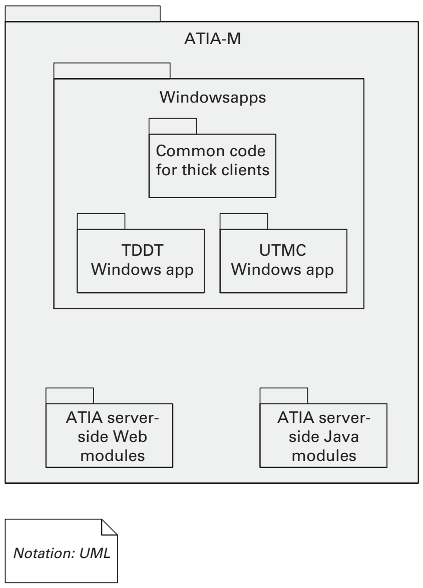

ATIA-M
Windowsapps
Common code for thick clients
TDDT Windows app
UTMC Windows app
ATIA server-side Web modules
ATIA server-side Java modules
Notation: UML

Figure 2.2
Top-level decomposition view for the ATIA system

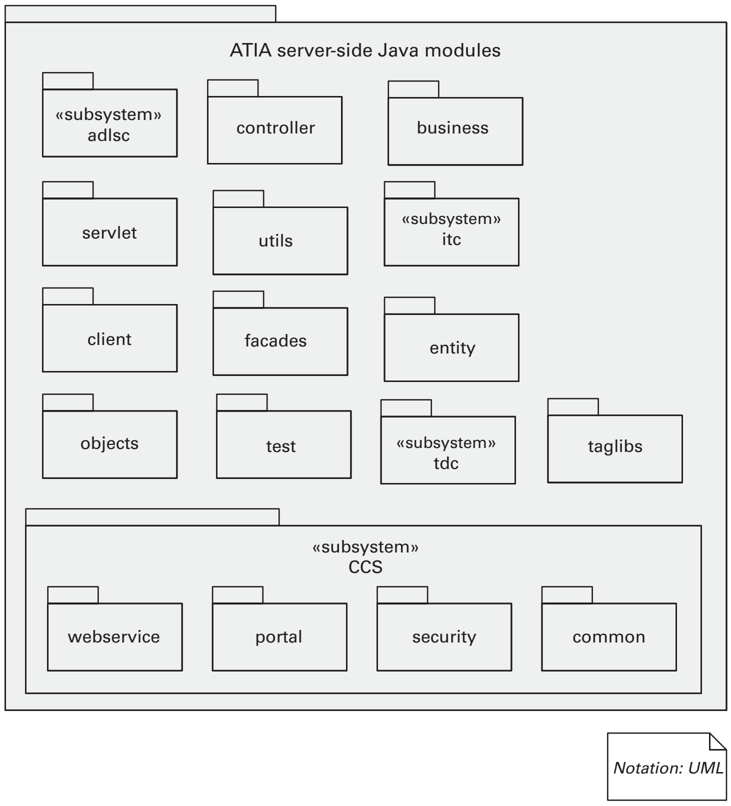


Figure 2.3
Refinement of ATIA-M server-side Java modules showing how it is further decomposed into submodules

A-7E Avionics System

An example of the decomposition style comes from the A-7E avionics software system described in Chapter 3 of the book by Bass, Clements, and Kazman (2003). Figure 2.4 shows the primary presentation part of the view. The figure names the elements and shows the *is-part-of* relation among them for the A-7E system. The *decomposition* relation is conveyed by indentation.

The primary presentation is the (typically) graphical portion of an architecture view, as described in Chapter 10.

In this example, the criterion for decomposition is the information-hiding principle, which holds that there should be a module to encapsulate responsibilities likely to change together. A module's responsibilities, then, are described in terms of the information-hiding secrets it encapsulates.

This diagram shows that in A-7E, the first-order decomposition produced three modules: Hardware Hiding, Behavior Hiding, and Software Decision Hiding. Each of these modules is decomposed into two to six submodules, which are in turn decomposed, and so forth, until the granularity is fine enough to be manageable.

The A-7E decomposition view documentation describes the responsibilities of the three highest-level modules in the element catalog as follows:

- *Hardware Hiding Module:* The Hardware Hiding Module includes the procedures that need to be changed if any part of the hardware is replaced by a new unit with a different hardware/software interface but with the same general capabilities. This module implements "virtual hardware" or an abstract device that is used by the rest of the software. The primary secrets of this module are the hardware/software interfaces. The secondary secrets of this module are the data structures and algorithms used to implement the virtual hardware.
- *Behavior Hiding Module:* The Behavior Hiding Module includes procedures that need to be changed if there are changes in requirements affecting the required behavior. Those requirements are the primary secret of this module. These procedures determine the values to be sent to the virtual output devices provided by the Hardware Hiding Module.
- *Software Decision Hiding Module:* The Software Decision Hiding Module hides software design decisions that are based upon mathematical theorems, physical facts, and programming considerations such as algorithmic efficiency and accuracy. The secrets of this module are not described in the requirements document. This module differs from the other modules in that both the secrets and the interfaces are determined by software designers. Changes in these

Hardware Hiding Module
- Extended Computer Module
  - Data Module
  - Input/Output Module
  - Computer State Module
  - Parallelism Control Module
  - Program Module
  - Virtual Memory Module
  - Interrupt Handler Module
  - Timer Module
- Device Interface Module
  - Air Data Computer Module
  - Angle of Attack Sensor Module
  - Audible Signal Device Module
  - Computer Fail Device Module
  - Doppler Radar Set Module
  - Flight Information Displays Module
  - Forward Looking Radar Module
  - Head-Up Display Module
  - Inertial Measurement Set Module
  - Input-Output Representation Module
  - Master Function Switch Module
  - Panel Module
  - Projected Map Display Set Module
  - Radar Altimeter Module
  - Shipboard Inertial Nav System Module
  - Slew Control Module
  - Switch Bank Module
  - TACAN Module
  - Visual Indicators Module
  - Waypoint Info. System Module
  - Weapon Characteristics Module
  - Weapon Release System Module
  - Weight on Gear Module

Behavior Hiding Module
- Function Driver Module
  - Air Data Computer Module
  - Audible Signal Module
  - Computer Fail Signal Module
  - Doppler Radar Module
  - Flight Information Display Module
  - Forward Looking Radar Module
  - Head-Up Display Module
  - Inertial Measurement Set Module
  - Panel Module
  - Projected Map Display Set Module
  - Shipboard Inertial Nav System Module
  - Visual Indicator Module
  - Weapon Release Module
  - Ground Test Module
  - Shared Services Module
- Mode Determination Module
  - Panel I/O Support Module
  - Shared Subroutine Module
  - Stage Director Module
  - System Value Module

Software Decision Hiding Module
- Application Data Type Module
  - Numeric Data Type Module
  - State Transition Event Module
- Data Banker Module
  - Singular Values Module
  - Complex Event Module
  - Filter Behavior Module
- Physical Models Module
  - Aircraft Motion Module
  - Earth Characteristics Module
  - Human Factors Module
  - Target Behavior Module
  - Weapon Behavior Module
- Software Utility Module
  - Power-Up Initialization Module
  - Numerical Algorithms Module
- System Generation Module
  - System Generation Parameter Module
  - Support Software Module

Figure 2.4
The decomposition of the A-7E software architecture results in three top-level modules (Hardware Hiding, Behavior Hiding, and Software Decision Hiding) and *is-part-of* relations (Bass, Clements, and Kazman 2003, p. 59). In this presentation, *is part of* is indicated by textual indentation.

modules are more likely to be motivated by a desire to improve performance or accuracy than by externally imposed changes.

The A-7E decomposition view documentation then goes on to describe the second-level modules.

In the case of the A-7E architecture, the second-level module structure was enshrined in many ways: Design documentation, configuration-controlled files, test plans, programming teams, review procedures, and project schedule and milestones all were pegged to this second-level module structure as their unit of reference.

If you use a module decomposition structure to organize your project, you will find it useful to focus on a specific level of the hierarchy as your organizing motif, chosen based on a manageable granularity.

COMING TO TERMS

Subsystem

When documenting a module view of a system, you may choose to identify certain aggregated modules as subsystems. A subsystem can be pretty much anything you want it to be, but it often describes a part of a system that (1) carries out a functionally cohesive subset of the overall system's mission, (2) can be executed independently, and (3) can be developed and deployed incrementally. The software system of a Mars exploratory robot, for example, may be divided into subsystems responsible for:

- Communication
- Motion
- Power management
- Navigation
- Monitoring its own health and status

Not just any portion of a system is a subsystem. In our exploratory robot example, a math utility library is certainly a portion of a system and an aggregation of modules and even has coherent functionality. But the library is unlikely to be called a subsystem, because it lacks the ability to operate independently to do work that's recognizably part of the overall system's purpose.

Subsystems do not partition a system into completely separate parts, because some parts are used in more than one subsystem. For example, suppose that the exploratory robot system has the layered design shown in Figure 2.5. In this case, a subsystem consists of one segment from the top layer, as well as any segments of any lower layers that it needs in order to carry out its responsibilities. A subset of the system formed in this way is often called a *slice*, or a *vertical slice*.

The "more or less independent" nature of a subsystem makes it ideal for dividing up a project's work. You may, for example, ask an analyst to examine the performance of a subsystem. A subsystem can often be fielded and accomplish useful

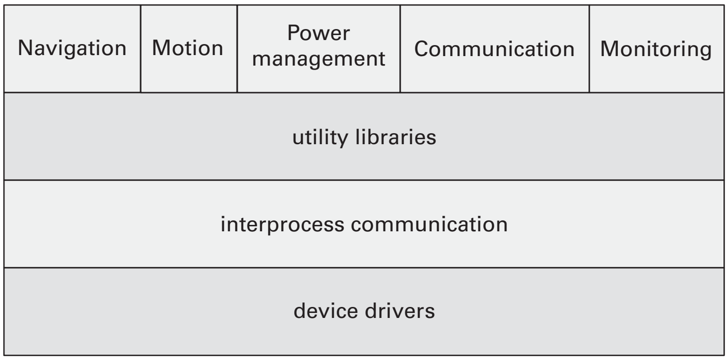


Figure 2.5
Layered design of a hypothetical exploratory robot system

work before the whole system is complete. A subsystem makes a convenient package to hand off to a team or a subcontractor to implement. The fact that it executes more or less independently allows that team to work more or less independently even through testing.

In the UML world, <<subsystem>> is a stereotype of component. It represents a large-scale component that embodies other components. According to the UML 2.2 specification, a subsystem is:

> A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly. Definitions of subsystems vary widely among domains and methods, and it is expected that domain and method profiles will specialize this construct.

In previous versions of UML, <<subsystem>> was a stereotype of package and still today it is common to find packages with that stereotype in UML diagrams. Regardless of the notation used, a subsystem can represent a group of modules (implementation units) or a group of components with runtime presence.

You may decide to identify subsystems in your design. If you do, make sure that your rationale explains why you chose the ones you did.

2.2 Uses Style

2.2.1 Overview

Uses is a form of dependency that can exist between two modules. A *uses* B if the correctness of A depends on the presence of a correct implementation of B.

The uses style results when the *depends-on* relation is specialized to ***uses***. A module *uses* another module if its correctness depends on the correctness of the other. Whereas the module decomposition style shows only the organization of the implementation units as modules and submodules, a uses style goes one step further to reveal which modules use which other modules. This style tells developers what other modules must exist for their portion of the system to work correctly. This style enables incremental development and the deployment of useful subsets of full systems.

2.2.2 Elements, Relations, and Properties

Table 2.2 summarizes the characteristics of the uses style. The elements of this style are the modules as described in Section 1.2. We define a specialization of the *depends-on* relation to be the *uses* relation, whereby one module requires the correct implementation of another module for its own correct functioning. This view makes explicit which modules use which other modules to achieve their responsibilities.

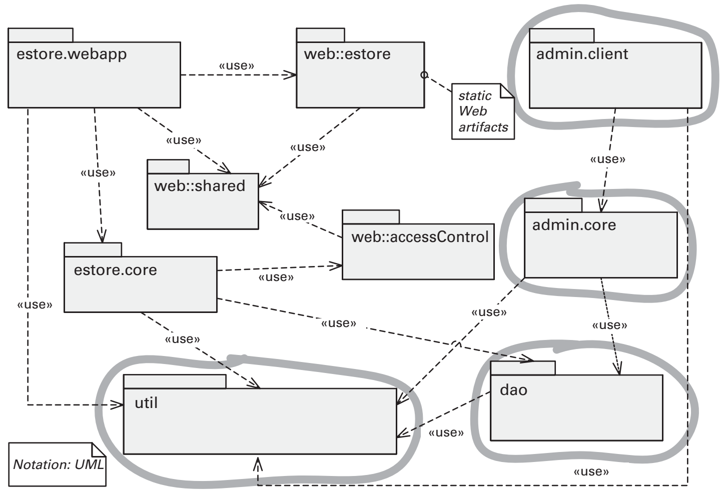
2.2.3 What the Uses Style Is For

This style is useful for planning incremental development, system extensions and subsets, debugging and testing, and gauging the effects of specific changes. Figure 2.6 shows the primary presentation of a uses view and how it can help with incremental development. To define incremental subsets, modules should be defined at the right level of granularity. In the example, `admin.core` may not need the entire dao package, only a submodule of it; the diagram should then show the submodules of `dao`.

Table 2.2 Summary of the uses style

Overview	The uses style shows how modules depend on each other; it is helpful for planning because it helps define subsets and increments of the system being developed.
Elements	*Module*
Relations	The *uses* relation, which is a form of the *depends-on* relation. Module A *uses* module B if A *depends on* the presence of a correctly functioning B to satisfy its own requirements.
Constraints	The uses style has no topological constraints. However, if *uses* relations present loops, broad fan-out, or long dependency chains, the ability of the architecture to be delivered in incremental subsets will be impaired.
What It's For	• Planning incremental development and subsets • Debugging and testing • Gauging the effect of changes

Figure 2.6
In this uses view, suppose the incremental development plan called for module admin.client in the next release. Based on the *uses* relation, the diagram highlights what other modules need to be present: admin.core, dao, and util.



See "Coming to Terms: Uses" on page 81, in this chapter, for more about loops in the *uses* relation.

The uses view also helps in managing the dependencies of a system that is being built or maintained. The goal of this task is to keep complexity under control and avoid degradation in the modifiability of the system due to the addition of undesirable dependencies.

2.2.4 Notations for the Uses Style

Informal Notations

The *uses* relation can be documented as a two-column table, with using elements on the left and the elements they use listed on the right. Alternatively, informal graphical notations can show the relation by using the standard box-and-line diagram with a key. For defining subsets, a tabular—that is, nongraphical—notation is sometimes a better alternative. It is easier to look up the detailed relations in a table than to find them in a diagram, which can rapidly grow too cluttered to be helpful unless the diagram is partitioned using decomposition refinement.

Decomposition refinement is discussed in Section 6.1.

Semiformal Notations

UML
The uses style is easily represented in UML. UML packages can be used to represent modules; the *uses* relation is depicted as a dependency with the stereotype <<use>>. In Figure 2.7(a), the User Interface module has a *uses* dependency on the Data Access module.

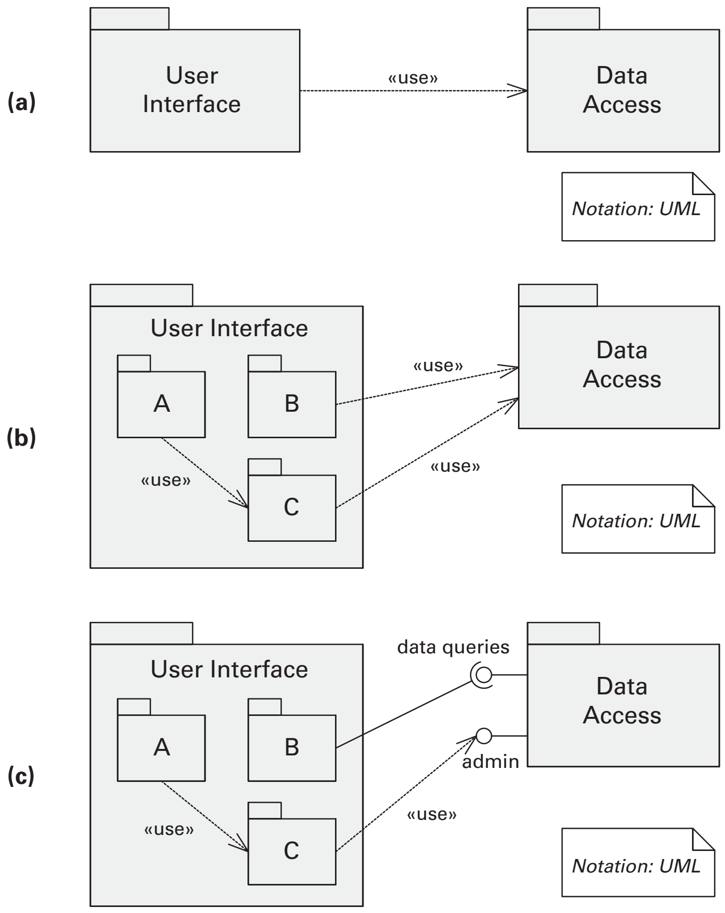


Figure 2.7
(a) The User Interface module is an aggregate module with a *uses* dependency on the Data Access module. We use UML package notation to represent modules and the specialized form of *depends-on* arrow to indicate a *uses* relation.

(b) Here is a variation of Figure 2.7(a) in which the User Interface module has been decomposed into modules A, B, and C. At least one of the modules must depend on the Data Access module or the decomposition would not be consistent.

(c) In UML we can represent the *uses* relations and also show interfaces explicitly. This version shows that the Data Access module has two interfaces, which are used by modules B and C, respectively. Both the socket lollipop connection and the <<use>> dependency connected to the lollipop indicate *uses* relations.

Dependency Structure Matrix

The *uses* relation can be documented as a square matrix, with the modules listed as rows and columns. A mark in the i^{th} column and j^{th} row indicates that module i uses module j. This simple representation has evolved and been used in automated tools to create dependency structure matrices (DSMs).

A diagram like the UML package diagram in Figure 2.8 can be seen as a directed graph; the packages are the vertices and the dependencies are the edges. A DSM is the matrix representation of a directed graph. The cell corresponding to column i and row j is nonzero if there is an edge from vertex i to vertex j in the graph (that is, module i uses module j). Figure 2.9 shows the DSM for the UML diagram in Figure 2.8.

DSMs need a key too! In the key, say whether a value in row *i* and column *j* means that module *i* depends on module *j* or module *j* depends on module *i*. Both alternatives are possible.

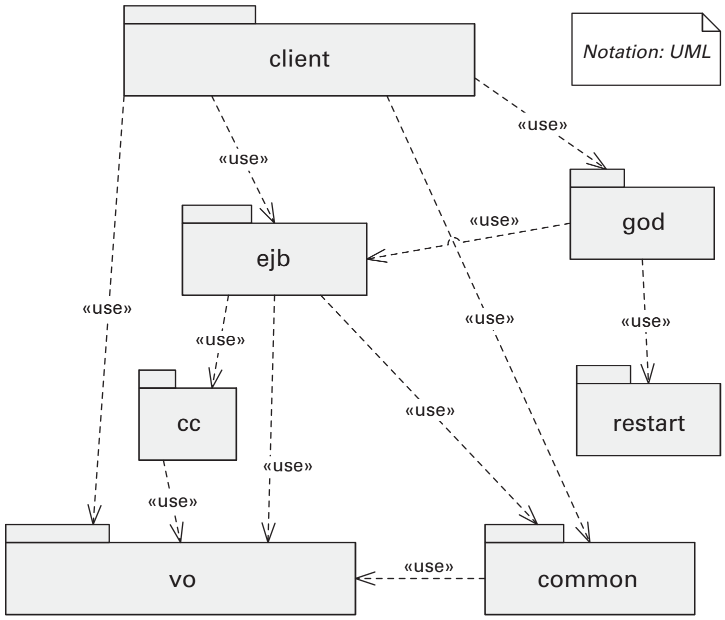


Figure 2.8
UML package diagram showing <<uses>> dependencies

Figure 2.9
DSM for the UML diagram in Figure 2.8

used module \ using module	client	ejb	cc	god	restart	common	vo
client	0	0	0	0	0	0	0
ejb	**1**	0	0	**1**	0	0	0
cc	0	**1**	0	0	0	0	0
god	**1**	0	0	0	0	0	0
restart	0	0	0	**1**	0	0	0
common	**1**	**1**	0	0	0	0	0
vo	**1**	**1**	**1**	0	0	**1**	0

Key: "1" means module in column uses module in row

2.2.5 Relation to Other Styles

The uses style also goes hand in hand with the layered style, with its *allowed-to-use* relation. An *allowed-to-use* relation usually comes first and contains coarse-grained directives defining the degrees of freedom for implementers. Once implementation choices have been made, the uses view emerges and governs the production of incremental subsets.

When a module contains submodules, the decomposition requires that any *uses* relation involving the aggregate module be mapped to a submodule using that relation. In Figure 2.7(b), the User Interface module is decomposed into modules A, B, and C. At least one of the modules must depend on the Data Access module; otherwise, the decomposition is not consistent.

A uses view can also show interfaces explicitly. In Figure 2.7(c), the Data Access module has two interfaces, which are used by modules B and C, respectively.

Chapter 7 has more information about interfaces.

2.2.6 Examples Showing the Uses Style

Adventure Builder

The example software architecture document accompanying this book online contains an example of a uses view for the Adventure Builder (2010) system. See wiki.sei.cmu.edu/sad.

The ATIA-M System

Figure 2.10 shows the diagram from a top-level uses view for the ATIA-M system (it also shows decomposition). In the architecture documentation, it could have superseded the decomposition view (see Figure 2.2) for the same system.

ECS

EOSDIS Core System (ECS) is a NASA system. A constellation of satellites collect measurements about Earth and send the data to ground stations. ECS controls spacecraft and instruments, processes data, and produces refined data that are stored in several distributed data centers and made available to scientists around the world. Figure 2.11 is a small excerpt of a uses view's primary presentation from the ECS system. The notation is textual, using the tabular format mentioned earlier. Like most primary presentations, this one names only the elements; they are defined in the view's supporting documentation (not shown here).

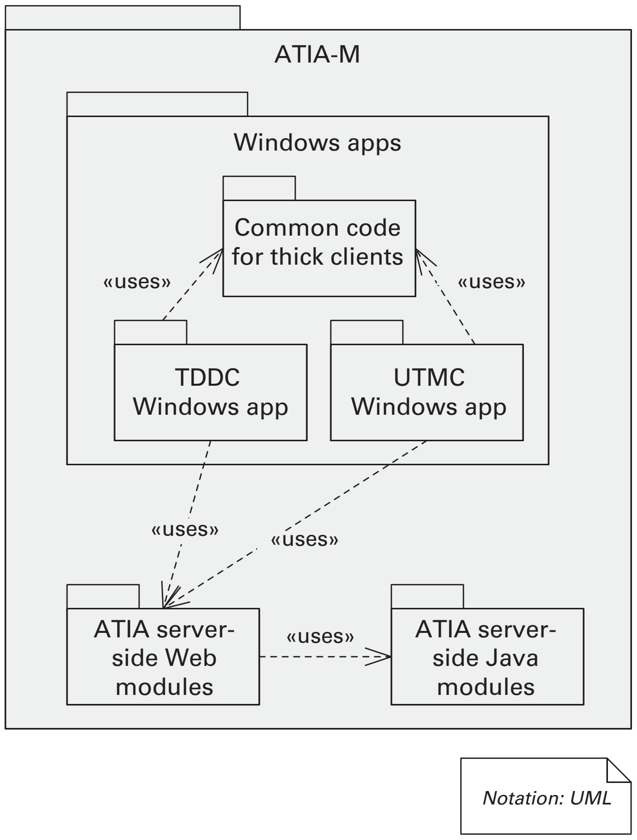


Figure 2.10
Top-level uses view for the ATIA-M system

Element	Uses This Element
Science Data Processing Segment	
Ingest Subsystem	
INGST CSCI	ADSRV CSCI in the Interoperability Subsystem
	STMGT CSCI in the Data Server Subsystem
	SDSRV CSCI in the Data Server Subsystem
	DCCI CSCI in the Communications Subsystem
(Continue for other CSCIs within the Ingest Subsystem)	
Data Server Subsystem	
DDIST CSCI	MCI CSCI in the System Management Subsystem
	DCCI CSCI in the Communications Subsystem
	STMGT CSCI in the Data Server Subsystem
	INGST CSCI in the Ingest Subsystem
(Continue for other CSCIs within the Data Server Subsystem)	
(Continue for other subsystems within the Science Data Processing Segment)	
(Continue for other ECS segments)	

Figure 2.11
Excerpt of the ECS system uses view, documented as a table. The left column mirrors the system's module decomposition structure.

COMING TO TERMS

Uses

Two of the module styles that we present in this book—the uses style and the layered style—are based on one of the most underutilized relations in software engineering: *uses*. The *uses* relation is a form of the *depends-on* relation. A unit of software P_1 is said to use another unit P_2 if P_1's correctness depends on a correct implementation of P_2 being present.

The *uses* relation resembles, but is decidedly not, the simple *calls* relation provided by most programming languages. Here's why.

- A program P_1 can use program P_2 without calling it. P_1 may assume, for example, that P_2 has left a shared device in a usable state when it finished with it. Or P_1 may expect P_2 to leave a computed result that it needs in a shared variable. Or P_1 may be a process that sleeps until P_2 signals an event to awaken it.
- A program P_1 might call program P_2 but not use it. If P_2 is an exception handler that was passed as a parameter[1] for P_1 to call when it detects an error, P_1 will usually not care what P_2 does. P_1 does not use P_2 because its own correctness does not depend on P_2.

So *uses* is not *calls* or *invokes*. Likewise, *uses* is different from other *depends-on* relations, such as *includes*, which deals with compilation dependencies but need not influence runtime correctness.

Because the *uses* relation takes many forms, a uses view usually cannot be automatically derived from other architecture views nor extracted from source code. To enjoy its benefits, the architect must engineer the relations and document the uses view explicitly.

The careful engineering of the *uses* relation imparts a powerful capability to a development team: It enables the building of small subsets of a total system. Early in the project, this allows incremental development, a development paradigm that allows early prototyping, early integration, and early testing. At every step along the way, the system carries out part of its total functionality, even if far from everything, and does it correctly. Fred Brooks (1995) writes about the "electrifying effect" on team morale when the system first succeeds at doing something. Absent incremental development, nothing works until everything works, and we are reduced to the waterfall model of development. Subsets of the total system are also useful beyond development. They provide a safe fallback in the event of slipped schedules: It is much better for the project manager to offer the customer a working subset of the system at delivery time rather than apologies and promises.

1. Or perhaps it calls a program whose name was bound by a parameter at system-generation time or a program whose name it looks up via a name server. Many schemes are possible.

Here's how it works. Choose a program that is to be in a subset; call it P_1. In order for P_1 to work correctly in this subset, correct implementations of the programs it uses must also be present. So include them in the subset. For them to work correctly, their used programs must also be present, and so forth. The subset consists of the transitive closure of P_1's uses.[2] Conceptually, you pluck P_1 out from the uses graph and then see what programs come dangling beneath it. There's your subset.

Loops in the relation—that is, for example, where P_1 uses P_2, P_2 uses P_3, and P_3 uses P_1—are the enemy of simple subsets. A large uses loop necessitates bringing in a large number of programs—every member of the loop—into any subset joined by any member. "Bringing in a program" means, of course, that it must be implemented, debugged, integrated, and tested. But the point of incremental development is that you'd like to bring in a small number of programs to each new increment, and you'd like to be able to choose which ones you bring in and not have them choose themselves. Generally speaking, any long list of used programs (caused by long dependency chains or broad fan-out in the relation) detracts from the ability to field small increments. They also decrease modifiability, because a change to a module could very well ripple into modules that it uses.

Besides managing subsets, the *uses* relation is also a helpful tool for debugging and integration testing. If you discover a program that's producing incorrect results, the problem is going to be either in the program itself or in the programs that it uses. The *uses* relation lets you instantly narrow the list of suspects. In a similar way, you can employ the relation to help you gauge the effects of proposed changes. If a program's external behavior changes as the result of a planned modification, you can backtrack through the *uses* relation to see what other programs may be affected by that modification.

2.3 Generalization Style

2.3.1 Overview

Even though this style shares the terms *parent* and *child* with the decomposition style, they are used differently. In decomposition, a parent consists of its children. In generalization, parents and children have things in common.

The generalization style results when the *is-a* relation is employed. This style is useful when an architect wants to support extension and evolution of architectures and individual elements. Modules in this style are defined in such a way that they capture commonalities and variations. When modules have a generalization relationship, the parent module is a more general version of the child modules. (The parent module owns the commonalities, and the variations are manifested in the children.) Extensions can be made by adding, removing, or chang-

2. Of course, *calls* and other *depends-on* relations must be given their due. If a program in the subset calls, includes, or inherits from another program but doesn't use it, the compiler is still going to expect that program to be present. But if it isn't *used*, there need not be a correct implementation of it: a simple stub, possibly returning a pro forma result, will do just fine.

ing children; a change to the parent will automatically change all the children that inherit from it, which could support evolution if the change is appropriate for all the children.

Generalization may represent inheritance of either interface, implementation, or both. Within an architecture description, the emphasis is on sharing and reusing interfaces and not so much on implementations.

2.3.2 Elements, Relations, and Properties

Table 2.3 summarizes the characteristics of the generalization style. The element of the generalization style is the module; the relation is generalization, which is the *is-a* relation defined in Section 1.2. In this relation, one module is a generalization (parent) of other modules (children), and these other modules are specializations of the first.

A module can be abstract. Such a module does not contain a complete implementation. Modules that are children of an abstract module need to provide the necessary implementations or else they should be abstract as well.

A module that inherits information is referred to as a descendant; the module providing the information is an ancestor. Cycles are not allowed. That is, a module cannot be an ancestor or a descendant of itself.

The fact that module A inherits from module B using *interface realization* is a promise that module A complies to interface B. This strategy is useful when variants of a module with different

Table 2.3 Summary of the generalization style

Overview	The generalization style employs the *is-a* relation to support extension and evolution of architectures and individual elements. Modules in this style are defined in such a way that they capture commonalities and variations.
Elements	*Module*. A module can have the "abstract" property to indicate it does not contain a complete implementation.
Relations	*Generalization*, which is a specialization of the *is-a* relation. The relation can be further specialized to indicate, for example, if it is class inheritance, interface inheritance, or interface realization.
Constraints	• A module can have multiple parents, although multiple inheritance is often considered a dangerous design approach. • Cycles in the *generalization* relation are not allowed; that is, a child module cannot be a generalization of one or more of its ancestor modules in a view.
What It's For	• Expressing inheritance in object-oriented designs • Incrementally describing evolution and extension • Capturing commonalities, with variations as children • Supporting reuse

implementations are needed and one implementation of the module can substitute for another implementation with little or no effect on other modules. In object-oriented designs, *class inheritance* indicates that a module inherits behavior from its ancestors and may modify it to achieve its specialized behavior. *Interface inheritance* is also possible when we want a child interface that adds operations to the list of operations defined by the parent interface.

2.3.3 What the Generalization Style Is For

The generalization style can be used to support

- *Object-oriented designs.* The generalization style is the predominant means for expressing an inheritance-based, object-oriented design for a system.
- *Extension.* It is often easier to understand how one module differs from another, well-known module rather than to try to understand a new module from scratch. Thus, generalization is a mechanism for producing incremental descriptions to form a full description of a module.
- *Local change or variation.* One purpose of architecture is to provide a stable global structure that accommodates local change or variation. Generalization is one approach to define commonalities on a higher level and to define variations as children of a module.
- *Reuse.* Finding reusable modules is a by-product of the other purposes. Suitable abstractions can be reused at the interface level alone, or the implementation can be included as well. The definition of abstract modules creates an opportunity for reuse.

2.3.4 Notations for the Generalization Style

UML

Expressing generalization lies at the heart of UML. Modules are typically shown as classes or interfaces. Figure 2.12 shows the basic notation available in UML for class or interface inheritance. Figure 2.13 shows how UML expresses interface realization.

Chapter 7 discusses how to document interfaces.

2.3.5 Relation to Other Styles

Inheritance and interface realization relationships complement other module relations and are often found in module views along with *uses* relations and package decompositions. But for designs that involve a complex hierarchy of modules, it is useful to show inheritance relationships in a diagram separate from other types of relationships.

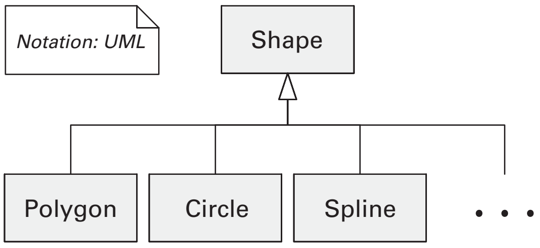


Figure 2.12
In UML, class or interface inheritance is represented by a solid line with a closed, hollow arrowhead. UML allows an ellipsis (. . .) in place of a submodule, indicating that a module can have more children than shown and that additional ones are likely. Module Shape is the parent of modules Polygon, Circle, and Spline, each of which is in turn a subclass, child, or descendant of Shape. Shape is more general; its children are specialized versions. The arrow points toward the more general entity.

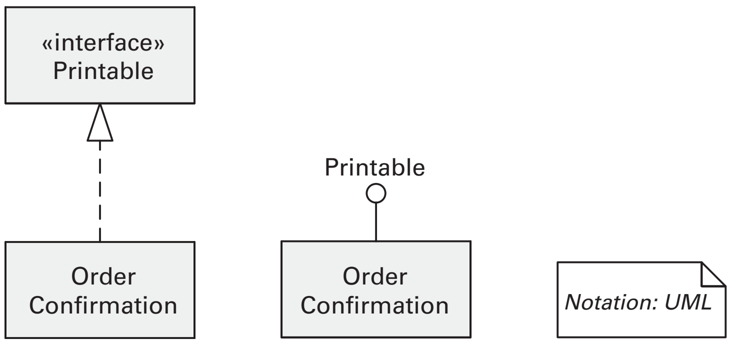


Figure 2.13
Interface realization (sometimes called interface implementation) is also a kind of generalization. It can be expressed in UML in two ways: (1) a dashed line with a closed hollow arrowhead going from the module to the interface it realizes; (2) a lollipop symbol for the interface connected to the module that implements it. Thus the two notations in the figure are equivalent. However, the one on the left is more convenient when multiple modules realize the same interface.

2.3.6 Examples Using the Generalization Style

ArchE

Figure 2.14 shows part of a generalization view from the SEI Architecture Expert (ArchE) tool. This tool allows an architect to create the architecture design for a system based on quality attribute requirements, feature requirements, and preexisting pieces of design. Internally, ArchE uses a rule engine that manipulates data elements called facts. Various operations are performed on any Fact object; other operations are specific to the subclasses of Fact.

PetStore

Figure 2.15 shows part of the generalization view of the Pet-Store application. This is a multi-tier, Web-based application

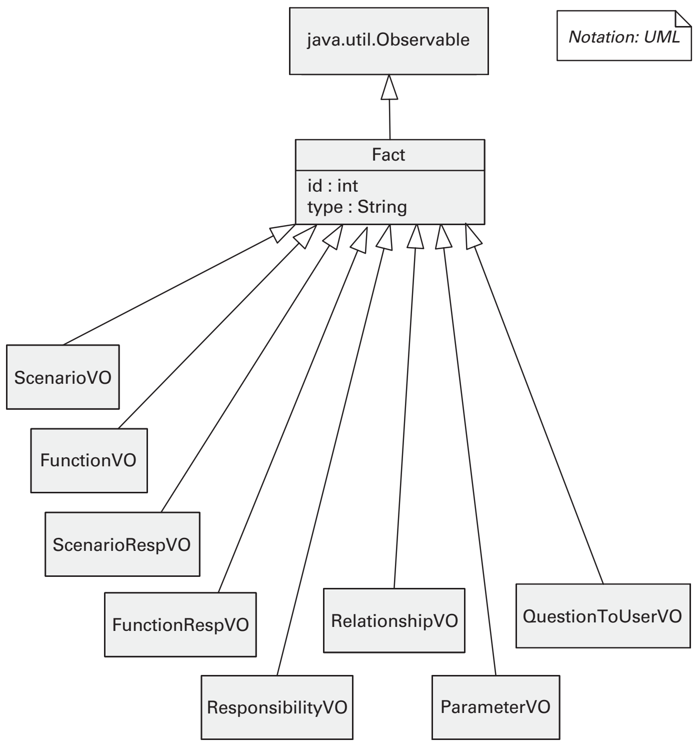


Figure 2.14
The primary presentation for ArchE's generalization view. This system uses internally a rule engine, and many operations are defined on a class called Fact. In addition, specific functionality exists to deal with different kinds of facts and hence the generalization in this figure. The classes shown here also appear in other diagrams, which show the attributes and operations available in each class, as well as *uses* relations among these and other modules that are part of the system.

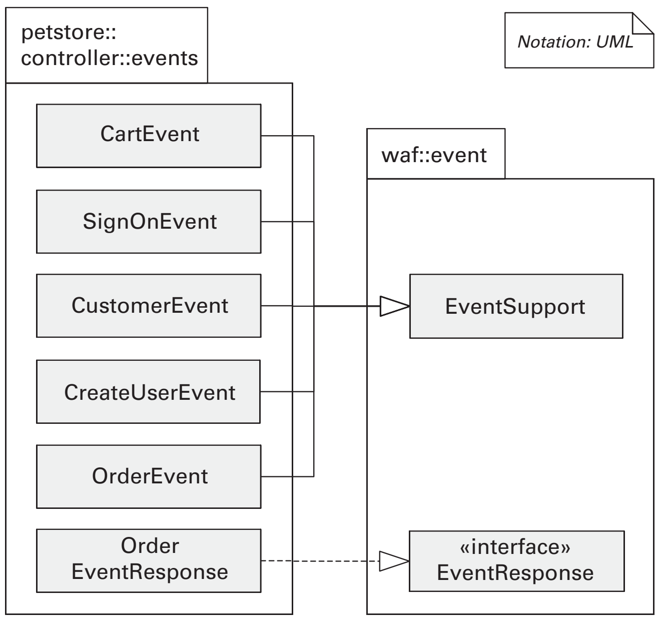


Figure 2.15
Part of the primary presentation of the generalization view for the PetStore application. It shows a hierarchy of classes that represent events in the system, and an interface realization. The package on the right is part of a Web application framework (waf), which offers an event-handling service. An application such as PetStore has to define the application-specific events. The events are used for the interaction of other modules in the system (not shown) following the model-view-controller pattern.

that implements an online pet store. The generalization view shows several important hierarchies in the system (Figure 2.15 shows a subset of them).

2.4 Layered Style

2.4.1 Overview

The layered style, like all module styles, reflects a division of the software into units. In this case, the units are **layers**. Each layer represents a grouping of modules that offers a cohesive set of services. There are constraints on the *allowed-to-use* relationship among the layers: the relations must be unidirectional. The layered view of architecture, shown with a layer diagram, is one of the most commonly used views in software architecture. However, it often is poorly defined, and so often misunderstood. Because true layered systems promote modifiability and portability, architects have an incentive to show their systems as layered, even if they are not.

A **layer** is a grouping of modules that together offer a cohesive set of services to other layers. The layers are related to each other by the strictly ordered relation *allowed to use*.

Layers completely partition a set of software, and each partition—through a public interface—provides a cohesive set of services. But that's not all. Figure 2.16, which is intentionally vague about what the units are and how they interact, shows three divisions of software—you'll have to take our word that each division provides a cohesive set of services—but none of them constitutes a layering. What's missing?

Layering has one more fundamental property: The layers are created to interact according to a strict ordering relation. Herein lies the conceptual heart of layers. If (A, B) is in this relation, we say that the implementation of layer A is allowed to use any of the public facilities provided by layer B.

Element A **uses** element B if A's correctness depends on a correct implementation of B being present.

By **uses**, we mean the very specific term defined in Section 2.2 for the uses style, but the definition has some loopholes. If A is implemented using the facilities in B, is it implemented using only B? Maybe or maybe not. For example, assume that layers are depicted horizontally, one on top of the other. Some layering schemes allow a layer to use the public facilities of *any* lower layer, not just the nearest lower layer. Other layering

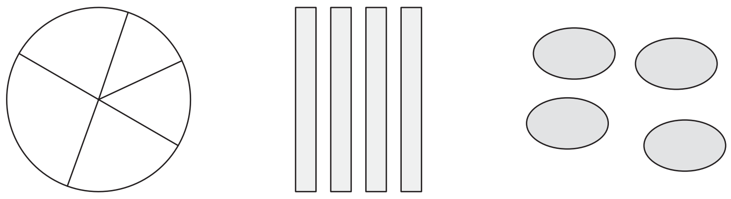


Figure 2.16
Three different divisions of software. Is any of them layered?

schemes have so-called layers that are collections of utilities and can be used by any layer. *But no architecture that can be validly called layered allows a layer to use, without restriction, the facilities of a higher layer.* Allowing unrestricted upward usage destroys the desirable properties that layering brings to an architecture; this will be discussed shortly. Usage in layers generally flows downward. A small number of well-defined special cases may be permitted, but these should be few and regarded as exceptions to the rule. Hence, the architecture in Figure 2.17 *resembles* a layering *but is not.*

Remember that a system with a *uses* relation from a lower layer to a higher layer is not a layered system, strictly speaking.

Figure 2.17 shows why layers have been a source of ambiguity for so long: architects have been calling such diagrams layered when they are not. There is more to layers than the ability to draw separate parts on top of each other.

In some cases, modules in a very high layer might be required to directly use modules in a very low layer where normally only next-lower-layer uses are allowed. The layer diagram or an accompanying document will have to show these exceptions. The case of software in a higher layer using modules in a lower layer that is not just the next lower layer is called *layer bridging.* If many of these are present, the system is poorly structured, at least with respect to the portability and modifiability goals that layering helps to achieve. Systems with upward usages are not, strictly according to the definition, layered. However, in such cases, the layered style may represent a close approximation to reality and also conveys the ideal design that the architect was trying to achieve.

Layers cannot be derived by examining source code. Layers are logical groupings that are wonderful aids in creating and communicating the architecture, but often they are not explicitly delimited in the source code. The source code may disclose what uses what, but the relation in layers is *allowed to use.*

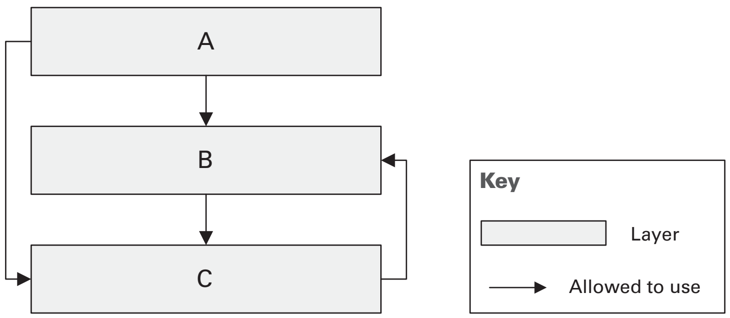


Figure 2.17
There may be three layers here, but this is not a design in the layered style, which forbids upward uses.

Some of the criteria used in defining the layers of a system are an expectation that they will evolve independently on different time scales, that different people with different sets of skills will work on different layers, and that different levels of reuse are expected of the different layers.

2.4.2 Elements, Relations, and Properties

Table 2.4 summarizes the characteristics of the layered style.

The elements of a layered view are *layers*. A layer is a cohesive collection of modules, each of which may be invoked or accessed. The modules in a layer can be anything: from modules that implement Web services to assembly-language subroutines to shared data. A requirement is that the modules have an interface by which their services can be triggered or accessed.

The relation among layers is *allowed to use*. For two layers having this relation, any module in the first is allowed to use any module in the second. Module A is said to use module B if A's correctness depends on B being correct and present.

Layers have the following properties, which should be documented in the element catalog accompanying the layer diagram.

Element catalogs are described in Section 10.1.

- *Contents*. The description of a layer should provide guidelines to what modules should be in a layer and how to implement them. It can also explicitly list the software modules

Table 2.4 Summary of the layered style

Overview	The layered style puts together layers (groupings of modules that offer a cohesive set of services) in a unidirectional *allowed-to-use* relation with each other.
Elements	*Layer*. The description of a layer should define what modules the layer contains.
Relations	*Allowed to use*, which is a specialization of the generic *depends-on* relation. The design should define the layer usage rules (for example, "A layer is allowed to use any lower layer.") and any allowable exceptions.
Constraints	• Every piece of software is allocated to exactly one layer. • There are at least two layers (typically three or more). • The *allowed-to-use* relations should not be circular (that is, a lower layer cannot use a layer above).
What It's For	• Promoting modifiability and portability • Managing complexity and facilitating the communication of the code structure to developers • Promoting reuse • Achieving separation of concerns

contained by each layer. Each module should be assigned to exactly one layer. Layers typically have labels that are descriptive but vague, such as "network communications layer" or "business rules layer"; a description is needed that identifies the complete contents of every layer.

- *The software a layer is allowed to use.* Is a layer allowed to use only the layer below, any lower layer, or some other? If a layer is segmented horizontally, are modules in a segment permitted to use modules in another segment of the same layer? This part of the documentation must also explain exceptions, if any, to the usage rules implied by the geometry.

Section 2.4.4 has more information about segmented layers.

See also "Perspectives: Calling Higher Layers" on page 100, in this chapter.

You should document the rationale for the choice of layer partitioning. Explain how each layer provides a cohesive set of responsibilities. This description helps to assign future modules to one layer or the other.

Suppose that module P_1 is allowed to use module P_2. Should P_2 be in a lower layer than P_1, or should they be in the same layer? Layers are not a function of just who uses what, but are the result of a conscious design decision that allocates modules to layers, based on such considerations as cohesion and the nature of likely changes. In general, P_1 and P_2 should be in the same layer if they are likely to be ported to a new application together or if together they provide different aspects of the same **virtual machine** to a usage community.

A **virtual machine** is a collection of modules that form an isolated, cohesive set of services that can execute programs. It's sometimes called an *abstract machine*.

See "Coming to Terms: Virtual Machines" on page 99, in this chapter.

The preceding is an operational definition of cohesion. The cohesion explanation can also serve as a *portability guide*, describing the changes that can be made to each layer without affecting other layers.

2.4.3 What the Layered Style Is For

Layers help to bring quality attributes of modifiability and portability to a software system. A layer is an application of the principle of information hiding. The theory is that a change to a lower layer can be hidden behind its interface and will not impact the layers above it. As with all such theories, both truth and caveats are associated with it. The truth is that this technique has been used with great success to support portability. Machine, operating system, or other platform dependencies are hidden within a layer; as long as the interface for the layer does not change, technology-specific or product-specific parts can be exchanged, and the upper levels that depend only on the interface will work successfully.

See "Coming to Terms: Signature, Interface, API" on page 280, in Chapter 7.

The caveat is that *interface* means more than just the application programming interface (API) containing program signatures. An interface embodies all the assumptions that an external entity—in this case, a layer—may make. Changes in a lower layer that affect, say, a performance assumption will leak through its interface and may affect a higher layer.

A common misconception is that layers introduce additional runtime overhead. Although this may be true for naive implementations, sophisticated compile/link/load facilities can reduce additional overhead.

We have already mentioned that in some contexts, a layer may contain unused services. These unused services may needlessly consume a runtime resource, such as memory to store the unused code or a thread that is never launched. If these resources are in short supply, a sophisticated compile/link/load facility that eliminates unused code will be helpful.

Layers are part of the blueprint role that architecture plays for constructing the system. Knowing the layers in which their software resides, developers know what services they can rely on in the coding environment. Layers might define work assignments for development teams, although not always.

Layers are part of the communication role played by architecture. In a large system, the number of modules and the dependencies among them rapidly expand. Organizing the modules into layers with interfaces is an important tool for managing complexity and communicating the structure to developers.

Grouping into layers those modules that have the same technology abstraction or are cohesive with respect to their responsibilities helps to assign the implementation work across more specialized teams. For example, the modules in a presentation layer can be assigned to skilled GUI developers.

Layers help with the analysis role played by architecture. They support the analysis of the impact of changes to the design by enabling some determination of the scope of changes.

Layers that provide a virtual machine promote portability. For this reason, it is important to scrutinize the interface of such layers to ensure that portability concerns are addressed. The interface should not expose functions that are dependent on a particular platform; these functions should be hidden behind a more abstract interface that is independent of platform.

See "Coming to Terms: Virtual Machines" on page 99, in this chapter.

Because the ordering relationship among layers has to do with "implementation allowed to use," the lower the layer, the fewer the facilities available to it. That is, the "worldview" of lower layers tends to be smaller and more focused on the computing platforms. Lower layers tend to be built using knowledge of the operating systems, communications channels, databases, and the like. These platform-specific layers are largely independent of the particular application that runs on them; they make the application more easily portable to a different platform.

2.4.4 Notations for the Layered Style

Informal Notations

Stack
Layers are almost always drawn as a stack of boxes. The *allowed-to-use* relation is denoted by geometric adjacency and is read from the top down, as in Figure 2.18 (note that the key could have said, "A layer is allowed to use any lower layer").

Layering is thus one of the few architecture styles in which connection among components is shown by geometric adjacency and not an explicit symbology, such as an arrow, although arrows can be used, as in Figure 2.19.

Segmented Layers
Sometimes layers are divided into segments denoting a finer-grained aggregation of the modules. Often, this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use* relation. When this happens, the creator of the diagram must specify what usage rules are in effect among the segments. Many usage rules are possible, but they must be made explicit. In Figure 2.20, the top and the bottom layers are

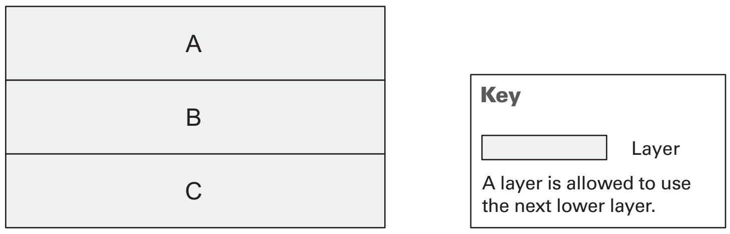


Figure 2.18
Stack of boxes notation for layered designs

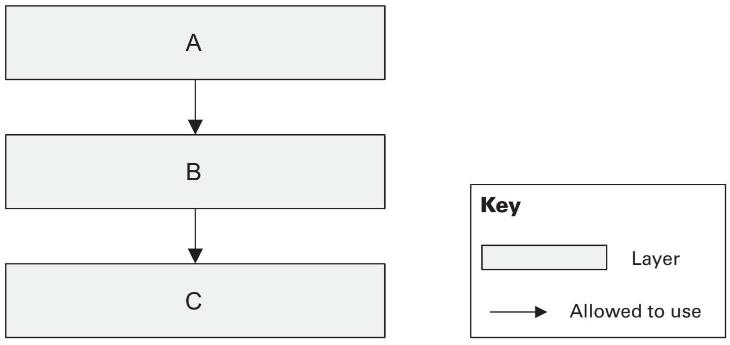


Figure 2.19
Layered design with *allowed-to-use* relations shown with arrows

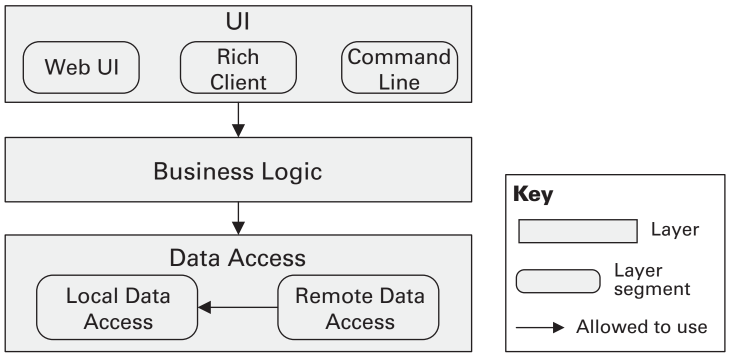


Figure 2.20
Layered design with segmented layers

segmented. Segments of the top layer are not allowed to use each other, but segments of the bottom layer are. If you draw the same diagram without the arrows, it will be harder to differentiate the usage rules within segmented layers. Layered diagrams are often a source of ambiguity because the diagram does not make explicit the *allowed-to-use* relations.

Rings
A notational variation is to show layers as a set of concentric circles, or rings. The innermost ring corresponds to the lowest layer; the outermost ring, the highest layer. A ring may be subdivided into sectors, meaning the same thing as the corresponding layer being segmented.

There is no semantic difference between a layer diagram that uses a stack of rectangles and one that uses the rings paradigm, except when segmented layers have restrictions on the *allowed-to-use* relation within the layer. In Figure 2.21, assume that ring segments that touch are allowed to use one another and that layer segments that touch are allowed to use one another. You cannot “unfold” the ring diagram to produce a stack diagram, such as the one on the right, with exactly the

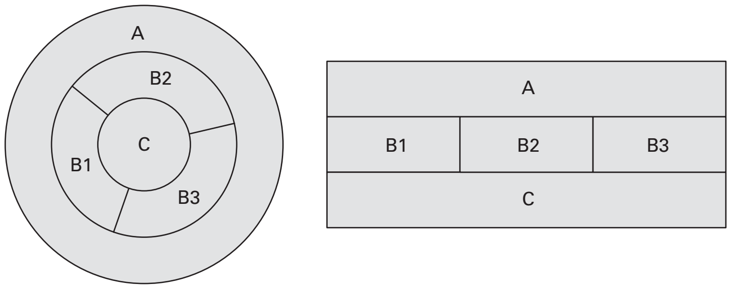


Figure 2.21
A layered design shown as concentric rings and as a stack of boxes. Are these two representations equivalent?

same meaning, because circular arrangements allow more adjacencies than do linear arrangements. (In the layer diagram, B1 and B3 are separate; in the ring diagram they are adjacent.) Cases like this are the only ones in which a ring diagram can show a geometric adjacency that a stack picture cannot.

Layers with a Sidecar

Many architectures that are described as layered look something like Figure 2.22. This type of notation could mean one of two things: (1) Modules in D can use modules in A, B, or C. (2) Modules in A, B, or C can use modules in D. (Technically, the diagram might mean that both are true, although this would arguably be a poor layered architecture.) The creator of the diagram must specify which usage rules pertain. A variation like this makes sense only for single-level usage rules in the main stack, that is, when A can use only B and nothing below. Otherwise, D could simply be made the bottommost layer in the main stack, and the "sidecar" geometry would be unnecessary.

In some cases, the layered architecture is depicted as a three-dimensional figure, to represent a layer that is accessible to all other layers, as shown in Figure 2.23.

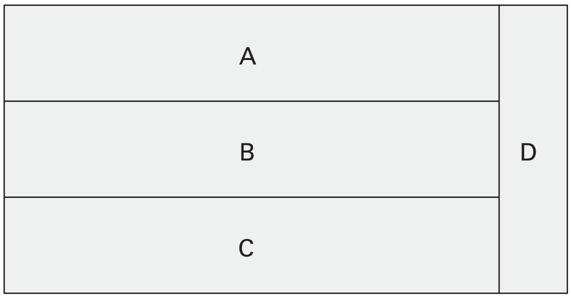


Figure 2.22
Layers with a "sidecar." The key should make clear what is allowed to use and be used by software in the box on the side.

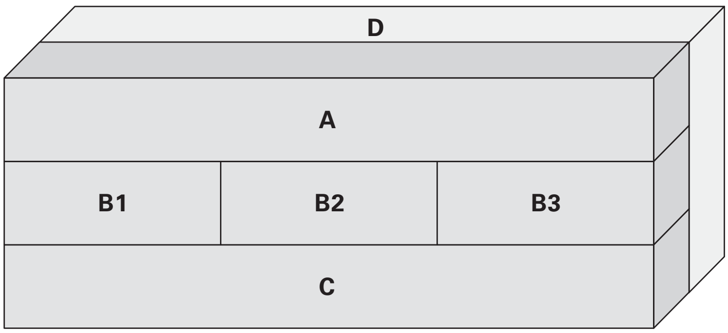


Figure 2.23
Three-dimensional layered diagram trying to show that layer D can be used by all other layers. The picture could just as well be showing that D can use all other layers. The ambiguity should be resolved by an annotation, or in the key.

Such layers on the side often represent utility libraries or platform services (such as the operating system or runtime environment).

Size and Color
Sometimes layers are colored to denote which team is responsible for them or to denote another distinguishing feature. Sometimes layers use different colors just to improve readability. Size is sometimes used to give a vague idea of the relative size of the modules constituting the various layers. If they carry meaning, size and color should be explained in the key accompanying the layer diagram.

UML
UML has no built-in primitive corresponding to a layer. However, layers can be represented in UML as stereotyped packages, as shown in Figure 2.24. A package is a general-purpose mechanism for organizing elements into groups, and it suits the notion of layers. The *allowed-to-use* relation can be a stereotyped dependency between layer packages.

Appendix A discusses how to use UML classes and packages to represent layers and more.

Access dependencies are not transitive. If package 1 can access package 2 and package 2 can access package 3, it does not automatically follow that package 1 can access package 3.

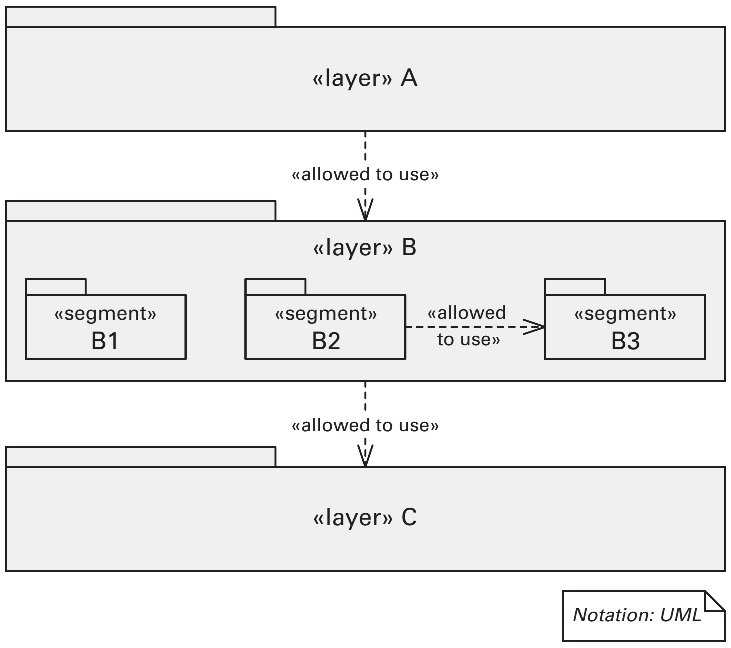


Figure 2.24
Documenting segmented layers in UML. If segments in a layer are allowed to use each other, then <<allowed to use>> dependencies must be added among them as well.

2.4.5 Relation to Other Styles

Layer diagrams are often confused with other architecture styles when information orthogonal to the *allowed-to-use* relation is introduced without conscious decision.

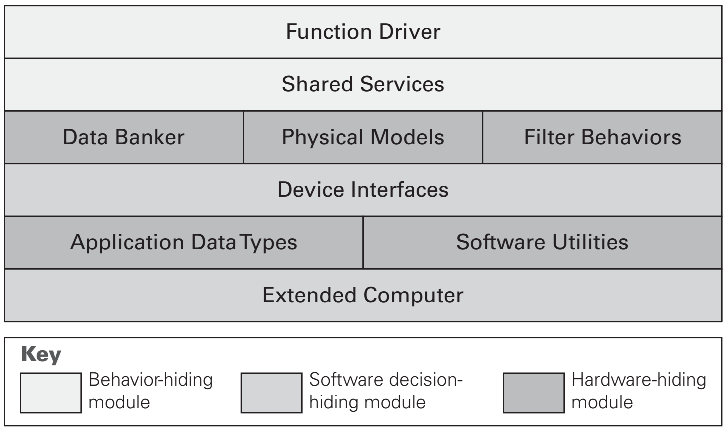
1. *Module decomposition.* Layers in a layered view and modules in a decomposition view are always related but almost never correspond one-to-one with each other. A layer may comprise more than one module. Two submodules of a module may be part of different layers. In any case, you should provide a mapping between layers and the modules in the decomposition view. If a module occurs in more than one layer, you can indicate this by using colors or fill patterns, as in Figure 2.25.

   In this example, once again borrowing from the A-7E architecture described previously, the mapping between layers and modules is not one-to-one. In this architecture, the criterion for partitioning into modules was the encapsulation of likely changes. The shading of the elements denotes the coarsest-grain decomposition of the system into modules; that is, Function Driver and Shared Services are both submodules of the Behavior Hiding module. Hence, in this system, layers correspond to parts of highest-level modules. It's also easy to imagine a case in which a module constitutes a part of a layer.

Section 2.1.6 has more information about the module decomposition in the A-7E avionics system.

2. *Tiers.* Layers are often confused with the tiers in a multi-tier architecture. Layers are not tiers. The layered style shows

Figure 2.25
A diagram showing layers and modules from a decomposition view from the A-7E software architecture



Software in a layer is allowed to use software in the same or any lower layer.

groupings of implementation units and hence is a kind of module style. The multi-tier style is a component-and-connector style because tiers congregate runtime components.

3. *Module "uses" style.* Because layers express the *allowed-to-use* relation, there is a close correspondence to the uses style. Of course, no *uses* relation is allowed to violate the *allowed-to-use* relation. If incremental development or the fielding of subsets is a goal, the architect will begin with a broad *allowed-to-use* specification. That specification gives the guidelines for designing with actual *uses* relations any subset of interest.

Section 4.6.2 discusses tiers.

The uses style is covered in Section 2.2.

See "Perspectives: Using a DSM to Maintain a Layered Architecture" on page 101, in this chapter, for a description of how layered architectures can be identified in a DSM based on existing code.

2.4.6 Examples Using the Layered Style

UNIX System V

A classic layered design is the UNIX System V operating system, as shown in Figure 2.26. The lower layers form the system kernel; top layers are user programs or libraries that access the kernel through system calls. The system call interface layer isolates the kernel implementation details and provides a virtual

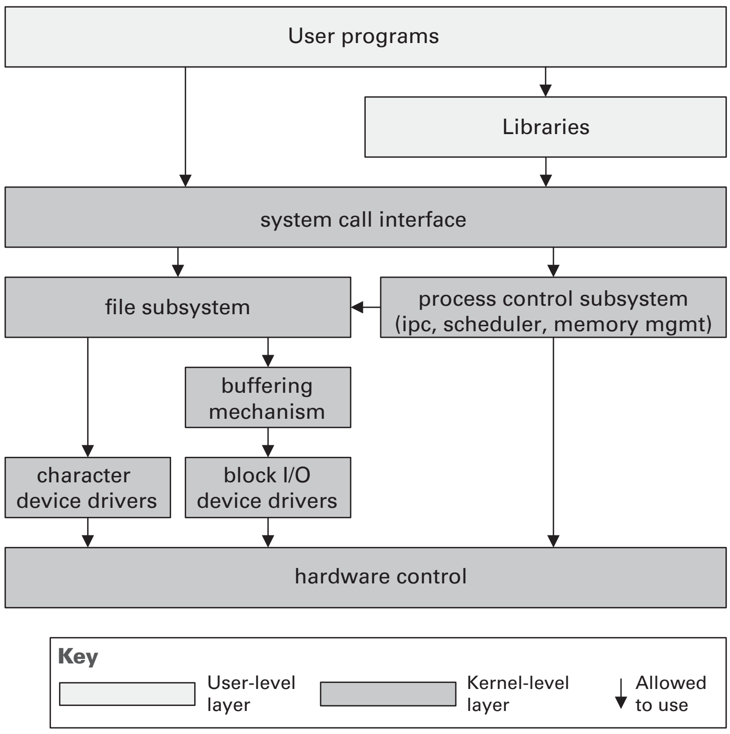


Figure 2.26
The primary presentation of a layered view of the UNIX System V operating system implementation (adapted from Bach 1986)

machine to user programs. The file subsystem is responsible for managing files (devices are treated as files), administering free space, controlling access, and reading/writing data. The process control subsystem is responsible for process scheduling, interprocess communication, process synchronization, and memory management. The hardware control layer is responsible for handling interrupts and communicating with the machine.

This design is presented in Chapter 2 of the classic book by Maurice Bach, *The Design of the UNIX Operating System* (Bach 1986), where a candid observation is made: "The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others." All such exceptions should be noted in your documentation.

This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level.

—H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs* (1996)

Java EE Application

Figure 2.27 is the primary presentation of the layered view of a set of integrated, multi-tier, Web-based applications that use the Java EE platform. All user operations in these applications follow this layered design. The topmost layer has presentation classes, which are servlets and JavaServer Faces (JSF) action classes. Servlet and JSF are Java component technologies for developing Web components. The second layer has controller classes, which implement the sequence of steps to carry on the functionality of a use case. An example of a controller class is CtlRetrievePtoDays. Controller classes interact with business service classes, which encapsulate the core business logic associated with domain objects. An example of a service class is SvcFullTimeEmployee. The lowermost layer has data access objects. These modules handle all interaction with the relational database.

There are two sets of auxiliary modules that are presented as sidecar layers. On the left are presentation data transfer objects (DTOs). They are simple classes that contain basic attributes corresponding to data elements required in different user screens. The right sidecar layer has the corporate DTOs and plain old Java objects (POJOs). Like presentation DTOs, these classes have a set of attributes to hold data. In this design, DTOs have attributes required by a particular transaction, whereas POJOs correspond to data entities stored in the database.

The key drivers for this layered design are modifiability and portability, which is achieved with separation of concerns. On top is the presentation layer. Changes to the user interface are

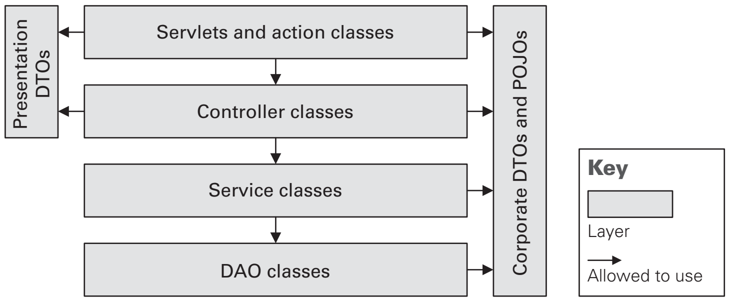


Figure 2.27
Part of the layered view of a set of Java EE applications. The top layer has servlets and JSF action classes responsible for the user interface. Controller classes handle the user operations by interacting with business service classes. Access to the database is done in the lowermost layer with the data access objects. Sidecar layers contain DTOs and POJOs that are used by the other layers to hold and transfer data.

addressed in that layer. If the technology used to implement the UI has to change from servlet and JSF to, say, Google Web Toolkit and Flash, this layer has to be rewritten, but the other layers should remain unchanged. The second layer implements the logic to handle the user actions by wiring the calls to services in the third layer, which is the core business logic layer. The bottom layer isolates database access operations and also enhances portability. If the application is migrated to a different database management system with a different SQL dialect, all modifications required would be confined to that layer.

COMING TO TERMS

Virtual Machines

A virtual machine, sometimes called an *abstract machine*, is a collection of modules that form an isolated, cohesive set of services that can execute programs. Early use of the term referred to a more abstract stand-in for a real computer, but current use includes virtual machines that have no direct correspondence to any real machine. Interpreters are good examples of virtual machines. The Common Language Runtime (CLR) of the Microsoft .NET platform is an example of a virtual machine. It provides services to execute bytecode produced by compiling C# or other .NET programming languages. The CLR converts the bytecode into code that is native to the operating system underneath. The Java Virtual Machine (JVM) does the same thing for the Java language. An operating system itself is a virtual machine that allows the execution of native code on the underlying hardware. Thus, a virtual machine is a software layer that can execute "programs," which can be sequences of calls to facilities of the virtual machine's interface. Hence some authors regard layers and virtual machines as synonyms.

PERSPECTIVES

Calling Higher Layers

We have been emphatic in saying that upward uses invalidate layering. We made allowances for documented exceptions but implied that too many of those would get you barred from the Software Architect's Hall of Fame.

Seasoned designers, however, know that in many elegantly designed layered systems, all kinds of control and information flow upward along the chain of layers, with no loss of portability, reusability, modifiability, or any of the other qualities associated with layers. In fact, one of the purposes of layers is to allow for the "bubbling up" of information to the units of software whose scope makes them the appropriate handlers of the information. One approach to error handling illustrates this upward flow. Suppose that we have a simple three-layer system, as in Figure 2.28. Say that program P_A in A uses program P_B in B, which uses program P_C in C. If P_C is called in a way that violates its specification, P_C needs a way to tell P_B, "Hey! You called me incorrectly!" At that point, (1) P_B can either recognize its own mistake and call P_C again, this time correctly, or take another action; or (2) P_B can realize that the error resulted because it was called incorrectly—perhaps it received bad data—by P_A. In the latter case, P_B needs a way to tell P_A, "Hey! You called me incorrectly!"

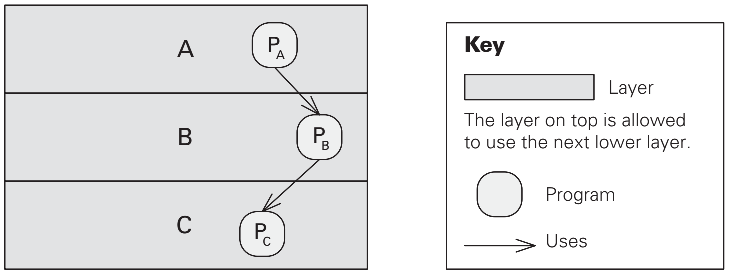


Figure 2.28
Layered design showing programs inside and their usage dependencies

Callbacks are a mechanism to manifest the protestation. We do not want P_C written with knowledge about programs in B or P_B written with knowledge about programs in A, as this would limit the portability of layers C and B. Therefore, the names of higher-level programs to call in case of error are passed downward as parameters. Then the specification for, say, P_B includes the promise that in case of error, it will invoke the program whose name has been made available to it.

Other situations where callbacks can be used include:

- When P_A uses P_B to obtain data to present in the user interface but P_A also wants P_B to announce future changes to the data. In other words, P_A sub-

scribes to events that can be emitted by P_B and provides to P_B the name of the operation that will handle the events.
- When P_A uses P_B and the interaction is asynchronous, but P_A needs to receive a response once P_B is done processing the request. In this case P_A provides P_B the name of the operation to call.

So there we have it: data and control flowing downward and upward in an elegant error-handling scheme that preserves the best qualities of layers. So much for our prohibition about upward uses. Right?

Wrong. Upward uses are still a bad idea, but the scheme we just described doesn't have any. It has upward data flow and upward invocation but not uses. The reason is that once a program calls its error handler, its obligation is discharged. The program does not *use* the error handler, because its own correctness depends not a whit on what the error handler does. This is how the callback mechanisms, built in to some programming languages, work and still allow true layered systems to be written in those languages.

Although this may sound like a mere technicality, it is an important distinction. *Uses* is the relation that determines the ability to reuse and to port a layer; "calls" or "sends data to" is not. Architects need to know the difference and need to convey the precise meaning of the relations in their architecture documentation.

—P.C. and P.M.

PERSPECTIVES

Using a DSM to Maintain a Layered Architecture

Tools based on the dependency structure matrix claiming to be the solution to managing complexity in large software projects have recently been capturing the attention of program analysts and software architects. The DSM concept has been adopted for use in software engineering from its origins with Donald Steward as the Design Structure System (Steward 1981), which he devised in 1967 to help manage complexity in the nuclear power industry. Over the past 15 years the DSM has been used in a wide variety of industries to aid in systems engineering and analysis as well as project planning and management.

When layer A depends on layer B and layer B depends on layer A, there is a codependence between these two layers, a situation that is forbidden in a layered architecture. In a DSM, circular dependencies are immediately visible as marked cells on both sides of the matrix's diagonal. A layered architecture is clearly discernable because the corresponding DSM is a lower triangular matrix (that is, one in which all the marked cells are below the diagonal). For example, consider the layered architecture in Figure 2.29. The key indicates that a layer is allowed to use only the next lower layer, so it's a strictly layered design. The corresponding DSM is shown in Figure 2.30(a). If a layer were allowed to use any lower layer, the DSM would be similar to Figure 2.30(b). When cells above the diagonal are marked, the architect can see the circular dependency and focus on what to change to reach the goal of a layered architecture.

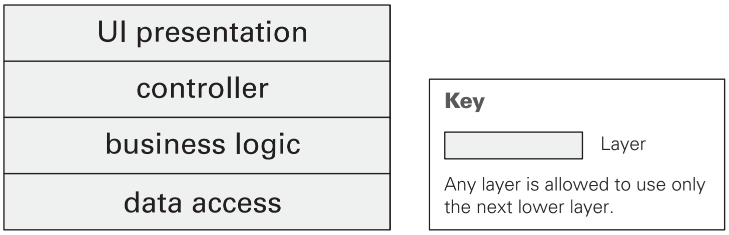
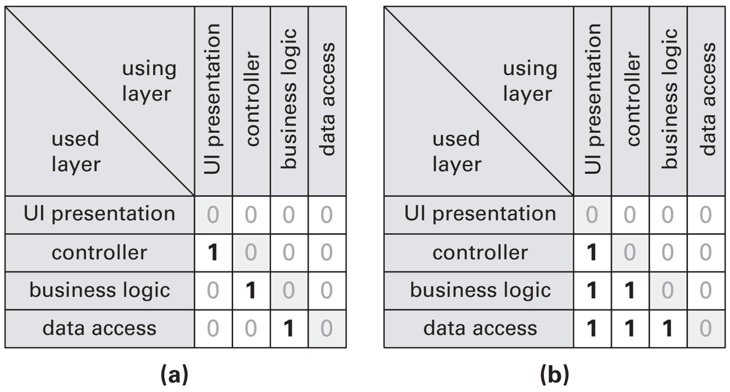


Figure 2.29
Simple layered architecture



(a)

used layer \ using layer	UI presentation	controller	business logic	data access
UI presentation	0	0	0	0
controller	**1**	0	0	0
business logic	0	**1**	0	0
data access	0	0	**1**	0

(b)

used layer \ using layer	UI presentation	controller	business logic	data access
UI presentation	0	0	0	0
controller	**1**	0	0	0
business logic	**1**	**1**	0	0
data access	**1**	**1**	**1**	0

Figure 2.30
DSM showing (a) strictly layered design and (b) layered design

In practice, layered designs are more complex. Figure 2.31 shows the layered design that was introduced in Figure 2.27, now with Java packages added for each layer. The DSM for this design is shown in Figure 2.32. In a DSM tool, the architect can mark the dependencies that violate the layered design: the highlighted cells above and below the diagonal in Figure 2.32. During the implementation of the system, the tool can create a DSM from the code and highlight any violations. If other constraints on interdependencies have been indicated by the architect, those will also be visible using the DSM representation. With good tool support, continuous integration builds can be subjected to DSM analysis, and architecture violations can be caught immediately. DSM tools also generally allow the user to perform "what-if" analysis by simulated restructuring of the system, providing immediate insight into the impact that a suggested change would have on the system's structure.

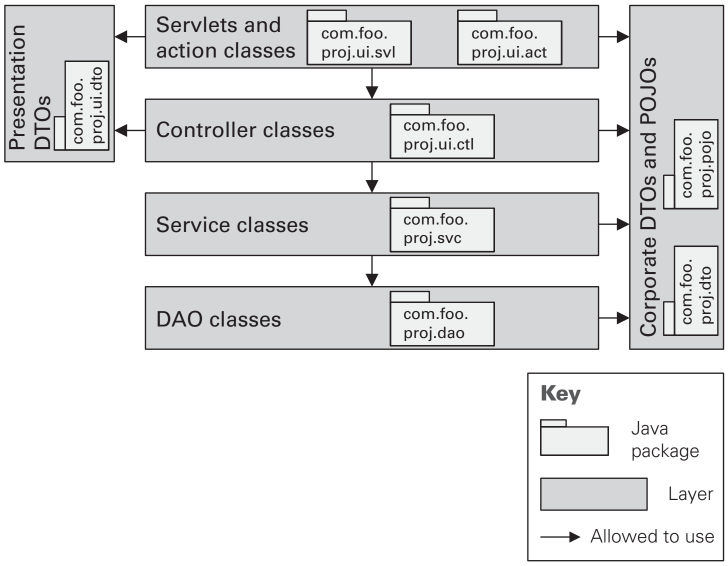


Figure 2.31
Layered design showing Java packages for each layer

used module \ using module	com.foo.proj.ui.svl	com.foo.proj.ui.act	com.foo.proj.ui.ctl	com.foo.proj.ui.dto	com.foo.proj.svc	com.foo.proj.dao	com.foo.proj.dto	com.foo.proj.pojo
com.foo.proj.ui.svl	0	0	0	0	0	0	0	0
com.foo.proj.ui.act	0	0	0	0	0	0	0	0
com.foo.proj.ui.ctl	1	1	0	0	0	0	0	0
com.foo.proj.ui.dto	1	1	1	0	0	0	0	0
com.foo.proj.svc	0	0	1	0	0	0	0	0
com.foo.proj.dao	0	0	0	0	1	0	0	0
com.foo.proj.dto	1	1	1	0	1	1	0	0
com.foo.proj.pojo	1	1	1	0	1	1	0	0

Figure 2.32
DSM for a layered design. The highlighted cells above and below the diagonal represent dependencies that are not allowed

—J.S. and P.M.

2.5 Aspects Style

2.5.1 Overview

If you haven't documented a commonality, it isn't likely to be one by the time you get done implementing.

—D. L. Parnas

The aspects style is a module style used to isolate in the architecture the modules responsible for crosscutting concerns.

When we implement software modules in general, the business logic code ends up intermixed with code that deals with crosscutting concerns. For example, if you're writing a bank automation system, there may be modules such as Account, Customer, and Atm. The Account module ideally would contain only the code to deal with the bank account business logic (open/close account, deposit, withdraw, transfer, and so on). But in practice we have to add code to handle crosscutting concerns, such as access control, transaction management, and logging.

The aspects style prescribes that the modules responsible for the crosscutting functionality should be placed in one or more aspect views. These modules are called aspects, based on the terminology introduced by aspect-oriented programming (AOP). The aspect views should contain information to bind each aspect module to the other modules that require the crosscutting functionality.

See "Coming to Terms: Aspect-Oriented Programming" on page 107, in this chapter.

The aspects style is particularly useful when you plan to use AOP in the implementation. However, it's also applicable when crosscutting functionality will be implemented in traditional ways through class inheritance and interfaces, macro insertion, dependency injection, utility libraries, or other alternatives. The goal of designing and implementing crosscutting concerns in separate aspect modules is to improve modifiability of the modules that deal with the business domain functionality.

2.5.2 Elements, Relations, and Properties

Table 2.5 summarizes the characteristics of the aspects style. The elements in the aspects style are aspect modules. As mentioned in Section 2.5.1, an aspect is a special type of module introduced by AOP. It contains the crosscutting code that affects other specific modules in the system.

The relation found in the aspects style is usually called *crosscuts*. An aspect crosscuts a module if the aspect contains crosscutting functionality that will affect the module. An aspect may contain the same properties of a regular module. In addition, it may contain a property that describes what target modules are affected by that aspect; in AOP terms, this property is called pointcut specification.

Table 2.5 Summary of the aspects style

Overview	The aspects style shows aspect modules that implement crosscutting concerns and how they are bound to other modules in the system.
Elements	*Aspect*, which is a specialized module that contains the implementation of a crosscutting concern
Relations	*Crosscuts*, which binds an aspect module to a module that will be affected by the crosscutting logic of that aspect
Constraints	• An aspect can crosscut one or more regular modules as well as aspect modules. • An aspect that crosscuts itself may cause infinite recursion, depending on the implementation.
What It's For	• Modeling crosscutting concerns in object-oriented designs • Enhancing modifiability

2.5.3 What the Aspects Style Is For

The aspects style can be used to model the implementation of crosscutting concerns. It promotes modifiability by increasing modularity and avoiding the tangling of crosscutting functionality and business domain functionality.

2.5.4 Notations for the Aspects Style

UML

Although UML does not have built-in symbols for aspects, it is a common choice for aspect views. In UML aspect modules are usually represented as stereotyped classes in a class diagram, as shown in Figure 2.33. Especially when the target implementation platform supports AOP, showing aspect modules as stereotyped classes makes sense because aspects are structurally similar to classes: they may contain attributes and operations, and they may extend another aspect in an inheritance relation.

The *crosscut* relation could be represented as a stereotyped dependency going from the aspect to each module it crosscuts. However, this alternative does not scale: by definition an aspect provides crosscutting functionality, and hence it may crosscut too many modules. Drawing a line between the aspect module and each of the crosscut modules is impractical in nontrivial systems and would clutter the diagrams. A better alternative is simply to omit the *crosscut* relation from the diagrams. Instead, just add a comment to the aspect module to characterize (in natural language or in a formal syntax) what other modules this aspect crosscuts. Figure 2.34 shows an example. Not showing

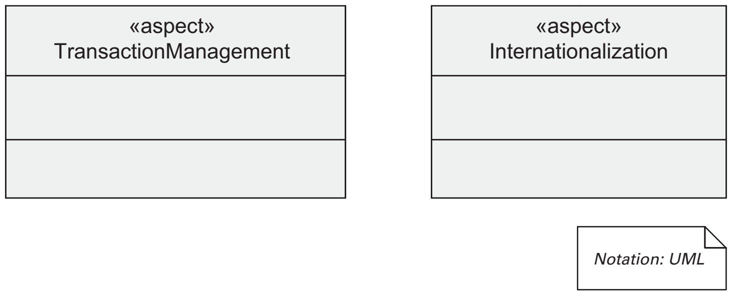


Figure 2.33
Aspect modules are often represented in UML as classes with stereotype <<aspect>>.

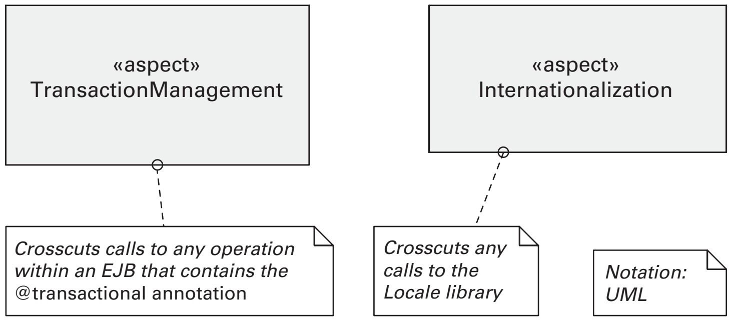

«aspect»
TransactionManagement
«aspect»
Internationalization
Crosscuts calls to any operation within an EJB that contains the @transactional annotation
Crosscuts any calls to the Locale library
Notation: UML

Figure 2.34
Instead of trying to draw a line from each aspect to every module it crosscuts, we simply add a comment box that characterizes what modules will be crosscut.

the crosscut relation in the diagram actually makes sense because in an AOP implementation, the developer doesn't have to identify each target class for a given aspect. The architecture representation should not be more detailed than the implementation!

2.5.5 Relation to Other Styles

In general, aspects allow inheritance. The aspects style may be combined with the generalization style when we want to show a hierarchy of aspects.

2.5.6 Examples Using the Aspects Style

Figure 2.35 is from the aspects view of an application called IkeWiki. The design prescribes the use of aspects for transaction management, exception handling, authorization check, and enforcement of architecture constraints. Drawing a line for each *crosscut* relation would be impractical, so the architect opted simply to indicate with comments what other modules should be crosscut by each aspect.

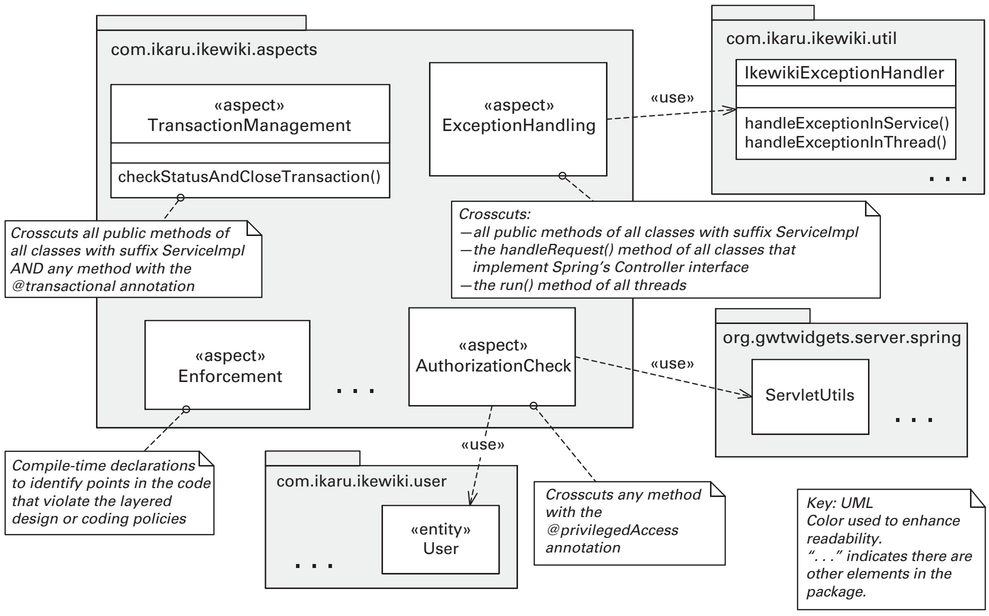

com.ikaru.ikewiki.aspects
«aspect» TransactionManagement
checkStatusAndCloseTransaction()
«aspect» ExceptionHandling
«use»
com.ikaru.ikewiki.util
IkewikiExceptionHandler
handleExceptionInService()
handleExceptionInThread()
. . .
Crosscuts all public methods of all classes with suffix ServiceImpl AND any method with the @transactional annotation
Crosscuts:
—all public methods of all classes with suffix ServiceImpl
—the handleRequest() method of all classes that implement Spring's Controller interface
—the run() method of all threads
«aspect» Enforcement
. . .
«aspect» AuthorizationCheck
«use»
org.gwtwidgets.server.spring
ServletUtils
. . .
«use»
Compile-time declarations to identify points in the code that violate the layered design or coding policies
com.ikaru.ikewiki.user
«entity» User
. . .
Crosscuts any method with the @privilegedAccess annotation
Key: UML
Color used to enhance readability.
". . ." indicates there are other elements in the package.

Figure 2.35
Primary presentation for the aspects view of the IkeWiki application. This Java EE application implemented with the Spring framework and Google Web Toolkit uses aspects for some crosscutting concerns. The TransactionManagement aspect makes sure all requests received by the server will close the transaction and release database resources properly, performing a rollback when an exception occurs. The ExceptionHandling aspect has code to log the error to the database, send e-mail notification if applicable, and wrap the exception with a proper user message to be displayed by the client application. This aspect is woven into server-side classes that are either threads or entry points to process HTTP requests. The AuthorizationCheck aspect is used to check if the current user has permission to execute a specific method. The Enforcement aspect is different from the others. It doesn't exactly implement a crosscutting concern, but rather it scans the source code at compile time looking for violations of the layered design, as well as violations of several coding policies.

COMING TO TERMS

Aspect-Oriented Programming

Aspect-oriented programming is an evolutionary implementation paradigm that complements object-oriented programming and facilitates the implementation of crosscutting concerns. AspectJ is probably the most widely known AOP package. Other implementations include Spring AOP, JBoss AOP, AspectC++, and Aspect#.

Suppose the bank automation example is implemented using a regular object-oriented language. The solution would contain classes such as Account, Customer, and Atm. In these classes, the code to handle crosscutting concerns such as logging or transaction management is tangled with the business logic code, making the classes more difficult to maintain. Moreover, the lines of code found in class Account to handle transaction management are very similar if not equal to the lines of code that handle the same concern in Customer, Atm, and other classes. The code for a particular concern is scattered across several classes; that poses a modifiability problem. Suppose you need to change the signature of a method used for logging. You'll need to change the corresponding lines of code in all classes where logging is needed. Code tangling and code scattering in traditional object-oriented applications is notionally represented in Figure 2.36.

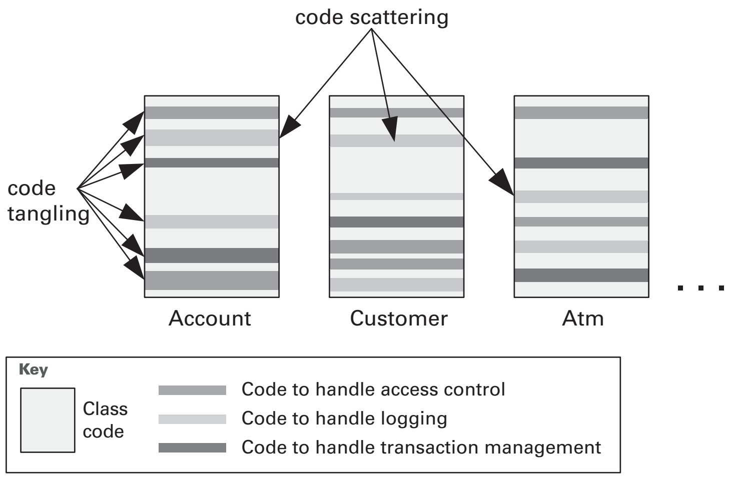


Figure 2.36
The traditional object-oriented implementation of a bank automation system would have several classes where the business logic is tangled with code that handles crosscutting concerns, such as access control, logging, and transaction management. In addition, the code that handles a particular crosscutting concern is repeated and scattered across several classes.

AOP brings an ingenious solution to improve modularity and resolve the code tangling and code scattering problems. The crosscutting code is factored out from the classes and placed in a special module called aspect, as represented in Figure 2.37. An aspect has two important parts: advices and pointcut specifications. Advices contain the code for the crosscutting concerns. Such code will be injected at certain points (called join points) of the classes through a process called weaving, carried on by the AOP compiler. The pointcut specifications contain declarations that map to specific sets of join points in the target classes. In the aspect code, advices are associated to pointcut specifications to let the AOP compiler know where exactly each advice code will be injected in the target classes.

AOP is the programming component of the larger aspect-oriented software development (AOSD) movement, which strives to factor out otherwise-redundant commonality in all kinds of software activities, including requirements engineering, design, and testing.

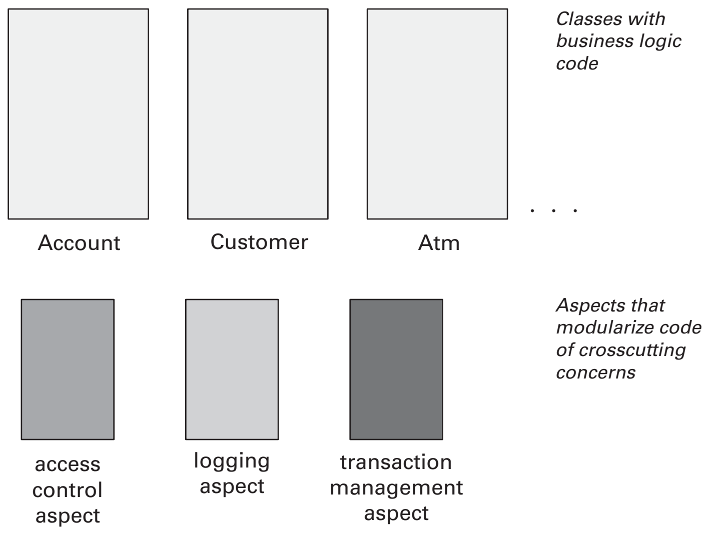

Classes with business logic code
Account
Customer
Atm
. . .
Aspects that modularize code of crosscutting concerns
access control aspect
logging aspect
transaction management aspect

Figure 2.37
In the aspect-oriented implementation of the same bank automation system, the classes don't contain code for logging, access control, transaction management, and other crosscutting concerns. The code to handle these concerns is now inside aspect modules. Classes such as Account, Customer and Atm contain the business logic only. The AOP compiler will use the weaving process to insert the code inside aspects at the locations in the classes where it's needed.

2.6 Data Model

2.6.1 Overview

Data modeling is a common activity in the software development process of information systems. The output of this activity is the data model, which describes the static information structure in terms of data entities and their relationships. For example, in a banking system, entities typically include Account, Customer and Loan. Account has several attributes, such as account number, type (savings or checking), status, and current balance. A relationship may dictate that one customer can have one or more accounts, and one account is associated to one or two customers. The data model is often represented graphically in entity-relationship diagrams (ERDs) or UML class diagrams.

The first draft of an architecture view typically has very little detail. Over time, as design decisions are made, the view is elaborated until the architect considers there's enough information captured in that architecture view. The same thing happens with the data model. Data modeling spans the evolution of the high-level model that displays the data entities in a given business domain into a model that shows details of how the data is stored, for example, in a relational database management system. As a result, different organizations focus the modeling and documentation effort on different stages of the data model evolution. Thus organizations sometimes use qualifiers to the data model to distinguish these stages. Examples of qualifiers include:

- *Conceptual.* The conceptual data model abstracts implementation details and focuses on the entities and their relationships as perceived in the problem domain. Figure 2.38 shows a fragment of a conceptual data model.
- *Logical.* The logical data model is an evolution of the conceptual data model toward a data management technology (such as relational databases). It is typically the subject of normalization (see Section 2.6.2). Figure 2.39 shows an example of a logical data model.
- *Physical.* The physical data model is concerned with the implementation of the data entities. It incorporates optimizations that may include partitioning or merging entities, duplicating data, and creating identification keys and indexes. For example, in Figure 2.40 a column named totalPrice was likely added to the entity Order as a performance optimization, since the total price could also be obtained by reading all order items and adding up their prices.

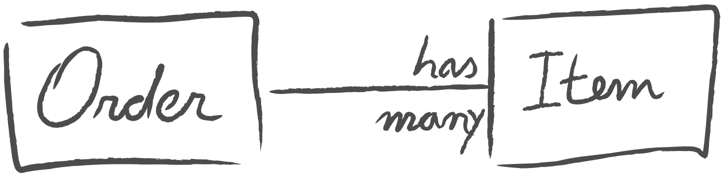


Figure 2.38
First draft of a conceptual data model. This and the next two diagrams are fragments of an online order-processing system at different stages.

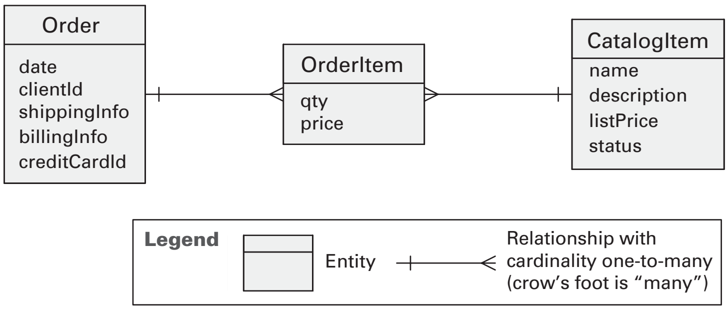


Figure 2.39
Logical data model that has evolved from the conceptual data model in Figure 2.38

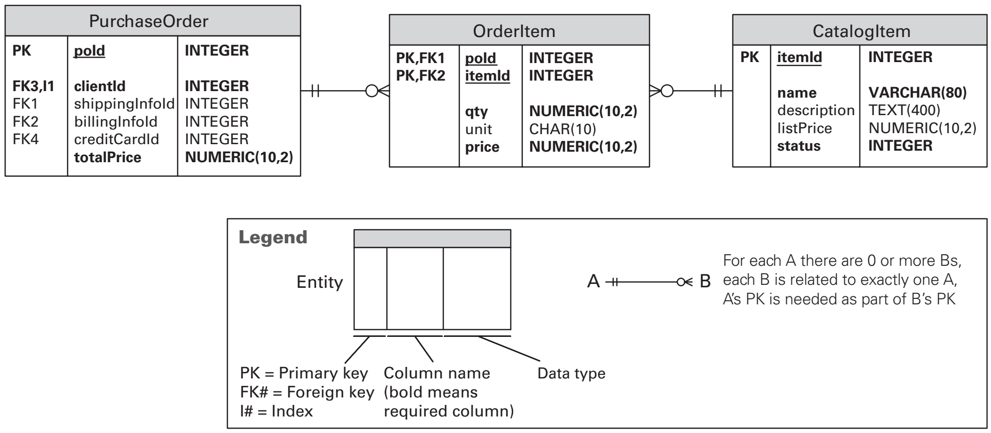


Figure 2.40
Physical data model that was created by adding implementation details and optimizations to the logical data model in Figure 2.39

In an early stage, the architecture documentation may contain the data model with the key entities and important relationships. Later on, this initial model is superseded by the detailed model approved by the data administrators.

2.6.2 Elements, Relations, and Properties

Table 2.6 summarizes the characteristics of the data model style.

The elements in a data model are called data entities or simply entities. Any distinguishable object that contains information to be stored or represented in the system can be an entity.

See “Coming to Terms: Entity” on page 118, in this chapter.

Table 2.6 Summary of the data model style

Overview	The data model describes the structure of the data entities and their relationships.
Elements	*Data entity*, which is an object that holds information that needs to be stored or somehow represented in the system. Properties include name, data attributes, primary key, and rules to grant users permission to access the entity.
Relations	• *One-to-one*, *one-to-many*, and *many-to-many* relationships, which are logical associations between data entities • *Generalization/specialization*, which indicate an *is-a* relation between entities • *Aggregation*, which turns a relationship into an aggregate entity
Constraints	Functional dependencies should be avoided.
What It's For	• Describing the structure of the data used in the system • Performing impact analysis of changes to the data model; extensibility analysis • Enforcing data quality by avoiding redundancy and inconsistency • Guiding implementation of modules that access the data

Properties of entities may include:

- Name of the entity.
- Description of the meaning and significance of the entity.
- List of data attributes of the entity. For example, a Car entity may have attributes year, manufacturer, model, mileage, price, and license. Each attribute may have properties, such as data type, size, and whether it's a required attribute or not.
- The attribute (or attributes) used to uniquely identify an entity (that is, the primary key).
- Whether an entity is weak. A weak entity, also known as a dependent entity, depends on the existence of another entity to exist. For example, an OrderItem requires the existence of a PurchaseOrder in Figure 2.40.
- Constraints and invariants on the values of individual or combined attributes. For example, "Returning date cannot be prior to arrival date."
- Rules that will be used to grant permissions to users or user groups to access the entity.
- Expected number of entity instances and growth rate.

Other properties concern the physical data model and are specific to the target implementation platform of the data model. Examples include:

- List of attributes that should be indexed to optimize access time.
- List of attributes that should be encrypted or compressed.
- Whether the entity should become a database view instead of a table. A view is a virtual table that is defined by a SQL query command on one or more tables.
- Whether the entity should become a materialized view, which means it will be implemented as a database table that stores a subset of the data copied from a master table. Like a regular view, the subset is defined by a query command.
- List of database triggers that will be implemented for that entity. A trigger is a special procedure that is automatically executed by the database management system when data is inserted, updated, or deleted.

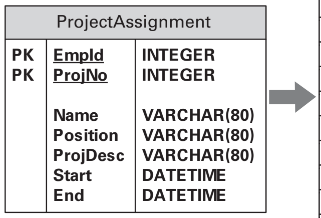
There are three types of relations found in data models:

- *Relationship.* Used to designate a logical association between entities. It is usually qualified by the cardinality of the participant entities: one-to-one, one-to-many, or many-to-many. In addition, a relationship can be *identifying* or *nonidentifying*. An identifying relationship from A to B means that the existence of B depends on the existence of A; that is, the primary key of B contains the primary key of A.

- *Generalization/specialization.* Indicates an *is-a* relation between entities. For example, entity Insurance is a generalization of different types of insurances; at the same time, entities Car Insurance and House Insurance are specializations of entity Insurance.
- *Aggregation.* An abstraction that turns a relationship between entities into an aggregate entity (Smith and Smith 1977). For example, a relationship between a patient, a physician, and a date can be abstracted as an aggregate entity called Appointment. In practice, this relation is rarely used.

Conceptually, there are no topological constraints with respect to the relations in a data model. However, the database normalization technique imposes restrictions on the data model based on the dependencies between entity attributes. Normalization is used by data administrators to avoid duplication of information, in order to safeguard the consistency (integrity) of the data. Figures 2.41 and 2.42 show an example of normalization.

For an explanation of the normalization technique and description of the various normal forms, refer to the classic book by C. J. Date, *An Introduction to Database Systems* (1999).



EmpId	Name	Position	ProjNo	ProjDesc	Start	End
100	Simpson	Analyst	23	DB design	Apr-02	Jul-02
140	Beeton	Technician	14	Network cabling	Sep-02	Oct-02
160	Davis	Technician	14	Network cabling	Sep-02	Nov-02
			36	Network testing	Nov-02	Dec-02
190	Berger	DBA	45	Physical design	Aug-02	Nov-02
			48	Space allocation	Nov-02	Dec-02
100	Simpson	Analyst	25	Reports	Oct-02	Nov-02
110	Covino	Analyst	31	Forms	Mar-02	May-02
			25	Reports	May-02	Jul-02
120	Brown	Analyst	11	Order entry	Jul-02	Sep-02
180	Smith	Programmer	31	Forms	Sep-02	Nov-02
			25	Reports	May-02	Jul-02
200	Rogers	Programmer	11	Order entry	Sep-02	Oct-02
			12	Inventory control	Oct-02	Dec-02
			13	Invoicing	Nov-02	Dec-02
100	Simpson	Analyst	31	Forms	Aug-02	Oct-02
130	Clemens	Analyst	23	DB design	Apr-02	Jun-02

Figure 2.41
Entity ProjectAssignment before normalization, along with sample data (adapted from Ponniah 2007). The attributes that uniquely identify a project assignment (that is, the primary key) are EmpId and ProjNo.

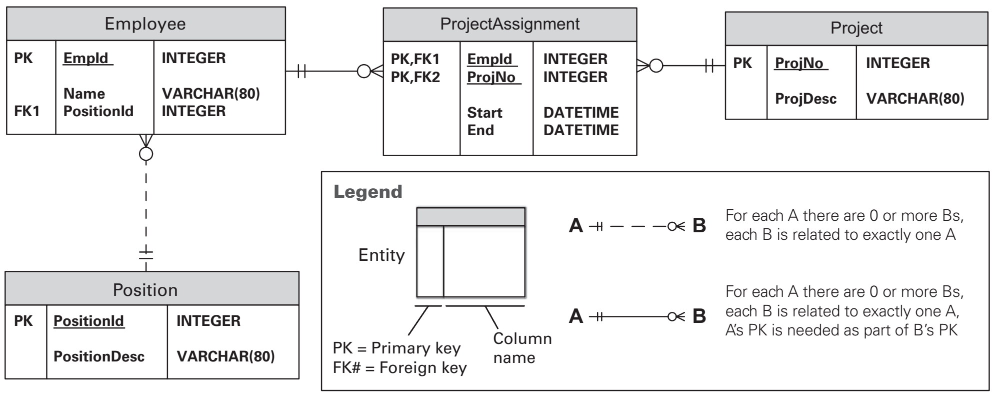


Figure 2.42
Data model for ProjectAssignment after normalization. One of the rules of normalization is that non-key attributes should have functional dependencies to the whole primary key only. Attribute ProjDesc has a functional dependency to ProjNo, which is not the whole primary key. After this and other violations of the normalization rules were fixed, this is the resulting data model diagram.

2.6.3 What the Data Model Is For

The data model facilitates stakeholder communication during domain analysis and requirements elicitation. But foremost, the data model is the blueprint for the implementation of the data entities, for example, in a relational database.

A carefully created data model also helps to achieve performance requirements in a software system. In data-centric applications, access to the data usually represents a significant amount of the time to process user requests. The architect and the data administrator should understand what kinds of data access operations will be more critical to the system and what their performance requirements are. Driven by these requirements, denormalizations, optimizations, and other design decisions are applied to the data model, aiming at improved system performance. Examples of these design decisions include:

- Merging two entities to avoid an expensive outer join or union operation in a query
- Adding a derived attribute to avoid scanning an entire data table to obtain the derived value
- Creating an index on attributes that are often parameters in a query
- Changing the granularity (such as table row or page) and type (such as optimistic) of locks on certain entities to avoid contention and deadlocks

After the software system is implemented, even when the data model is carefully created, it's common to find performance bottlenecks in data access operations. To remove these bottlenecks, the data model comes in handy once again, in a task called query optimization.

In information systems, the data model is essential input to modifiability analysis. To analyze the impact of required modifications to a system, one cannot look exclusively at the code structure. Many modifications require altering the data model and hence its physical implementation. Modifications to the data model can be costly, as they may require changing the code of multiple applications that share the same data. A simple change such as making a certain attribute of an entity mandatory (for example, requiring a customer's date of birth) may require changes to all screens and functions that allow creating or updating that information. Versioning and redeployment of applications is more complicated when data model changes are involved. Moreover, larger data model modifications, such as merging with the data model of a legacy system, may also require the implementation of extract, transform, and load (ETL) operations to fix the data itself. Indeed, the data model is an important input to data warehouse projects and to the integration of data schemas required by some business partnerships (for example, an airline company needs to share data with a car rental company).

The data model is an architecture view that should ideally be created with a thorough understanding of incremental development plans, future extensions, and integration of data across information systems. Data is a valuable asset, and the existence of an enterprise data model and a data administration group helps to enforce **data integrity**. If a new system needs to retrieve sales information, the enterprise data model may already contain that information. The architect of the new system may not be aware of the data entities that hold sales information, but the data administrator should and can point out those entities instead of creating new ones in the database. Disparate, redundant data contribute to poor data quality.

Data integrity refers to the consistency and accuracy of the data shared across all applications in a system.

Based on the data model, data modeling tools can generate scripts to create the physical database. Some tools can also generate application code to access the data tables, classes to hold the data, forms for end users to enter data, message schemas, and simple reports.

Finally, the data model can help application developers to write code to access the database. It is easier to understand an entity-relationship diagram than to browse through the table creation commands or the database management system dictionary.

2.6.4 Notations for the Data Model Style

The data model can be described graphically using informal or semiformal visual notations that include:

- Peter Chen's entity-relationship diagram notation (Chen 1976)
- Crow's foot entity-relationship diagram notation
- IDEF1X
- UML class diagram

The first three notations are ERD variations, and the last one is the UML alternative to ERD. Crow's foot and UML class diagrams are more widely used in industry and more commonly supported by tools.

Crow's Foot ERD Notation

One of the most popular ERD notations for relationships uses lines with special symbols at each end to indicate cardinality. These symbols include a dash (indicating one), a ring (indicating zero), and a crow's foot (indicating many). The crow's foot ERD notation was initially used in the 1980s by Richard Barker (1990), as well as in the Information Engineering approach developed by James Martin and Clive Finkelstein (1981). The symbology found in today's tools provides slight variations on Baker's original notation and the Information Engineering notation. Figure 2.43 shows an example.

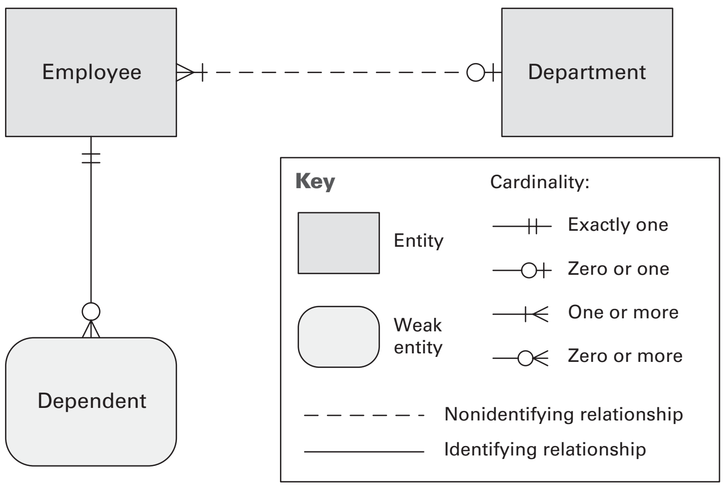


Figure 2.43
Data model (simplified) of a human resource system using crow's foot ERD notation

UML

The data model can be represented as a UML class diagram, where the classes correspond to data entities. The attribute compartment lists the entity attributes, and the operation compartment is empty. UML associations represent the relationships between entities and the multiplicity intervals shown at both ends of the association lines (for example, "1..*") indicate the cardinality of the relationship. Figure 2.44 shows an example.

UML was originally created for object-oriented modeling, not for data modeling. Therefore, it doesn't provide built-in mechanisms for indicating primary keys, weak entities, or foreign keys. In addition, class diagrams are more flexible than ERDs. For example, a class Order may include a list of items as an attribute, whereas in an ERD, Item would naturally be a separate entity. Some constraints are needed in order to use UML class diagrams as an ERD alternative.

The Object Management Group has a draft specification for an Information Management Metamodel that contains a UML 2 profile for entity-relationship modeling. It is available online at omgwiki.org/imm.

2.6.5 Relations to Other Styles

The entities in the data model are intrinsically connected to some of the modules in other module views, especially the modules that contain the in-memory representation of the data. In object-oriented systems that use a relational database to store data, we typically find classes that correspond to the persisted entities. The mapping is not always one-to-one, because the relational paradigm is fundamentally different from the object-oriented paradigm. This problem is known as the object-relational impedance mismatch (Ambler 2006) and is addressed by object-relational mapping (ORM) tools and

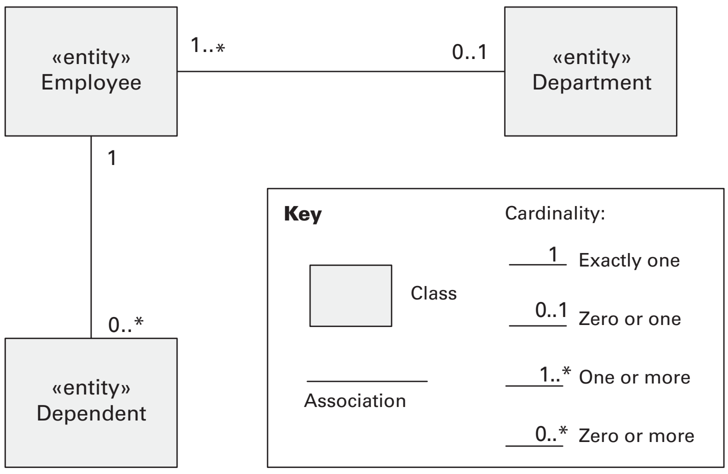


Figure 2.44
Data model (simplified) of a human resource system shown as a UML class diagram

frameworks, such as Hibernate for Java and LLBLGen for Microsoft .NET.

The architect may find it useful to indicate what modules (in a module view), what components (in a component-and-connector view), or even what use cases from the functional requirements do use which data entities. Such mapping of the data model to other views can be recorded as a table, as described in Section 10.2. Moreover, the architect can indicate whether each element creates, reads, updates, or deletes data (CRUD, for short) from each data entity. This generic mapping can be represented as a CRUD matrix.

Mapping between views is discussed in Section 10.2.

The data model describes the structure of data entities and relationships that will typically be deployed to a shared-data-store component such as an Oracle database. Data stores are typically depicted in a shared-data view of the architecture, along with the other runtime components that access them. Also, a deployment view typically shows what machine(s) the data stores are allocated to. Documenting the mapping of entities in a data model to different data stores and respective machines is especially useful when the solution uses distributed or replicated databases.

The shared-data style is covered in Section 4.5.1.

The deployment style is described in Section 5.2.

2.6.6 Examples

Figure 2.45 shows the data model reconstructed and adapted from the Microsoft .NET Pet Shop application (Microsoft 2002), a Web store that keeps a catalog of pets and takes purchase orders from registered Web users. The data is persisted in a relational database. The majority of the functionality consists of retrieving, creating or updating the data elements shown in the data model.

COMING TO TERMS

Entity

The elements in the data model style are data entities or, as most data administrators and developers call them, entities. The original paper that proposed the entity-relationship model initially describes an entity in a purely conceptual way: an entity is a “thing” that can be distinctly identified (Chen 1976). Later in the paper, the author adds a practical caveat: “From now on, we shall consider only the entities and relationships (and the information concerning them) which are to enter into the design of a database.” Thus an entity can be related to any object in the real world: a car, a person, an event, a company, and so on. But for practical reasons, data modeling in general is concerned with only those entities and their respective attributes that are relevant to the software system and hence will be represented in the system, possibly in the database. The same focus is true in the context of software architecture documentation.

Strictly speaking, an entity is a particular instance of an entity set or entity type. For example, Earth is an entity of entity set Planet. For simplicity, most people don't make that distinction and refer to entity sets as entities.

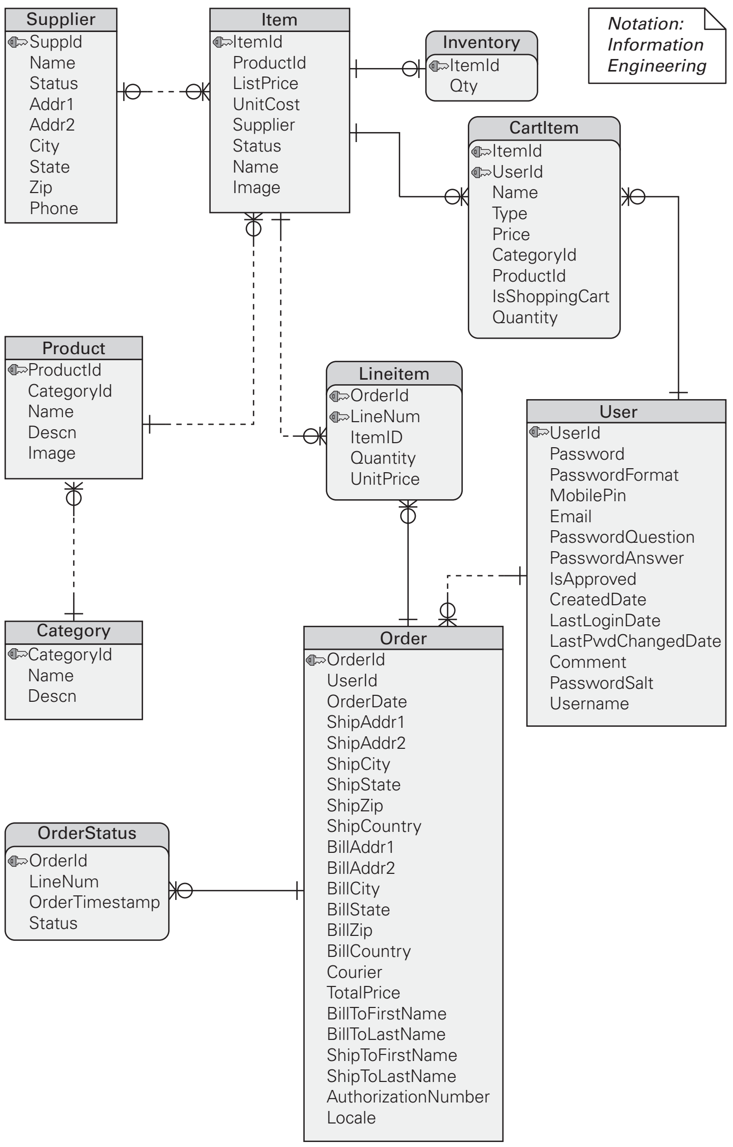


Figure 2.45
Data model for the Pet Shop application using Information Engineering crow's foot ERD notation

2.7 Summary Checklist

- A decomposition view shows how responsibilities are allocated across modules and submodules.
- A uses view shows how modules depend on one another. This view helps achieve incremental development and is especially suitable for performing change-impact analysis.
- A generalization view relates modules by showing how one is a generalization or specialization of the other. This view is widely used in object-oriented systems, where inheritance is used to exploit commonality among modules.
- A layered view divides a system into groups of modules that provide cohesive responsibilities. These groups are called layers and relate to each other unidirectionally by the *allowed-to-use* relation. A layered design helps a system achieve portability and modifiability.
- An aspects view shows special modules called aspects, which are responsible for crosscutting concerns. This view is particularly useful if the system implementation is going to use AOP.
- A data model view describes the structure of the data used in the system in terms of data entities and their relationships. It guides implementation and helps to improve performance and modifiability in data-centric systems.

2.8 Discussion Questions

1. Can you think of a system that cannot be described using a layered view? If a system is not layered, what would this say about its *allowed-to-use* relation?
2. How does a UML class diagram relate to the styles given in this chapter? Does that diagram show decomposition, uses, generalization, or another combination?
3. We consciously chose the term *generalization* to avoid the multiple meanings that the term *inheritance* has acquired. Find two or three of these meanings, compare them, and discuss how each is a kind of generalization. (*Hint*: You may wish to consult books by Booch and Rumbaugh, respectively.)
4. Suppose that your system will include commercial off-the-shelf (COTS) software modules. In which module views might you show them and why?
5. Would you create a data model using an entity-relationship notation for a system that will not contain a database? In what situations and why?

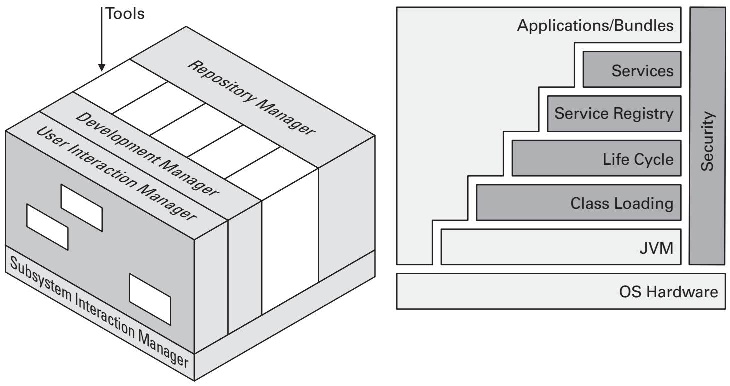


Figure 2.46
ECMA "toaster model" (left) and OSGi framework layered design (right)

6. Crosscutting concerns have been implemented in object-oriented systems without using AOP constructs. Would you create an aspect view for your system if your implementation will not use AOP? Would you use aspect modules in your design in this case?
7. The two layered diagrams in Figure 2.46 are from real systems. The first is called the ECMA "toaster model," which has slots for pluggable tools. The second is the layered architecture of the OSGi framework. How is the *allowed-to-use* relation represented in these diagrams? How would you create the key to each diagram so that any ambiguity in the notation is removed?

2.9 For Further Reading

Most of the styles in this chapter can be traced to a foundational paper in the annals of the software engineering literature. An architect interested in the roots of the discipline may find the original ideas refreshing in their simplicity and purposefulness. These papers, seen as a group, express the then-revolutionary idea that there is more to a computer program than getting the right answer: how it is structured also matters.

In 1968, Edsger Dijkstra wrote about designing an operating system as a set of abstract virtual machines, giving us the concept of layers (Dijkstra 1968). David Parnas showed how decomposing a system into modules based on likely changes, as opposed to steps in the processing, resulted in systems vastly easier to modify (Parnas 1972). Parnas also introduced the *uses* relation and showed how it could lead to software that was easy to extend or to develop incrementally (Parnas and Weiss 1979).

In the 1960s the fundamental concepts of object-oriented programming, including objects, inheritance, and dynamic binding, were invented by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo (Nygaard and Dahl 1981). The concepts were introduced in the programming language Simula-67, which, although never widely used itself, laid the foundation for the development of popular object-oriented languages such as Smalltalk and C++. In 1986–1987, two widely influential papers by Alan Snyder and Barbara Liskov, respectively, tied together two concepts that had been drifting apart: inheritance and encapsulation (Snyder 1986, Liskov 1987). Liskov in particular argued convincingly that undisciplined inheritance that violated objects' abstractions was harmful. Between them, they set the object-oriented community on its present path.

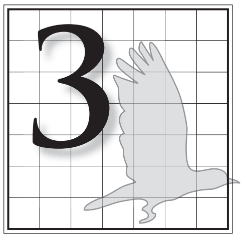
A software engineering demonstration project that paid special attention to the use of separate architectural structures was the A-7E avionics system built by the U.S. Navy in the 1980s. A case study is presented in the book by Bass, Clements, and Kazman (2003). The example employs decomposition (using information hiding as the criterion [Parnas, Clements, and Weiss 2001]), layers, and uses, and it shows how a subset is built from the *uses* relation.

The authoritative source of information about the UML language and notation is the specification published by the Object Management Group (OMG 2009), which at the time of writing is on version 2.2. However, there are many UML books that are far more digestible. Two valuable references are *The Unified Modeling Language User Guide*, by Booch, Rumbaugh, and Jacobson (2005), and *UML Distilled*, by Martin Fowler (2003). UML classes, packages, and their relations are especially relevant to module styles.

The seminal paper on aspect-oriented programming was written by Gregor Kiczales and colleagues at Xerox PARC (Kiczales et al. 1997). It describes the concepts and terminology that were later used to create AspectJ and other AOP languages. The second edition of the book by Ramnivas Laddad (2008) is an excellent guide to AspectJ and provides a nice introduction to AOP. Recently, aspect orientation has been investigated in the realm of domain analysis, requirements engineering, and software architecture. Resources about the use of aspects in early phases of software development can be found at early-aspects.net.

Data modeling is a well-established discipline. Entity-relationship modeling was originally proposed by Peter Chen (1976). In addition to Chen's original paper, the book by C. J. Date (2003) has been an important reference to relational theory, normalization, and data modeling since the publication of the first edition in 1975.

3 Component-and-Connector Views



In this chapter, we look at these aspects of component-and-connector (C&C) views:

- Elements, relations, and properties
- Purpose
- Notation
- Relation to other views

3.1 Overview

In this chapter we discuss C&C views in their most general form, and we look at notations for representing C&C views. In Chapter 4, we explore some important C&C styles.

A C&C view shows elements that have some runtime presence, such as processes, objects, clients, servers, and data stores. These elements are called *components*. Additionally, component-and-connector views include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage. Such interactions are represented as *connectors* in C&C views.

Component-and-connector views are ubiquitous in practice; indeed, box-and-line diagrams depicting these views are often the graphical medium of choice as a principal first-look explanation of the architecture of a system. But such informal C&C views can be misleading, ambiguous, and inconsistent. Some problems follow from the usual pitfalls of visual documentation and are equally applicable to any of the view types discussed in this book. Other problems derive specifically from the use of components and connectors to portray a system's execution structure. In this chapter, we provide guidelines for documenting C&C views, and we highlight common pitfalls.

The primary presentation is the (typically) graphical portion of an architecture view. Documentation that explains the primary presentation is called supporting documentation. Both are described in Chapter 10.

Figure 3.1 illustrates the primary presentation of a C&C view of a system's runtime architecture. What is this diagram (and the documentation that explains it) attempting to convey? It shows a picture of the system as it appears at runtime. The system contains a shared repository of customer accounts (`Account Database`) accessed by two servers and an administrative component. A set of client tellers can interact with the account repository servers, embodying a client-server style. These client components communicate among themselves by publishing and subscribing to events. The purpose of the two servers is to enhance availability: If the main server goes down, the backup can take over. Finally, an administrative component allows an administrator to access and maintain the shared-data store.

The system illustrated in Figure 3.1 is built from an amalgamation of different styles: client-server is described in Section 4.3; the shared-data style is described in Section 4.5; and publish-subscribe is described in Section 4.4. This picture is a result of combining views, which is discussed in Section 6.6.

Each of the three types of connectors shown in Figure 3.1 represents a different form of interaction among the connected parts.

- Client-server connectors allow a set of concurrent clients to retrieve data synchronously via service requests. This variant of the client-server style supports transparent failover to a backup server.
- The database access connector supports transactional, authenticated access for reading, writing, and monitoring the database.
- The publish-subscribe connector supports asynchronous event announcement and notification.

Each of these connectors represents a complex form of interaction and will likely require nontrivial implementation mechanisms. For example, the client-server connector type represents a protocol of interaction that prescribes how clients initiate a client-server session, how and when failover is achieved, and how sessions are terminated. Implementation of this connector will probably involve runtime mechanisms that detect when a server has gone down, queue client requests, handle attachment and detachment of clients, and so on.

Connectors need not be binary. Two of the three connector types in Figure 3.1 can involve more than two participants: the publish-subscribe bus and the failover client-server connectors.

It may also be possible to carry out both qualitative and quantitative analyses of system properties such as performance, reliability, and security based on this view. For instance, the design decision that causes the administrative user interface to be the only way to change the database schema would improve the security of the system. But that decision also might affect serviceability or availability. For example, does the use of

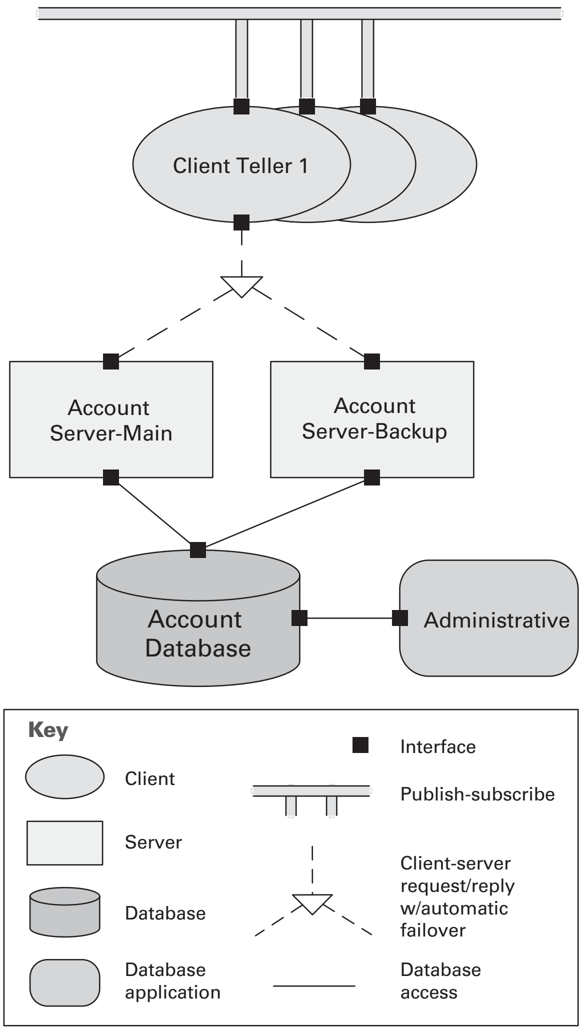


Figure 3.1
A bird's-eye-view of a system as it appears at runtime. This system contains a shared repository that is accessed by servers and an administrative component. A set of client tellers can interact with the account servers and communicate among themselves through a publish-subscribe connector.

the administrative interface lock out the servers? Similarly, by knowing properties about the reliability of the individual servers and the database, you might be able to produce numeric estimates of the overall reliability of the system, using some form of reliability analysis.

Here are some things to note about the nature of C&C graphical documentation, as illustrated in Figure 3.1:

- It acts as a key to the associated supporting documentation (not shown here), where details about the elements, relations, and their properties can be found.

Supporting documentation is discussed in Section 10.1.

- It's restricted to information that can be simply presented in—and comprehended from—a single diagram.
- It's explicit about its vocabulary of component-and-connector types in the diagram's key.
- It indicates the number and kind of interfaces on its components and connectors.
- It uses component-and-connector abstractions that may have rich semantics and complex implementations.

The documentation explaining the diagram should elaborate on the elements shown. Supporting documentation should explain, for example, how `Account Server-Backup` improves the availability of the system. Some of the elements of this Figure 3.1 may themselves represent subsystems that have their own subarchitectures, shown elsewhere.

The combination of C&C diagrams and their supporting documentation provide an essential vehicle for communicating an architect's design intent, supporting reasoning about the runtime behavior of the system, and justifying design decisions in terms of their impact on relevant quality attributes.

3.2 Elements, Relations, and Properties of C&C Views

Table 3.1 summarizes the elements, relations, and properties that can appear in C&C views. It is followed by a more detailed discussion of these concepts, together with guidelines concerning their documentation.

Table 3.1 Summary of C&C views

Elements	• *Components*: principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors). • *Connectors*: pathways of interaction between components. Connectors have a set of roles that indicate how components may use a connector in interactions.
Relations	• *Attachments*: component ports are associated with connector roles to yield a graph of components and connectors. • *Interface delegation*: in some situations component ports are associated with one or more ports in an "internal" subarchitecture. Similarly for the roles of a connector.
Constraints	• Components can be attached only to connectors, not other components. • Connectors can be attached only to components, not other connectors. • Attachments can be made only between compatible ports and roles. • Interface delegation can be defined only between two compatible ports (or two compatible roles). • Connectors cannot appear in isolation; a connector must be attached to a component.

Table 3.1 Summary of C&C views (*continued*)

What C&C Views Are For	• Showing how the system works • Guiding development by specifying the structure and behavior of runtime elements • Helping architects and others to reason about runtime system quality attributes, such as performance, reliability, and availability

3.2.1 Elements

The elements of a C&C view are components and connectors. Each element in a C&C view of a system has a runtime manifestation, consuming execution resources and contributing to the execution behavior of that system. Attachment relations of a C&C view associate components with connectors (via their respective ports and roles) to form a graph that represents a runtime system configuration.

Components

Components represent the principal computational elements and data stores that are present at runtime. Each component in a C&C view has a name. The name should indicate the intended function of the component. The name also allows you to relate the graphical element with any supporting documentation for that component.

Components are the principal computational elements and data stores that execute in a system.

Components have interfaces called **ports**. A port defines a specific point of potential interaction of a component with its environment. A port usually has an explicit type, which defines the kind of behavior that can take place at that point of interaction. A component may have many ports of the same type. In this respect, ports differ from interfaces of modules, whose interfaces are never replicated. For example, a filter might have several input ports of the same type to handle multiple input streams, or a server might provide a number of request ports for client interactions. The database in Figure 3.1 has two ports for two kinds of access.

A **port** is an interface of a component. A port defines a point of interaction of a component with its environment.

You can annotate a port with a number or range of numbers to indicate replication. For example, a port annotated with "[3]" stands for three occurrences of that port. A port annotated with "[0..10]" means that there are from 0 to 10 instances of that port. That form is useful when defining component types, allowing component instances to bind the exact number, or for components that dynamically create new points of interaction.

To indicate multiple ports of the same type in a diagram using an informal notation, you can draw each one separately or you can show a single port but append a bracketed number (for example, [5]) after the port's name to indicate its degree of replication. UML provides a similar convention.

A component's ports should be explicitly documented, by showing them in the diagram and defining them in the diagram's supporting documentation.

See Chapter 7 for a more complete discussion of types of information that can be used to define a port.

Section 6.1 contains more detail on guidelines for documenting hierarchical relationships and refinement.

See Section 3.2.3 for more information on how to document substructure using an interface delegation relation.

A component in a C&C view may represent a complex subsystem, which itself can be described as a C&C subarchitecture. This subarchitecture can be depicted graphically *in situ* when the substructure is not too complex, by showing it as nested inside the component that it refines. Often, however, it is documented separately. A component's subarchitecture may be in a style different from the one in which the component appears.

When a component has such a substructure, you should also document the relationship between the "internal" and "external" ports. As we describe later, this relationship is captured using an interface delegation relation.

Connectors

A **connector** is a runtime pathway of interaction between two or more components.

Connectors are the other kind of element in a C&C view. Simple examples of connectors are service invocation, asynchronous message queues, event multicast, and pipes that represent asynchronous, order-preserving data streams. But as we noted earlier, connectors often represent much more complex forms of interaction, such as a transaction-oriented communication channel between a database server and a client, or an enterprise service bus that mediates interactions between collections of service users and providers.

A **role** is an interface of a connector. A role defines a point of interaction of a connector and indicates how components may use a connector in interactions.

Connectors have **roles**, which are its interfaces, defining the ways in which the connector may be used by components to carry out interaction. For example, a client-server connector might have *invokes-services* and *provides-services* roles. A pipe might have *writer* and *reader* roles. Like component ports, connector roles differ from module interfaces in that they can be replicated, indicating how many components can be involved in its interaction. A publish-subscribe connector might have many instances of the *publisher* and *subscriber* roles.

A protocol specification or a pattern of events can be described using behavioral notations, described in Chapter 8.

A role typically defines the expectations of a participant in the interaction. For example, an *invokes-services* role might require that the service invoker initialize the connection before issuing any service requests. The semantics of the interaction represented by a connector is often documented as a protocol specification prescribing what patterns of events or actions are allowed to take place over the connector.

Refinement is described in Section 6.1.

Like components, complex connectors may in turn be decomposed into collections of components and connectors that describe the architectural substructure of those connectors. For example, a decomposition of the failover client-server connector of Figure 3.1 would probably include components that are responsible for buffering client requests, determining when a server has failed, and rerouting requests.

ADVICE

Connectors

- Connectors need not be binary. That is, they need not have exactly two roles. For example, a publish-subscribe connector (as illustrated in Figure 3.1) might have an arbitrary number of publisher and subscriber roles. Even if the connector is ultimately *implemented* using binary connectors, such as a procedure call, it can be useful to adopt *n*-ary connector representations in a C&C view.
- If a component's primary purpose is to mediate interaction between a set of components, consider representing it as a connector. Such components are often best modeled as part of the communication infrastructure.
- Connectors can—and often should—represent complex forms of interaction. What looks like a semantically simple procedure call can be complex when carried out in a distributed setting, involving runtime protocols for time-outs, error handling, data marshaling, and locating the service provider—for example, as provided by SOAP.
- Connectors embody a protocol of interaction. When two or more components interact, they must obey conventions about order of interactions, locus of control, and handling of error conditions and time-outs. The protocol of interaction should be documented.

3.2.2 Component-and-Connector Types and Instances

The components and connectors depicted in a C&C view are *instances* of component-and-connector *types*. A type is an incompletely defined component or connector. Type definitions often express a set of choices, such as using a multiplicity indicator like [1..5] to indicate that a component may have from 1 to 5 ports.

An instance is the result of completing the definition by binding the choices that the types create. Each instance must conform to its type in terms of behavior, interfaces, substructure (if any), properties, and topological restrictions. As a result of this conformance requirement, all instances of a given type are more or less identical. For example, the type may define a set of allowable behaviors. An instance can restrict this set, perhaps through instantiation parameters, but an instance can't add behaviors.

When documenting a C&C view:

- Make clear in the view what architecture style is being used. Refer the reader to the appropriate style guide for more information about the style.
- Document any additional component or connector type specializations introduced in the view.

It is usually not a good idea to mix types and instances in the same diagram.

A C&C view's primary presentation depicts only instances; no component or connector types should appear in the view's primary presentation. Mixing types and instances in the same diagram is generally ill-advised. Although it may seem convenient ("I'll just add a little inheritance information to clarify a relationship between different instances"), it is more likely to add confusion.

Types are found in style guides. However, type definitions given in style guides, including the ones in this book, are too general to sufficiently constrain an implementation or support useful analysis. It would make no sense to instantiate them as they are without specializing them first. Type definitions like these define the essence of the elements of the style. For example, a style guide for the client-server style will define the component types *client* and *server*, define the connector type *request/reply connector*, and specify how their interfaces differ (for example, that clients make requests of servers, who in turn reply to clients). Such abstract types, however, do not provide any application-specific semantics for the components (for example, whether a server supplies Web pages or processes banking transactions).

Types might specialize more general types in domain-specific ways, such as a controller servlet that takes requests from ATMs in a banking system, or a sensor component type, used in an avionics application. Or they might be technology-specific, such as an ASP.NET component, a Java servlet, an Enterprise JavaBean (EJB), a MySQL database, or a database connector. Like an abstract class in Java, these are usually still too general to drive an implementation or support useful analysis.

A style is a **specialization** of another style if it is consistent with that style—that is, doesn't violate it—and adds more constraints to its element types, relation types, and/or topological restrictions.

Architects need to define application-specific **specializations** of those types that contain enough information so that instances that populate a view can be implemented and analyzed. We'll call these *application-specific types*. Document these types in your view's supporting documentation. Application-specific types provide application-specific semantics, such as a detailed behavior specification (such as showing how a request is processed) or refined interfaces (such as refining the general notion of a "request" with a specific set of request types). The type definition should also characterize the number of interface types (ports for components, roles for connectors) that instances of the type can have.

Document application-specific types you introduce in the view's element catalog, part of a view's supporting documentation. Element catalogs are discussed in Section 10.1.

Component-and-connector types, whether introduced in style guides or as application-specific specializations, are useful to identify elements with common behavior, interfaces, substructure, relations to implementation elements, and so on. Localizing this information in a type definition (as opposed to replicating it across each instance of an implied type) improves understandability and simplifies the overall documentation.

This flow from a style's types to application-specific types to instances constitutes a spectrum of design, which is discussed in Section 6.1.3.

In many cases the use of component-and-connector types allows one to conveniently map a component type (and by extension, all of its instances) to its implementation in a module view. For example, if a set of name-lookup servers in a C&C

view are defined as instances of the NameLookupServer type (a specialization of the client-server style's Server type), one might expect to find a corresponding module that implements the behavior of all instances of such a server. A mapping between some module and the NameLookupServer type would indicate that every instance of the NameLookupServer corresponds to that module.

Mappings between C&C and module views are discussed in more detail in Section 3.5.

ADVICE

Component-and-Connector Types

- When several components or connectors of a view share the same form and behavior (beyond what is specified in the style), define a common, application-specific type for them.
- Define application-specific types that cover all your components. This gives the reader one place to look for all component-and-connector details, rather than sometimes looking at type definitions and sometimes looking at instance information.
- The definition of a component type or connector type should explain the general computational nature and form of each of its instances.
- Application-specific types should provide enough information so that an architecture built from their instances can be correctly implemented and usefully analyzed.
- The component-and-connector types instantiated in a particular C&C view should be explained by referring to the appropriate style guide that enumerates and defines them, or through a catalog of application-specific types defined as part of the architecture.
- The definition of a component or connector type should characterize the number and type of interfaces (ports for components, roles for connectors) that instances of the type can have.
- A C&C view's primary presentation depicts only component-and-connector instances; no component types should appear in the view's primary presentation.
- When mapping between views, map modules to C&C types (not instances).

3.2.3 Relations

The primary relation within a C&C view is *attachment*. Attachments indicate which connectors are attached to which components, thereby defining a system as a graph of components and connectors. Specifically, an attachment is denoted by associating (attaching) a component's port to a connector's role.

A valid attachment is one in which the ports and roles are compatible with each other, under the semantic constraints defined by the style. For example, in a call-return architecture, you should confirm that all "calls" ports are attached to some call-return connector. At a deeper semantic level, you should ensure that a port's protocol is consistent with the behavior expected by the role to which it is attached.

ADVICE

Use the following guidelines when attaching components to connectors:

- You can depict attachment simply by connecting the ports of components in the diagram. In this case, or in any case where the context makes clear what roles are being attached, you don't need to represent roles explicitly in the diagram.
- Attach a connector to a port of a component, not directly to a component.
- If it might not be clear that it is valid to attach a given port to a given role, provide a justification in an annotation in the diagram or in the rationale section for the view.
- Attaching connectors between ports annotated with a multiplicity factor (such as [5] or [0..10]) is a great source of ambiguity. For example, if you connect a port of multiplicity 3 to a port of multiplicity 22, what does that mean? If you connect two ports with the same multiplicity (greater than 1), which ports on one component are connected to which ports on the other? If you use this notation, explain what you mean.

It is possible to establish an interface delegation between two ports of different types. It is also possible to relate multiple internal ports to a single external port. If you do either of these, make sure to explain what that delegation means and why it's valid, in the element catalog entry for that view.

Describing C&C views with UML is covered in detail in Section 3.4.3 and Appendix A.

A second kind of relation is *interface delegation*. When a component or connector has a subarchitecture, it is important to document the relationship between the internal structure and the external interfaces of that component or connector. The relationship can be documented using interface delegation relations. Such relations map internal ports to external ports (for components) or internal roles to external roles (for connectors). Some notations provide specific graphical elements to characterize this relationship. Figure 3.2 shows an example of interface delegation in UML notation. UML "delegation connectors" are used to represent interface delegation.

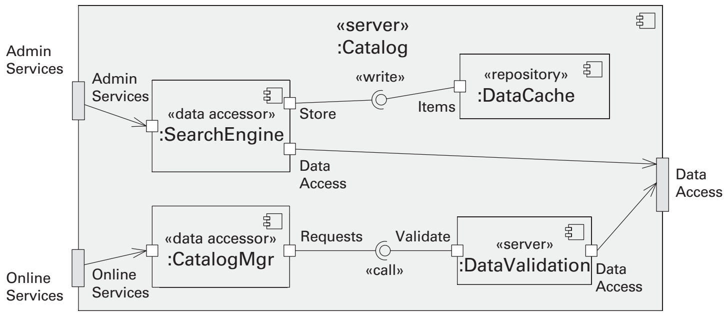


Figure 3.2
A UML component diagram showing the subarchitecture of a component called `Catalog`. UML delegation connectors associate the ports of `Catalog` with the ports of internal components.

3.2.4 Properties

An element (component or connector) of a C&C view will have various associated properties. Every element should have a name and type. Additional properties depend on the type of component or connector. The properties are needed to guide the implementation and configuration of components and connectors, but the architect should also define values for the properties that support the intended analyses for the particular C&C view. For example, if the view will be used for performance analysis, latencies, queue capacities, and thread priorities may be necessary. The following are examples of some typical properties and their uses:

- *Reliability.* What is the likelihood of failure for a given component or connector? This property might be used to help determine overall system reliability.
- *Performance.* What kinds of response time will the component provide under what loads? What kinds of latencies and throughputs can be expected for a given connector? This property can be used with others to determine system properties such as response times, throughput, and buffering needs.
- *Resource requirements.* What are the processing and storage needs of a component or a connector? This property can be used to determine whether a proposed hardware configuration will be adequate.
- *Functionality.* What functions does an element perform? This property can be used to reason about overall computation performed by a system.
- *Security.* Does a component or a connector enforce or provide security features, such as encryption, audit trails, or

authentication? This property can be used to determine system security vulnerabilities.

- *Concurrency.* Does this component execute as a separate process or thread? This property can help to analyze or simulate the performance of concurrent components and identify possible deadlocks.
- *Tier.* For a tiered topology, what tier does the component reside in? This property helps to define the build and deployment procedures, as well as platform requirements for each tier.

Tiers are defined and discussed in Section 4.6.2.

Ports and roles also may have properties associated with them. For example, maximum sustainable request rates may be specified for a server port.

ADVICE

To illustrate what *not* to do, Figure 3.3 presents an example of a poorly documented C&C view diagram.

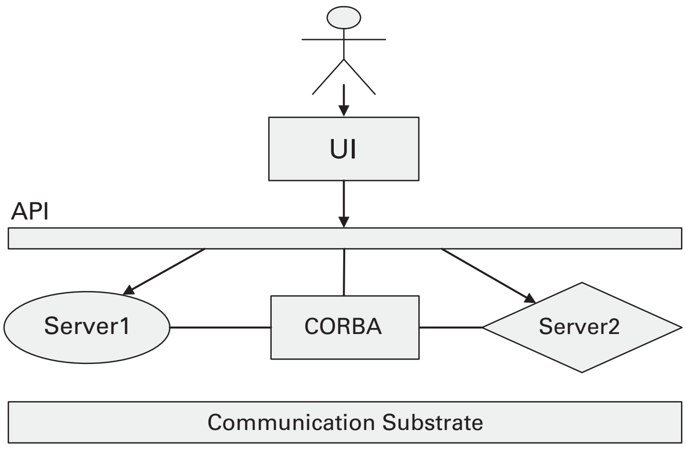


Figure 3.3
A poorly documented C&C view diagram. It does not have a key; it portrays an interface (assuming that "API" has the common meaning of an interface) as a component; it uses different shapes for the same type of component; it uses the same shape for different types of components and connectors; it confuses the context with the system to be built; its use of arrows is not explained; and its components do not have ports.

PERSPECTIVES

Are Complex Connectors Necessary?

In this book we treat connectors as first-class design elements for documenting runtime-oriented views: Connectors can represent complex abstractions; they have types and interfaces, or roles; and they require detailed semantic documentation. But couldn't one simply use a mediating component for a complex connector? For example, in Figure 3.4, the complex connector `Connector 1` gets replaced by the component `Component 1` and two (presumably) simpler connectors. For instance, `Connector 1` might be a pipe that implements buffered data flow between components. On the other hand, `Component 1` might be a buffer, and `Connector 1A` and `Connector 1B` might be simple procedure calls to read or write to the buffer.

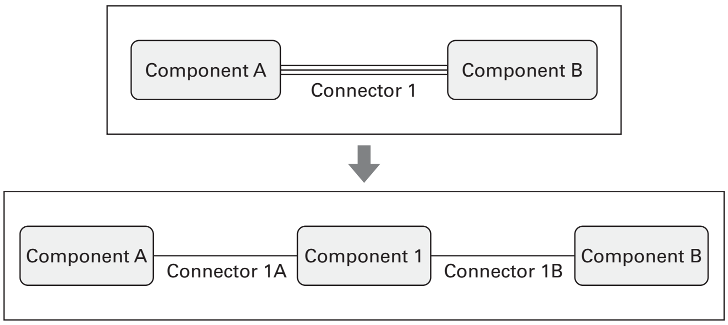


Figure 3.4
A complex connector and the alternative of representing it as a component with two simpler connectors

In other words, are complex connectors needed? The answer is yes. Here's why.

First, complex connectors are rarely realizable as a single mediating component. Although most connector mechanisms do involve runtime infrastructure that carries out the communication, that is not the only thing involved. In addition, a connector implementation requires initialization and finalization code; special treatment in the components that use the connector, such as using certain kinds of libraries; global operating system settings, such as registry entries; and others.

Second, use of complex connector abstractions often supports analysis. For example, reasoning about a data flow system is greatly enhanced if the connectors are pipes rather than procedure calls or another mechanism, because well-understood calculi are available for analyzing the behavior of data flow graphs. Additionally, allowing complex connectors provides a single home where one can talk about their semantics. For example, in Figure 3.4, I could attach a single description of the protocol of interaction to the complex connector. In contrast, the lower model would require me to combine the descriptions of two connectors and a component to explain what is going on.

Third, using complex connectors helps convey an architect's design intent. When components are used to represent complex connectors, it is often no longer clear which components in a diagram are essential to the application-specific computation and which are part of the mediating communication infrastructure.

Fourth, complex connector abstractions can significantly reduce clutter in an architecture model. Few would argue that the lower of the two diagrams in Figure 3.4 is easier to understand. Magnify this many times in a more complex diagram, and it becomes obvious that clarity is served by using connectors to encapsulate details of interaction.

—D.G.

3.3 What C&C Views Are For

Component-and-connector views are commonly used to show developers and other stakeholders how the system works. The C&C views (with associated behavior documentation) specify the structure and behavior of the runtime elements. In particular, these views allow you to answer questions, such as the following:

It's a good idea to provide comprehensive behavior documentation for each component (or component type). Each such model documents the possible behaviors of a component. When combined with the topological information in a C&C view, you can trace possible behaviors throughout the system, rather than just within a component.

- What are the system's principal executing components, and how do they interact?
- What are the principal shared-data stores?
- Which parts of the system are replicated, and how many times?
- How does data progress through a system as it executes?
- What protocols of interaction are used by communicating entities?
- What parts of the system run in parallel?
- How can the system's structure change as it executes?

Component-and-connector views are also used to reason about runtime system quality attributes, such as performance, reliability, and availability. In particular, a well-documented view allows architects to predict overall system properties, given estimates or measurements of properties of the individual elements and interactions. For example, to determine whether a system can meet its real-time scheduling requirements, you usually need to know the execution time of each process component (among other things). Timing behavior such as this would be represented as properties of the elements. Similarly, documenting the reliability of individual elements and communication channels supports an architect when estimating or calculating overall system reliability. In some cases, analyses such as these are supported by formal, analytical models and tools. In others, it is achieved by judicious use of rules of thumb and past experience.

PERSPECTIVES

Choosing Connector Abstractions

If you've committed to a particular C&C style, then the types of connectors to use in documenting a C&C view are already prescribed. But in other cases the architect has some freedom to determine what kinds of connectors to use and how to represent them in documentation. This choice often revolves around how much implementation structure to expose. On the one hand, a connector might be used to encapsulate a complex interaction as a single abstraction. On the other hand, a complex form of interaction can be represented as a set of components and connectors that implement it.

The publish-subscribe style is described in Section 4.4.1.

To illustrate, consider two ways of documenting a publish-subscribe system shown in Figure 3.5. The first version shows five components communicating through an event bus, which describes an interaction that ensures that each published event is delivered to all subscribers of that event. The second version shows the same five components communicating with the assistance of a centralized dispatcher component responsible for distributing events via procedure calls to the other components.

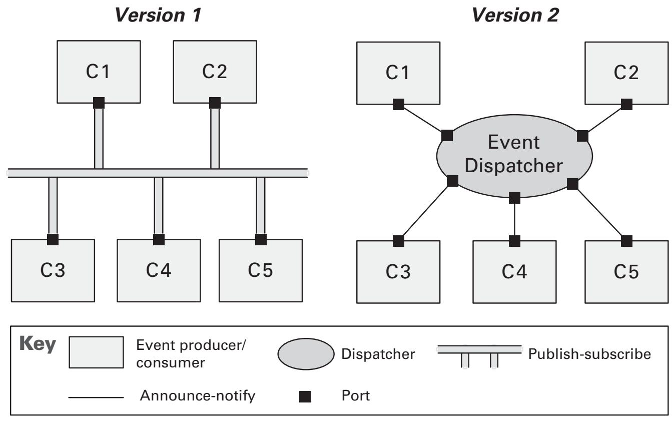


Figure 3.5
Two potential versions of a publish-subscribe system. In Version 1, all communication takes place over an event bus; in Version 2, communication occurs with the assistance of a dispatcher component.

There are several advantages to using the first approach:

- It simplifies the description, since there are fewer elements in the view.
- It clearly distinguishes the parts of the architecture that are used for interaction (the connectors) and the parts that are used to provide the computational functions of the system (the components).
- It permits a variety of implementations to be used to effect the event-based interactions. For instance, instead of a single dispatcher, there could be several, or alternatively each component could be responsible for sending its events to the required listeners.
- It provides a natural way to decompose documentation into multiple views, where the specific implementation would be represented in its own view as a refinement of the event bus connector.

Refinement is discussed in Section 6.1.

On the other hand, the second approach has some advantages:

- It clearly indicates what kinds of mechanisms are being used to carry out event announcement.
- It may better support reasoning about runtime properties, such as delays, order guarantees, and so on, where knowledge of the specific mechanisms for dispatch is needed.
- It fits with what your chosen notation allows: For instance, because UML does not provide a way to represent rich connectors, we are forced to adopt the second approach.

Thus the choice of connector abstraction will depend on taste, needs for analysis, and the amount of implementation detail known to the architect when the architecture is documented. In practice, however, documentation usually errs on the side of putting in too much detail, using low-level communication mechanisms and additional components instead of defining the higher-level interaction abstractions that they represent.

—D.G.

3.4 Notations for C&C Views

3.4.1 Informal Notations

As always, box-and-line drawings are available to represent C&C views. Figure 3.1 is an example of a C&C diagram that uses an informal notation (explained in the diagram's notation key). Although informal notations can convey limited semantics, following some guidelines can lend rigor and depth to the descriptions. The primary guideline is to assign each component type and each connector type a separate visual form (symbol), and to list each of the types in a key.

Beyond just naming the types, however, their meaning should be specified. For example, Figure 3.1 shows a connector of type Publish-Subscribe, but the diagram does not show the connector's capacity, the type of data it can transmit, whether or not delivery is guaranteed, or a host of other important considerations. These details can be documented in the style guide in which the type is defined, or as properties in the C&C view's element catalog.

Element catalogs document the architecture elements that appear in a view. They are discussed in Section 10.1.

Take special care with connectors. A common source of ambiguity in most existing architecture documents is the meaning of connectors, especially ones that use arrows as their visual symbol. Make sure to say what the arrow's direction means.

See "Perspectives: Quivering at Arrows" on page 41, in the prologue.

3.4.2 Formal Notations

Most, if not all, architecture description languages (ADLs) can be used to describe component-and-connector types, constraints on topologies of component-and-connector graphs, and properties that can be associated with the elements of the graph. Tools may then process an architecture description by referring to the meanings of the types, the constraints, and the properties. For example, some ADL-associated tools can tell you if a set of processes can be scheduled so that, given the resources of the CPU, they will all meet their processing deadlines.

Consider the following criteria if selecting an ADL: How standardized is it? What analysis or code generation does it enable? Does it lend itself only to representing certain styles, and if so, are those styles the ones you need for your architecture? Will it let you represent all of the views of the architecture that you need? Is it extensible? How robust are its tools? Is it commercially supported? Is there a large and active user community with whom you can interact?

3.4.3 Semiformal Notations: UML

This section introduces some basic UML modeling constructs for representing components and connectors. Appendix A goes into more depth about using UML to represent other facets of architecture.

Components in UML

UML components are a good semantic match to C&C components because they permit intuitive documentation of important information such as interfaces, properties, and behavioral descriptions. UML components also distinguish between component types and component instances, which is useful when defining view-specific component types.

Section 3.2.2 discusses types and instances of components and connectors.

Because C&C components that appear in a view are instances, they should be represented using UML component instances, as shown in Figure 3.6. The visual distinction between UML component types and instances is found in the naming convention. Names that do not include a colon (:) are types; names that include a colon are instances, with the instance name appearing to the left of the colon. Anonymous instances, such as the instance of `Account Database` in Figure 3.6, are shown by starting the name with a colon.

The element catalog of an architecture view provides information about the elements in that view. Element catalogs are described in Section 10.1.

You can define a component type in a UML diagram in a style guide you're writing or in a view's element catalog for a view-specific type. You should specify attributes common to all instances on the component type. If creating a view-specific type, you should link the type definition to a type defined in your style guide, such as by placing a stereotype on the type definition, as shown in Figure 3.7.

UML ports are a good semantic match to C&C ports. A UML port can be decorated with a multiplicity, as shown in the left portion of Figure 3.8, though this is typically done only on component types. The number of ports on component instances, as shown in the right portion of Figure 3.8, is typi-

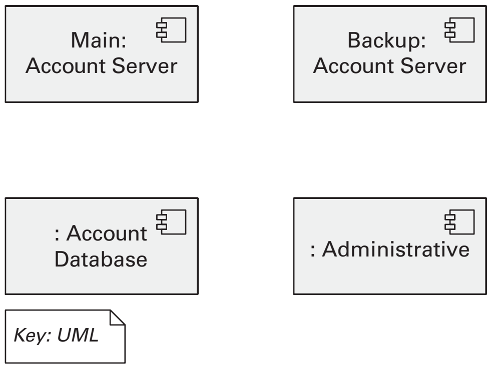


Figure 3.6
A UML representation of a portion of the C&C view originally presented in Figure 3.1. This fragment only shows how four components are represented in UML. `Main` and `Backup` are instances of the same component type (`Account Server`).

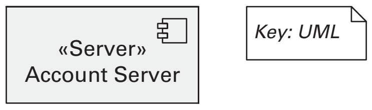


Figure 3.7
A UML representation of a C&C component type. The `Account Server` component type is a specialization of the Server component type from the client-server style (see Section 4.3.1).

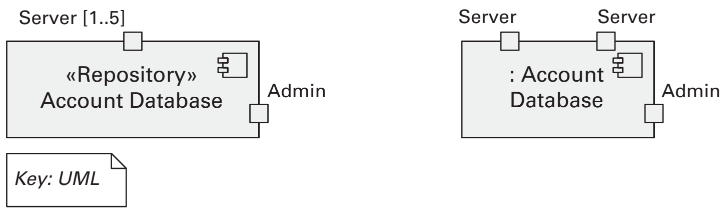


Figure 3.8
A UML representation of the ports on a C&C component type (left) and a component instance (right). The `Account Database` component type has two types of ports, `Server` and `Admin` (denoted by the boxes on the component's border). The `Server` port is defined with a multiplicity, meaning that multiple instances of the port are permitted on any corresponding component instance.

cally bound to a specific number. Components that dynamically create and manage a set of ports should retain a multiplicity descriptor on instance descriptions.

UML provides a lollipop/socket notation for showing provided and/or required interfaces attached to ports. Each port can have an arbitrary number of provided and required interfaces. Figure 3.9 shows the same components in Figure 3.8, but now each port of the `Account Database` type includes one provided interface (the lollipop), which can be further elaborated in UML by supplying additional information, such as methods or attributes. The instance of `Account Database` on the right has exactly two `Server` ports, and the interfaces are omitted.

The lollipop/socket notation of UML can be confusing if not used carefully. If the style of connector interaction is some form of call-return, then the lollipop and socket correspond to calls that are provided and required, respectively. In a client-server connector, a single port might provide and require something at the same time, in which case you would adorn the same port with both a lollipop and a socket. But in other cases, where "provides" and "requires" are the wrong intuition, the notation should be avoided. In a pipe-and-filter system, for example, what does a filter interface "provide" and what does it "require?" In that case, just document the port by itself.

Even where appropriate, you normally omit lollipops and sockets from a C&C view (which shows instances) and use them only on the component type definitions. Often, full interface

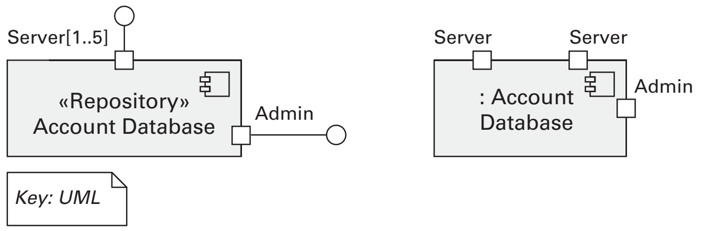


Figure 3.9
Each port on the `Account Database` type now includes one supplied interface (the lollipop), which can be further elaborated in UML by supplying additional information, such as methods or attributes. The instance of `Account Database` on the right has exactly two `Server` ports, and the interfaces are omitted.

The primary presentation is the (typically) graphical portion of an architecture view, as described in Chapter 10.

details will be provided with a component type definition, and only ports will be shown in a C&C primary presentation. This reduces visual clutter without losing the instances' precise interface definitions.

Connectors in UML

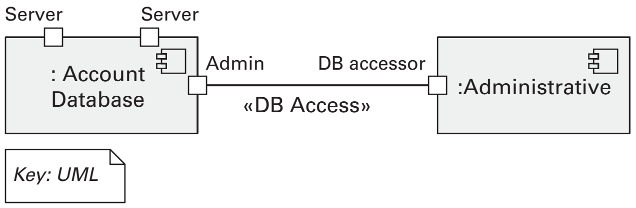
While C&C connectors are as semantically rich as C&C components, the same is not true of UML connectors. UML connectors cannot have substructure, attributes, or behavioral descriptions. This makes choosing how to represent C&C connectors more difficult, as UML connectors are not always rich enough.

You should represent a "simple" C&C connector using a UML connector—a straight line. Many commonly used C&C connectors have well-known, application-independent semantics and implementations, such as function calls or data-read operations. If the only information you need to supply is the type of the connector, then a UML connector is adequate. Call-return connectors can be represented by a UML assembly connector, which links a component's required interface (socket) to the other component's provided interface (lollipop). You can use a stereotype to denote the type of connector. If all connectors in a primary presentation are of the same type, you can note this once in a comment rather than explicitly on each connector, to reduce visual clutter. Attachment is shown by connecting the endpoints of the connector to the ports of components. Figure 3.10 illustrates some of these points.

Connector roles cannot be explicitly represented with a UML connector because the UML connector element does not allow the inclusion of interfaces (unlike the UML port, which does allow interfaces). The best approximation is to label the connector ends and use these labels to identify role descriptions that must be documented elsewhere.

If you also need to supply simple descriptive information, such as attribute-value pairs, attach it to a UML connector by using tagged values or a comment.

Figure 3.10
A UML representation of a "simple" C&C connector between two components. The type of the connector is noted by a stereotype (<<DB Access>> in this case).



You should represent a "rich" C&C connector using a UML component, or by annotating a straight-line UML connector with a tag or other auxiliary documentation that explains the meaning of the complex connector.

Figure 3.11 shows an example of representing a C&C connector using a UML component. In this approach, roles are represented using UML ports. Attachment relations are represented by attaching the UML ports of the components and the connector using a UML connector. Although it's not ideal to use the same graphical convention as for a C&C component, it is sometimes necessary in UML.

See "Perspectives: Are Complex Connectors Necessary?" on page 135, in this chapter.

Sometimes it is better to use a straight line (possibly stereotyped) with a tag that explains the complex connector. For example, suppose you have ten clients, each of which is talking over the same nontrivial asynchronous protocol to some server. Introducing ten extra components would make for a lot of clutter, when a stereotyped straight-line connector would be at least as clear.

A C&C Primary Presentation in UML

The C&C primary presentation found in Figure 3.11 is an example of a combined view that combines the client-server, publish-subscribe, and shared-data styles presented in Chapter 4. Figures 3.12 and 3.13 show how to represent the same information using UML.

Figure 3.12 defines the component-and-connector subtypes that are view specific. Each type uses a UML stereotype to identify the corresponding component or connector type defined in one of the three cited style guides. Multiplicities are attached

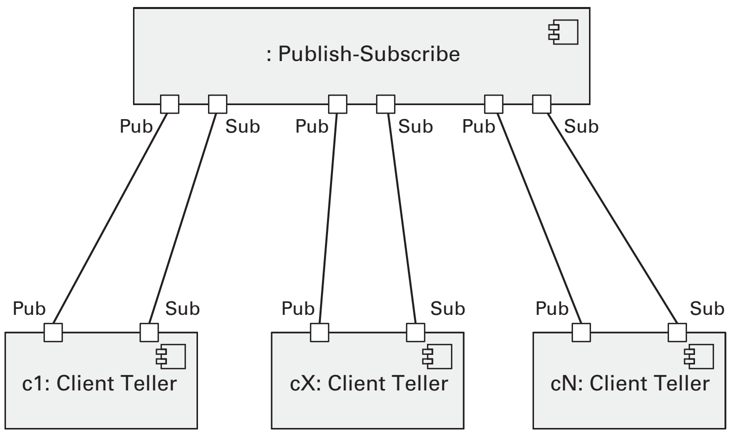


Figure 3.11
A UML representation of a "rich" C&C connector used to connect three components. The Publish-Subscribe connector is represented using a UML component. Its roles are represented using UML ports. Attachments between C&C ports and roles is represented using UML connectors between the respective UML ports.

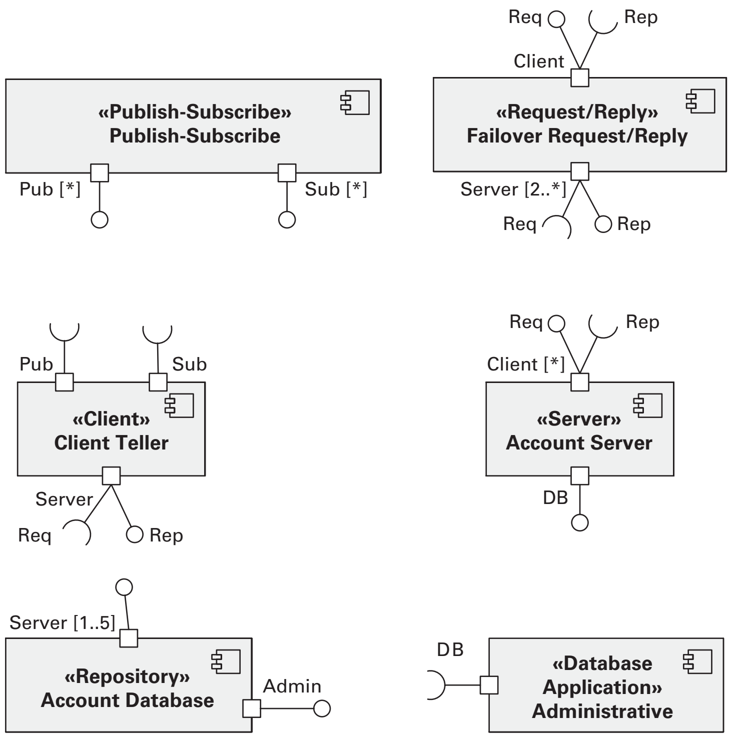


Figure 3.12
A UML representation of component-and-connector types for Figure 3.11. Each type uses a stereotype to link the view-specific subtypes to the types defined in the style guides.

to some of the ports to note where multiple connections are permitted and to set bounds on the number of connections. This information should be in the view's element catalog.

Figure 3.13 shows the view's primary presentation, as represented using UML. Like the `Publish-Subscribe` connector, the `Failover Request/Reply` connector is represented using a UML component; this allows the details of the failover semantics to be formally documented, and it simplifies the representation of an *n*-ary connector.

In addition to the advice presented on representing basic C&C concepts in UML, we had to decide how to represent the implied variability from Figure 3.11. That figure gives the intuition of a variable number of `Client Teller` components, any of which may be connected to one or both of the `Account Server` components at some point in time.

Using a semiformal notation like UML forces us to be more precise about the meaning that was largely implied in the informal version. Representing a variable number of components is not easy using a UML instance diagram. We opted for a naming convention of using `Client Teller` components `c1`, `cX`,

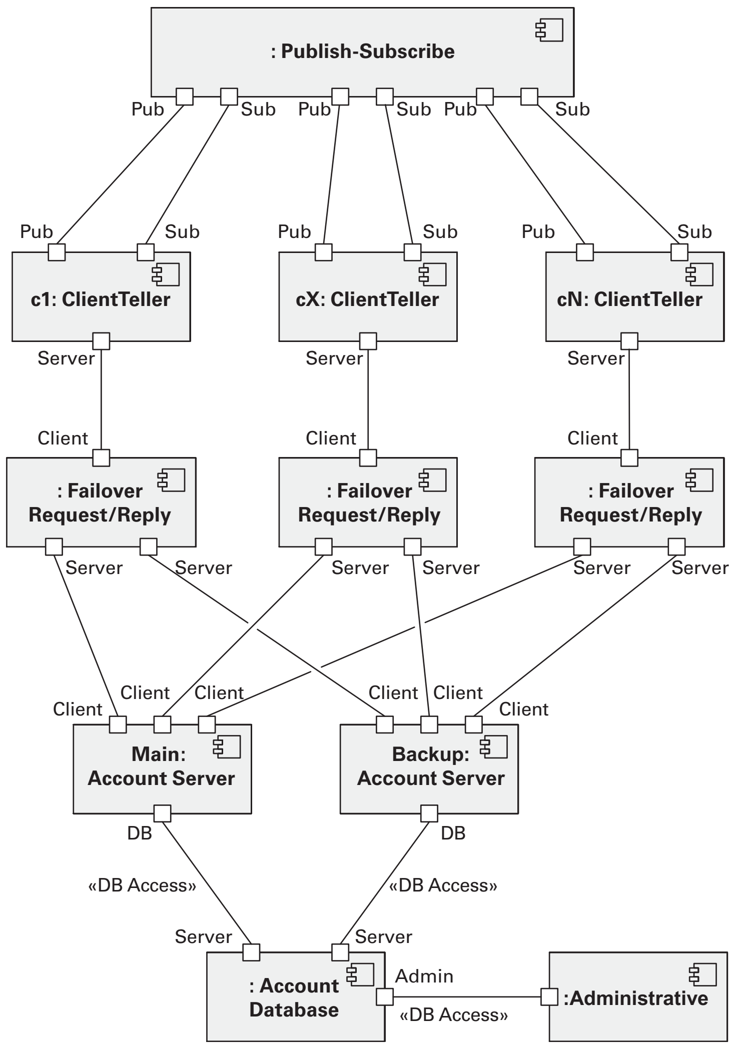


Figure 3.13
A UML representation of the primary presentation found in Figure 3.11

and cN to fill in for an arbitrary number of clients (1..N). The meaning of this convention would have to be documented in the view, as it is not a standard UML convention.

UML contains many of the right modeling elements to document C&C components in an intuitive way, but it suffers from visual blandness. Where an informal C&C notation could use different shapes for different component types to highlight important distinctions, all UML component types are graphically depicted using the same rectangular box. UML permits such visual customization in theory, but tool support is lacking. Similarly, different types of connectors cannot be quickly distinguished by, for example, noting different line conventions; instead, the reader must distinguish between textual descriptions on lines or in boxes, which also tends to introduce visual clutter.

ADVICE

UML for C&C

- Use UML components and ports to model C&C components and ports.
- Always show a component's ports explicitly, even though UML doesn't require it.
- Use a <<stereotype>> to indicate the type of a component or connector instance in a view, if that type was defined in a style guide. If the type is specific to a view, its name appears after the colon in the instance name.
- Represent a simple C&C connector with a straight-line UML connector or (if it's a call-return connector) with a UML assembly connector (a lollipop/socket pair).
- Represent a more complex C&C connector as a UML component, possibly with substructure, or with a straight-line UML connector annotated by a tag that explains the meaning of the connector.
- Use the lollipop/socket connector in UML only for call-return connectors. Avoid it otherwise.
- Don't attach connectors directly to a component; attach connectors to a specific port of a component.

PERSPECTIVES

Data Flow and Control Flow Models

Two representations that have long been used to document software systems—for so long, in fact, that we might consider them archaic today—are data flow and control flow models. These models show how data and control flow through a system during execution. Remember data flow diagrams from Structured Analysis? They're an example, and probably the best known example, of a notation for a data flow model. Going back still farther in time, flow charts are a notation for control flow models. Once ubiquitous forms of software documentation, both have receded in usage, but they can still be found in pockets of practice. Many software engineers trained in, for example, data flow diagrams see similar-looking C&C views of architecture and ask "What's the difference?"

Plenty. First, if the nodes in the diagrams are not architecture elements—pieces of programs, for example—then the diagrams are simply not architectural. But what if the elements shown are architectural elements? Then they can be said to be architecture diagrams, but they are still not full-fledged architecture views. A C&C view would show ports, feature-rich connectors with specified connector protocols, behavioral and interface documentation, mechanisms for variability, and design rationale.

The flow chart is a most thoroughly oversold piece of program documentation. . . . The detailed blow-by-blow flow chart . . . is an obsolete nuisance suitable only for initiating beginners into algorithmic thinking.

—Fred Brooks, *The Mythical Man-Month* (1995)

Both data flow models and control flow models can be seen as derivatives of a corresponding C&C view. You can derive a data flow model from a C&C view by examining the connector protocols to determine in which direction data can flow between components, then replacing the C&C connectors that carry data with simple one- or two-headed arrows indicating flow of data and eliminating C&C connectors that don't carry data. You can take a similar approach to deriving a control flow model.

But why would you? First of all, replacing connectors with arrows isn't as easy as it sounds; see "Perspectives: Quivering at Arrows" in Section P.5 for a discussion of the difficulties associated with even a simple connector. Now imagine replacing a complex connector with an arrow when that connector involves exception handling, timeouts, callbacks, or multistage negotiated protocols.

See "Perspectives: Quivering at Arrows" on page 41, in the prologue.

Second, for all but the simplest architectures, it's hard to imagine you'd want an architecture document to contain the derived models but not their full-fledged C&C view counterparts. Granted, a data flow model or a control model highlights only certain aspects of a view in order to simplify discussion or to focus on specific properties, but those properties can be highlighted in the full view. And keeping them separate means having more documentation to maintain, because it's unlikely that a tool will keep the view and the derived model consistent with each other; you'll have to do that manually when either changes.

Third, for most analysis that you'd want to perform using a data flow or control flow model, you're going to need the information in the full C&C view that those models throw away. For example, control flow diagrams are useful for tracking down bugs in a federation of components. But so are the protocol specifications that dictate how those components interact.

Data flow and control flow models are only architectural if their nodes are architecture elements. But even if they are, they are at best only shadows of full-fledged architecture views. Think carefully before you invest in creating and maintaining them.

—D.G. and P.C.

3.5 Relation to Other Kinds of Views

Component-and-connector views differ from module views in fundamental ways. In particular, the elements of a C&C view represent instances of runtime entities, whereas the elements of a module view represent implementation entities. For example, consider a system that has 10 identical clients connected to a single server. That's 11 components and 10 connectors—but exactly 2 modules (assuming the simplest mapping between views).

An important consideration is how to relate the C&C and module views of a system. Often, the relationship between a system's C&C views and its module views may be complex.

- The same code module might be executed by many of the elements of a C&C view.
- A single component of a C&C view might execute code defined by many modules.
- A C&C component might have many points of interaction with its environment, each defined by the same module interface.
- Since not every module is necessarily shown in every module view, a component in a C&C view may not map to any module in a particular module view at all.

Figure 3.14 shows both a module view and a C&C view of the same system:

- The module view represents a typical implementation that one might find using the C programming language. In this view, the relation between modules is *uses*, as described in Chapter 2. The module `main` starts things off, using the facilities of four modules—`To-upper`, `To-lower`, `Split`, and `Merge`—

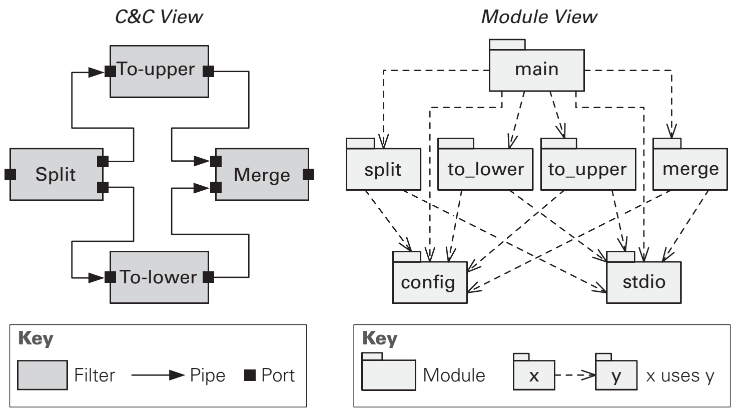


Figure 3.14
Component-and-connector and module views of a simple system that accepts a stream of characters as input and produces a new stream of characters identical to the original but with uppercase and lowercase characters alternating

that do the bulk of the work. The `main` module determines how inputs from one are fed to others, using a configuration module, `Config`. `To-upper`, `To-lower`, `Split`, and `Merge` use a standard I/O library (`stdio`) to carry out the communication. Note that from a code perspective, those worker modules do not directly use the services of one another, but rather do so via the I/O library.

- The C&C view shows the same system described in the pipe-and-filter style. Each of the components is a filter that transforms character streams. Pathways of communication between the components are explicit, indicating that during runtime, the pipe connectors will mediate communication of data streams among those components.

The pipe-and-filter style is described in Section 4.2.1.

The mapping between these two views is illustrated in Table 3.2. It shows which modules contribute to the implementation of which C&C elements. As you can see, there is an *m*-to-*n* relationship for many of the elements of each view.

The correspondence between the elements in a system's module views and the elements in its C&C views should be documented as part of the documentation that applies to more than one view. This mapping between views is described in Section 10.2.

Table 3.2 Mapping between module and C&C views for the example in Figure 3.14

C&C View	Module View
System as a whole	main
Split	split, config, stdio
To-lower	to_lower, config, stdio
To-upper	to_upper, config, stdio
Merge	merge, config, stdio
Each pipe	stdio

In many situations, however, module and C&C views have a more straightforward relationship. Indeed, systems that have natural correspondences between these two kinds of views are often much easier to understand, maintain, and extend. Here are two examples:

- Each component has a type that can be associated with an implementation module, such as a class. In this case the name of the component type will typically be taken to be the same as the corresponding module, making it trivial to relate the two views.
- Each module has a single runtime component associated with it, and the connectors are restricted to *calls procedure* connectors. This would be the case for an object-oriented implementation in which each class has a single instance.

Deployment views are discussed in Section 5.2.

In addition to relations between C&C views and module views, there is often a close correspondence between C&C views and deployment views. Because C&C views represent runtime elements, it is useful to relate these elements to the physical platforms and communication channels on which they execute using an allocation view.

3.6 Summary Checklist

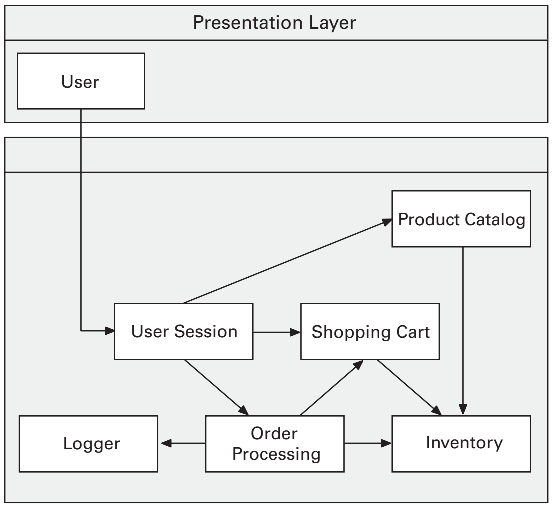
- Component-and-connector views describe structures consisting of elements that have runtime presence, such as processes, objects, clients, servers, and data stores. Additionally, C&C views include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage.
- Component-and-connector views show instances, not types. Style-specific types are defined in a style guide; application-specific types are described in the view documentation.
- Components have interfaces, which are called ports.
- Connectors have interfaces, which are called roles.
- Connectors need not be binary: they may have more than two roles.
- If a component's primary purpose is to mediate interaction between a set of components, consider representing it as a connector instead.
- Connectors can, and often do, represent complex forms of interaction. What seems to be a semantically simple procedure call can be complex when carried out in a distributed setting, involving runtime protocols for time-outs, error handling, and locating the service provider.

- Be clear about which style you are using, by referring to an appropriate style guide.
- Where helpful, define component-and-connector types specific to the view as specializations of the types defined in the corresponding C&C style. These can help indicate semantic relations between similar components, and to establish correspondence between the module types that implement their functionality.
- Always show a component's ports explicitly. Always attach a connector to a port of a component, not directly to a component.
- If it is not clear that it is valid to attach a given port with a given role, provide a justification in the rationale section for the view or mark the attachment to be revisited later.
- Make clear which ports are used to connect the system to its external environment.
- Data flow and control flow models are best thought of as projections of C&C views, but they are not views. When creating such models, be explicit about the semantic criteria used to determine where the arrows go. Data flow and control flow arrows are at best approximations to the connectors, which define more completely the components' interactions.
- It is often important to understand the mapping between components in a C&C view and their respective implementation units in module views. In general, this mapping is many-to-many.
- You can document a C&C style using a spectrum of formality, from informal box-and-line depictions to fully formal, analyzable descriptions. UML is an example of a semiformal notation for representing C&C styles.

3.7 Discussion Questions

1. It is said that a C&C view illustrates a system in execution. Does this mean that it shows a snapshot of an execution, a trace of an execution, the union of all possible traces, some combination, or something else?
2. As we have mentioned, *component* is an overloaded term. Discuss the relationship between a component in a C&C view and (a) a UML component and (b) a component in the sense of the component-based software engineering community.
3. A communication framework, such as enterprise service bus (ESB), CORBA, or COM, can be viewed as a connector

Figure 3.15
An overview architecture diagram. Where is it misleading? What questions does the diagram fail to answer?



among components or as a component with its own substructure. Which is appropriate, and why?

4. Figure 3.15 shows an overview architecture diagram for an electronic commerce store. Assume that you are new on the job, without knowledge of the symbology the organization uses, or perhaps you wrote this some time ago but now have to go back and review the system. Critique the diagram. List places where you think it is misleading, and list the questions that need to be asked—and that the diagram fails to answer—before you can understand its meaning.
5. After you have critiqued Figure 3.15 and have enumerated the information you believe is missing, augment the diagram to make it tell a coherent story. Did you decide that the diagram is describing code-based entities, runtime entities, or both? Did you decide that the boxes called layers are, in fact, layers, or something else? What did you decide the arrows mean?

3.8 For Further Reading

We are awash in stories of architects who thought they could plug two components together with a connector, only to find out that the component didn't implement the right protocol, or was otherwise badly matched with the expectations of that connector. This is why we prescribe writing a justification where the matchup is less than obvious. For a thoughtful treatment of element mismatch, see the paper by Garlan, Allen, and Ockerbloom (1995).

It is tempting to treat architecture simply as an assembly of components, but there are great conceptual advantages to be gained from elevating connectors to the status of first-class architecture. Mary Shaw (1996b) makes an eloquent argument for doing so. Shaw and Garlan (1996) treat software architecture in terms of components and connectors and address concerns such as constructing systems as assemblies of components. Allen and Garlan (1997) lay out the semantic foundations for connectors as first-class entities.

Component-and-connector views can provide a basis for formal analysis of qualities such as performance, reliability, security, and privacy. Garlan and Schmerl (2006) provide a broad introduction to such analyses.

A swarm of architecture description languages were created in the 1990s. Medvidovic and Taylor (1997) give a tour of them and compare members of that generation. Today only a small number deserve mention. Acme is of that earlier generation (see www.cs.cmu.edu/~acme [Acme 2009]). The Architecture Analysis and Design Language (AADL) is a direct descendant of one from that generation. Appendix C gives an architecture-oriented overview of AADL, and the Web site at aadl.info offers full coverage. Yahoo! Pipes can be considered an ADL, albeit a very style-specific one; see pipes.yahoo.com/pipes (Yahoo! 2010) and the Yahoo! Pipes example in Chapter 4.

This page intentionally left blank

4 A Tour of Some Component-and-Connector Styles

4.1 An Introduction to C&C Styles

A component-and-connector (C&C) style introduces a specific set of component-and-connector types and specifies rules about how elements of those types can be combined. Additionally, given that C&C views capture runtime aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style.

In Section 4.9 we provide references for reading about dozens of C&C styles.

The choice of a C&C style (or styles) will usually depend on the nature of the runtime structures in the system. For example, if the system will need to access a set of legacy databases, the style will likely be based on a shared-data style. Alternatively, if a system is intended to perform data stream transformation, a data flow style will likely be chosen.

The choice of style will also depend on the intended use of the documentation. For example, if high performance is a critical property, the style will likely be chosen to enable analysis of performance, so that trade-offs affecting that system quality can be assessed.

Many C&C styles exist. To make sense of the space of these styles, we begin by describing some broad categories of commonly used C&C styles, and then we consider in more detail one or more example styles in each category.

The space of C&C styles is quite large. For example, C&C styles can differ dramatically in terms of the types of the connectors that they support. Styles based on asynchronous event broadcast (such as publish-subscribe) are quite different from those based on synchronous service invocation. Similarly, styles may differ in terms of the types of components that they permit or require. For instance, some styles require a database component to be present. Other styles may require a registry component to enable components to find others at runtime. Styles may differ in terms of topological restrictions, such as whether the components are assigned to tiers. They may also differ in terms of their level of domain specificity. For example, a style to support automotive control systems will likely involve connectors that represent specific protocols for real-time coordination. Similarly, there exist dozens of client-server styles that differ in subtle (or not-so-subtle) ways, depending on the nature of the application domain they are addressing. For example, some client-server styles allow late binding of requests for services, where the recipient of a request is determined dynamically; others insist on a static configuration determined when a system is built or deployed.

Section 6.1.4 discusses how styles can be progressively specialized from generic styles to domain-specific styles and product line.

One way to impose some conceptual order on the space of C&C styles is to consider several broad categories of styles, differentiated primarily by their underlying computational model. In this chapter we consider examples in four such categories.

- *Call-return styles.* Styles in which components interact through synchronous invocation of capabilities provided by other components.
- *Data flow styles.* Styles in which computation is driven by the flow of data through the system.
- *Event-based styles.* Styles in which components interact through asynchronous events or messages.
- *Repository styles.* Styles in which components interact through large collections of persistent, shared data.

Section 4.6.1 describes communicating processes, which is a way to add concurrency to a C&C style. Section 4.6.2 describes the notion of tiers, which are common in some C&C architectures.

Additionally we consider several crosscutting style issues, such as the imposition of a tiered topology, and augmentations that allow one to reason about concurrency.

Figure 4.1 provides a birds-eye view of part of the terrain. This figure can be interpreted as a kind of C&C style specialization hierarchy. At the top is the most general and unconstrained form of C&C view: namely, one that uses generic components and connectors, with no particular constraints on topology, behavior, and element properties. Below this are the general categories of C&C styles distinguished largely by their underlying computational model. Below these are specializations of these general styles. Note that a specific style may specialize more than one general category, as is the case of the service-oriented architecture (SOA) style.

Naturally this is only a partial representation of the space of C&C styles: there are other general categories, and there are many styles that are specializations of these categories. Addi-

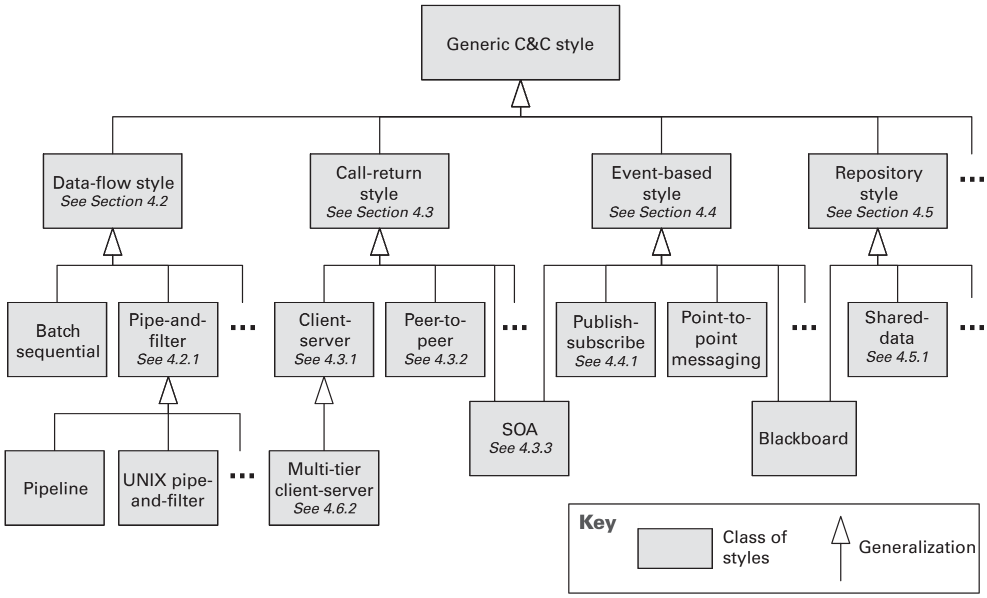


Figure 4.1
A partial representation of the space of C&C styles

tionally, in most real systems several styles may be used together, often from across categories. For example, enterprise IT applications are frequently a combination of client-server and shared-data styles.

See Section 6.6 for a discussion of documenting a view that combines more than one style.

4.2 Data Flow Styles

Data flow styles embody a computational model in which components act as data transformers and connectors transmit data from the outputs of one component to the inputs of another. Each component type in a data flow style has some number of input ports and output ports. Its job is to consume data on its input ports and write transformed data to its output ports.

A variety of data flow styles appear in practice. In the early days of computing, one common data flow style was “batch sequential,” a style in which each component transforms all of its data before the next component can consume its outputs. Later a form of data flow style was invented in which components run concurrently and data is incrementally processed: the pipe-and-filter style. Today data flow styles are common in domains where stream processing occurs, and where the overall computation can be broken down into a set of transformational steps.

4.2.1 Pipe-and-Filter Style

Overview

The pattern of interaction in the pipe-and-filter style is characterized by successive transformations of streams of data. Data arrives at a filter's input ports, is transformed, and then is passed via its output ports through a pipe to the next filter. A single filter can consume from, or produce data to, multiple ports. Modern examples of such systems are signal-processing systems, systems built using UNIX pipes, the request-processing architecture of the Apache Web server, the map-reduce paradigm for search engines, Yahoo! Pipes for processing RSS feeds, and many scientific computation systems that have to process and analyze large streams of experimental data.

Elements, Relations, and Properties

The basic form of pipe-and-filter style, summarized in Table 4.1, provides a single type of component—the *filter*—and a single

Table 4.1 Summary of the pipe-and-filter style

Elements	• *Filter*, which is a component that transforms data read on its input ports to data written on its output ports. Filters typically execute concurrently and incrementally. Properties may specify processing rates, input/output data formats, and the transformation executed by the filter. • *Pipe*, which is a connector that conveys data from a filter's output ports to another filter's input ports. A pipe has a single data-in and a single data-out role, preserves the sequence of data items, and does not alter the data passing through. Properties may specify buffer size, protocol of interaction, and data format that passes through a pipe.
Relations	The *attachment* relation associates filter output ports with data-in roles of a pipe, and filter input ports with data-out roles of pipes.
Computational Model	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters.
Constraints	• Pipes connect filter output ports to filter input ports. • Connected filters must agree on the type of data being passed along the connecting pipe. • Specializations of the style may restrict the association of components to an acyclic graph or a linear sequence—sometimes called a pipeline. • Other specializations may prescribe that components have certain named ports, such as the *stdin*, *stdout*, and *stderr* ports of UNIX filters.
What It's For	• Improving reuse due to the independence of filters • Improving throughput with parallelization of data processing • Simplifying reasoning about overall behavior

type of connector—the *pipe*. A filter transforms data that it receives through one or more pipes and transmits the result through one or more pipes. Filters typically execute concurrently and incrementally. A pipe is a connector that conveys streams of data from the output port of one filter to the input port of another filter. Pipes act as unidirectional conduits, providing an order-preserving, buffered communication channel to transmit data generated by filters. In the pure pipe-and-filter style, filters interact only through pipes.

Because pipes buffer data during communication, filters can act asynchronously and concurrently. Moreover, a filter need not know the identity of its upstream or downstream filters. For this reason, pipe-and-filter systems have the nice formal property that the overall computation can be treated as the functional composition of the computations of the filters, allowing the architect to reason about the end-to-end behavior as a simple composition of the behaviors of the parts.

ADVICE

Typical properties to document for pipes include

- Pipe capacity (that is, buffer size)
- How end-of-data is signaled
- What form of blocking occurs when writing to a pipe whose buffer is full or reading from a pipe that is empty

Properties of filters can include

- Whether or not each filter is a separate process
- The data stream transformation each performs

What the Pipe-and-Filter Style Is For

Systems conforming to a pipe-and-filter style are typically used in data transformation systems, where the overall processing can be broken down into a set of independent steps, each responsible for an incremental transformation of its input data. The independence of the processing done by each step supports reuse, parallelization, and simplified reasoning about overall behavior.

Often such systems constitute the front end of signal-processing applications. These systems typically receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial filtering. "Downstream" filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example, providing input to modeling or visualization tools.

Analyses associated with pipe-and-filter systems include deriving the aggregate transformation provided by a graph of filters and reasoning about system performance: input/output stream latency, pipe buffer requirements, and throughput.

Relation to Other Styles and Models

Data flow models are discussed in "Perspectives: Data Flow and Control Flow Models," on page 146, in Chapter 3.

A pipe-and-filter view of a system is not the same as a data flow model. In the pipe-and-filter style, lines between components represent connectors, which have a specific computational meaning: They transmit streams of data from one filter to another. In data flow models, the lines represent relations, indicating the communication of data between components. Flows in a data flow model have little computational meaning: They simply indicate that data flows from one element to the next. This flow might be realized by a connector, such as a procedure call, the routing of an event between a publisher and a subscriber, or data transmitted via a pipe. The reason that these views might be confused is that the data flow model of a pipe-and-filter style looks almost identical to the original pipe-and-filter view.

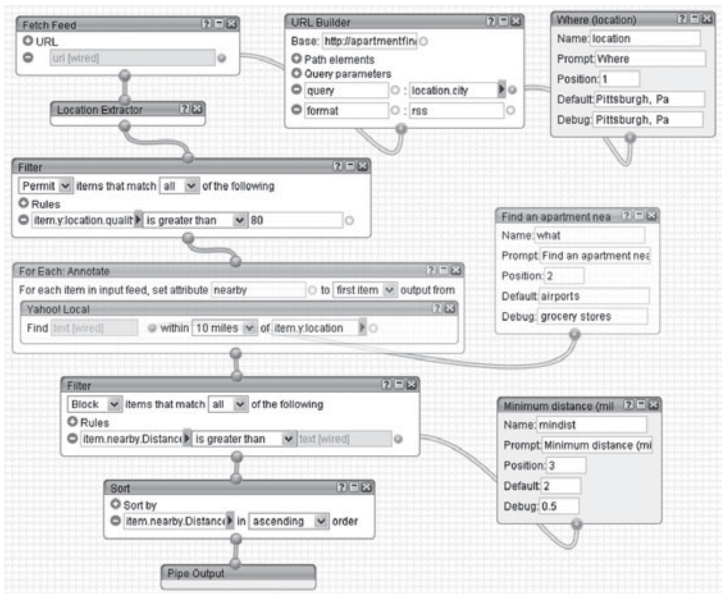
Data flow styles are often combined with other styles by using them to characterize a particular subsystem. A good example of this is the filter processing chains of the Apache Web server.

Example of the Pipe-and-Filter Style: Yahoo! Pipes

"Rewire the Web" is the motto of Yahoo! Pipes, a composition tool that lets Web users combine simple functions quickly and easily into pipe-and-filter applications that aggregate and manipulate content from around the Web.

The basis of Yahoo! Pipes is the many RSS feeds available from sites on the Internet. These data streams form the input to the applications that users build, applications that combine and manipulate the data in the streams to form useful results. Many of the building blocks to perform general-purpose filtering and manipulation of the data streams are made available in the composition environment itself, rather like library functions.

For example, you can take an RSS stream from a financial news site and filter it so that only news items related to stocks that you own are shown. Or you can take an RSS stream from a sports site and filter it so that you see news about your favorite teams or athletes.



Fetch Feed
URL
url [wired]

Location Extractor

Filter
Permit items that match all of the following
Rules
item.y:location.qualit is greater than 80

For Each: Annotate
For each item in input feed, set attribute nearby to first item output from
Yahoo! Local
Find text [wired] within 10 miles of item.y:location

Filter
Block items that match all of the following
Rules
item.nearby.Distanc is greater than text [wired]

Sort
Sort by
item.nearby.Distanc in ascending order

Pipe Output

URL Builder
Base: http://apartmentfin
Path elements
Query parameters
query : location.city
format : rss

Where (location)
Name: location
Prompt: Where
Position: 1
Default: Pittsburgh, Pa
Debug: Pittsburgh, Pa

Find an apartment nea
Name: what
Prompt: Find an apartment nea
Position: 2
Default: airports
Debug: grocery stores

Minimum distance (mil
Name: mindist
Prompt: Minimum distance (mi
Position: 3
Default: 2
Debug: 0.5

Figure 4.2
A Yahoo! Pipes application for finding apartments for rent near a given location (shown using the notation of the Yahoo! Pipes editor). The pipe-and-filter flow runs from top to bottom through the seven "modules" down the left-hand side (each representing what our pipe-and-filter style calls a filter); this is indicated by the thick solid lines (the pipes) connecting the output port of one to the input port of the next. The other "modules" supply inputs to the mainline components; this is indicated by the thinner, hollow lines. The `Fetch Feed` component uses the RSS output from an apartment-finder search; it is fed the search site URL and the search parameters by the helper modules to its right. The `Location Extractor` and the `Filter` component extract high-quality (well-formed) addresses from the apartment-finder search. That stream feeds `Yahoo! Local`, which finds businesses of a given type (supplied by its helper module) near a given location. (The `For Each` component applies the function shown in its interior to every item in the input stream.) The second `Filter` removes listings that aren't a minimum distance from our search term. The `Sort` component orders the stream in ascending order of distance for viewing via the `Pipe Output` component.

Yahoo! Pipes uses terminology not quite the same as that in this book. It calls a complete application a pipe; the building blocks are called modules. A filter is a special kind of module that removes values from a stream based on given comparison criteria.

Figure 4.2 shows an application that finds an apartment for rent that is near a given type of business, such as a movie theater. This is based on one of the teaching examples on the Yahoo! Pipes Web site.

4.3 Call-Return Styles

Call-return styles embody a computational model in which components provide a set of services that may be invoked by other components.[1] A component invoking a service pauses (or is blocked) until that service has completed. Hence, call-return is the architectural analog of a procedure call in programming languages. The connectors are responsible for conveying the service request from the requester to the provider and for returning any results.

1. The term *service* here designates a generic operation or function that can be invoked via a call-return connector; it does not refer to services as in service-oriented architecture.

Call-return styles differ among each other in a variety of ways. Some variants differ in terms of the behavior of their connectors. For example, connectors in some call-return styles may support error handling (such as when the service provider is not available). Other differences relate to constraints on topology. Some call-return architectures are organized in tiers. Others partition the set of components into disjoint sets of components that can make requests and those that can service them.

The organization of components in tiers and multi-tier architectures are discussed in Section 4.6.2.

Examples of call-return styles include client-server, peer-to-peer, and representational state transfer (REST) styles.

Wikipedia provides a nice description of the REST architecture style, at en.wikipedia.org/wiki/REST (Wikipedia 2010b).

4.3.1 Client-Server Style

Overview

As with all call-return styles, client-server style components interact by requesting services of other components. Requesters are termed clients, and service providers are termed servers, which provide a set of services through one or more of their ports. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

Typical examples of systems in the client-server style include the following:

- Information systems running on local networks, where the clients are GUI applications (such as Visual Basic) and the server is a database management system (such as Oracle)
- Web-based applications where the clients run on Web browsers and the servers are components running on a Web server (such as Tomcat)

Elements, Relations, and Properties

In the client-server style, summarized in Table 4.2, component types are *clients* and *servers*. The principal connector type for the client-server style is the request/reply connector used for invoking services. When more than one service can be requested on the same connector, a protocol specification is often used to document ordering relations among the invocable services over that connector. Servers have ports that describe the services they provide. Clients have ports that describe the services they require. Servers may in turn act as clients by requesting services from other servers. A component that has both service-request and service-reply ports can function as both a client and a server simultaneously.

A protocol of interactions can be described using notations such as sequence diagrams and state diagrams, which are covered in Chapter 8.

The computational flow of pure client-server systems is asymmetric: clients initiate interactions by invoking services of servers.

Table 4.2 Summary of the client-server style

Elements	• *Client*, which is a component that invokes services of a server component. • *Server*, which is a component that provides services to client components. Properties will vary according to concerns of the architect but typically include information about the nature of the server ports (such as how many clients can connect) and performance characteristics (such as maximum rates of service invocation). • *Request/reply connector*, which is used by a client to invoke services on a server. Request/reply connectors have two roles: a request role and a reply role. Connector properties may include whether the calls are local or remote, and whether data is encrypted.
Relations	The *attachment* relation associates client service-request ports with the request role of the connector and server service-reply ports with the reply role of the connector.
Computational Model	Clients initiate interactions, invoking services as needed from servers and waiting for the results of those requests.
Constraints	• Clients are connected to servers through request/reply connectors. • Server components can be clients to other servers. • Specializations may impose restrictions: – Numbers of attachments to a given port – Allowed relations among servers • Components may be arranged in tiers.
What It's For	• Promoting modifiability and reuse by factoring out common services • Improving scalability and availability in case server replication is in place • Analyzing dependability, security, and throughput

Thus, the client must know the identity of a service to invoke it, and clients initiate all interaction. In contrast, servers do not know the identity of clients in advance of a service request and must respond to the initiated client requests.

Service invocation is synchronous: the requester of a service waits, or is blocked, until a requested service completes its actions, possibly providing a return result. Variants of the client-server style may introduce other connector types. For example, in some client-server styles, servers are permitted to initiate certain actions on their clients. This might be done by allowing a client to register notification procedures, or callbacks, that the server calls at specific times. In other systems service calls over a request/reply connector are bracketed by a "session" that delineates the start and end of a set of client-server interactions.

What the Client-Server Style Is For

The client-server style presents a system view that separates client applications from the services they use. This style supports system understanding and reuse by factoring out common services. Because servers can be accessed by any number of clients, it is relatively easy to add new clients to a system. Similarly, servers may be replicated to support scalability or availability.

ADVICE

Useful properties to document about components include whether new clients and servers can be introduced dynamically, as well as any limitations on the number of clients that can interact with a given server. Connector properties deal with the request/reply protocol: How are errors handled? How are client-server interactions set up and taken down? Are there sessions? How are servers located? What kinds of middleware, if any, are relied upon?

Client-server system analyses include the following:

- *Dependability*. For example, to understand whether a system can recover from a service failure
- *Security*. For example, to determine whether information provided by servers is limited to clients with the appropriate privileges
- *Performance*. For example, to determine whether a system's servers can keep up with the volume and rates of anticipated service requests

Relation to Other Styles

Like many C&C styles, the client-server style decouples producers of services and data from consumers of those services and data. Other styles, such as peer-to-peer, involve a round-trip form of communication. However, these styles do not have the asymmetric relationship between clients and servers found in the client-server style.

Clients and servers are often grouped and deployed on different machines in a distributed environment to form a multitier hierarchy.

Examples of the Client-Server Style

The World Wide Web may be the best known example of a system that is, at its heart, a client-server system. It is a hypertext-based system that allows clients (Web browsers) to access information from servers distributed across the Internet. Clients access the information, written in Hypertext Markup Language

(HTML), provided by a Web server using Hypertext Transfer Protocol (HTTP). HTTP is a form of request/reply invocation. HTTP is a stateless protocol; the connection between the client and the server is terminated after each response from the server.

For another example, Figure 4.3 uses informal notation to describe the client-server view of an ATM banking system developed in the early 1990s. At that time, client-server architectures were the modern alternative to mainframe-based systems. (J2EE and .NET application servers didn't exist and multi-tier was not yet described as a style.)

In this architecture, there are three types of components:

- The FTX server daemons are processes running in the background on the fault-tolerant UNIX (FTX) server. Each daemon creates one or more socket ports using predefined TCP ports, through which calls from client components arrive.
- ATM OS/2 client processes are concurrent processes that run on the ATMs, which were powered with the IBM OS/2 operating system. Although it can't be inferred from the diagram, each ATM runs one instance of the ATM main process and one instance of the Reconfigure and update process.

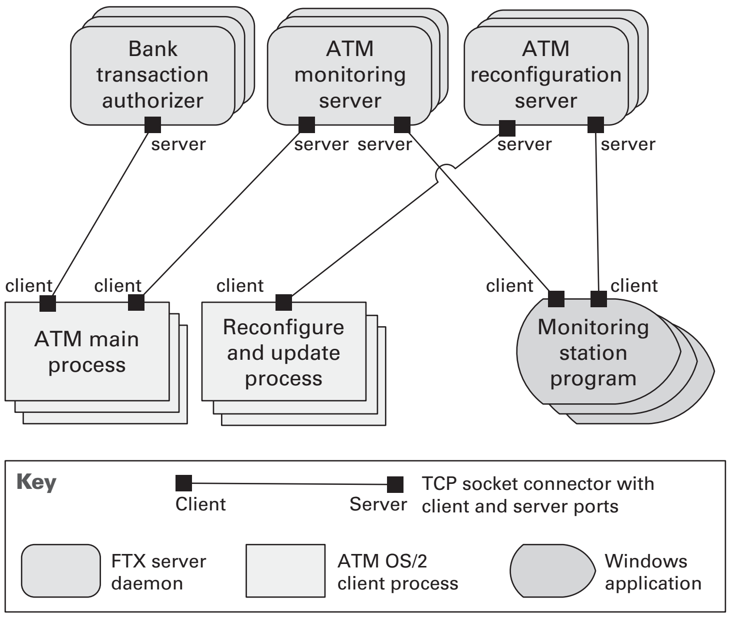


Figure 4.3
Client-server architecture of an ATM banking system. The `ATM main process` sends requests to `Bank transaction authorizer` corresponding to user operations (such as deposit, withdrawal). It also sends messages to `ATM monitoring server` informing the overall status of the ATM (devices, sensors, and supplies). The `Reconfigure and update process` component sends requests to `ATM reconfiguration server` to find out if a reconfiguration command was issued for that particular ATM. Reconfiguration of an ATM (for example, enabling or disabling a menu option) and data updates are issued by bank personnel using the `Monitoring station program`. `Monitoring station program` also sends periodic requests to `ATM monitoring server` to retrieve the status of the range of ATMs monitored by that station.

- The Windows application component was also a client component. It was a Windows 3.x GUI program developed using the Borland OWL API. Each instance was used by an operator to monitor a group of ATMs from his or her workstation.

When this system was developed, the TCP socket connector was very often used for communication in client-server and distributed applications. Today HTTP is far more common. In the protocol used in this TCP socket connector, the client opens a connection to a server identified by an IP address and port number. The client then sends a nonblocking request, after which it may display in the UI a "Please wait…" message to the user. Then the client calls an operation to receive the response from the server. For both client and server, sending and receiving messages are separate steps. Therefore, the connector implementation has to handle the correlation of request and response messages, as well as time-outs and communication errors.

4.3.2 Peer-to-Peer Style

Overview

The peer-to-peer architecture style has inspired new models for industrial production, community knowledge, political movement, property ownership, and an economic alternative to capitalism. See en.wikipedia.org/wiki/Peer-to-peer_(meme).

In the peer-to-peer style, components directly interact as peers by exchanging services. Peer-to-peer communication is a kind of request/reply interaction without the asymmetry found in the client-server style. That is, any component can, in principle, interact with any other component by requesting its services. Each peer component provides and consumes similar services, and sometimes all peers are instances of the same component type. Connectors in peer-to-peer systems may involve complex bidirectional protocols of interaction, reflecting the two-way communication that may exist between two or more peer-to-peer components.

Examples of peer-to-peer systems include file-sharing networks, such as BitTorrent and eDonkey; instant messaging and VoIP applications, such as Skype; and desktop grid computing systems.

Elements, Relations, and Properties

Table 4.3 summarizes the peer-to-peer style. The component types in this style are peers, which are typically independent programs running on network nodes. The principal connector type is the call-return connector. Unlike in the client-server style, the interaction may be initiated by either party: each peer component acts as both client and server. Peers have interfaces that describe the services they request from other peers and the services they provide. The computational flow of peer-to-

Table 4.3 Summary of the peer-to-peer style

Elements	• *Peer* component • *Call-return connector*, which is used to connect to the peer network, search for other peers, and invoke services from other peers
Relations	The *attachment* relation associates peers with call-return connectors.
Computational Model	*Computation* is achieved by cooperating peers that request services of one another.
Properties	Same as other C&C views, with an emphasis on protocols of interaction and performance-oriented properties. Attachments may change at runtime.
Constraints	• Restrictions may be placed on the number of allowable attachments to any given port, or role. • Special peer components can provide routing, indexing, and peer search capability. • Specializations may impose visibility restrictions on which components can know about other components.
What It's For	• Providing enhanced availability • Providing enhanced scalability • Enabling highly distributed systems, such as file sharing, instant messaging, and desktop grid computing

peer systems is symmetric: Peers first connect to the peer-to-peer network and then initiate actions to achieve their computation by cooperating with their peers by requesting services from one another.

Often a peer's search for another peer is propagated from one peer to its connected peers for a limited number of hops. A peer-to-peer architecture may have special peer nodes (called ultrapeers, ultranodes, or supernodes) that have indexing or routing capability and allow a regular peer's search to reach a larger number of peers.

Constraints on the use of the peer-to-peer style might limit the number of peers that can be connected to a given peer or impose a restriction about which peers know about which other peers.

What the Peer-to-Peer Style Is For

Peers interact directly among themselves and can play the role of both service caller and service provider, assuming whatever role is needed for the task at hand. This partitioning provides flexibility for deploying the system across a highly distributed platform. Peers can be added and removed from the peer-to-peer network with no significant impact, resulting in great scalability for the whole system.

Typically multiple peers have overlapping capabilities, such as providing access to the same data. Thus, a peer acting as client can collaborate with multiple peers acting as servers to complete a certain task. If one of these multiple peers becomes unavailable, the others can still provide the services to complete the task. The result is improved overall availability. The load on any given peer component acting as a server is reduced, and the responsibilities that might have required more server capacity and infrastructure to support it are distributed. This can decrease the need for other communication for updating data and for central server storage, but at the expense of storing the data locally.

Peer-to-peer computing is often used in distributed computing applications, such as file sharing, instant messaging, and desktop grid computing. Using a suitable deployment, the application can make efficient use of CPU and disk resources by distributing computationally intensive work across a network of computers and by taking advantage of the local resources available to the clients. The results can be shared directly among participating peers.

Relation to Other Styles

The absence of hierarchy means that peer-to-peer systems have a more general topology than client-server systems.

In late 2007, [Gnutella] was the most popular file sharing network on the Internet with an estimated market share of more than 40%.

—Wikipedia (en.wikipedia.org/wiki/Gnutella)

Examples of the Peer-to-Peer Style

Gnutella is a peer-to-peer network that supports bidirectional file transfers. The topology of the system changes at runtime as peer components connect and disconnect to the network. A peer component is a running copy of a Gnutella client program connected to the Internet. Upon startup, this program establishes a connection with a few other peers. The Web addresses of these peers are kept in a local cache.

The Gnutella protocol supports request/reply messages for peers to connect to other peers and search for files. Peers are identified by their IP address, and the Gnutella protocol messages are carried over dedicated UDP and TCP ports. To perform a search, a Gnutella peer requests information from all of its connected peers, which respond with any information of interest. The connected peers also pass the request to their peers successively, up to a predefined number of "hops." All the peers that have positive results for the search request reply directly to the requester, whose IP address and port number go along with the request. The requester then establishes a connection directly with the peers that have the desired file and initiates the data transfer using HTTP (outside the Gnutella network).

Later versions of Gnutella differentiate between leaf peers and ultrapeers. An ultrapeer runs on a computer with a fast Internet connection. A leaf peer is usually connected to a small number (say, three) of ultrapeers, and an ultrapeer is connected to a large number of other ultrapeers and leaf peers. The ultrapeers are responsible for routing search requests and responses for all leaf peers connected to them.

Figure 4.4 shows part of a peer-to-peer view of a Gnutella network using an informal C&C notation. For brevity, only two leaf peers and four ultrapeers are identified. Each of the identified leaf peers uploads and downloads files directly from other peers.

Documentation of behavior is discussed in Chapter 8.

4.3.3 Service-Oriented Architecture Style

Overview

Service-oriented architectures consist of a collection of distributed components that provide and/or consume services. In SOA, service provider components and service consumer components can use different implementation languages and platforms. Services are largely standalone: service providers and service consumers are usually deployed independently, and often belong to different systems or even different organizations.

Elements, Relations, and Properties

Table 4.4 summarizes the SOA style. The basic component types in this style are service providers and service consumers,

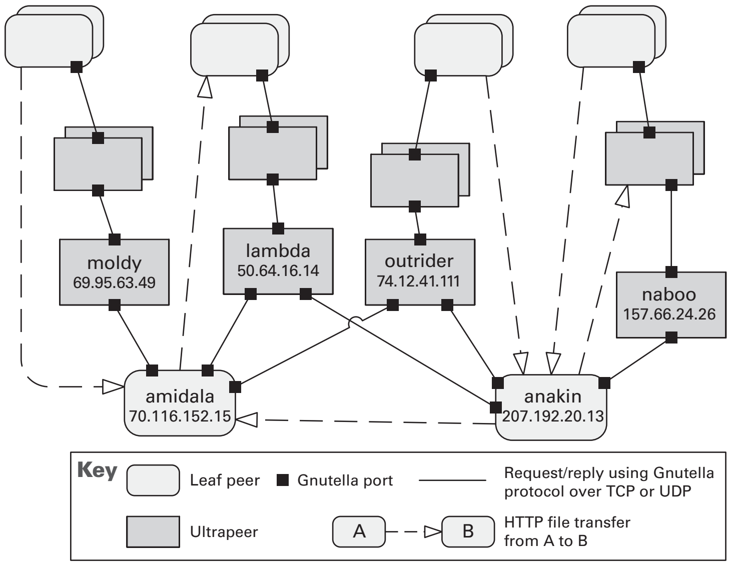


Figure 4.4
A C&C diagram of a Gnutella network, using informal notation

which in practice can take different forms, from JavaScript running on a Web browser to CICS transactions running on a mainframe.

In addition to the service provider and consumer components that you develop, your SOA application may use specialized components that act as intermediaries and provide infrastructure services:

Including an ESB in your architecture of a service-oriented system improves interoperability, security, and modifiability.

- Service invocation can be mediated by an *enterprise service bus* (ESB). An ESB routes messages between service consumers and service providers. In addition, an ESB can convert messages from one protocol or technology to another, perform various data transformations (for example, format, content, splitting, merging), perform security checks, and manage transactions. When an ESB is in place, the architecture follows a hub-and-spoke design, and interoperability, security, and modifiability are improved. When an ESB is not in place, service providers and consumers communicate to each other in a direct point-to-point fashion.
- To improve the transparency of location of service providers, a *service registry* can be used in SOA architectures. The registry is a component that allows services to be registered and then queried at runtime. It increases modifiability by making the location of the service provider transparent to consumers and permitting multiple live versions of the same service.
- An *orchestration server* (or orchestration engine) is a special component that executes scripts upon the occurrence of a specific event (for example, a purchase order request arrived). It orchestrates the interaction among various service consumers and providers in an SOA system. Applications with well-defined business workflows that involve interactions with distributed components or systems gain in modifiability, interoperability, and reliability by using an orchestration server. Many orchestration servers support the Business Process Execution Language (BPEL) standard.

There are many possibilities for communication between components in an SOA architecture, such as SOAP, REST, JMS, MSMQ, and SMTP. Try to indicate in your C&C diagram what protocol or technology is used for each component interaction by using labels or different arrow types.

The basic types of connectors used in SOA are these:

- *Call-return connectors.* Two of the most common such connectors are SOAP and REST:
  - SOAP is the standard protocol for communication in Web services technology. Service consumers and providers interact by exchanging request/reply XML messages, typically on top of HTTP.
  - With the REST connector, a service consumer sends synchronous HTTP requests. These requests rely on the four

basic HTTP commands (post, get, put, and delete) to tell the service provider to create, retrieve, update, or delete a resource (a piece of data). Resources have a well-defined representation in XML, JSON, or a similar language/notation.

- *Asynchronous messaging*. Components exchange asynchronous messages, usually through a messaging system such as IBM WebSphere MQ, Microsoft MSMQ, or Apache ActiveMQ. The messaging connector can be point-to-point or publish-subscribe. Messaging communication typically offers great reliability and scalability.

Components have interfaces that describe the services they request from other components and the services they provide. Components initiate actions to achieve their computation by cooperating with their peers by requesting services from one another.

In practice, SOA environments may involve a mix of the three connectors listed above, along with legacy protocols and other communication alternatives (such as SMTP).

Table 4.4 Summary of the service-oriented architecture style

Elements	• *Service providers*, which provide one or more services through published interfaces. Properties will vary with the implementation technology (such as EJB or ASP.NET) but may include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement (SLA). • *Service consumers*, which invoke services directly or through an intermediary. • *ESB*, which is an intermediary element that can route and transform messages between service providers and consumers. • *Registry of services*, which may be used by providers to register their services and by consumers to query and discover services at runtime. • *Orchestration server*, which coordinates the interactions between service consumers and providers based on scripts that define business workflows. • *SOAP connector*, which uses the SOAP protocol for synchronous communication between Web services, typically over HTTP. Ports of components that use SOAP are often described in WSDL. • *REST connector*, which relies on the basic request/reply operations of the HTTP protocol. • *Messaging connector*, which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
Relations	Attachment of the different kinds of ports available to the respective connectors

continues

Table 4.4 Summary of the service-oriented architecture style (*continued*)

Computational Model	Computation is achieved by a set of cooperating components that provide and/or consume services over a network. The computation is often described as a kind of workflow model.
Constraints	• Service consumers are connected to service providers, but intermediary components (such as ESB, registry, or BPEL server) may be used. • ESBs lead to a hub-and-spoke topology. • Service providers may also be service consumers. • Specific SOA patterns impose additional constraints.
What It's For	• Allowing interoperability of distributed components running on different platforms or across the Internet • Integrating legacy systems • Allowing dynamic reconfiguration

What Service-Oriented Architectures Are Good For

The main benefit and the major driver of SOA is interoperability. Because service providers and service consumers may run on different platforms, service-oriented architectures often integrate different systems and legacy systems. Service-oriented architecture also offers the necessary elements to interact with external services available over the Internet. Special SOA components such as the registry or the ESB also allow dynamic reconfiguration, which is useful when there's a need to replace or add versions of components with no system interruption.

Example of a Service-Oriented Architecture

Figure 4.5 was taken from the example software architecture document accompanying this book online, at wiki.sei.cmu.edu/sad. It shows the SOA view of the Adventure Builder system (Adventure Builder 2010). This system interacts via SOAP Web services with several other external service providers. Note that the external providers can be mainframe systems, Java systems, or .NET systems—the nature of these external components is transparent because the SOAP connector provides the necessary interoperability.

4.4 Event-Based Styles

Event-based styles allow components to communicate through asynchronous messages. Such systems are often organized as a loosely coupled federation of components that trigger behavior in other components through events.

A variety of event styles exist. In some event styles, connectors are point-to-point, conveying messages in a way similar to

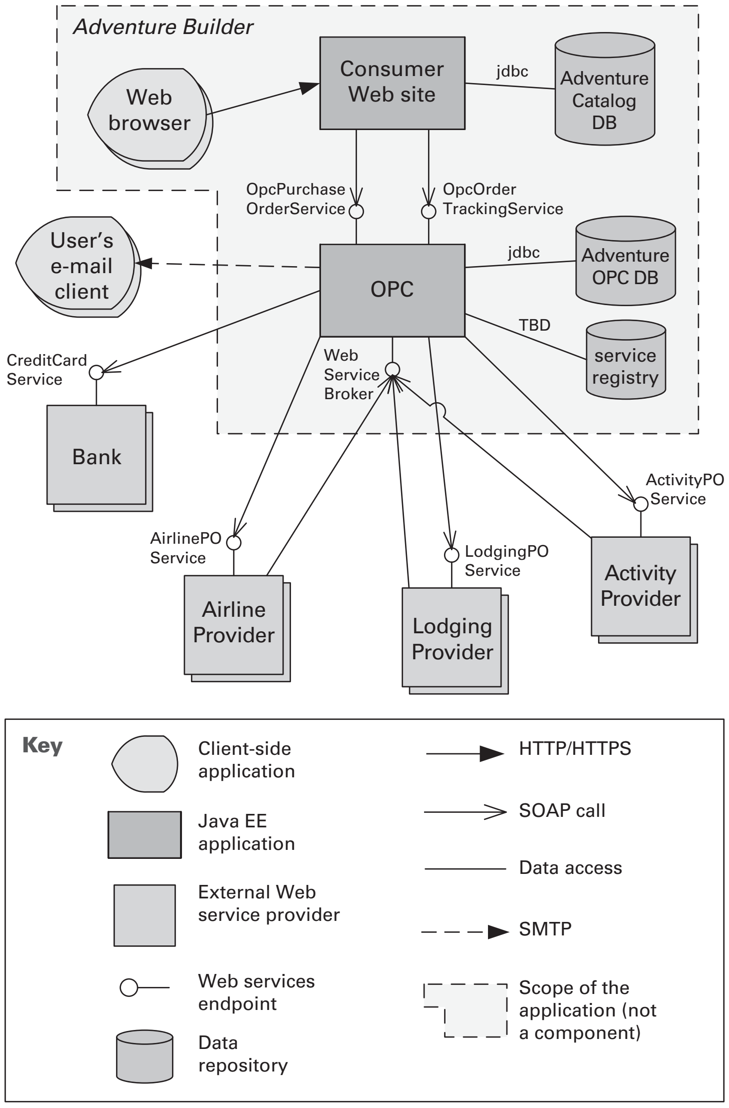


Figure 4.5
Diagram of the SOA view for the Adventure Builder system. The OPC (Order Processing Center) component coordinates the interaction with internal and external service consumers and providers

call-return, but allowing more concurrency, because the event sender need not block while the event is processed by the receiver. In other event styles, connectors are multi-party, allowing an event to be sent to multiple components. Such systems are often called publish-subscribe systems, where the event announcer is viewed as publishing the event that is subscribed to by its receivers.

4.4.1 Publish-Subscribe Style

Overview

In the publish-subscribe style, summarized in Table 4.5, components interact via announced events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus the main form of connector in this style is a kind of event bus. Components place events on the bus by announcing them; the connector then delivers those events to the components that have registered an interest in those events.

The computational model for the publish-subscribe style is best thought of as a system of independent processes or objects, which react to events generated by their environment, and which in turn cause reactions in other components as a side effect of their event announcements.

Examples of systems that employ the publish-subscribe style are the following:

- Graphical user interfaces, where a user's low-level input actions are treated as events that are routed to appropriate input handlers
- Applications based on the model-view-controller (MVC) pattern, where view components are notified when the state of a model object changes
- Extensible programming environments, in which tools are coordinated through events
- Mailing lists, where a set of subscribers can register interest in specific topics
- Social networks, where "friends" are notified when changes occur to a person's Web site

Table 4.5 Summary of the publish-subscribe style

Elements	• Any C&C component with at least one publish or subscribe port. Properties vary, but they should include which events are announced and/or subscribed to, and the conditions under which an announcer is blocked. • *Publish-subscribe connector*, which will have announce and listen roles for components that wish to publish and/or subscribe to events.
Relations	*Attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components have registered to receive events.
Computational Model	Components subscribe to events. When an event is announced by a component, the connector dispatches the event to all subscribers.

Table 4.5 Summary of the publish-subscribe style (*continued*)

Constraints	All components are connected to an event distributor that may be viewed as either a bus—that is, a connector—or a component. Publish ports are attached to announce roles, and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system. A component may be both a publisher and a subscriber, by having ports of both types.
What It's For	• Sending events to unknown recipients, isolating event producers from event consumers • Providing core functionality for GUI frameworks, mailing lists, bulletin boards, and social networks

Elements, Relations, and Properties

The publish-subscribe style can take several forms. In one common form, called *implicit invocation*, the components have procedural interfaces, and a component registers for an event by associating one of its procedures with each subscribed type of event. When an event is announced, the associated procedures of the subscribed components are invoked in an order usually determined by the runtime infrastructure. Graphical user-interface frameworks, such as Visual Basic, are often driven by implicit invocation: User code fragments are associated with predefined events, such as mouse clicks.

In another publish-subscribe form, events are simply routed to the appropriate components. It is the component's job to figure out how to handle the event. Such systems put more of a burden on individual components to manage event streams, but also permit a more heterogeneous mix of components than implicit invocation systems do.

In some publish-subscribe systems, an event announcer may block until an event has been fully processed by the system. For example, some user-interface frameworks require that all views be updated when the data they depict has been changed. This is accomplished by forcing the component that announces a "changed-data" event to block until all subscribing views have been notified.

ADVICE

Useful properties to document for components include these:

- Which events a component announces or subscribes to
- Conditions under which an announcer is blocked

- Whether components can change their subscriptions dynamically
- Whether new event types can be created dynamically, or the event vocabulary is fixed at build or deployment time
- Whether one can add new publishers to the system dynamically

Connector properties often describe the semantics of the event dispatch mechanism:

- Can a subscriber queue up new events when it's busy handling an event?
- Is the connector synchronous or asynchronous?
- Do events have priorities?
- Is temporal or causal ordering enforced?
- Is event delivery reliable?
- What are the semantics of each event?
- Does the connector support other distributed component management, such as starting and stopping publish-subscribe components at the same time?

What the Publish-Subscribe Style Is For

The publish-subscribe style is used to send events and messages to an unknown set of recipients. Because the set of event recipients is unknown to the event producer, the correctness of the producer cannot depend on those recipients. Thus new recipients can be added without modification to the producers.

Publish-subscribe styles are often used to decouple user interfaces from applications. They may also be used to integrate tools in a software development environment: tools interact by announcing events that trigger invocation of other tools. Other applications include systems such as bulletin boards, social networks, and message lists, where some dynamically changing set of users are notified when the content that they care about is modified.

Relation to Other Styles

The publish-subscribe style is similar to a blackboard repository style, because in both styles components are automatically triggered by changes to some component. However, in a blackboard system, the database is the only component that generates such events; in a publish-subscribe system, any component may generate events.

Implicit invocation is often combined with call-return in systems in which components may interact either synchronously by service invocation or asynchronously by announcing events.

For example, many service-oriented architectures and distributed object systems (such as CORBA and Java EE) support both synchronous and asynchronous communication. In other object-based systems, synchronous procedure calls are used to achieve asynchronous interaction using the MVC pattern or the observer pattern.

Example of the Publish-Subscribe Style

Figure 4.6 is a publish-subscribe view of the SEI ArchE tool. There are three different publish-subscribe interactions in this architecture:

Sections 2.3.6 and 6.6.4 have more information about the ArchE tool.

1. `Eclipse UI event manager` acts as an event bus for user-interface events (such as button clicks). Subscription information—that is, what UI events are relevant to the system and what components handle them—is defined at load time when the event manager reads the `SEI.ArchE.UI plug-in config` XML file. From then on a UI event generated by the user working on a view or editor is dispatched via implicit invocation to the action handler objects that subscribe to that event.
2. The data manipulated in ArchE is stored using a rule engine called `Jess`. Data elements are called facts. When a user action creates, updates, or deletes a fact, that action generates respectively an assert, modify, or retract fact

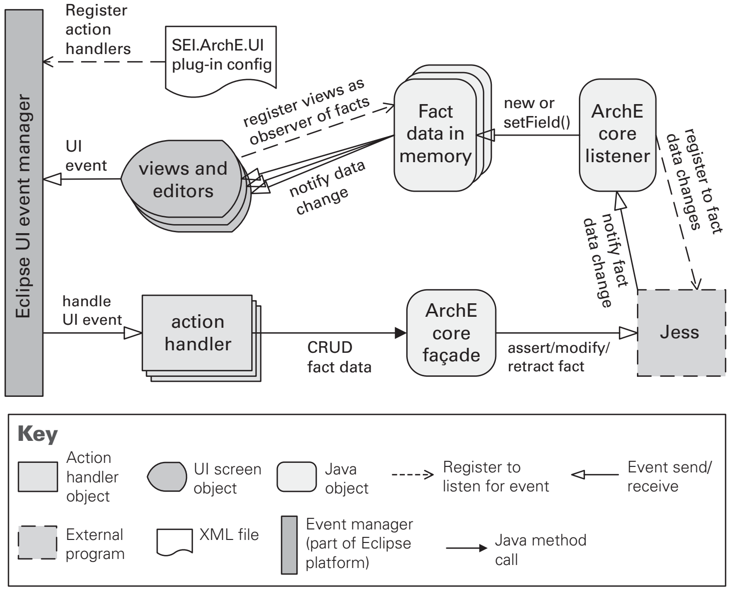


Figure 4.6
Diagram for a publish-subscribe view of the SEI ArchE tool

event that is sent to `Jess`. When `Jess` processes that event, changes to many other facts may be triggered. `Jess` also acts as an event bus that announces changes to facts. In the ArchE architecture, there is one component that subscribes to all data changes: `ArchE core listener`.

The observer design pattern is described in the book by Gamma et al. (1995).

3. ArchE keeps in memory copies of the fact data elements persisted in the rule engine. These copies are observable Java objects. User-interface screens (that is, views) that display those elements are observers of the fact data objects. When facts in memory are created or updated, the views are notified.

4.5 Repository Styles

Repository views contain one or more components, called repositories, which typically retain large collections of persistent data. Other components read and write data to the repositories. In many cases access to a repository is mediated by software called a database management system (DBMS) that provides a call-return interface for data retrieval and manipulation. MySQL is an example of a DBMS. Typically a DBMS also provides numerous data management services, such as support for atomic transactions, security, concurrency control, and data integrity. In C&C architectures where a DBMS is used, a repository component often represents the combination of the DBMS program and the data repository.

Repository systems where the data accessors are responsible for initiating the interaction with the repository are said to follow the shared-data style. In other repository systems, the repository may take responsibility for notifying other components when data has changed in certain prescribed ways. These systems follow the blackboard style. Many database management systems support a triggering mechanism activated when data is added, removed, or changed. You can employ this feature to create an application following the blackboard style. But if your application uses the DBMS for retrieving and changing data in the repository but doesn't employ triggers, you're following the pure shared-data style.

4.5.1 Shared-Data Style

Overview

In the shared-data style, the pattern of interaction is dominated by the exchange of persistent data. The data has multiple accessors and at least one shared-data store for retaining persistent data.

Database management systems and knowledge-based systems are examples of this style.

Elements, Relations, and Properties

The shared-data style, summarized in Table 4.6, is organized around one or more shared-data stores, which store data that other components may read and write. Component types include *shared-data stores* and *data accessors*. The connector type is *data reading and writing*. The general computational model associated with shared-data systems is that data accessors perform operations that require data from the data store and write results to one or more data stores. That data can be viewed and acted on by other data accessors. In a pure shared-data system, data accessors interact only through one or more shared-data stores. However, in practice shared-data systems also allow direct interactions between data accessors. The data-store components of a shared-data system provide shared access to data, support data persistence, manage concurrent access to data through transaction management, provide fault tolerance, support access control, and handle the distribution and caching of data values.

Specializations of the shared-data style differ with respect to the nature of stored data: existing approaches include relational, object structures, layered, and hierarchical structures.

Table 4.6 Summary of the shared-data style

Elements	• *Repository component*. Properties include types of data stored, data performance-oriented properties, data distribution, number of accessors permitted. • *Data accessor component*. • *Data reading and writing connector*. An important property is whether the connector is transactional or not.
Relations	*Attachment* relation determines which data accessors are connected to which data repositories.
Computational Model	Communication between data accessors is mediated by a shared-data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
Constraints	Data accessors interact with the data store(s).
What It's For	• Allowing multiple components to access persistent data • Providing enhanced modifiability by decoupling data producers from data consumers

What the Shared-Data Style Is For

The shared-data style is useful whenever various data items have multiple accessors and persistence. Use of this style decouples the producer of the data from the consumers of the data; hence this style supports modifiability, as the producers do not have direct knowledge of the consumers.

ADVICE

Useful properties to document about data stores include the following:

- Restrictions on the number of simultaneous connections to the data store.
- Whether or not new accessors can be added at runtime.
- Access-control enforcement policies.
- Whether concurrent access to the same data element is permitted, and if so, what kinds of synchronization mechanisms are used.
- Administrative concerns, such as whether one modifies the types of data stored, and if so, who has access, when those changes can be performed, and via what interface.
- Replication of data in a distributed setting.
- Age of data.
- If the repository system supports both query-based and triggered modes of interaction, it is important to clearly document what form of interaction is intended, for example, by using different connector types.

Analyses associated with this style usually center on qualities such as performance, security, privacy, availability, scalability, and compatibility with, for example, existing repositories and their data. In particular, when a system has more than one data store, a key architecture concern is the mapping of data and computation to the data. Use of multiple stores may occur because the data is naturally, or historically, partitioned into separable stores. In other cases data may be replicated over several stores to improve performance and/or availability through redundancy. Such choices can strongly affect the qualities noted above.

Relation to Other Styles

This style has aspects in common with the client-server style, especially the multi-tiered client-server. In information management applications that use this style, the repository is often a relational database, providing relational queries and updates using client-server interactions. The clients of the relational database (that is, the accessors) connect to the DBMS using a network port and protocol specified by the DBMS. A bridge module or DBMS driver, built into the client components, provides database operations.

The shared-data style is closely related to the data model style. While a shared-data view of the system depicts the data repositories and their accessors, the data model shows how data is structured inside the repositories, in terms of data entities and their relations.

The data model style is described in Section 2.6.

Akin to other C&C styles, the shared-data style is also related to the deployment style. Very often systems that have a shared repository are distributed applications where one or more dedicated server machines host the repositories. A deployment view of the system shows the allocation of the repositories and other components to the hardware nodes.

Example of the Shared-Data Style

Figure 4.7 shows the diagram of a shared-data view of a corporate access-management system. There are three types of acces-

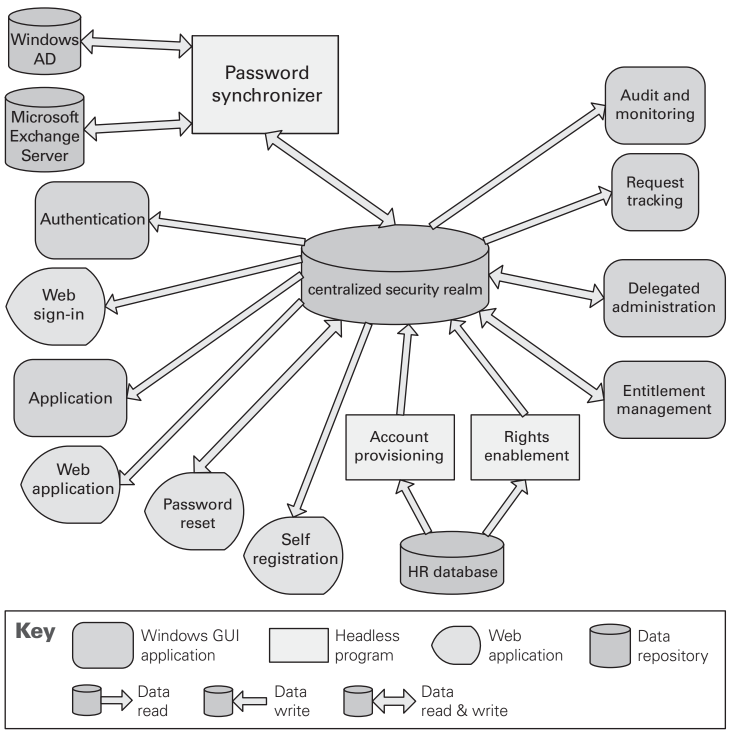


Figure 4.7
The shared-data diagram of an enterprise access-management system. The centralized security realm is a repository for user accounts, passwords, groups of users, roles, permissions, and related information. User IDs and passwords are synchronized with external repositories shown on the top left. The accounts of the enterprise employees are created/deactivated and permissions are granted/revoked based on status changes in `HR database`.

sor components: Windows applications, Web applications, and "headless" programs (that is, programs or scripts that run in the background and don't provide any user interface).

4.6 Crosscutting Issues for C&C Styles

There are a number of concerns that relate to many C&C styles in a similar way. It is helpful to treat these as crosscutting issues, since the requirements for documenting them are similar for all styles. One such issue is concurrency: indicating which components in the system execute as concurrent threads or processes. Another crosscutting issue is the use of tiers: aggregating components into hierarchical groupings and restricting communication paths between components in noncontiguous groups. Another issue is dynamic reconfiguration: indicating which components may be created or destroyed at runtime.

In these and other cases, the crosscutting issues can be documented by augmenting the element types of a style with additional semantic detail to clarify how instances of those types address the crosscutting issues. By adding this additional detail, we effectively create a specialized variant of the original style, because the augmentation will typically introduce new constraints on the components and connectors, their properties, and system topologies.

4.6.1 Communicating Processes

Communicating processes are common in most large systems and necessary in all distributed systems. A communicating-processes variant of any C&C style can be obtained by stipulating that each component can execute as an independent process. For instance, clients and servers in a client-server style are usually independent processes. Similarly, a communicating-processes variant of the pipe-and-filter system would require that each filter run as a separate process. The connectors of a **communicating-processes style** need not change, although their implementation will need to support interprocess communication.

A **communicating-processes style** is any C&C style whose components can execute as independent processes.

A common variant on this scheme (for components with substructure) is to require that top-level components run as separate processes but allow their internal components to run in their parent's process. Another variant is to use threads, instead of processes, as the concurrency unit. Still other variants mix threads and processes.

For communicating-processes styles, there are additional things that often must be documented, including the following:

- Mechanisms for starting, stopping, and synchronizing a set of processes or threads
- Preemptability of concurrent units, indicating whether the execution of a concurrent unit may be preempted by another concurrent unit
- Priority of the processes, which influences scheduling
- Timing parameters, such as period and deadline
- Additional components, such as watchdog timers and schedulers, for monitoring and controlling concurrency
- Use of shared resources, lock mechanism, and deadlock prevention or detection techniques

Communicating processes are used to understand (1) which portions of the system could operate in parallel, (2) the bundling of components into processes, and (3) the threads of control within the system. Therefore this style variant can be used for analyzing performance and reliability, and for influencing how to deploy the software onto separate processors. Behavioral notations such as activity diagrams and sequence diagrams are particularly useful to understand interactions among elements running concurrently.

The deployment style is described in Section 5.2.

Chapter 8 covers documentation of behavior.

4.6.2 Tiers

The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a **tier**. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose.

A **tier** is a mechanism for system partitioning. Usually applied to client-server-based systems, where the various parts (tiers) of the system (user interface, database, business application logic, and so forth) execute on different platforms.

The use of tiers may be applied to any C&C style, although in practice it is most often used in the context of client-server styles. Tiers induce topological constraints that restrict which components may communicate with other components. Specifically, connectors may exist only between components in the same tier or residing in adjacent tiers. The multi-tier style found in many Java EE and Microsoft .NET applications is an example of organization in tiers derived from the client-server style.

Additionally, tiers may constrain the *kinds* of communication that can take place across adjacent tiers. For example, some tiered styles require call-return communication in one direction but event-based notification in the other.

You can depict tiers graphically by overlaying tier boundaries on top of an existing C&C diagram. Alternatively, or in addition, you can document tiers by associating a property with each component to indicate the tier to which it belongs.

Don't confuse tiers with layers! Layering is a module style, while tiers apply to C&C styles. In other words, a layer is a grouping of implementation units while a tier is a grouping of runtime elements.

Tiers are not components; they are logical groupings of components.

Example of a Multi-tiered System

Figure 4.8 uses informal notation to describe the multi-tier architecture of the Consumer Website Java EE application. This application is part of the Adventure Builder system (Adventure Builder 2010). Many component-and-connector types are specific to the supporting platform, which is Java EE in this case.

4.6.3 Dynamic Creation and Destruction

Many C&C styles allow components and connectors to be created or destroyed as the system is running. For example, new server instances might be created as the number of client requests increases in a client-server system. In a peer-to-peer system, new components may dynamically join the system by connecting to a peer in the peer-to-peer network. Because any style can in principle support the dynamic creation and destruction of elements, this is another crosscutting issue.

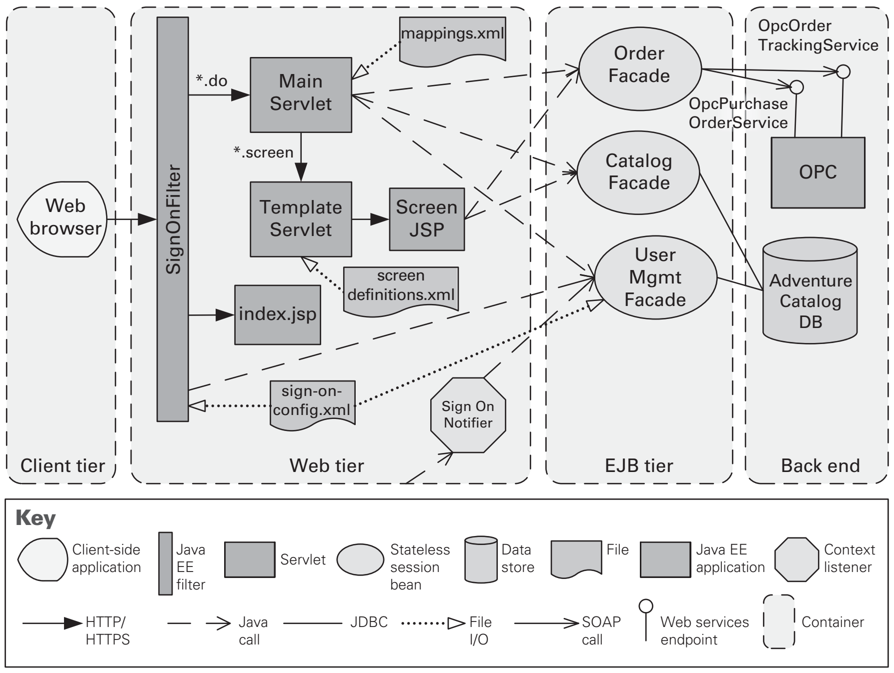


Figure 4.8
Diagram of the multi-tier view describing the Consumer Website Java EE application, which is part of the Adventure Builder system

To document the dynamic aspects of an architecture, you should add several pieces of information, including the following:

Section 6.4.3 discusses documentation of dynamic systems.

- What types of components or connectors within a style may be created or destroyed.
- The mechanisms that are used to create, manage, or destroy elements. For example, component "factories" are a common mechanism for creating new components at runtime.
- How many instances of a given component may exist at the same time. For example, some Web applications use a pool of instances of Web server components, and the number of instances in a pool is parameterized by a minimum and a maximum value.
- What is the life cycle for different component types. Under what conditions new instances are created, activated, deactivated, and removed. For example, some styles require that all or part of a system be brought to a stable, "quiescent" state before new components can be added.

4.7 Summary Checklist

- Component-and-connector styles specialize C&C views by introducing a specific set of component-and-connector types and by specifying rules about how elements of those types can be combined. A C&C style is typically associated with a computational model that prescribes how execution, data, and control flow through systems in this style.
- Component-and-connector styles can be grouped into a number of general categories on the basis of their underlying computational model. Each of these categories contains a variety of specific C&C styles, a number of which were illustrated in this chapter.
- In a pipe-and-filter system, filters process the data input serially and send the output to the next filter through a pipe.
- In client-server systems, client components make synchronous requests to services from server components.
- In peer-to-peer solutions, many instances of the same component cooperate to achieve the desired goal by exchanging synchronous request/reply messages.
- Service-oriented architecture involves distributed components that act as service providers and/or service consumers and are highly interoperable. Intermediaries such as ESB, service registry, and BPEL server may be used.

- In publish-subscribe systems, publishers send events to a pub-sub connector that dispatches the event to all subscribers that have registered to receive that event.
- The shared-data style shows how a shared data repository is accessed for reading and/or writing by independent components called accessors.
- Many C&C views involve communicating components that run as concurrent processes or threads. In these cases, it's important to document how these processes or threads are scheduled or preempted, and how access to shared resources is synchronized.
- Component-and-connector architectures can be structured in tiers, which are logical groupings of components. The multi-tier style found in Java EE and Microsoft .NET applications is a specialization of the client-server style

4.8 Discussion Questions

1. Peer-to-peer, client-server, and other call-and-return styles all involve interactions between producers and consumers of data or services. If an architect is not careful when using one of these styles, he or she will produce a C&C view that simply shows a request flowing in one direction and a response flowing in the other. What means are at the architect's disposal to distinguish among these styles?
2. Some forms of publish-subscribe involve runtime registration; others allow only pre-runtime registration. How would you represent each of these cases?
3. A user invokes a Web browser to download a file. Before doing so, the browser retrieves a plug-in to handle that type of file. How would you model this scenario in a C&C view?
4. If you wanted to show a C&C view that emphasizes the system's security aspects, what kinds of properties might you associate with the components? With the connectors?
5. Suppose that the middle tier of a three-tier system is a data repository. Is this system a shared-data system, a three-tier system, a client-server system, all of them, or none? Justify your answer.
6. To help you see why layers and tiers are different, sketch a layered view for a system you're familiar with, and then sketch a multi-tier client-server view for the same system.

4.9 For Further Reading

There is not widespread agreement about what to call C&C styles or how to group them. While this might seem like an issue of importance only to the catalog purveyors, it has documentation ramifications as well. For instance, suppose you choose a peer-to-peer style for your system. In theory, that should free you of some documentation obligations, because you should be able to appeal to a style catalog for details. However, it is difficult to find an authoritative source for the style definition; different authors describe the same style with slightly different component-and-connector types and properties. But many good style catalogs are available. The reader interested in finding out more about a particular style can look at the book by Shaw and Garlan (1996) and any of the five volumes of the *Pattern-Oriented Software Architecture* books (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; and Buschmann, Henney, and Schmidt 2007a and 2007b). Wikipedia is also a good source of information about styles.

The SEI report titled *Evaluating a Service-Oriented Architecture* (Bianco, Kotermanski, and Merson 2007) describes many different component and connector types available in SOA, and it discusses how the different design alternatives affect the quality attribute properties of the solution. A comprehensive description of various event-based styles is found in the *Enterprise Integration Patterns* book (Hohpe and Wolff 2003). An excellent description of blackboards and their history in system design can be found in the article by Nii (1986). One of the first systems to employ the blackboard style was a speech-understanding system called Hearsay II. A more modern variation is provided by "tuple spaces," as exemplified by the Linda programming language (Gelernter 1985) and JavaSpaces technology (Freeman, Hupfer, and Arnold 1999). High Level Architecture (HLA) uses a publish-subscribe mechanism as an integration framework for distributed simulations (IEEE 1516.1 2000).

To learn more about Yahoo! Pipes, visit pipes.yahoo.com/pipes.

This page intentionally left blank

5 Allocation Views and a Tour of Some Allocation Styles

In this chapter, after a brief overview of allocation views, we look at these aspects of allocation views and styles:

- Deployment style
- Install style
- Work assignment style
- Other allocation styles

5.1 Overview

Software elements in a software architecture interact with nonsoftware elements in the environment in which the software is developed, deployed, and executed. Computing and communication hardware, file management systems, and development teams all interact with the software architecture. Because of this, the "set of structures needed to reason about the system" (from our definition of software architecture given in the prologue) includes structures that show the relations between software and nonsoftware elements. It is through the mapping between the software architecture and the hardware that the performance of the system can be analyzed; it is through the mapping between the software architecture and a file structure that the management of the system in production can be done; and it is through the mapping between the software architecture and the team structure that project management activities can proceed.

You can think of an allocation view as the result of combining a software architecture view with a view from a different kind of architecture—for example, a hardware architecture or an organizational architecture. Section 6.6 describes techniques for combining otherwise-separate views, and why you might want to do so.

These structures have a first-class place in the Views and Beyond approach, and this chapter focuses on the views and styles that represent them. *Allocation views* present a mapping between software elements (from either a module view or a component-and-connector [C&C] view) and nonsoftware elements in the software's environment.

We begin by considering the most general form of the mapping between the software architecture and its environment. We then identify three common allocation styles, as shown in Figure 5.1.

These are not the only allocation styles; many others are possible and useful. Examples can be found in Section 5.5 and "Perspectives: Coordination Views," on page 209, in this chapter.

- The deployment style describes the mapping between the software's components and connectors and the hardware of the computing platform on which the software executes.
- The install style describes the mapping between the software's components and structures in the file system of the production environment.
- The work assignment style describes the mapping between the software's modules and the people, teams, or organizational work units tasked with the development of those modules.

Table 5.1 summarizes the characteristics of the allocation styles. The elements of allocation styles are software elements plus environmental elements. Examples of environmental elements are a processor, a disk farm, a file or folder, or a group of developers. The software elements come from a module or C&C style.

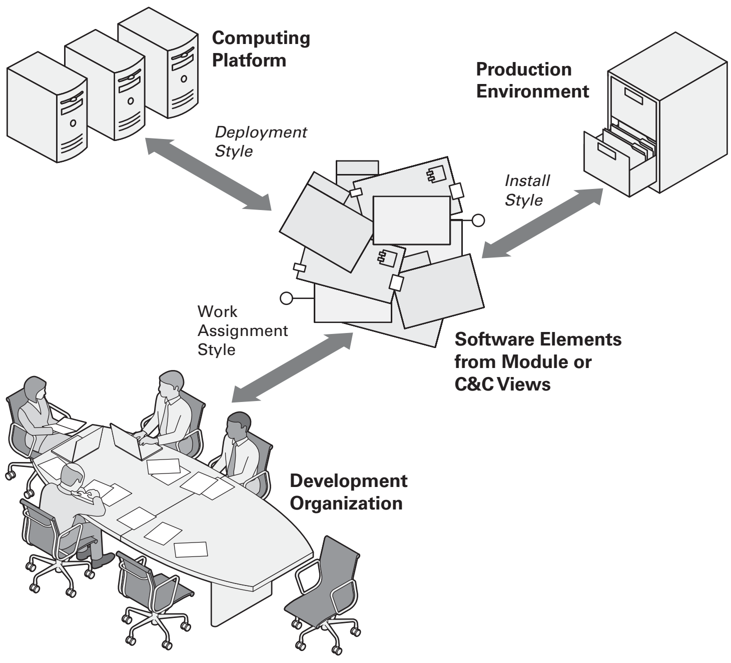


Figure 5.1
Three allocation styles are deployment (mapping software architecture to the hardware of the computing platform), install (mapping it to a file system in the production environment), and work assignment (mapping it to the teams in the development organization).

Table 5.1 Summary of the characteristics of the allocation styles

Overview	Allocation styles describe the mapping between the software architecture and its environment.
Elements	*Software element* and *environmental element*. A software element has properties that are *required* of the environment. An environmental element has properties that are *provided* to the software.
Relations	*Allocated-to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular style.
Constraints	Varies by style

The relation in an allocation style is the *allocated-to* relation. We usually talk about allocation styles in terms of a mapping from software elements to environmental elements, although the reverse mapping would also serve the same purposes. A single software element can be allocated to multiple environmental elements, and multiple software elements can be allocated to a single environmental element. If these allocations change over time, during either development or execution of the system, then the architecture is said to be dynamic with respect to that allocation.

Software elements and environmental elements have properties in allocation styles. The specific properties you should include in an allocation view will, as always, depend on the purpose of that view. The usual goal of an allocation view is to compare the properties required by the software element with the properties provided by the environmental elements to determine whether the allocation will be successful or not. For example, to ensure a component's *required* response time, it has to execute on (be allocated to) a processor that *provides* sufficiently fast execution times, where "sufficiently fast" might be defined in terms of a requirement that an IEEE 754 single-precision floating-point multiply must execute in 50 microseconds. Or a computing platform might not allow a task to use more than 10 kilobytes of virtual memory. In this case, an execution model of the software element in question can be used to determine the required virtual memory usage.

The specific uses and notations for allocation styles are style specific and are covered in their respective sections.

5.2 Deployment Style

5.2.1 Overview

In the deployment style, software elements native to a C&C style are allocated to the hardware of the computing platform on which the software executes. A valid allocation ensures that the requirements expressed by the software elements are satisfied by the characteristics of the hardware element(s).

5.2.2 Elements, Relations, and Properties

Table 5.2 summarizes the characteristics of the deployment style. Environmental elements in a deployment style are entities that correspond to physical units that store, transmit, or compute data. Physical units include processing nodes (CPUs), communication channels, memory stores, and data stores.

The software elements in this style are typically elements that would be documented in a C&C view. When represented in a deployment view, the software elements are assumed to run on a computer. Therefore, software elements in this style correspond to runtime entities of the computing platform (such as processes, threads, ports, or shared memory).

The typical relation depicted in a deployment view is a special *allocated-to* form that shows on which physical units the software elements reside at a given moment in time. The relation can be dynamic; that is, the allocation can change as the system

Although the deployment style in its general form imposes no topological-form restrictions, specializations (substyles) of the deployment style might. See Section 5.5 for examples.

Table 5.2 Summary of the deployment style

Overview	The deployment style describes the mapping of components and connectors in the software architecture to the hardware of the computing platform.
Elements	• *Software element: elements from a C&C view.* Useful properties to document include the significant features required from hardware, such as processing, memory, capacity requirements, and fault tolerance. • *Environmental elements: hardware of the computing platform—processor, memory, disk, network (such as router, bandwidth, firewall, bridge), and so on.* Useful properties of an environmental element are the significant hardware aspects that influence the allocation decision.
Relations	• *Allocated-to*. Physical units on which the software elements reside during execution. Properties include whether the allocation can change at execution time or not. • *Migrates-to*, *copy-migrates-to*, and/or *execution-migrates-to* if the allocation is dynamic. Properties include the trigger that causes the migration.
Constraints	The allocation topology is unrestricted. However, the required properties of the software must be satisfied by the provided properties of the hardware.

executes. In this case, additional relations, such as the following, may be shown:

Section 6.4.3 describes how to document dynamism and dynamic architectures.

- *Migrates-to.* A relation from a software element on one processor to the same software element on a different processor, this relation indicates that a software element can move from processor to processor but does not simultaneously exist on both processors.
- *Copy-migrates-to.* This relation is similar to the *migrates-to* relation, except that the software element sends a copy of itself to the new processor while retaining a copy on the original processing element.
- *Execution-migrates-to.* Similar to the previous two, this relation indicates that execution moves from processor to processor but that the code residency does not change. A copy of a process exists on more than one processor, but only one is active at any particular time. The execution of the process "migrates" when the active process is changed.

It is also possible for the allocation to change over time as a result of manual reconfiguration brought about by exercising a variation point built in to the architecture.

Documenting variation points is discussed in Section 6.4.

The important properties of the elements are the significant hardware features that affect the allocation of the software to the physical units. How a physical unit satisfies a software element requirement is determined by the properties of both. For example, if a software element requires a minimum storage capacity, any environmental element that has at least that capacity is a candidate for a valid allocation.

Moreover, the types of analyses to be performed via a deployment view also determine the particular properties the elements must possess. For example, if a memory capacity analysis is needed, the necessary properties of the software elements must describe memory consumption aspects, and the relevant environmental element properties must depict memory capacities of the various hardware entities.

Below are some environmental element properties relevant to physical units:

- *CPU properties.* The properties relevant to the various processing elements (such as processor clock speed, number of processors, memory capacity, bus speed, cache size, and instruction execution speed).
- *Memory properties.* The properties relevant to the memory stores (such as memory size and speed characteristics).

- *Disk or other storage unit capacity.* The storage capacity and access speed of disk units: individual disk drives, disk farms, and redundant arrays of independent disks (RAIDs).
- *Bandwidth.* The data transfer capacity of communication channels.
- *Fault tolerance.* Multiple hardware units may perform the same function, and these units may have a failover control mechanism.

Properties that are relevant to software elements include the following:

- *Resource consumption.* For example, a computation takes 32,123 instructions always, or at most, or on average, or under nominal (error-free) conditions, and so on.
- *Resource requirements and constraints that must be satisfied.* For example, a software element must execute in no more than 0.1 second.
- *Safety critical.* For example, this would be true if a software element must always be running.

The following property is relevant to the allocation:

- *Migration trigger.* If the allocation can change as the system executes, this property specifies what must occur for a migration of a software element from one processing element to another.

5.2.3 What the Deployment Style Is For

A deployment view is useful for analyzing performance, availability, reliability, and security. Testers use this view to understand runtime dependencies, and integrators use it to plan integration and integration testing. A deployment view may also be used to support cost estimation when evaluating purchasing options for hardware.

Performance is tuned by changing the allocation of software to hardware. Optimal or improved allocation decisions could be those that eliminate bottlenecks on processors or that distribute work more evenly (for example, processor utilization is roughly even across the system). Often performance improvement is achieved by collocating deployment units that require frequent and/or high-bandwidth communications with one another. The volume and frequency of communication among deployable units on different processing elements, which takes place along the communication channels between those elements, is the focus for much of the performance engineering of a system. The architect can employ additional hardware or replace hardware elements with more powerful versions when requirements cannot be met no matter how the allocation is optimized.

Availability and reliability are directly affected by the system's behavior in the face of faulty or failed processing elements or communication channels. If a processor or a channel can fail without warning, copies of software components can be placed on separate processors. If a warning will precede a failure, then components can be migrated at runtime when a failure is imminent. If every processing element has enough memory to host a copy of every deployable unit, runtime migration need not occur. When a failure occurs, a different copy of the no-longer-available deployable unit becomes active, but no migration of code occurs.

Security and attack resistance are influenced by the configuration of the hardware and the allocation of software to it. Limit the services available on each host to limit exposure. Firewalls and router and bridge protections can be employed to limit access to sensitive areas. Physical security measures can be used to limit exposure of a processor to physical attack.

Modern software architectures seek to make deployment decisions transparent, and thus changeable. For example, a goal is to carry out interprocess communication in exactly the same fashion whether the processes reside on the same or on different processors. If the deployment changes, the code need not. Thus, although a deployment view is invaluable in helping to analyze and achieve quality attributes, be careful not to let the software implementers assume too much about the deployment.

ADVICE

An incorrect use of a deployment view is to treat it as the entire software architecture of a system. A single view in this style, in isolation, is not a complete description of a software architecture. Although this observation is true of every style, allocation styles seem especially susceptible. When asked to show their software architecture, architects sometimes present an impressive diagram that shows a network of computers with all their properties and protocols used and the software components running on those computers. Although these diagrams fill an important role by helping to organize the work and to understand the software, they do not fully represent the software architecture.

Don't try to force a relationship between modules and hardware units. For instance, it is usually a design error to force each layer of a layered system onto its own processor. (Remember that layers are not tiers.)

5.2.4 Notation for the Deployment Style

Informal Notations

Informal graphical notations contain boxes, circles, lines, arrows, and so on to represent the software and environmental elements. In many cases, stylized symbols or icons are used to represent the environmental elements. The symbols are frequently pictures of the hardware devices in question. Additionally, shading, color, border types, and fill patterns are often used to indicate the type of element. Software elements can be listed inside or next to the hardware to which they're allocated to show the *allocated-to* relation. If the deployment structure is simple, a table that lists the software units and the hardware element on which each executes may be adequate. Figure 5.2 shows an example of a deployment view primary presentation using an informal notation.

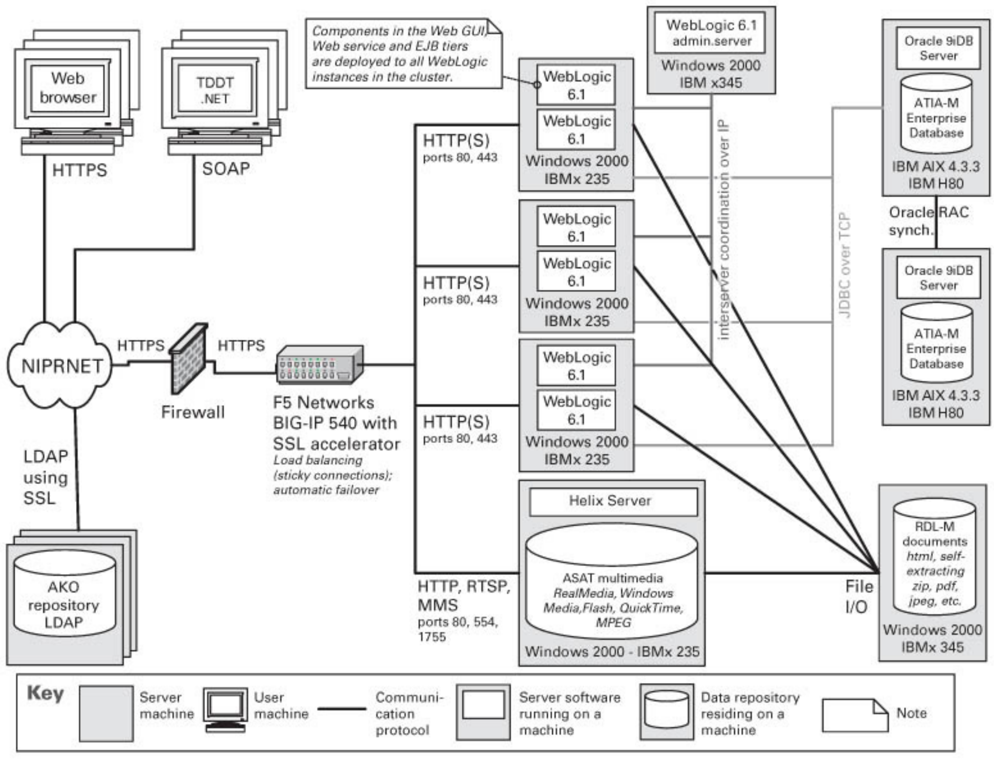

Web browser
TDDT .NET
HTTPS
SOAP
Components in the Web GUI, Web service and EJB tiers are deployed to all WebLogic instances in the cluster.
WebLogic 6.1
WebLogic 6.1
Windows 2000 IBMx 235
WebLogic 6.1 admin.server
Windows 2000 IBM x345
HTTP(S) ports 80, 443
interserver coordination over IP
JDBC over TCP
Oracle 9iDB Server
ATIA-M Enterprise Database
IBM AIX 4.3.3 IBM H80
Oracle RAC synch.
NIPRNET
HTTPS
HTTPS
Firewall
F5 Networks BIG-IP 540 with SSL accelerator
Load balancing (sticky connections); automatic failover
LDAP using SSL
AKO repository LDAP
Helix Server
ASAT multimedia RealMedia, Windows Media, Flash, QuickTime, MPEG
Windows 2000 - IBMx 235
HTTP, RTSP, MMS ports 80, 554, 1755
File I/O
RDL-M documents html, self-extracting zip, pdf, jpeg, etc.
Windows 2000 IBMx 345
Key
Server machine
User machine
Communication protocol
Server software running on a machine
Data repository residing on a machine
Note

Figure 5.2
Example of a deployment view in an informal notation. This example comes from the U.S. Army Training Information Architecture-Migrated (ATIA-M) System and uses distinctive symbols for different types of hardware. The connecting lines are physical communication channels that allow the components to communicate with one another. The allocation of components is shown by overlaying their names on the symbol. The allocation of connectors is done by writing their names adjacent to the channels to denote the communication protocol. ATIA is a Java Platform, Enterprise Edition (Java EE) application comprising hundreds of components (mostly servlets and Enterprise JavaBeans [EJBs]). The ATIA architecture has a client-server multi-tier view with a Web GUI tier, a Web service tier, and an EJB tier. All components inside those tiers are deployed to WebLogic, as indicated by the annotation. NIPRNET is an Internet-like network owned by the Department of Defense.

Formal Notations

The Architecture Analysis and Design Language (AADL) and SysML are examples of architecture description languages that provide formal notations for describing deployment views. AADL provides a vocabulary for representing the hardware and binding software to hardware elements such as processors, memory, and connections. The language supports analysis of performance, reliability, safety-critical, and security requirements. In SysML, graphical representation is supported using a modified version of UML block diagrams. In addition, it provides a tabular form for representing deployment and other forms of allocation.

SysML and AADL are described in Appendices B and C, respectively.

UML

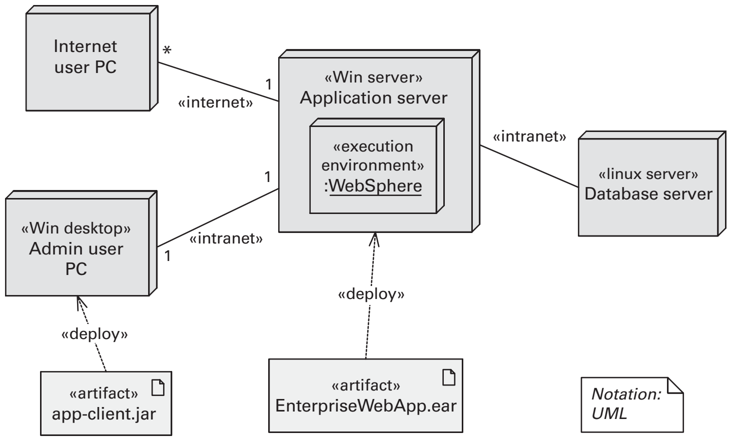
In UML, a deployment diagram is a graph of nodes connected by communication associations. Nodes correspond to processing elements, usually having a memory and a processing capability. Nodes may contain component instances, indicating that the component resides on the node. Components can be connected to each other by dependency arrows. In a UML deployment diagram, components may contain objects, meaning that the objects are part of those components. Migration of components from node to node (or objects from component to component) is shown by the <<becomes>> dependency stereotype. A node is shown using a symbol that looks like a three-dimensional box, with an optional name inside. Nodes are connected by associations that stand for communication paths. The precise nature of the communication path can be indicated by a stereotype on the association (for example, <<10-T Ethernet>>, <<RS-232>>). Properties are represented as attribute name-value pairs (for example, processorSpeed = 300 mHz, memory = 128 MB). A deployment specification specifies the parameters guiding deployment of a component, such as the mode of concurrency (for example, thread, process, none).

Figure 5.3 shows an example of UML notation for a deployment view.

5.2.5 Relation to Other Styles

The deployment style is related to the C&C styles that provide the software elements that are allocated to the hardware of the computing platform. It is also closely related to the install style, which shows the contents of the files deployed to hardware nodes.

Figure 5.3
A deployment view in UML, showing the hardware platform supporting a Java EE system. The <<deploy>> dependency shows which artifacts are deployed to which nodes. <<execution environment>> is a node that offers an environment to run specific types of components. To know what components are deployed to a specific node, you need to look at the install view to see what components go inside each artifact.


Internet user PC
*
1
«internet»
«Win server» Application server
«execution environment» :WebSphere
«intranet»
«linux server» Database server
1
«Win desktop» Admin user PC
«intranet»
1
«deploy»
«deploy»
«artifact» app-client.jar
«artifact» EnterpriseWebApp.ear
Notation: UML

5.3 Install Style

5.3.1 Overview

The install style allocates components of a C&C style to a file management system in the production environment. Once a software system is implemented, the resulting files have to be packaged to be installed on the target production platform (such as a desktop computer or a server machine running an application server). These files include libraries, executable files, data files, log files, configuration and version control files, license files, help files, deployment descriptors, scripts, and static content (for example, HTML files and images). For a large software system, the number of files installed in the production environment can reach the thousands. These files need to be organized so as to retain control over and maintain the integrity of the system build and package process, as well as to help deployers and operators locate and manipulate the files when necessary. Configuration management techniques, build tools, and installation tools usually help to get this job done. But an architecture description shows how the installed system is organized as a structure of files and folders, and describing how software elements map to that structure is important to assist developers, deployers, and operators.

The install style helps describe what specific files should be used and how they should be configured and packaged to produce different versions of the system. Maintaining multiple versions simultaneously is a common practice for many systems. Different versions of the same system may

- Support internationalization
- Offer different pricing (for example, a free version and a commercial version)
- Accommodate customizations for different clients
- Support clients in a distributed system that still send old-version message requests

Once the implementation is in place, configuration management tools and build scripts help to automate the process of selecting, configuring, and packaging the right configuration items for different versions. But the architecture describing this, possibly quite intricate, structure of files and folders should be initially captured in an install view.

Managing multiple versions involves not only the artifacts packaged for deployment but also implementation artifacts (such as source files). The implementation view, introduced in Section 5.5, describes the structure of files and folders in the development environment. The implementation and install views together describe the structures containing all software artifacts that are version controlled.

5.3.2 Elements, Relations, and Properties

Table 5.3 summarizes the principal characteristics of the install style. Environmental elements in an install view are configuration items: files and folders in a file system, which are organized in a tree structure. The software elements are C&C components, such as processes, threads, servlets, or data stores.

Two relations in the install style are

- *Allocated-to.* A relation between components and configuration items. This relation connects a component with the file or folder that stores that component in the file system.
- *Containment.* A folder in the file system contains other folders and/or files. Likewise, a file (such as a zip file) may contain other files and folders. Also, a given file or folder may be contained in multiple files or folders—for example, for multiple installed versions.

Table 5.3 Summary of the install style

Overview	The install style describes the mapping of components in the software architecture to a file system in the production environment.
Elements	• *Software element: a C&C component*. *Required* properties of a software element, if any, usually include requirements on the production environments, such as a requirement to support Java or a database, or specific permissions on the file system. • *Environmental element: a configuration item, such as a file or a folder. Provided* properties of an environmental element include indications of the characteristics provided by the production environments.
Relations	• *Allocated-to*. A component is allocated to a configuration item. • *Containment*. One configuration item is contained in another.
Constraints	Files and folders are organized in a tree structure, following an *is-contained-in* relation.

As with the deployment style, the important properties of the software and environmental elements of the install style are those that affect the allocation of the software to configuration items. For example, how a configuration management system deals with histories and branches is a configuration item property; a specific version of the Java runtime environment to use might be a required property of a software component. An install view might be designed to make extensive use of variation points, because installation requirements will likely be different on different platforms.

Section 6.4 explains what variation points are and how to document them.

5.3.3 What the Install Style Is For

Understanding the organization of the files and folders of the installed software can help developers, deployers, and operators carry out the following tasks:

- Create build-and-deploy procedures
- Navigate through a large number of files and folders that constitute the installed system, to locate specific files that require attention (such as a log file or configuration file)
- Select and configure files to package a specific version of a software product line
- Update and configure files of multiple installed versions of the same system
- Identify the purpose or contents of a missing or damaged file, which is causing a problem in production
- Design and implement an "automatic updates" feature

The required properties of the software elements in the install style can also be used to support the analysis of purchasing options for production environments.

5.3.4 Notations for the Install Style

Any notation for an install view must show components, the files and folders, and the mapping between them. The tree structure organization of the files and folders should also be shown. UML provides a number of built-in facilities to aid in showing an install view, including the <<artifact>> stereotype to denote a file (configuration item) and the <<manifest>> artifact to indicate containment.

Duke's Bank is an example application used in Sun's online Java tutorial. See java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html.

Figure 5.4 shows an install view diagram from the Duke's Bank application using an informal notation, and Figure 5.5 shows the same diagram rendered in UML.

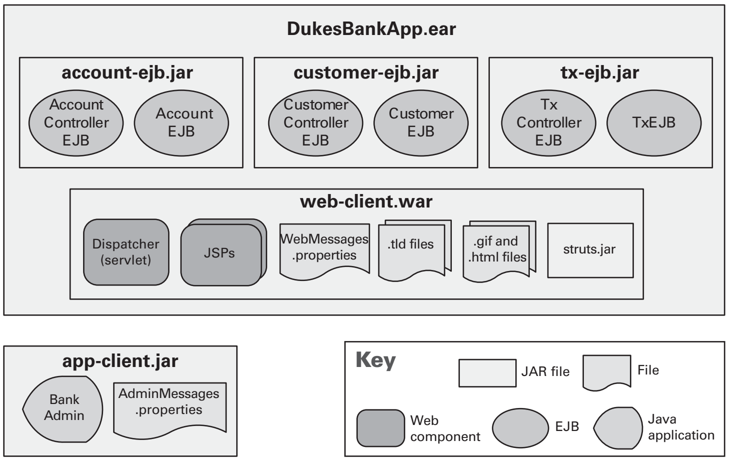


Figure 5.4
An install view in an informal notation, from the Duke's Bank Java EE application. Java applications are usually deployed in Java archive (JAR) files. Like a zip file, a JAR file may contain other files. Enterprise JavaBean JAR files contain EJB classes and other files that the EJBs may need. Web archive (WAR) files contain Web components (servlets and Java-Server Pages [JSPs]); very often, they also contain HTML, JPEG, and other files used in Web pages for "static content." Enterprise archive (EAR) files are a packaging of zero or more JAR and zero or more WAR files. All server-side components are inside DukesBankApp.ear, which is deployed to the application server. The diagram also shows that the client-side BankAdmin Java application is deployed in app-client.jar, which is deployed to the admin user's machine.

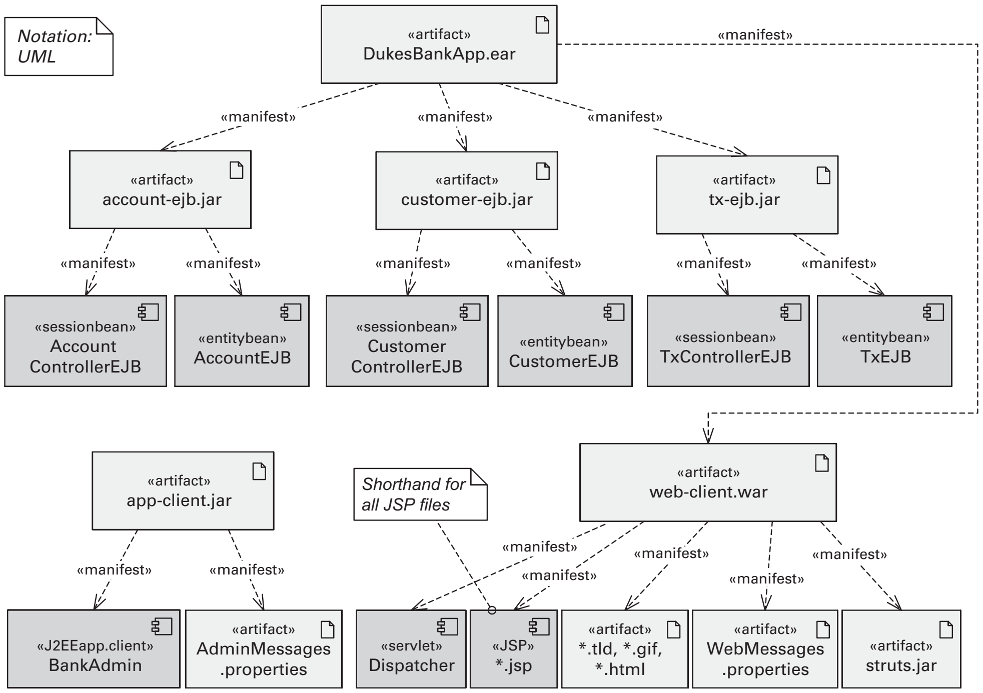

Notation: UML
«artifact» DukesBankApp.ear
«manifest»
«artifact» account-ejb.jar
«artifact» customer-ejb.jar
«artifact» tx-ejb.jar
«sessionbean» Account ControllerEJB
«entitybean» AccountEJB
«sessionbean» Customer ControllerEJB
«entitybean» CustomerEJB
«sessionbean» TxControllerEJB
«entitybean» TxEJB
«artifact» app-client.jar
Shorthand for all JSP files
«artifact» web-client.war
«J2EEapp.client» BankAdmin
«artifact» AdminMessages .properties
«servlet» Dispatcher
«JSP» *.jsp
«artifact» *.tld, *.gif, *.html
«artifact» WebMessages .properties
«artifact» struts.jar

Figure 5.5
The install view of Figure 5.4 rendered in UML. The <<artifact>> stereotype denotes a file of any kind. The <<manifest>> stereotype indicates that a given component, class, or other artifact is inside a given artifact.

5.3.5 Relation to Other Styles

The install style is most strongly related to the C&C styles that describe the software elements for the allocation. The deployment style is also closely related because it shows the hardware elements where the files in an install view are deployed to.

5.4 Work Assignment Style

5.4.1 Overview

The work assignment style allocates modules of a module style to the groups and individuals who are responsible for the realization of a system. This style defines the responsibility for implementing and integrating the modules to the appropriate development teams. The style is typically used to link activities to resources to ensure that the modules are each assigned to an individual or team. The architecture in combination with process determines the actual allocations.

A common managerial tool is the work breakdown structure (WBS). This tool defines a project and groups the project's discrete work elements in a way that helps organize and define the total work scope of the project. Software WBSs have always been based on some decomposition of the system being built into parts: the modules of a module style.

Because work assignments represent a mapping of the software architecture onto groups of humans, it is an important allocation style. Teams—and hence work assignments—are not simply associated with writing code that will run in the final system. There are many more tasks that humans must perform: configuration management, testing, evaluation of potential commercial off-the-shelf products, ongoing product sustainment, and so on.

Even if a module is purchased in its entirety as a commercial product without the need for any implementation work, someone still has to be responsible for procuring it, testing it, and understanding how it works, and someone has to "speak" for it during integration and system testing. The team responsible for that has a place in a work assignment view, just as do teams responsible for implementing "homegrown" modules.

Moreover, software written to support the building of the system—tools, environments, test harnesses, and so on—and the responsible team have a first-class place in a work assignment view.

5.4.2 Elements, Relations, and Properties

The elements of this style are software modules and the groups of people in the development organization.

Table 5.4 Summary of the work assignment style

Overview	The work assignment style describes the mapping of the software architecture to the teams in the development organization.
Elements	• *Software element: a module.* Properties include the required skill set and available capacity (effort, time) needed. • *Environmental element: an organizational unit, such as a person, a team, a department, a subcontractor, and so on.* Properties include the provided skill set and the capacity in terms of labor and calendar time available.
Relations	*Allocated-to*. A software element is allocated to an organizational unit.
Constraints	In general, the allocation is unrestricted; in practice, it is usually restricted so that one module is allocated to one organizational unit.

In this style, the *allocated-to* relation maps from software elements to organizational units.

A well-formed work assignment relation has the property of completeness—all work is accounted for—and no overlap—no work is assigned to two places. Properties of the software elements may include a description of the required skill set, whereas properties of the people elements may include provided skill sets.

Table 5.4 summarizes the characteristics of the work assignment style.

5.4.3 What a Work Assignment Style Is For

The work assignment style shows the major units of software that must be present to form a working system and who will produce them, as well as the tools and environments in which the software is developed (and their assignments to environmental elements). The work assignment style helps with planning and managing team resource allocations, assigning responsibilities for builds, and explaining the structure of a project—to a new hire, for example. The work assignment style can give each team its charter.

This style is the basis for work breakdown structures and for budget and schedule estimates.

5.4.4 Notations for the Work Assignment Style

No special notations exist for showing work assignment views. Among informal notations, a table showing software elements and responsible teams is often sufficient.

Tabular notes are a very simple and clear form of description for work assignment views. The architect doesn't need to choose the team but rather provide information to management. Later, the actual team assignments can be added.

ECS Element (Module)		
Segment	**Subsystem**	**Organizational Unit**
Science Data Processing Segment (SDPS)	Client	Science team
	Interoperability	Prime contractor team 1
	Ingest	Prime contractor team 2
	Data Management	Data team
	Data Processing	Data team
	Data Server	Data team
	Planning	Orbital vehicle team
Flight Operations Segment (FOS)	Planning and Scheduling	Orbital vehicle team
	Data Management	Database team
	User Interface	User interface team
. . .	. . .	. . .

Figure 5.6
Work assignment view using a tabular notation. The left two columns echo the system's module decomposition structure.

Figure 5.6 shows the primary presentation for a work assignment view of a NASA system called ECS. In the decomposition view for ECS, the highest level modules are called segments; those are decomposed into units called subsystems.

5.4.5 Relation to Other Styles

The decomposition style is discussed in Section 2.1.

The work assignment style is strongly related to the decomposition style, because that is the most common basis for its allocation mapping. A work assignment view may extend the module decomposition by adding modules that correspond to development tools, test tools, configuration management systems, and so forth, whose procurement and day-to-day operation must also be allocated to an individual or a team.

Combining views is discussed in Section 6.6.

A work assignment view is often combined with other views. For example, team work assignments could be the modules in a decomposition view, the layers in a layered view, the software associated with tiers in an *n*-tier architecture, or the software associated with tasks or processes in a multi-process system. You could augment those views by annotating the various software elements with the name of the team assigned to each. Or you could document the assignments as an additional property of the software elements.

The creation of a work assignment view—whether maintained separately or combined with another—enables the architect and the project manager to give careful thought to the best way to divide the work into manageable chunks. This approach also helps keep explicit the need to assign responsibility for all software, such as the development environment that will not be part of the deployed system. A danger of combining work assignments with other views is that the work assignments associated with tool building may be lost; in many situations, the ancillary software tools are not part of the actual system and do not appear in any of other views.

PERSPECTIVES

Why Is a Work Assignment View Architectural?

A work assignment view maps software elements (modules) to environment elements (units in a development or acquisition organization). It shows who is responsible for developing each piece of the system. Some people, when confronted with our prescription to consider designing and documenting a work assignment view as part of the architecture, balk. “Wait,” they say. “It is not part of the architect’s responsibilities to assign work to people. That’s what project management is for.” It’s a fair question.

About four years ago, I was part of a large U.S. government defense project that was just getting off the ground. It was a system of large interacting systems, each complex and, in several cases, unprecedented. The government decided that it needed to choose a major contractor to develop a key part of this project, and to oversee the development and integration of the rest of it. After that, it needed to award participating contracts to many other companies to build the other pieces of the system.

This project was predicted to comprise several tens of millions of lines of code, with a price tag in the billions of dollars. Contracts, especially sizable ones, take a long time to go through the competitive procurement process. There are massive “requests for proposals” publicly circulated, which precipitate massive bid proposals in response, which in turn trigger massive source selection processes. Even if there are no protests filed by any of the losing bidders, which can send the process back to the beginning, it takes months or years to award a contract and begin work. Government acquisition keeps legions of lawyers on both sides gainfully employed.

Knowing all of this, the government agency procuring the system had a tangible incentive to get the contract process under way as soon as possible. The clock was ticking; this system was going to replace several others whose withdrawal from service had already been planned. Contracting for this project occupied everyone's attention for well over a year.

Eventually the prime contractor and the major subcontractors were in place, and a huge sigh of relief was palpable throughout parts of Washington. "Now," said the project manager, "let's find an architect to design the system."

Do you see the flaw here? The first day on the job, the architect was confronted with a *de facto* decomposition, based on contracting concerns. It is fair to say that this was not the decomposition the architect would have wanted, and if he had been able to say so earlier, the contracts awarded might well have been different. The architect was concerned with exploiting the inherent commonality and managing the variation across the several versions of the system he knew were going to be deployed. He was also concerned with injecting some commonality across the subsystems; in particular, he knew they all needed a common look and feel. He might have created an architecture element to provide that. But with nothing in any of the contracts to cover this, he was reduced to writing it into the architecture "guidelines," which were not always followed with hoped-for rigor. We know that the module decomposition structure of a software system is primarily where its modifiability is created. It is doubtful that the government contracting experts either possessed the domain expertise or conducted a domain analysis to see what likely changes were in store and design the decomposition accordingly.

This example and others make me believe that a work assignment view is an important architectural contribution. And yet the skeptics have a point. Aren't we asking architects to make project management decisions? In this example, project managers were making *de facto* architectural decisions, and the result, predictably, was a poor architecture. The solution seems obvious: architects and project managers should work together on this and other issues. In particular, the architect can inform management about the decomposition and the skill set needed to ensure the successful development of each piece. Having the architect involved in the beginning ensures that the module decomposition drives the work assignments and not the other way around. Having a place for a work assignment view all ready and waiting in the architecture document can help the architect engage his or her project manager in a conversation about filling it in.

—P.C.

5.5 Other Allocation Styles

So far in this chapter you've seen that hardware, file management systems, and team structure all interact with the software architecture. We've shown a style that captures each of these allocations from software to external-to-software structures.

There are many other useful mappings of this variety. Here are a few for you to consider:

- *Implementation style.* The implementation style describes how the development environment is organized in a tree structure of files and folders and how modules from a module view map to that structure. When you apply the implementation style to a system, the resulting implementation view shows how files and folders should be arranged to host the implementation units: classes, programs, scripts, test cases, make files, documentation files, and any other artifacts created when the system is developed. The implementation view helps developers to navigate and locate *development* artifacts, and to place new artifacts in the proper place. The implementation view also helps in the implementation of version control and configuration management policies. The implementation style is similar to the install style, but instead of showing files and folders in the production environment, it shows the organization of files and folders in the development environment. A screenshot of your development environment tool (which manages the implementation environment) often makes a very useful and sufficient diagram for your implementation view.

If your development organization will create multiple software systems and wants all of them to follow the same structure for the files and folders in the development environment, you should document an implementation view that serves as a reference for all these software projects.

- *Data stores style.* The data stores style describes the mapping between the software's data entities and the hardware of the data servers on which the software resides. When you apply the data stores style to a system, the resulting data stores view shows how the tables containing data described in the data model style are distributed over servers. It might show to which servers stored procedures have been allocated. It might show geographic distribution of the database or database replication. It might also show the machines that host data warehouses and the data stores that feed them. These and other similar relations are important for addressing concerns about data availability, resilience of data to physical attack or cyberattack, as well as how data accesses affect overall system performance. The data stores style is similar to a deployment style, except that (instead of C&C components) it shows data entities allocated to hardware.

Section 2.6 discusses the data model style.

Other allocation styles are possible. You could define a requirements-allocation style that maps between system requirements and the software elements of the architecture that satisfy them; that's one way to document a mapping between requirements and design. And for projects spread across many teams and sites, a coordination view can be an important tool to bring the architecture and the development organization into alignment.

Section 10.3 discusses ways to capture mappings from requirements to software.

See "Perspectives: Coordination Views" on page 209, in this chapter.

A style is a **specialization** of another style if it is consistent with that style—that is, doesn't violate it—and adds more constraints to its element types, relation types, and/or topological restrictions.

See the MSDN Web site, msdn.microsoft.com/en-us/library/ms978694.aspx.

See the IBM Redbooks Web site, www.redbooks.ibm.com/abstracts/sg246446.html.

There are also useful **specializations** of the styles discussed in this chapter. For example:

- *Specializing the deployment style.* The deployment style as presented comes with no inherent topological restrictions, but you might find certain patterns of deployment to be particularly useful. Microsoft publishes a "Tiered Distribution" pattern, which prescribes a particular allocation of components in a multi-tier architecture to the hardware they will run on. This pattern specializes the generic deployment style. If you adopt and document this pattern for your system, the result will be a Tiered Distribution view. Similarly, IBM's WebSphere handbooks describe a number of what they call "topologies" along with the quality attribute criteria for choosing among them. There are 11 topologies (specialized deployment views) described for WebSphere version 6, including the "single machine topology (stand-alone server)," "reverse proxy topology," "vertical scaling topology," "horizontal scaling topology," and "horizontal scaling with IP sprayer topology."
- *Specializing the work assignment style.* You can also document often-used team structure patterns as specializations of the work assignment style. In Urdangarin et al. 2008, the authors describe a number of team-organization approaches for globally distributed Agile projects. Each constitutes a specialized work assignment style:
  - *Platform style.* In a software product line development, one site is tasked with developing reusable core assets of the product line, and other sites develop applications that use the core assets.
  - *Competence-center style.* Work is allocated to sites depending on the technical or domain expertise located at a site. For example, user-interface design is done at a site where usability engineering experts are located.
  - *Open-source style.* Many independent contributors develop the software product in accordance with a technical integration strategy. Centralized control is minimal, except when an independent contributor integrates his code into the product line.

  They also identify two other organizational allocation schemes that technically do not qualify as specializations of the work assignment style, because they allocate something other than modules to organizational units:

  - *Process-steps style.* Work is allocated across the sites in accordance with the phases of the software development process;

for example, design may be done at one site, development at another site, and testing at yet another site.

– *Release-based style.* The first product release is developed at one site, the second at another site, and so on. Often the releases will be overlapped to meet time-to-market goals; for example, one site is testing the next release, another site is developing a later release, and yet another site is defining or designing an even later release.

PERSPECTIVES

Coordination Views

With Jim Herbsleb

A *coordination view* can be an important tool to bring the architecture and the development organization into alignment, particularly for projects spread across many teams and sites.

The motivation for a coordination view stems from the limitations of communication as a coordination mechanism. Conway (1968) observed decades ago: "Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure." Small teams can coordinate their work rather simply through frequent communication. But since the number of potential communication paths increases as the square of the number of team members, this strategy does not scale. The usual solution is to divide a system into parts that have limited, well-specified interactions, so that developers working on one part do not need to coordinate their work with developers working on other parts. In the software domain, Parnas (1972) observed long ago that in thinking about criteria for partitioning code into modules, they should be thought of not as subprograms but as "work items" that can be assigned to teams.

Modularization is an essential strategy for allowing development projects to coordinate their work, but it is generally not sufficient. Modules are not completely independent—after all, they form a single system and must therefore interact in some way—and for this reason, the need for teams to coordinate is rarely eliminated completely. In some cases, only minimal coordination will be required, but in other cases, intensive coordination is necessary. The full picture is more complicated, as coordination can happen through shared representations, by prearranged plans and interfaces, and even by a shared work history that enables teams to predict the actions of other teams.

Just as the *need to coordinate* development work can vary dramatically across modules, the *capacity to coordinate* can vary dramatically across teams in a project. The key to successful project coordination is to *ensure that the coordination required among teams never exceeds the capacity of those teams to coordinate* (Cataldo et al. 2006). A coordination view is a tool that can be used to help ensure this condition is not violated.

The key to a coordination view is representing *complexity* and *uncertainty* in the relations between modules. Complexity implies that a module and its interfaces are likely to be difficult to understand and to use correctly. Uncertainty means that potentially complicated communication and negotiation between teams and with architects must occur as interfaces are worked out or the allocation of functionality to modules is determined. Representing complexity and uncertainty separately is important, since the means for addressing them are generally quite different. Complexity is generally addressed by detailed documentation, a tactic that is much less useful for handling uncertainty. Frequent Agile-style communication is often an effective way to address uncertainty, but it is easily overwhelmed and ineffective at high levels of complexity. It is important for a useful coordination view to represent both.

Dependency structure matrices are discussed in Section 1.4.3.

One straightforward form a coordination view can take is derived from matrices that represent the relations among modules and the coordination capacities of the project teams. Module relations are represented by two square matrices (like dependency structure matrices), of dimension in the number of modules, with each entry taken from the domain <0, 1, 2, 3>, representing an uncertainty (**UM**) or complexity (**CM**) relation between two modules. Zero values indicate modules are not related in any substantial way, while 1, 2, and 3 represent, respectively, low, moderate, and high levels of complexity or uncertainty in the interaction of the modules. Values can be assigned in a variety of ways, for example by an expert such as the lead architect.

These matrices can be used in conjunction with square matrices representing the relevant coordination capacity of pairs of development teams. The *communication capacity matrix* (**CCM**) represents the ease and facility with which two teams can be expected to communicate. This expectation depends on such factors as facility in a common language, cultural similarity, degree of overlap in work hours, use of similar communication technologies, and past experience successfully communicating with each other or similar teams. The *documentation capacity matrix* (**DCM**) represents the ease and facility with which two teams can be expected to create relevant documentation and achieve a common understanding of it. This expectation depends on such factors as experience with relevant notations (for example, are both teams experienced in UML if that is the chosen format), history of creating and maintaining detailed and accurate documentation of APIs, and the demonstrated willingness to publish and read documentation. For both matrices, the values can again be taken from the domain <0, 1, 2, 3>, representing approximate levels of communication or documentation capacity of pairs of teams.

We now have four square matrices: two of dimension number of modules (**UM**, **CM**), and two of dimension number of teams (**CCM**, **DCM**). In order to compare coordination needs with coordination capacities, it is necessary to express both as relations among teams. Some additional computation with **UM** and **CM** will achieve this. All that is required is to use an allocation view in the form of a binary matrix **AM** of teams by modules, where an entry of 1 indicates that a team is responsible for a given module. The following multiplication represents the degree to which each pair of teams can expect to be required to coordinate uncertainties (where $\mathbf{AM}^T$ is the transpose of **AM**).

$$\mathbf{AM} \times \mathbf{UM} \times \mathbf{AM}^T = \mathbf{CR}_U$$

The product $\mathbf{CR}_U$ is a square matrix of dimension in the number of teams, where entries give an indication of the extent to which each pair of teams is working on modules that interact with uncertain interfaces and/or uncertain allocation of functionality. This indication is very approximate, but a comparison of values in $\mathbf{CR}_U$ and **CCM** should give useful indication where much communication is going to be required (relatively large entries in $\mathbf{CR}_U$) and little communication capacity exists (relatively small entries in comparable cells of **CCM**). Such mismatches should trigger discussions about how communication can be supported or how work can be reassigned in order to sidestep communication problems. An analogous computation substituting **CM** for **AM** and **DCM** for **CCM** will provide a comparison of the need of teams to coordinate through documentation and their capacity to do so.

Work assignment views are discussed in Section 5.4.

Let's illustrate with a small example. Let **AM**, a matrix of dimension teams by modules, represent assignment of code modules to teams for development. This is simply a work assignment view. **UM** is an uncertainty matrix, representing the lead architect's judgment about the relative degree of uncertainty, for each pair of modules, of the interface and the allocation of functionality between the modules. $\mathbf{CR_U}$ represents the extent to which each pair of teams can expect to be required to coordinate uncertainties.

$$\overset{\mathbf{AM}}{\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}} \times \overset{\mathbf{UM}}{\begin{bmatrix} - & 0 & 3 & 0 & 0 \\ 0 & - & 0 & 1 & 1 \\ 3 & 0 & - & 1 & 3 \\ 0 & 1 & 1 & - & 2 \\ 0 & 1 & 3 & 2 & - \end{bmatrix}} \times \overset{\mathbf{AM^T}}{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}} = \overset{\mathbf{CR_U}}{\begin{bmatrix} - & 4 & 7 \\ 4 & - & 2 \\ 7 & 2 & - \end{bmatrix}}$$

$\mathbf{CR_U}$ can now be compared with communication capacities of the teams, **CCM**, or used to plan how the work is assigned. It is a good bet, for example, because of the work they are performing, teams 1 and 3 will require a very robust communication capacity, if not collocation. This because of the considerable uncertainty between modules 1 and 3 as well as 3 and 5. Teams 2 and 3 will have relatively little need to work out uncertainties, meaning they can probably be located anywhere and will need no special communication technologies. Teams 1 and 2 will have a moderate need to communicate, suggesting they should work in time zones that allow overlapping work hours and have adequate teleconferencing and perhaps instant-messaging technologies. Their coordination success should be carefully monitored to ensure they don't get out of sync.

Additional experience with coordination views will eventually tell us when this simple construction is sufficient, and when more nuanced schemes, perhaps attuned to architecture styles or other key attributes, will add value. We may also need more systematic ways of assigning values to both need and capacity. Such issues are the subject of ongoing research (Urdangarin et al. 2008; Avritzer, Cai, and Paulish 2008).

5.6 Summary Checklist

- Allocation styles map software elements to elements in the environment of the software.
- A deployment view describes the mapping of runtime software elements to the hardware of the computing platform on which the software executes.
- An install view describes the tree structure of files and folders in the production environment and how the software components are mapped to that structure.
- A work assignment view describes the mapping of modules onto the people, groups, or teams tasked with the development of those modules.

5.7 Discussion Questions

1. Consider a network diagram created by the network administrator in the IT department of your organization. How does that diagram compare with a deployment view? What is missing?
2. Suppose that you needed to map the modules under test to the test harness that generates inputs, exercises the modules, and records the outputs. Sketch an allocation style that addresses this concern.
3. In one project, short identifiers were assigned to every module. A module's full name consisted of its identifier, prefixed by its parent's identifier, separated by a period (.). The project's file structure was defined by a short memorandum stating the path name of a root directory and further stating that each module would be stored in the directory obtained by changing each period in the module's full name to a slash (/). Did this memorandum constitute an implementation view for this system? Why or why not? What are the advantages and disadvantages of this scheme?
4. Suppose that your system can be deployed on a wide variety of computing platforms and configurations. How would you represent that?

5. Besides the ones in this chapter, identify as many other structures in the environment of a software system as you can. Pick a few and answer the following: What software elements would map to it? Create an example primary presentation for a corresponding view. Discuss to whom such a view would be useful and what concerns it would address.
6. Many deployment tools and integrated development environments provide views of the development and production environments that allow you to easily understand and navigate the tree structure of files and folders. Do you think these tools can fill the need for creating install views or implementation views in the architecture documentation? Why so, or why not?

5.8 For Further Reading

Both the install style and the implementation style are aligned with the broad topic of software configuration management (SCM). An in-depth treatment of SCM is far beyond the scope of this book, but you can begin investigating the topic by looking at the documentation of SCM and version-control tools, such as Subversion, CVS, Perforce, ClearCase, and Visual SourceSafe. The Siemens Four View model defines a code architecture view that explains how the software implementing the system is organized into source, intermediate, and deployment components and related decisions regarding build and installation procedures and configuration management (Hofmeister, Nord, and Soni 2000).

In the 1960s Conway (1968) formulated a law that the architectural structure mirrors the organizational structure. He based his law on ease of communication within as opposed to across groups. This law is an organizational articulation of coupling and coherence. Architecture-based management of software projects is also discussed in the book by Paulish (2002). He has observed that accurate time and budget estimates depend on basing them on the software architecture. This is the place where a work assignment view comes into play; Paulish's observation has a strong intuitive base, as the time and budget estimates depend on the work breakdown structure, which in turn depends on the software architecture. More recently, Avritzer and others have observed many different organizational approaches to assign work in globally distributed teams (for example, product structure, process steps, release-based, computing platform structure, competence center, and open source) (Avritzer, Cai, and Paulish 2008). Avritzer explicitly discussed assigning work in globally distributed teams.

Beyond Structure: Completing the Documentation

PART II

Part I presented a substantial repertoire of useful architecture styles. An architect can choose from among these styles, pick styles in other style catalogs, or design a new style. Once a style is chosen, the view based on it needs to be designed and documented. The chapters in Part I presented ways to document the elements and relations that populate a view.

But documenting a view involves more than just writing down (or more often, drawing) the elements and their relations. Elements have interfaces, and those need to be documented so that teams developing other elements can interact with them correctly. Elements have behavior, and confederations of elements have collective behavior, which needs to be documented so that implementers know what the elements they're coding should do, and so that analysts can tell if the architecture is satisfying the system's behavioral requirements. Architects need a way to explain their design—what drove them to make the design decisions they did. Documenting rationale is a critical but often underpracticed part of an architect's duties.

Great things are not done by impulse, but by a series of small things brought together.

—Vincent van Gogh

These and other kinds of information are important parts of the architecture document. Part II deals with those.

- Chapter 6 explores documentation techniques such as refinement and chunking of information, context diagrams, creating and documenting combined views, documenting variability and dynamism, and documenting the rationale behind architectural decisions.
- Chapter 7 tells how to document the interfaces of architecture elements. It provides ways to document the existence of interfaces, the syntax (or signature) of an interface, and the semantics of an interface.

- Chapter 8 explores another essential technique for architects: documenting the behavior of an element or an ensemble of elements. Documenting behavior is an essential counterpoint to documenting static structure. This chapter covers the techniques and notations available for expressing the behavior of elements, groups of elements, and the system as a whole.

Beyond the Basics 6

This chapter contains guidelines for dealing with several aspects of documentation that either span views or are not specific to any particular category of views:

6.1 *Refinement.* Refinement is a way to reveal more information over time as it becomes available. Refinement reflects how architectures develop over time, and it lets architects present information in more or less detail to serve various audiences. This section discusses two kinds: decomposition refinement and implementation refinement.

6.2 *Descriptive completeness.* Does your architecture document tell the truth, the whole truth, and nothing but the truth? There may be good reasons why it doesn't.

6.3 *Documenting context diagrams.* A context diagram establishes the boundaries for the information contained in a view. A context diagram for the entire system defines what is and is not in the system, thus setting limits on the architect's tasks. This section discusses how to document context diagrams, and how to tailor context diagrams for each view.

6.4 *Documenting variation points.* Some architectures provide built-in variation points to facilitate building a family of similar but architecturally distinct systems. Other architectures are *dynamic,* in that the systems they describe change their basic structure while they are running.

6.5 *Documenting architectural decisions.* Why we made architectural decisions the way we did is just as important as the results of those decisions. This section discusses how to record the rationale behind your design.

6.6 *Combining views.* Prescribing a given set of rigidly partitioned views is naive; there are times and good reasons for combining two or more views into a single combined view.

6.1 Refinement

Refinement is the process of gradually disclosing information across a series of descriptions.

Architects need a way to carry out their designs and present information in a view in manageable chunks. **Refinement** allows the architect to present information in separate, digestible pieces. A refinement elaborates on (adds information to) an existing representation. Refinement allows the architect to capture and present information with more or less detail. Less detail is useful in early stages of design, and excellent for introductions, overviews, and early conceptualizing.

There are two important kinds of refinement: decomposition refinement and implementation refinement.

6.1.1 Decomposition Refinement

Decomposition refinement is a refinement in which a single element is elaborated to reveal its internal structure. Each member of that internal structure may be recursively refined.

A **decomposition refinement** elaborates a single element to reveal its internal structure and then recursively refines each member of that internal structure. The text-based analogy of this is the outline, whereby major sections (denoted by roman numerals) are decomposed into subsections (denoted by capital letters), which are decomposed into sub-subsections (denoted by Arabic numerals), and so forth.

Using decomposition refinements in a view carries an obligation to maintain consistency with respect to the relation(s) native to that view. For example, suppose that the relation shown in Figure 6.1(a) is *send-data-to*. Because element B is shown as both receiving and sending data, the refinement of B in Figure 6.1(b) must show where data can enter and leave B: in this case, via B1.

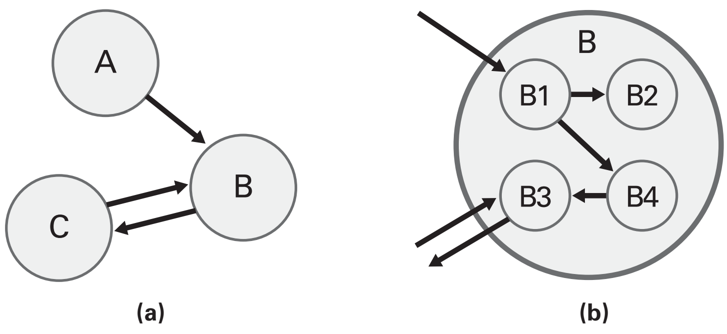


Figure 6.1
(a) A hypothetical system consisting of three elements: A, B, and C. Arrows signify data flow.
(b) Element B is refined to show that it consists of elements B1, B2, B3, and B4. Because B has two inputs and one output, B's decomposition refinement must satisfy that obligation. Children B1 and B3 receive the inputs; B3 produces the output.

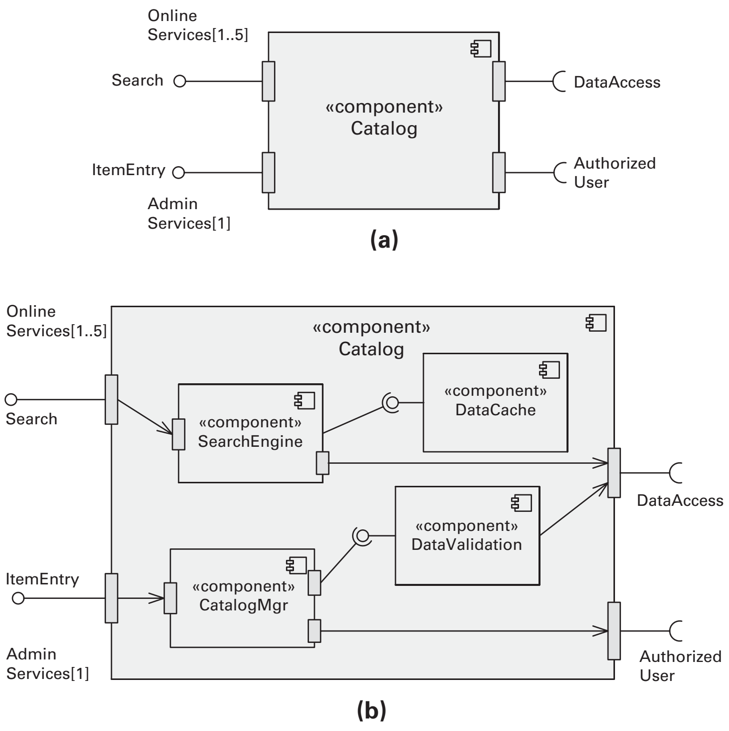


Figure 6.2
Showing decomposition refinement in UML 2.x. Figure 6.2(b) is a decomposition refinement of Figure 6.2(a).

Decomposition refinement is straightforward to depict in UML if the UML construct representing the elements supports nesting, such as a component or a package. Inside the refined element, use delegation connectors to show the association between the outer element's interfaces and the inner elements. Figure 6.2 shows an example.

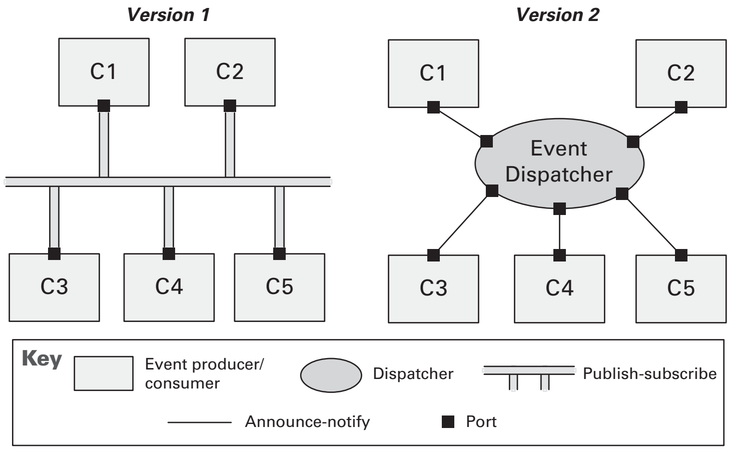
6.1.2 Implementation Refinement

Another kind of refinement, called **implementation refinement**, shows the same system—or portion of the system—in which many or all the elements and relations are replaced by new ones, usually of a different type. Unlike a decomposition refinement, the scope doesn't zoom in, but remains fixed. The implementation refinement reveals information showing how the original construct will be realized.

Implementation refinement is a refinement in which some or all of the elements and relations are replaced by other, more implementation-specific, elements and relations.

For example, imagine two views of a publish-subscribe system, as shown in Figure 6.3. In one view, components are connected by a single event bus. In the refined view, the bus is replaced by an event dispatcher to which the components make explicit calls to achieve their event announcements.

Figure 6.3
Version 2 is an implementation refinement of version 1, showing that the publish-subscribe bus is actually realized by an event dispatcher.



6.1.3 Spectrum of Design

Through the use of refinement, architects can manage the specificity of their architecture documentation (and its underlying architecture design). This varies depending on a variety of factors, such as the stage of design, the amount of resources available to nail down the design and produce the corresponding documentation, the audience for whom that documentation is being written, and the maturity of the system. The result is a spectrum of design.

A marketecture . . . is a one page, typically informal depiction of the system's structure and interactions. It shows the major components, their relationships and has a few well chosen labels and text boxes that portray the design philosophies embodied in the architecture. A marketecture is an excellent vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process. It's easy to understand and explain, and serves as a starting point for deeper analysis.

—Ian Gorton, *Essential Software Architecture* (2006, p. 6)

At the left end of the spectrum, the designs (and their documentation) are broad, very abstract, and unrefined. Early in the design, broad information is all the architect has. Happily, the documentation of these early design stages is not wasted. Architects often need to convey broad architectural understanding quickly to an audience that includes nontechnical stakeholders: sponsors, managers, chief information officers, visitors, and others. Such stakeholders do not want to pore over a complete architecture document. The description doesn't have to be precise, it may not even need to be completely accurate, and the intent is not to instill deep understanding in the audience. Sometimes the intent is to instill a sense that the people doing the presentation know what they're talking about.

Of course, other stakeholders (such as developers and those who need to analyze specific properties of the architecture) need the whole picture. They are the consumers of the documentation after detail and elaboration have been added through the progressive refinement that happens as the architect collects more information and makes more design decisions.

One of the specific ways that the design moves to the right along the spectrum is through *style specialization.*

I've often found the need for four different architecture presentations: the slides for the 10-minute presentation, the slides for the 1-hour presentation, the 50-page document, and finally the full document.

—Philippe Kruchten

6.1.4 Style Specialization

When picking a style for a view, one important dimension of choice is how specialized that style is. The more specialized a style is, the more constrained the architecture design space that uses it will be. In exchange for limiting the class of systems that are in the scope of that style, specialization has a number of benefits, including the following:

- Stronger guidance for the architect, through the inclusion of constraints associated with the style
- The ability to exploit specialized analyses, by leveraging semantic properties of the system, such as computational model and style-specific properties
- Reuse of implementation, such as middleware to support communication and common services for components in that style

"Advice: Building an Architecture Overview Presentation," on page 364 in Chapter 10, shows how to build a viewgraph presentation from a software architecture document.

Figure 6.4 illustrates the idea for component-and-connector (C&C) views. Moving to the right, styles become progressively more specific and constrained. At the left end of the spectrum are the most generic, and hence least constrained, styles. There a C&C style uses only generic components and connectors, allowing complete freedom of expression, but carrying none of the benefits mentioned above. Here the vocabulary consists of the generic categories of C&C style (call-return, data flow, event-based, and others) that impose constraints over component-and-connector types and support a specific computational model. Moving farther to the right are specializations of those styles, such as the examples described in

Generic Styles	**Generic Style Specializations**	**Generic Component Integration Standards**	**Domain-Spec Component Integration Standards**	**Organization-Specific Style Specializations**
Data Flow	Pipe & Filters	UNIX pipes	Yahoo! Pipes	…
Call-Return	Multi-tier	Java EE	Spring framework	
Implicit invocation	SOA	.NET	Ruby on Rails	
…	…	ESB	…	
		…		

Degree of Specialization →

Figure 6.4
Style specialization

Chapter 4 (client-server, pipe-and-filter, publish-subscribe, tiered, service-oriented, and more).

Farther right are styles that make stronger commitments to a particular domain, and typically provide an increasing basis for code reusability. For example, a Java EE-based style specializes tiered systems, introducing component types such as servlet, Enterprise JavaBean, and container components, while providing considerable implementation support for distribution, remote method invocation, transaction support, and persistence. One step to the right we find further specialization of the styles. For example, the Spring framework defines a specific way to implement Java EE applications, adopting patterns such as inversion of control and model-view-controller, and introducing element types such as Controller, View, and View-Resolver. Farther to the right, we might see architecture styles for product lines, which are targeted to the needs of systems within a particular company.

The choice of a domain-specific style often relates to the maturity of a family of architectures within a company, business segment, or engineering domain. For example, in the early days of client-server-based information systems, there was very little architecture guidance and reusability, beyond the need for clients, servers, and some form of remote invocation. Developers of such systems had to rely on relatively primitive forms of support for distributed communication, such as sockets and remote procedure call. As the field matured, so did frameworks such as .NET and Java EE, enabling far greater use of infrastructure, exploitation of common services, and guidance for construction of systems using these frameworks.

6.2 Descriptive Completeness

Descriptive completeness is a property of architecture documentation. Documentation has descriptive completeness if it documents all elements and relations in the system that are in the documentation's scope.

Related to refinement is the concept of **descriptive completeness**. Figure 6.5 shows an architecture diagram for an imaginary system. Element A is related to element B in some way—the diagram does not disclose how—B is related to C, and C is related to B. If you're a "consumer" of this diagram, what can you conclude about whether A and C are related?

You might say A and C are not related, because the diagram shows no arrow between A and C. Or you might say that this diagram reveals no relationship between A and C, but it is possible that this information was considered too detailed or tangential for the diagram. Subsequent documentation may reveal that A and C share this relation.

Either answer might be correct, as each represents a different strategy for documentation. The first strategy says that the

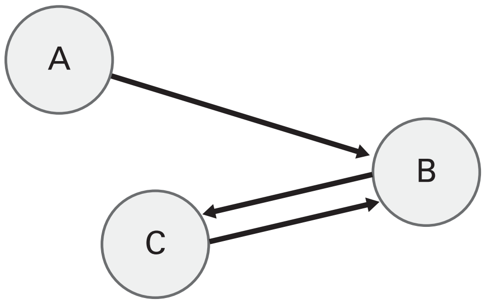


Figure 6.5
Element A is related to B, B is related to C, and C is related to B. What is the relation between A and C?

views are written with descriptive completeness; the second says they are not.

The same question can be asked about elements. In Figure 6.5, can we then presume that A, B, and C are the only elements involved? If the figure reflects descriptive completeness, then yes. Otherwise, no; perhaps in an elaboration or an augmentation of this view, another element will be shown, as in Figure 6.6.

Why would an architect omit some elements and relations in a view? There are some good reasons:

- It's early in the design. We don't know yet all the elements and relations that are part of the solution. Or we don't have time to complete the diagram right now, so we focus on the most important elements and relations.
- We want to show the most important parts of the view (and may produce an accompanying refinement showing more

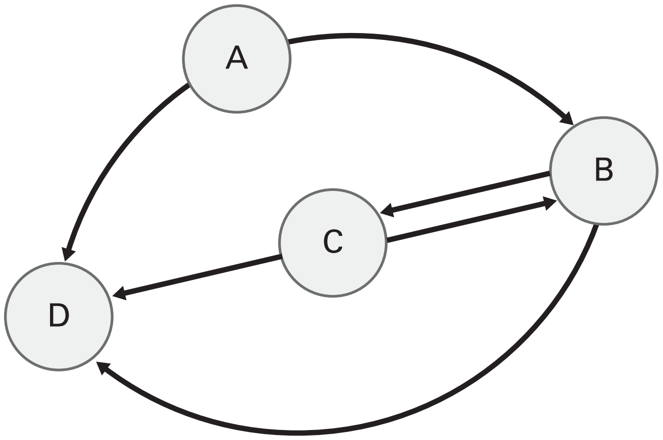


Figure 6.6
An elaboration to Figure 6.5 showing an additional element, D

of the design separately). Perhaps it's for an overview. Perhaps an element or relation is used only in special situations (such as error recovery) and we don't want to clutter the diagram to cover these special cases. Or maybe an element or relation is simply deemed less important and is left out.

- We want to reduce clutter in our diagrams. Maybe the same relation exists between most or all elements in the diagram, so we explain that in text (perhaps in a comment box) rather than graphically to avoid cluttering the diagram.

In Section P.5, we admonished you to explain your notation. The issue of descriptive completeness is a special case of that. You simply need to specify which of the two strategies your documents follow.

ADVICE

If you create a diagram that is not complete, here a few things you can do to inform the reader:

- Use ellipses ("...") to indicate in the diagram that there are other elements or relations not shown. In the key, explain the meaning of the ellipses. Figure 6.7 is an example.

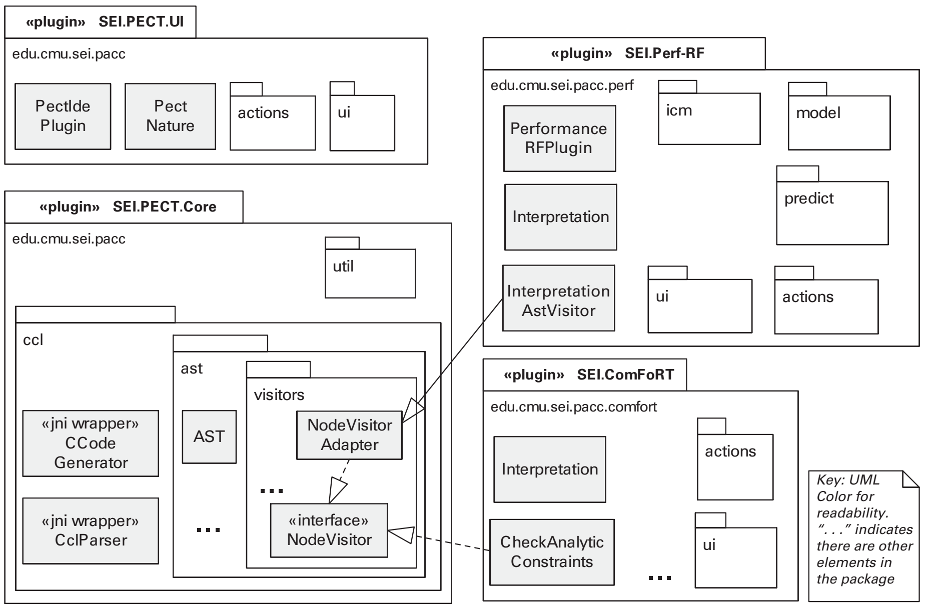


Figure 6.7
Module decomposition diagram that is not complete, as indicated by the ellipses ("..."). For packages whose submodules are shown and there is no "..." , the reader can assume all submodules are displayed.

- Use a comment box in the diagram to explain to the reader that not all elements or relations are being exhibited. Figure 6.8 is an example.
- Put a note in the key that says the diagram may not be complete and that other elements or relations may exist in subsequent refinements.

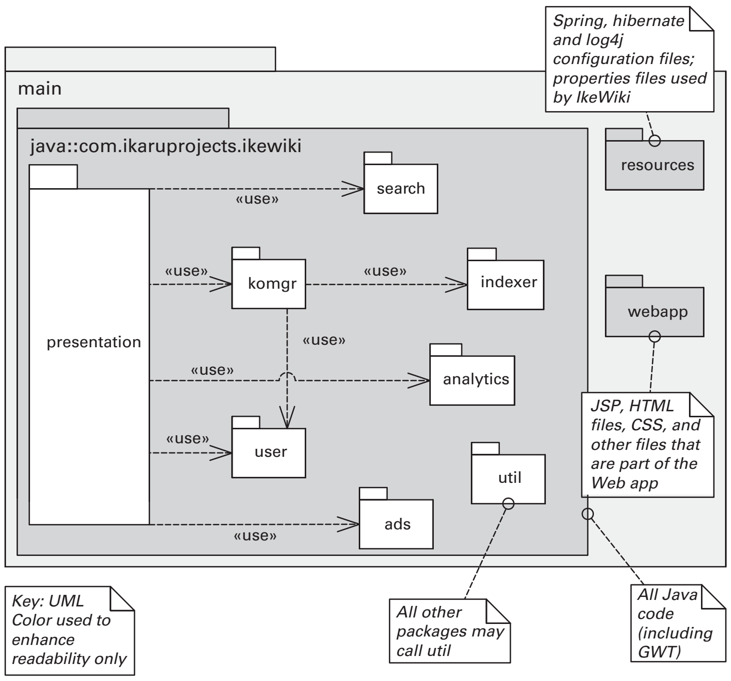

Spring, hibernate and log4j configuration files; properties files used by IkeWiki
main
java::com.ikaruprojects.ikewiki
resources
search
«use»
komgr
indexer
webapp
presentation
analytics
user
util
ads
JSP, HTML files, CSS, and other files that are part of the Web app
Key: UML Color used to enhance readability only
All other packages may call util
All Java code (including GWT)

Figure 6.8
Module uses diagram that does not show all usage dependencies, as indicated by the comment box attached to package util. To avoid cluttering the diagram, the author decided to use that comment box instead of drawing <<use>> dependencies from all other packages to util.

6.3 Documenting Context Diagrams

The purpose of a **context diagram** is to depict the scope of a view. Many, if not most, context diagrams in practice are **top-level context diagrams** (TLCDs), but context diagrams are also useful when an architecture document is explaining a subset of the system, such as a subsystem or even a single architecture element. Those smaller pieces have context as well, and understanding the context helps understand the subsystem or element. Here, "context" means an environment with which the part of the system interacts.

A **context diagram** defines the boundary between a system (or part of a system under consideration) and its environment, showing the entities in its environment with which it interacts.

A **top-level context diagram** is a context diagram in which the scope is the entire system.

Entities in the environment may be humans, other computer systems, or physical objects, such as sensors or controlled

A top-level context diagram makes a good first introduction to a system and its architecture description. It can serve as the jumping-off point for delving into deeper architecture detail in any number of directions.

devices. In the case of a context diagram for a subset of the whole system—that is, when the context diagram is not a TLCD—the entities in the environment may well be other entities that belong to the same system as the subset.

A context diagram is useful because it clarifies what are the parts of the whole solution you have to develop. Sometimes an organization is asked to develop a system that is part of a larger system, and a context diagram (in this case, a TLCD) depicts that. Sometimes supporting frameworks and libraries, external Web services, off-the-shelf software, other systems of the same organization, or some other tangential software is considered outside the scope of the system being developed. A context diagram clarifies what is in and what is out.

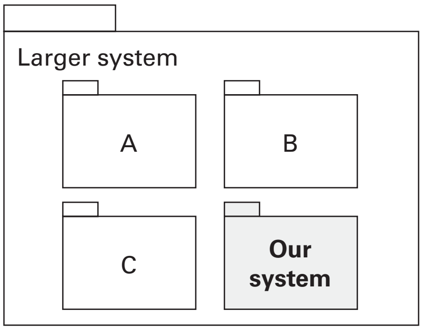
6.3.1 Create Context Diagrams Using the Vocabulary of the View

Describe the context of the system being developed using the vocabulary of the view that you're documenting.

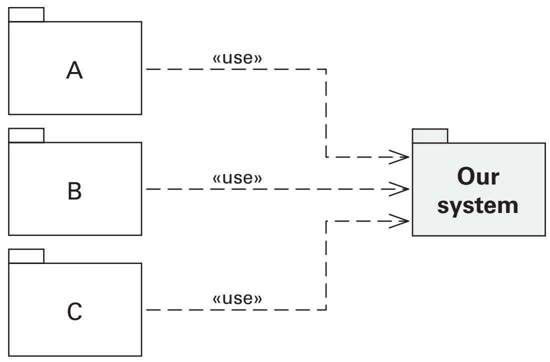
Remember that your architecture document will consist of a number of different views, and each view will include a context diagram. Will each of these context diagrams be the same? No! That would be unnecessary repetition.

Instead, let the vocabulary of the view—that is, its element types and relation types—determine what its context diagram should show. For example:

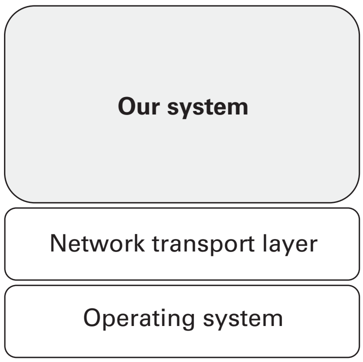
- The vocabulary of a decomposition view is "module" and "is part of." Sometimes an organization is asked to develop a system that is part of a larger system, and a context diagram depicts that. If so, then this relationship between what is being developed and the larger system is shown in the context diagram for the decomposition view. The system being developed can be shown as nested inside the larger system.



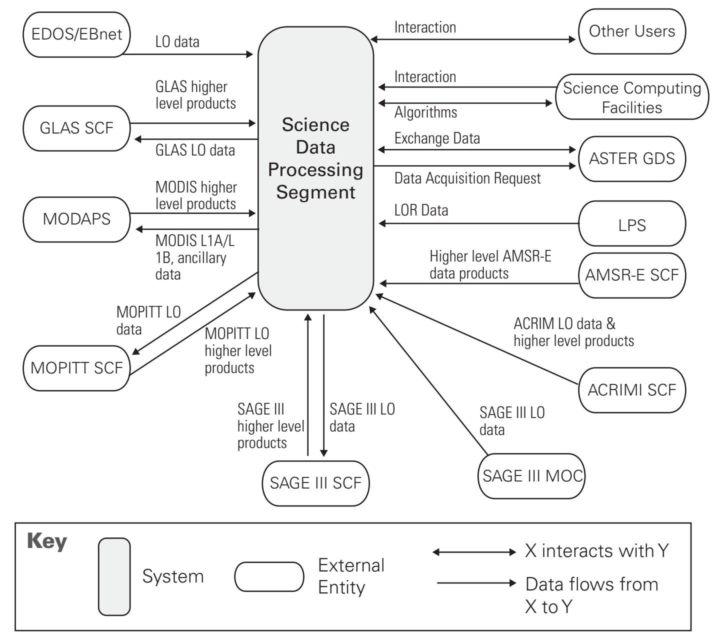
- The vocabulary of a uses view is “module” and “uses.” The context diagram for a uses view shows what external entities use or are used by the system under development.



- The vocabulary of a layered view is “layer” and “is allowed to use.” Sometimes the system being developed sits atop a layer provided externally, or sometimes the system being developed is the infrastructure or computing layer that can be used by application software developed elsewhere. In that case, the context diagram for a layered view would show the system under development as a layer above or below somebody else’s layers.



- The vocabulary of any kind of C&C view is, generally speaking, components and connectors and runtime interaction. The context diagram of a C&C view will show runtime interaction between the system being developed and external entities, specialized as appropriate. The “traditional” context diagram is, in fact, a context diagram for a C&C view.



- The vocabulary of a deployment view is the "is allocated to" relation between software and runtime hardware. Thus the context diagram for a deployment view will show any software external to the system being developed that is also allocated to the same hardware.

If the context diagram for a particular view doesn't apply, mark it as "Not applicable."

If you are documenting a view and the context diagram for it does not apply—for instance, if you're documenting a layered view and there are no external layers above or below the system being developed—then simply mark the context diagram for that view as "Not applicable."

6.3.2 Content of a Context Diagram

Use some sort of distinguished symbol, such as a thick outline or a hashed interior, to clearly denote the system whose context is being shown.

Context diagrams show the following:

- A depiction of the system—or part of the system—whose architecture is being documented.
- External entities.
- Relations with external entities that the system has. The external entities are shown outside the distinguished symbol for the system being described; the relations are expressed in the vocabulary of the category of the containing view.
- A key that explains the notation used in the context diagram, as is the case for all graphical figures.

A pure context diagram does not disclose any architecture detail about the system—it just appears as an undecomposed block—although in practice, context diagrams may show some internal structure of the system being put in context. Context diagrams do not show any temporal information, such as order of interactions or data flow. They do not show the conditions under which data is transferred, stimuli fired, messages transmitted, and so on.

6.3.3 Context Diagrams and Other Supporting Documentation

Context diagrams impart some obligations on the other supporting documentation in a view.

- The view's element catalog should include a description of the external elements shown in the context diagram. You should give a reference to the documentation in which the external entities' interfaces are documented.
- The view's rationale section should explain the reasons for drawing the boundary where it is.
- If the system has an interface with its environment shown in the context diagram, that interface needs to be "assigned" to one of the system's architecture elements. So every interface between the system and its environment that appears in a context diagram should also appear on one of the elements shown in the primary presentation.

Element catalogs are described in Section 10.1.

6.3.4 Notations for Context Diagrams

Informal Notations

Informally, context diagrams consist of a circle-and-line or box-and-line drawing, with the entity being defined depicted in the center as a distinguished circle or box, the entities external to it depicted as various shapes, and lines depicting relations connecting the entities as appropriate.

Structured analysis, the software design discipline that brought context diagrams into the mainstream, uses an informal notation to depict what we would call a C&C-type context diagram. The system is represented by a distinguished symbol in the middle, external entities are boxes, and the lines connecting them indicate data flow and runtime interaction.

Because context diagrams are often used to explain systems to people who know more about the externals of the application than the internals, such diagrams can be quite elaborate and use all sorts of idiomatic symbols for entities in the environment.

Figure 6.9 shows a context diagram created using an informal box-and-line diagram. Because the relation shown in the

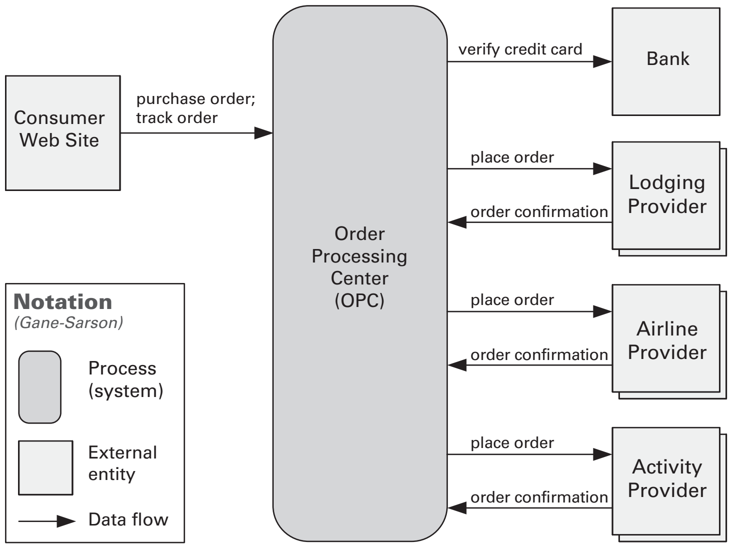


Figure 6.9
Context diagram for a C&C view using an informal notation. The example is taken from the Adventure Builder system (Adventure Builder 2010).

diagram is data flow (a runtime relation), we can tell that this is the context diagram for a C&C view of some kind.

Context diagrams can be depicted easily using tables. This is useful when there are too many interactions conveniently to show graphically. For example, a table depicting the data flow context diagram in Figure 6.9 would give the following:

- The identifier for each piece of data transferred across the environment boundary (such as a message identifier)
- A description
- The element that sends it
- The element that receives it
- Some information about it, such as what you would find in a data dictionary

Some software development standards prescribe a document with a name such as "Interface Requirements Specification," whose contents consist chiefly of long tables describing messages sent to and from the system. These documents are effectively context diagrams.

UML

UML does not have an explicit mechanism for a context diagram. However, diagrams that are appropriate for the various views are also good for showing the context of a given view.

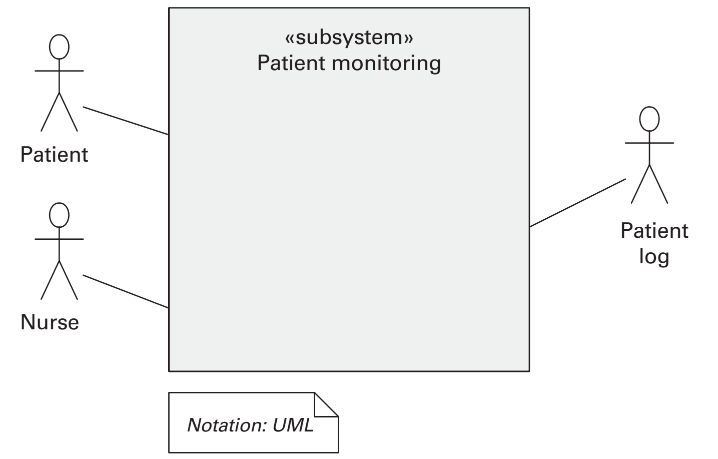


Figure 6.10
Description of a system context, using a UML class diagram. The class stereotyped as <<subsystem>> depicts the system whose context is shown; Patient, Nurse, and Patient log are external entities.

Recalling the principle that the context diagram for a view should describe the context using the element-type/relation-type vocabulary of the view that you're documenting, the same UML notation you use in a view's primary presentation can be used in that view's context diagram.

For instance, you can use component diagrams to show a C&C view's context diagram. Or you can show the context diagram of a decomposition view with nested packages. Or you can show the context diagram of a layered view using packages and <<allowed to use>> dependency arrows. And so forth.

A more general, though less informative, way to show context in UML is with a combination of use case and class diagrams as shown in Figure 6.10. Here the system's distinguished symbol is an appropriately stereotyped class and environment elements are shown as actors.

6.4 Documenting Variation Points

6.4.1 What Are Variation Points?

Variation points are places in the architecture where specific instances of flexibility have been built in. The flexibility is achieved by intentionally leaving specific architectural decisions open, but in a way so that they can be easily bound later, almost always by someone other than the architect. Architects design variation points into an architecture to achieve **variability**, which is the ability quickly to achieve change in preplanned ways.

A **variation point** is a place in the architecture where a specific kind of flexibility has been built in.

Variability is the ability to quickly achieve change in preplanned ways.

Providing variation points in an architecture is desirable in the following situations:

- Some set of decisions has not yet been made during the design process for a single system, but options have been explored.
- The architecture for a single system is prepared for envisioned future changes.
- The architecture provides basic functionality that can be extended easily.
- The architecture is for a family or product line of systems, and the option taken will depend on the specifics of the particular member of the family to be constructed.
- The architecture is a reference architecture for a collection of systems and contains explicit places where configurations and extensions to the reference architecture can occur.

See "Coming to Terms: Product-Line Architectures" on page 234 in this chapter.

Variation points can occur at any place in an architecture. They can affect elements and relations, the properties of those elements and relations, as well as their behavior. They can even affect the relations between views. For example, a simple element may run on the same processor with other elements, but a more complicated variant might need to run on its own dedicated processor.

Document variation points where they occur: in diagrams, element catalogs, behavioral descriptions, interface descriptions, and so forth. But fully describing the effects and ramifications of each variation point, as well as how to exercise the choice offered by a variation point, is best done in one place, called a **variability guide**.

A **variability guide** is the place in an architecture document that explains what variation points have been designed into the architecture and gives advice about how to exercise them.

Documenting variation points where they occur throughout the architecture documentation has the advantage that the description is available where it is needed. But it also has the disadvantage that pretty soon no one has the complete overview of which variation points exist in the system. Just as an element catalog serves as a complete repository of elements in a view, the variability guide will list and explain all of the variation points in a view.

A **variation mechanism** is a built-in software mechanism for making a change that, when exercised, results in a new instance of the architecture. The place where a variation mechanism occurs marks a variation point.

6.4.2 Variation Mechanisms

Architects design a variation point by selecting a **variation mechanism** that can be exercised to achieve one of the options provided. Some of the more prominent architecture variation mechanisms include the following:

- *Element substitution.* Replacing the implementation of a module or component with a different implementation that still honors (or "realizes") the same interface. This might provide one version of a system with a feature that behaves one way, whereas the second version's feature would behave in a different way.
- *Component replication.* Creating multiple instances of a component to provide greater capability in some fashion. For instance, Web-based systems may allow the deployment of Web components to multiple machines and the configuration of the number of instances on each machine. Such configuration is tuned to achieve the desired throughput and availability.
- *Optional inclusion.* In some versions of a system, a component might be present, whereas in another it might be omitted. This allows a system to have, or not have, a particular feature. Optional components are many times called plug-ins or add-ons.
- *Frameworks.* A **framework** is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.
- *Parameterization.* To allow variation in a wide range of constructs. Common examples include values of file names, URLs, user credentials, and lower-limit or upper-limit values.
- *Element composition.* Assembling new elements by putting together existing elements. (A tool that does this is sometimes called a configurator.)
- *Templates.* Providing a generic body that is almost, but not quite, complete. Downstream designers fill in the open parts as needed. Templates are often for code, but they can also be architectural: for instance, an architecture diagram that has "empty" parts that need to be filled in.
- *Inheritance.* Defining generic classes and interfaces. Different variations can be implemented (possibly by different vendors) by creating specific subclasses or classes that realize the interfaces.
- *Generator.* A generator is a software program that takes as input some specification of a desired program and produces as output a program that meets that specification.

A **framework** is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.

COMING TO TERMS

Product-Line Architectures

A product-line architecture is the poster child for architectures with built-in variation points. A software product line "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop 2001). Each product in the product line may have a slightly different architecture; these architectures are instances of the product-line architecture. The product-line architecture has decisions that have been intentionally left open; the architecture for a product (sometimes called a product architecture or an "instance" architecture) comes about when a product builder exercises the variation mechanisms that the product-line architect has put in place exactly for the purpose of building any one of a number of specific products.

For example, in a product line of personal income tax software, some products go to the Web and download the latest calculation software to reflect changes in the tax code; others might not. Some products might offer secure login and encryption to allow higher data confidentiality; others might not. And so forth. Product-line designers deal with extensive feature lists, and an individual product is usually defined by the features it does and does not support. Together, the family of products covers all of the targeted market segments. Individual products are differentiated by feature and price. A developer building one of these products for, say, testing or shipping, will exercise the variation mechanisms in such a way as to derive the desired product. For example, if the architect has chosen optional inclusion as the variation mechanism, the product builder will check out the calculation-downloading component and the encryption component and include them in the build, if the product includes the corresponding features.

To design a product-line architecture, an architect relies heavily on the product line's scope, which is a statement of what all of the products in the product line will have in common and the specific ways that they will vary from each other. Choosing variation mechanisms involves a trade-off between the cost of building in the variation mechanisms and the cost of exercising them. For example, a generator that takes as input a description of the product you want and—*poof!*—produces that product is usually very expensive to build but very cheap to use. There are situations where the economics favor that approach, and others where they do not.

6.4.3 Dynamism and Dynamic Architectures

A **dynamic architecture** is one in which architecture variation points are exercised at runtime.

When the binding time of a variation point is runtime, we say that this is a **dynamic architecture**. Architectures change during runtime in response to user requirements or to better enable the achievement of particular quality attributes. A Web browser that can go to a Web site, download a plug-in, and then start using it to handle a new media type has a dynamic architecture; its runtime architecture comprises more components after the download than before. An architecture can change dynamically by creating (including) or deleting (dropping) components and connectors, including replicas. For example, when a new user enters an environment and wants new services, components to provide those services would be created. When the user leaves the environment, the components would be deleted. The created component or connector may be a replica or a singleton. In any case, the architect should document the number of allowable replicas, the conditions under which the creation or deletion occurs, and the connectors or components that are created.

Another way an architecture can change dynamically is by reallocation of resources or responsibilities. Components may be moved from one processor to another to offer better performance. Responsibilities may be shifted among components: perhaps a backup could assume primary status in the event of a failure.

Happily, documenting a dynamic architecture is no different than documenting other kinds of variation points; the binding time is always runtime.

6.4.4 Documenting Variation Points

Variation points should be documented in two ways. First, their existence should be noted in the appropriate places throughout the view (primary presentation, element catalog, context diagram, and so on) for the view in which they are visible. Second, the variation point should be explained in the view's variability guide.

Showing variation points graphically can lead to diagrams that are cluttered and hard to read, especially if you try to show dependencies among variations graphically. Instead, you can annotate your diagram with a pointer to an entry in the variability guide (described in Section 10.1).

To show a variation point in a diagram, you can attach an annotation to the area affected by a variation point. With a suitable identifier (for example, "VP12"), the annotation can point to the location in the variability guide where the variation point is explained in full.

Other graphical approaches for showing the existence of variation tend to depend on the variation mechanism that the architect has chosen. For example:

- *Element substitution.* The UML relation "realizes" is a good way to depict this by showing that an interface can be realized by any number of implementations. Graphically, this is shown in Figure 6.11.
- *Component replication.* In an informal graphical notation, component replication is almost always documented show-

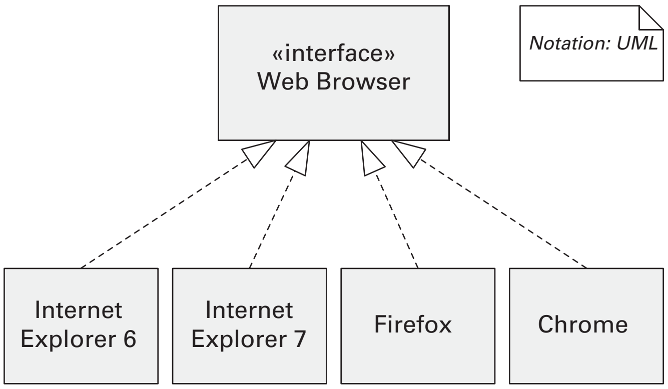


Figure 6.11
Element substitution as expressed by the "realizes" relation

ing shadow boxes: Almost always lacking are an indication of the possible range of replication and when the actual number is bound. Figure 6.12 includes this information in the annotation; it could equally well have referred to the variability guide.

- *Optional inclusion.* To show optional inclusion, you can employ the notations for component replication; simply confine the range of instances to 0 or 1.
- *Creation and deletion of elements.* Chapter 8 describes notations that can be used to indicate how elements can be created and deleted when the system is executed. An example is a UML sequence diagram, in which a time line underneath an object indicates the existence of that object.
- *Reallocating resources.* Some forms of reallocation of resources, such as the migration of objects, can be described by a UML stereotyped dependency <<*becomes*>>. The dependency tail is on the original location of an object and the head is on the subsequent location.

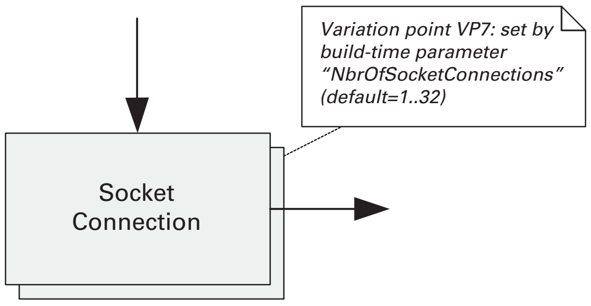

Variation point VP7: set by build-time parameter "NbrOfSocketConnections" (default=1..32)
Socket Connection

Figure 6.12
Component replication using shadow boxes

- *Frameworks.* Extension points need to be documented. An extension point is a place in the framework where additional elements can be added or abstract elements can be replaced with concrete ones. Each extension point is documented by an interface description of what the framework provides and the extension requires.

The variability guide for a view should contain the following information for each variation point that is present in the view:

- *Description of the variation point.* What decision has been left open by this variation point? The description should be architectural (for example, a particular component can be swapped in and out) but also meaningful to the stakeholders (for example, choosing different implementations results in different feature behavior).
- *Available options and their effects.* What is the range of choices available to exercise this variation point? What is the stakeholder-visible effect of each? What are the architectural effects of each option?
- *Condition of applicability.* Each variation point has a condition associated with it that describes a state that must be true for a variation point to apply. For example, to create an entertainment system for a car, the decision of which type of DVD player to use depends on the decision that the system actually has a DVD player.
- *The binding time of an option.* Possible binding times include design time, compile time, link time, or runtime. If runtime, more choices are possible: system start-up or restart time, when the component containing the variation point starts, or at other distinguished times during execution.
- *How the option is exercised.* This describes what someone has to do in order to choose an option of the variation point: set a build-time parameter, for instance, or replace one implementation of a module with another. This section is the step-by-step "how-to" guide for making the choice presented by the variation point.
- *Dependencies among variation point options.* Sometimes when an option is chosen for one variation point, it constrains other choices. For example, suppose your supply-chain management system stores images of the items that are in your inventory, and image format (such as JPEG or PNG) is a variation point. Suppose customers can access your inventory on a handheld device such as a pocket PC or cell phone. The list of devices that your system supports constrains the image formats that can be used and vice versa.

A variability guide can be conveniently presented as a table. Figure 6.13 shows an example.

Variation Point	Affected Element or Relation	Variants	Condition	Binding Time
VP1: Host name of the SMTP server used by the system to send e-mail messages	emailer	Any valid host name.	Whenever the SMTP host changes. Also used to switch between development and test environments.	Load time
VP2: External storage and content delivery services in use	filemanager	List of Web services; shall differentiate whether the service is available in the development, test, and/or production environments.	At least one storage service must be configured. A submodule that handles communication with a given new service must be available prior to enabling a service.	Load time
VP3: Access keys to storage and content delivery Web services	filemanager	For each storage and content delivery service enabled (such as Scribd, S3, YouTube), there will be a set of access parameters (such as URL, user ID, password).	Only used if the corresponding service is enabled in VP2.	Runtime
VP4: Performance-monitoring switch	util; logging; aspects	True if the response time of requests needs to be monitored and recorded; false otherwise. Must be configurable separately for development, test, and production environments.	Must be set to true when the SLA is in effect or for debugging purposes. If false, the corresponding submodules and artifacts may be excluded from the build.	Build time

Figure 6.13
Excerpt from a variability guide showing variation points of a Web application. This variability guide is part of a module uses view (not shown), where the description of the affected elements is found.

6.5 Documenting Architectural Decisions

With Jeff Tyree and Art Akerman

6.5.1 Why Document Architectural Decisions?

The process of developing a complex software architecture involves making hundreds of big and small decisions. The *results* of these decisions are reflected in the views that document the architecture—the structures with their elements and relations and properties, and the interfaces and behavior of those elements—but most of the time the decisions themselves are sadly neglected. And in that case, the **rationale**, especially the rationale behind the most important decisions, is irrevocably lost.

Rationale is an explanation of the reasoning that lies behind an architectural decision.

Most decisions are made in a complex environment and almost always involve trade-offs, and the environment and the trade-offs are likely to be completely invisible to someone who "inherits" the architecture. Generally, there were circumstances, constrained by cost and schedule, under which these decisions made sense. However, looking back, after all the dust has settled and the original system designers are long gone, we have no context around the critical decisions; we have no history; we have no guidance from the architect to take us forward. All we can do is just shake our heads (sometimes in disbelief) and ask "What was he thinking?" Rationale tells us exactly that: What he (or she) was thinking.

The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.

—Philippe Kruchten

In the Views and Beyond approach, documenting architectural decisions enjoys first-class status. When we introduce the templates for software architecture documentation in Chapter 10, you will see that they contain dedicated places to record architectural decisions.

Documenting architectural decisions as you go results in an architecture that is demonstrably aligned with the business and technical goals of the system. This is a theme we have tried to emphasize throughout the book. Documentation isn't something you do after the architecture is finished. *Documenting the architecture helps you design the architecture.* Documenting the decisions as you make them helps you make them correctly.

6.5.2 A Template for Documenting Architectural Decisions

Following a minimalist approach, only those issues that need addressing at various points in the life cycle should be addressed and thus documented. For example, decisions with many and far-reaching implications are prime candidates to be documented.

Like all templates in this book, use this one as a starting point. Add or subtract rows or sections so that it best fits your organization, your stakeholders, and their needs.

What follows is a template for capturing essential information about a key architectural decision.

1. **Issue**. State the architectural design issue being addressed. This should leave no questions about the reason why this issue is to be addressed now.
2. **Decision**. Clearly state the solution chosen. It is the selection of one of the positions that the architect could have taken.
3. **Status**. State the status of the decision, such as pending, decided, or approved. (This is not the status of implementing the decision.)
4. **Group**. Name a containing group. Grouping allows for filtering based on the technical stakeholder interests. A simple group label, such as "integration," "presentation," "data," and so on can be used to help organize the set of decisions. For example, the data architects reviewing the decisions can focus only on the decisions classified as data.
5. **Assumptions**. Clearly describe the underlying assumptions in the environment in which a decision is being made. These could be cost, schedule, technology, and so on. Note that constraints in the environment (such as a list of accepted technology standards, an enterprise architecture, or commonly employed patterns) may limit the set of alternatives considered.
6. **Alternatives**. List alternatives (that is, options or positions) considered. Explain alternatives with sufficient detail to judge their suitability; refer to external documentation to do so if necessary. Only viable positions should be described here. While you don't need an exhaustive list, you also don't want to hear the question "Did you think about . . . ?" during a final review, which might lead to a loss of credibility and a questioning of other architectural decisions. Listing alternatives espoused by others also helps them know that their opinions were heard. Finally, listing alternatives helps the architect make the right decision, because listing alternatives cannot be done unless those alternatives were given due consideration.
7. **Argument**. Outline why a position was selected. This is probably as important as the decision itself. The argument for a decision can include items such as implementation cost, total cost of ownership, time to market, and availability of required development resources.
8. **Implications**. Describe the decision's implications. For example, it may
   - Introduce a need to make other decisions

   - Create new requirements
   - Modify existing requirements
   - Pose additional constraints to the environment
   - Require renegotiation of scope
   - Require renegotiation of the schedule with the customers
   - Require additional training for the staff

   Clearly understanding and stating the implications of the decisions has been a very effective tool in gaining buy-in.

9. **Related Decisions**. List decisions related to this one. A traceability matrix or decision tree is useful, as is showing complex relations diagrammatically such as with object models. Useful relations among decisions include causality (which decisions caused other ones), structure (showing decisions' parents or children, corresponding to architecture elements at higher or lower levels), or temporality (which decisions came before or after others).
10. **Related Requirements**. Map decisions to objectives or requirements, to show accountability. Each architecture decision is assessed as to its contribution to each major objective. We can then assess how well the objective is met across all decisions, as part of an overall architecture evaluation.
11. **Affected Artifacts**. List the architecture elements and/or relations affected by this decision. You might also list the effects on other design or scope decisions, pointing to the documents where those decisions are described. You might also include external artifacts upstream and downstream of the architecture, as well as management artifacts such as budgets and schedules.
12. **Notes**. Capture notes and issues that are discussed during the decision process.

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

—Donald Knuth

ADVICE

Using the Template for Documenting Architectural Decisions

Assumptions (number 5 in the template). The architect should document key assumptions he or she made when crafting the design. Assumptions are usually about either environment or need. Assumptions about the environment document what the architect assumes is available in the environment and what can be used by the system being designed. Assumptions are also made about invariants in the environment. For example, a navigation system architect might make assumptions about the stability of the earth's geographic and/or magnetic poles. Finally, assumptions about the environment can pertain to the development environment: tool suites available or the skill levels of the implementation teams, for example. Assumptions about need state why the design provided is sufficient for what's needed. For example, if a navigation system's software interface provides location information in a single geographic frame of reference, the architect is assuming that it is sufficient and that alternative frames of reference are not useful.

Assumptions can play a crucial role in the validation of an architecture. The design that an architect produces is a function of these assumptions, and writing them down explicitly makes it vastly easier to review them for accuracy and soundness than trying to ferret them out by examining the design.

Alternatives (number 6). Unless mandated, do not explicitly register the name of the person who suggested each alternative. Finding the best alternative for a design problem should be seen as a team effort. The solution has collective ownership; it's not important that Carlos's solution won over Julia's.

Argument (number 7). Analysis or formal review results often make excellent fodder for rationale, in that they illuminate goals and requirements driving the architecture and provide the connection between those constraints and the architectural decisions that satisfy them. If trade studies were performed to support a decision, or analysis performed to validate a decision, these can be conveniently referenced here.

Alternatives are often conveniently documented using a table listing relevant objectives or decision criteria and showing how well each alternative addresses them.

6.5.3 Documenting Alternatives

Often, early and major architectural decisions involve selecting from among a set of available alternatives. A table shows and quickly contrasts the pros and cons of each alternative. Table 6.1 shows an example of a table comparing three strategic options available to a financial organization trying to meet the listed business objectives.

It is hard to claim that you know what you are doing unless you can present your act as a deliberate choice out of a possible set of things you could have done as well.

—E. W. Dijkstra (1972, pp. 39–41)

6.5.4 Which Decisions to Document

Which of the hundreds or thousands of design decisions comprising an architecture should be documented? Certainly not all of them. It's simply too time-consuming, and many decisions do not warrant the effort. So how do you select which decisions are important enough to warrant documentation?

The goal is to receive a positive "return on investment" for the effort you expend recording the decision. That is, you should document an architectural decision if, in your judgment, you think it's cheaper to capture it now than not capturing it will be later.

Table 6.1 Analysis of alternatives for implementation of interactive approval processing

ID	Concerns	Alternative 1: Re-architect System A	Alternative 2: Extend System B	Alternative 3: Replace System A
N1	Provide interactive approval of credit applications	Yes	Yes	Yes
N2	Deliver in 6 months	Yes	Yes	No
N3	Reduce time to market for future enhancements	No	Yes	Yes
N4	Reduce costs	No	Yes	No
N5	Reduce risks	No	Yes	No
N6	Will not disrupt business operations	Unknown	Unknown	No
N7	System qualities	No	Yes	Unknown
N8	Reuse existing infrastructure, buy before build	Yes	Yes	No
N9	Use proven technologies	Yes	Yes	No

Here are some guidelines to help you identify the architectural decisions worth capturing. Document an architectural decision if:

- It has an important effect on the system. For instance, it strongly affects the system's business goals, or one or more system quality attributes (performance, availability, modifiability, security, and the like). Or the decision has some other widespread effect that will be difficult to undo. Or the decision implies spending (or saving) a significant amount of time (such as buying an expensive product).
- The design team spent significant time and effort evaluating options before making a decision. For example, the decision comes after performing technical experiments or implementing prototypes or trade-off studies. Or you performed a focused group analysis or conducted a survey with a user base or established some sort of user forum.
- The decision is complex or confusing. For instance, the decision seems not to make sense at first but becomes clear when more background is considered. Or on several occasions, you've been asked, "Why did you do that?" Or the issue is confusing to new team members.
- Decisions that were unusual or unexpected should be documented because these are very likely to be broken by mistake by people who would not have considered such a resolution.

Often an architecture decision creates more issues. We document these issues as implications, which automatically become concerns for the new architecture decisions.

PERSPECTIVES

"It may sound like a lot of effort to do this, but here's how we do it in the trenches."

We've worked with dozens of architects who have written thousands of decisions and conducted countless review sessions defending their technical recommendations and rationales. Did these efforts produce enough tangible results? Did they justify the significant investments of time and resources that were made? Would these architects have been better off spending more time developing reference architectures, patterns, or standards? It is hard to tell. One thing is certain, however: By demonstrating relentless focus on aligning architectures with business problems, by bringing partners and customers along on a journey to develop the "right" solutions, and by being clear about the implications of their choices, we've seen architects build much stronger relations with their business and IT stakeholders. It is difficult to accuse any of these architecture teams of living in an ivory tower. Their work is well integrated with the strategy, development, and operations of their respective companies. How do you put a price on that?

Sometimes we do hear complaints from architects about the extra work involved to document their decisions. In such cases we usually find that they've gone too far, by documenting decisions that had very little impact or that had no viable alternatives. We reiterate with them the intent of documenting decisions, which has very little to do with the number of decisions captured. At the end of our discussions, architects usually leave the room much happier.

Modern enterprises have a characteristically flat organization and unclear lines of authority. Such places gather highly intelligent people who live to challenge the status quo, to innovate and to excel. Very seldom are people are given direct orders anymore. It is even more seldom that they would follow such orders willingly. The only way to drive change in such an environment is to obtain buy-in. Architecture decisions and rationale are essential tools for achieving that goal. Of course, even the most rational arguments are useless if we don't consider human factors, such as personal and organizational agendas, relationships, trust, and so on. But having a strong rationale is a minimum requirement for successful conversations.

—A.A. and J.T.

6.5.5 The Payback for Documenting Architectural Decisions

Documenting architectural decisions can be seen as informing the cost/benefit formula for architecture documentation given in the prologue. That formula lets you decide whether the payback for producing sound architecture documentation outweighs the effort it takes to produce it. Documenting architectural decisions, like architecture documentation at large, helps stakeholders do their jobs more effectively and efficiently, avoid wasting time on known technical dead ends, and maintain and evolve the architecture in a manner consistent with its underlying design concepts and constraints. That savings is the payback for the effort it takes for an architect to say, "This is what I was thinking."

Section P.2.4 in the prologue shows a formula describing the payoff point for architecture documentation.

Documenting the architecture will also help ensure that the architecture is properly aligned with the prevailing business and technical goals, by compelling the architect to document that alignment as the architecture is being crafted. Here the savings shows up as prevention of rework, which might be necessary if the architecture were discovered to be the wrong one for the job, because the architect didn't understand what "the job" actually was.

Although maintainers and future architects are primary consumers of architecture rationale, they are not the only stakeholders. Developers can gain important insights from reading the architect's reasoning. Testers can design tests to validate the architect's precepts and assumptions. Customers can examine the documented architecture decisions to convince themselves that their business goals are being met by the design. These stakeholders, and others, can read the rationale to make sure their interests have been addressed.

Here are some of the paybacks you can expect:

- *Socializing decisions.* Once a final architectural decision has been reached, the team will need to socialize the result and convince the rest of the organization that it has chosen appropriately. The architecture decision template provides a common language for discussing decisions. Reviewers can easily see the status of the decision, the reasoning behind it, and the impacts. In practice, this is more powerful than reviewing, say, box-and-line diagrams. In practice, controversial decisions should be socialized early and often.
- *External memory for the architect.* The stakeholder with perhaps the most vested interest in capturing the motivation and background for design decisions is the architect. In the maelstrom of developmental activities, the architect needs some way to remember the conceptual path he or she has taken, as well as a way not to repeat dead-end design paths.

- *Conveying risk.* Without properly documenting the major decisions, understanding the implications of the architecture is difficult. If recorded using a structure such as the one given in Section 6.5.2, decisions describe more than just a solution. They also communicate the essential risks and issues. The team has information on where it should focus attention.
- *Heading off redundant discussion.* Without documented rationale, stakeholders may ask the same questions about a decision that have long been answered. People may still challenge some decisions, but they will do so from a more informed footing.
- *Supporting timely development.* Each decision can be communicated separately, with a caveat that it is subject to change due to the impacts of downstream work. As long as these relations and risks are understood, a team can start using the decisions. This provides the opportunity to let development proceed in the face of not-fully-worked-out decisions.
- *Support for communication.* By turning the rationale into a viewgraph presentation, management or business stakeholders can understand the major architectural decisions along with their implications.

PERSPECTIVES

From Documenting Architectures to Architecting As Decision Making

With Rik Farenhorst

What is being proposed in this section reflects a decision-based school of thought for how we go about laying down an architecture. Until now, the norm has been to create an architecture and then document it, usually as a set of views. Rationale, if captured at all, was an after-the-fact exercise in trying to describe the reasoning behind a *fait accompli*.

But architecture can be seen as the outcome of a sequence of decisions, each one rationally made in response to context and need. "Here is a major decision we must make now," they say. "Let's write about it as we make it." And they capture both, at the same time, with tooling.

Many architecture tools let you extend the underlying metamodel that the tools provide out of the box. Akerman and Tyree (2005) have a metamodel for describing architectural decisions that can be loaded into such tools. As shown in Figure 6.14, the metamodel defines a direct association between requirements or stakeholder concerns, the architecture decisions that satisfy them, and the architecture assets (systems, components, modules, interfaces, and so on) that those decisions make manifest. "Architecture asset" is typically part of an architecture tool's default metamodel, and so provides the anchor point between architecture decisions and architecture.

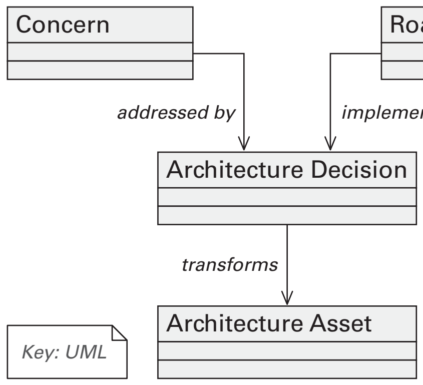


Figure 6.14
Akerman and Tyree's metamodel relates architecture decisions to architecture assets. (Adapted from A. Akermann and J. Tyree, "Position on Ontology-based Architecture," *Proceedings of the Fifth Working IEEE/IFIP Conference on Software Architecture* [November 2005]. ©2005 IEEE)

Their metamodel goes on to elaborate each node. The node "decision" is elaborated to define the information fields laid out in Section 6.5.2. A "concern" can be a required capability, a change case, a quality attribute, a risk, or a business need.

Now it becomes straightforward to include both decisions and views in a single model for the architecture. The chosen solution is captured right along with the rationale that produced it. Rationale behind a decision is no longer second-class; the why and the what are two sides of the same coin.

This approach is in accord with a growing community of researchers in a field called "architectural knowledge." They focus on managing architectural design decisions, their rationale, and related knowledge concepts (Araujo and Weiss 2002). The approach described here is an example of this shift put into everyday practice.

—R.F.

PERSPECTIVES

An Ontology of Architecture Decisions

Philippe Kruchten, well known for his work in creating and describing the Rational Unified Process, is one of the most experienced and thoughtful software architects in the world. Some extremely talented architects never share what they know. Others share all the time, but without having the experience to back it up. When I think of that rare group of architects who speak and write usefully and with insight from years of front-line experience, Philippe is at the top of the list. A current interest of his is the capturing and sharing of architecture knowledge, and toward this end he has created a classification scheme for architectural decisions (Kruchten 2004; Kruchten, Lago, and van Vliet 2006), summarized in Table 6.2.

Table 6.2 Kruchten's classification scheme for architectural decisions

Kind of Decision	Description	Examples
Existence decisions ("ontocrises")	An existence decision states that some element/artifact will positively show up; that is, it will exist in the system's design or implementation. Structural decisions lead to the creation of architecture elements of some kind. Behavioral decisions decide how the elements interact.	• "The logical view is organized in three layers: data layer, business logic layer, and user-interface layer." • "Communication between classes uses Remote Method Invocation (RMI)."
Ban or nonexistence decisions ("anticrises")	This is the opposite of an existence decision, stating that some element will not appear in the design or implementation.	• "The system does not use MySQL as its relational database system." • "The system does *not* reuse the flight management system from project ASIEW."
Property decisions ("diacrises")	A property decision states an enduring, overarching trait or quality of the system. Property decisions can be design rules or guidelines (when expressed positively) or design constraints (when expressed negatively), as some trait that the system will not exhibit.	• "All domain-related classes are defined in Layer #2." • "The implementation does not make use of open-source components whose license restricts closed redistribution."
Executive decisions ("pericrises")	These are the decisions that do not relate directly to the design elements or their qualities, but are driven more by the business environment (financial) and affect the development process (methodological), the people (education and training), the organization, and to a large extent the choices of technologies and tools.	• *Process decisions*: "All changes in subsystem exported interfaces (APIs) must be approved by the Change Control Board and the architecture team." • *Technology decisions*: "The system is developed using Java EE." • *Tool decisions*: "The system is developed using the System Architect Workbench."

Philippe also proposes an outline for describing an architectural decision. Here are some descriptive items not contained in the outline we presented in Section 6.5.2. Add them to your template if you find them useful.

- **Scope.** Some decisions may have limited scope, in time, in the organization, or in the design and implementation (see the *overrides* relationship, later in this sidebar). By default (if scope is not documented) the decision is universal. Scope might delimit the part of the system, a life-cycle time frame, or a part of the organization to which the decision applies.
- **Author, Time Stamp, History.** The person who made the decision, and when the decision was taken. Ideally we collect the history of changes to a design

decision. Important are the changes of state, or course, but also changes in formulation or in scope, especially when we run incremental architecture reviews.

- **Categories.** A design decision may belong to one or more categories. The list of categories is open ended; categories are useful for queries, and for creating and exploring sets of design decisions that are associated to a specific concern or quality attribute.
- **Cost.** Some design decisions have a cost associated with them, which is useful to reason about alternatives.
- **Risk.** Documented traditionally by exposure—a combination of impact and likelihood factors—this is the risk associated with taking that decision. It is often related to the uncertainty in the problem domain or to the novelty of the solution domain, or to unknowns in the process and organization. If the project is using a risk management tool, this should simply link to the appropriate risk in that tool.

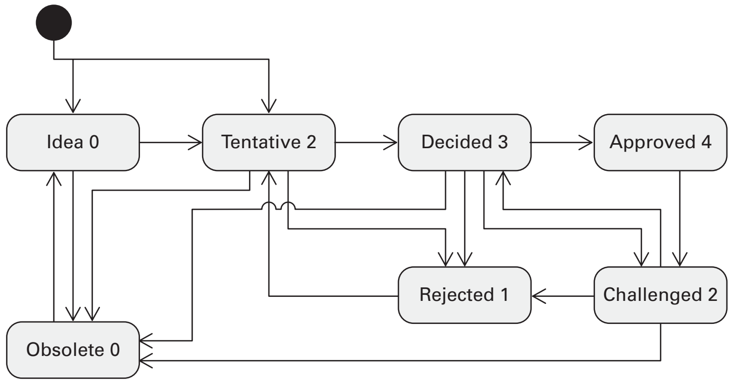


Figure 6.15
Kruchten's state machine for an architectural design decision (Kruchten 2009)

As shown in Figure 6.15, Philippe has a richly defined notion of a decision's state:

- **Idea.** Just an idea, captured so as not to be lost, when doing brainstorming, looking at other systems, and so on. It cannot constrain other decisions other than ideas.
- **Tentative.** Allows running "what-if" scenarios, when playing with ideas.
- **Decided.** Current position of the architect or architecture team; must be consistent with other, related decisions.
- **Approved.** By a review, or a board (not significantly different than decided in low-ceremony organizations).
- **Challenged.** Previously approved or decided decision that is now in jeopardy; it may go back to approved without ceremony, but it can also be demoted to tentative or rejected.

- **Rejected.** Decision that does not hold in the current system; but we keep such decisions around as part of the system rationale (see *subsumes* in the next list).
- **Obsolesced.** Similar to rejected, but the decision was not explicitly rejected (in favor of another one, for example) but simply became "moot"—for example, as a result of some higher level restructuring.

Finally, Philippe has worked out the ways in which decisions can be related to each other:

- **Constrains.** The decision "Must use Java EE" constrains the decision "Use JBoss."
- **Forbids.** Synonymous with *excludes*.
- **Enables.** The decision "Use Java" enables the decision "Use Java EE."
- **Subsumes.** "All subsystems are coded in Java" subsumes "Subsystem XYZ is coded in Java."
- **Conflicts With.** "Must use .NET" conflicts with "Must use Java EE."
- **Overrides.** "The Comm subsystem will be coded in C++" overrides "The whole system is developed in Java."
- **Comprises.** Synonymous with *is made of* and *decomposes into*. "Design will use UNAS as middleware" decomposes into "Rule: cannot use Ada tasking" and "Message passing must use UNAS messaging services" and "Error logging must use UNAS error logging services," and so on.
- **Is Bound To.** Decision A constrains decision B, and decision B constrains decision A.
- **Is an Alternative To.** Decisions A and B address the same issue but propose different choices.
- **Is Related To.** There is a relation of some sort between the two design decisions, but it is not of any kind listed previously and is kept mostly for purposes of documentation and illustration.

So, when you're filling in your template for an architectural decision and you come to the table row holding the decision's scope, or its current state, or its related decisions, you may want to refer to Philippe's categories in these areas.

—P.C.

6.6 Combining Views

The basic principle of documenting an architecture as a set of separate views brings a divide-and-conquer advantage to the task of documentation, but if the views were irrevocably different, with no association with one another, nobody would be able to understand the system as a whole.

Because all views in an architecture are part of that same architecture and exist to achieve a common purpose, many of them do have strong associations with each other. Managing how views are associated is an important part of the architect's job, and documenting that association is an important part of the documentation that applies beyond views.

6.6.1 Types of Associations Between Views

Views are associated with each other in a variety of ways.

In a *many-to-one* association (see Figure 6.16), multiple elements in one view are associated with a single element in another view. Implementation units are frequently associated with the runtime components they become. The association should make clear which module maps to which component.

In a *one-to-many* association (see Figure 6.17), a single element is associated from one view to multiple elements in another view. For example, a shopping cart module maps to multiple components in a tiered view of a Web store application.

Finally, a *many-to-many* association associates a set of elements in one view to a set of elements in another. This kind of association reflects the inherent complexity in relating two views to each other, each of which was crafted to show its own important aspects that in many ways might be orthogonal to those in the other view.

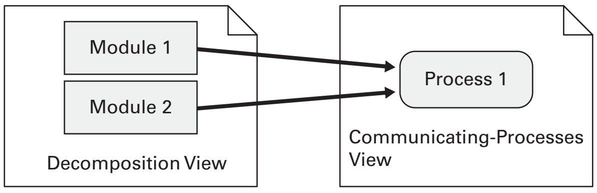


Figure 6.16
Many-to-one association. Multiple elements from one view are associated with a single element of another view. As shown here, two modules from a decomposition view are designed to run in a single process, shown in the communicating-processes view.

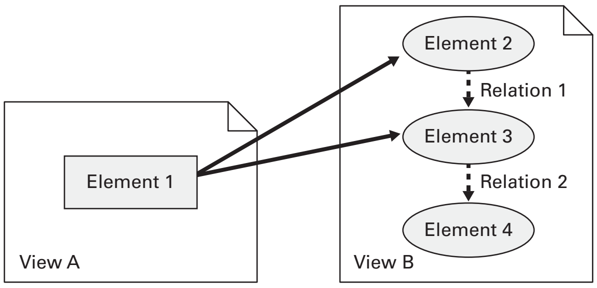


Figure 6.17
One-to-many mapping. An element of one view can be associated with multiple elements in another view.

6.6.2 Combined Views

A **combined view** is a view that contains elements and relations that come from two or more other views.

Sometimes the most convenient way to show a strong association between two views is to collapse them into a single **combined view**. A combined view nominally reduces the number of views in an architecture document because it replaces the views that it combines.

Figure 6.16 showed how multiple modules might map to a single process. Figure 6.18 shows how that mapping might be documented using a combined view.

In Figure 6.17 we showed how an element of one view mapped to more than one element of a second view. In Figure 6.19, we show how to represent this as a hybrid view. If Elements 2, 3, and 4, for example, are components of a C&C view and Element 1 is the functionality to store and to retrieve data within a component, designed as a class in a decomposition view, mapping Element 1 onto Elements 2 and 3 makes those elements "persistent components." Combined views can be very useful as long as you do not try to overload them with too many mappings.

There are two ways to produce a combined view.

An **overlay** is a view that combines the primary presentations of two or more views followed by supporting documentation for that combined primary presentation.

- Create an **overlay** that combines the information in what would otherwise have been in two separate views. This works well if the coupling between the two views is tight; that is, there are strong associations between elements in one view and elements in the other view. If that is the case, the structure described by the combined view will be easier to under-

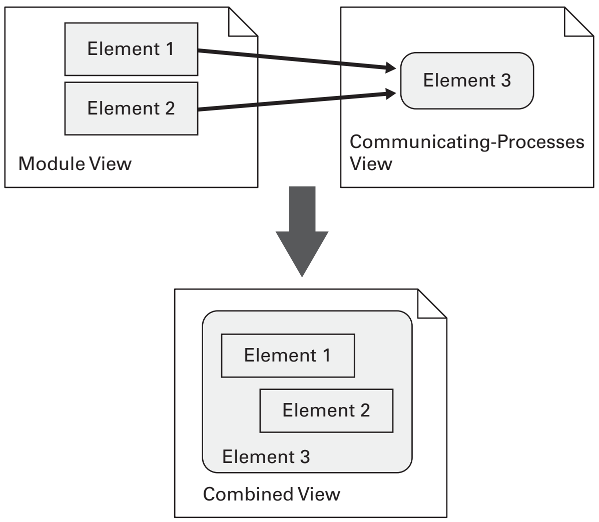


Figure 6.18
Multiple elements from one view can be mapped to a single element of another view. Here Elements 1 and 2 from a module view are designed to run in a single process—Element 3—shown in the communicating-processes view. The resulting combined view shows all three elements of the module and communicating-processes views, and their association as containment.

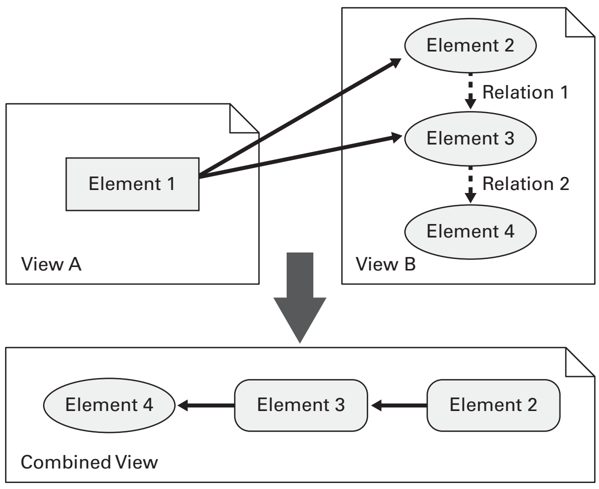


Figure 6.19
In this example, mapping Element 1 of View A onto Elements 2 and 3 of View B resulted in a new type of element, depicted as a new shape. This required the definition of a new element type in a style.

stand than the two views seen separately. For an example, see the overlay of decomposition and uses diagrams shown in Figure 6.20. In an overlay, the elements and the relations keep the types as defined in their constituent styles.

- Create a **hybrid style** by combining two existing styles and creating a style guide that indicates what styles were combined and describes any new or hybrid element and relation types, their properties, and constraints. Do this if the style is important and will be used in a variety of analyses and communication contexts in the system at hand or in other systems you expect to build. A view showing the hybrid style applied to a system is a combined view.

  In a hybrid style, element and relation types of the constituent styles can "meld" into new types with new properties. Therefore, hybrid styles require the definition of the resulting new element and relation types. For example, if a hybrid style combines layered style and a communicating-processes style, a new element type could be *layered process*, and this type would need to be defined in the hybrid style's style guide.

  Similarly, the relation types of a hybrid style are derived from the relation types of the constituent styles and their associations. Not all relation types of the constituent styles need to be preserved.

A **hybrid style** is the combination of two or more existing styles. Hybrid styles are documented using a style guide, as shown in "Style Guides: A Standard Organization for Explaining a Style," in Section I.1, in the introduction to Part I. Hybrid styles are like other styles in that, when applied to a particular system, they produce views.

If you create a hybrid style, document it using a style guide, following a template like the one that appears in Section I.1, in the introduction to Part I.

A hybrid style is useful to create if the style is used over and over again in the same system or in the kinds of systems developed in your organization, and if many stakeholders need to be familiar with it.

Sometimes, however, a combined view is created for a single, short-term purpose: for analysis or communication, for example. For these short-term purposes, creating the required documentation for a new style is burdensome overhead, and an overlay will serve nicely.

If two views can be associated with one another, you should show that association, whether using a hybrid style, an overlay, or a separate piece of the documentation showing the association.

So now an architect has three ways to establish the association between otherwise stand-alone views:

- Document a mapping between separate views. Do this as part of the documentation that applies beyond views.
- Create a hybrid style and then produce views of your architecture using that style.
- Create an overlay from two otherwise separate views.

In fact, there's a fourth way that sometimes works well. Augment the property list of one view with a property that lets you specify the important information from the second view. For example, in a decomposition view, you can add "Organizational unit" and "Development folder" as properties to document for each module. When you fill in those property values, you effectively have a combined module decomposition, work assignment, and implementation view. Or in a communicating-processes view, you can add a property named "Processor" and another named "Installation file." The result is a combined communicating-processes/deployment/install view.

Finally, you can think of allocation views such as those in Chapter 5 as a kind of combined view. One of the views they combine is not a view from software architecture, but rather a view from outside—runtime hardware, development environment, or organization.

6.6.3 When to Combine Views

The set of views used for a system is the result of a trade-off between the clarity of many views, each of which has a small number of concepts, and the reduced cost associated with having a small number of views, each dealing with multiple concepts.

If the association is clear and straightforward, and the combined view won't be overly complex, and a consumer group for the combined view has been identified, and that group is the same group consuming the constituent views, then it makes sense to adopt the combined view in place of the separate constituents.

When considering a combined view, make sure that the association among the constituents is clear and straightforward. Otherwise, these views are probably not good candidates to be combined, as the result will be a complex and confusing view. In this case, it would be better to manage the association separately, as in a table that relates the views while keeping them separate. A table has the space to make the complex associations among the constituents clear and complete.

Even if the associations are strong, too many different concepts clutter up combined views. Keys and the plethora of relations shown in the primary presentation all become difficult to understand. Before committing to a combined view, sketch it to see whether it passes the "elevator speech" test: Could you explain the idea behind it to someone in the time it takes to ride an elevator up a dozen or so floors?

Different groups of workers need different types of information. Make your choice of views responsive to the needs of your stakeholders. Before committing to a combined view, make sure that there is a stakeholder "market" for it.

Tool support influences the choice and number of views. The cost of maintaining multiple views is partially a function of the sophistication of the available tools. If your tools understand how a change in one view should be reflected in another view, it is not necessary to manage this change manually. The more sophisticated the tools, the more views can be supported.

These views often combine naturally:

- *Various C&C views.* Because C&C views all show runtime relations of various types among components and connectors of various types, they tend to combine well. Different (separate) C&C views tend to show different parts of the system, or tend to show decomposition refinements of components in other views. The result is often a set of views that can be combined easily.
- *Deployment view with either service-oriented or communicating-processes views.* A service-oriented view shows services, and a communicating-processes view shows processes. In both cases, these are components that are deployed onto processors. Thus there is a strong association between the elements in these views.
- *Deployment view and install view.* The combined view shows the installation files and what hardware elements they are deployed to.
- *Decomposition view and any of work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses; and they populate layers.
- *Generalization and aspects.* Both views deal with classes and objects and the relations among them—hence, these are two views with a strong association.

6.6.4 Examples of Combined Views

Decomposition, Uses, and Generalization

Figure 6.20 is the primary presentation for one view of the software architecture for the SEI's Architecture Expert (ArchE) tool. This tool allows an architect to create the architecture design for a system based on three types of input: quality

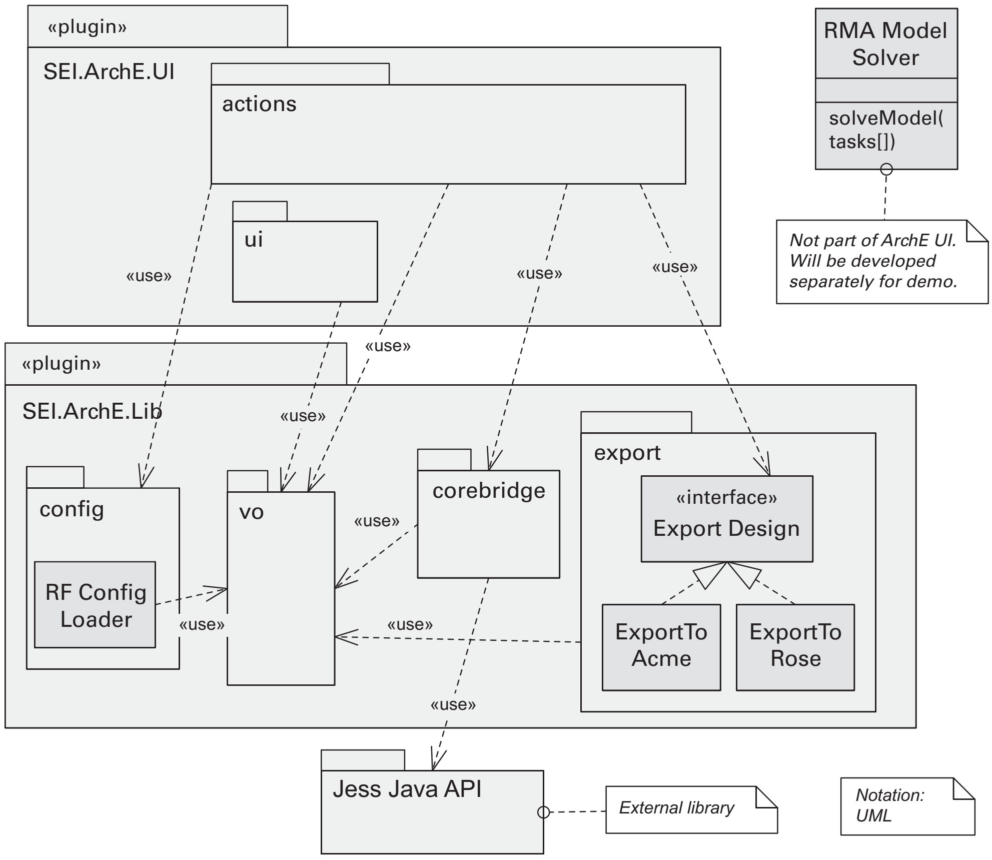


Figure 6.20
Decomposition-uses-generalization combined view for ArchE

attribute requirements, features of the system being designed, and preexisting pieces of design. Internally, ArchE constructs a representation of the responsibilities of the system and the dependencies among them. ArchE is powered by reasoning framework plug-ins that can create quality attribute models and use them to analyze performance, modifiability, and other properties. Based on the inputs—the results of quality attribute analyses and responses provided by the architect to questions that ArchE raises interactively—ArchE creates an architecture design.

Figure 6.20 is the primary presentation of a combined view showing ArchE's module decomposition, uses, and generalization. ArchE is an Eclipse-based tool that uses the Jess rule engine. The <<plugin>> stereotype indicates that contained modules are packaged as Eclipse plug-ins.

Tiered Client-Server and Deployment

The use of tiers is explained in Section 4.6.2. The deployment style is discussed in Section 5.2.

In Chapter 4, client-server was discussed as a C&C style. There are many alternatives for allocating the components in each tier to the supporting hardware infrastructure. The network topology and the deployment structure of the software affect several quality attributes, such as availability and throughput (enhanced by replication and clustering of machines), and performance (components on different machines require remote calls to interact).

Figures 6.21 and 6.22 show a multi-tier client-server view and a deployment view, respectively, of a banking application. Figure 6.23 shows the combined view.

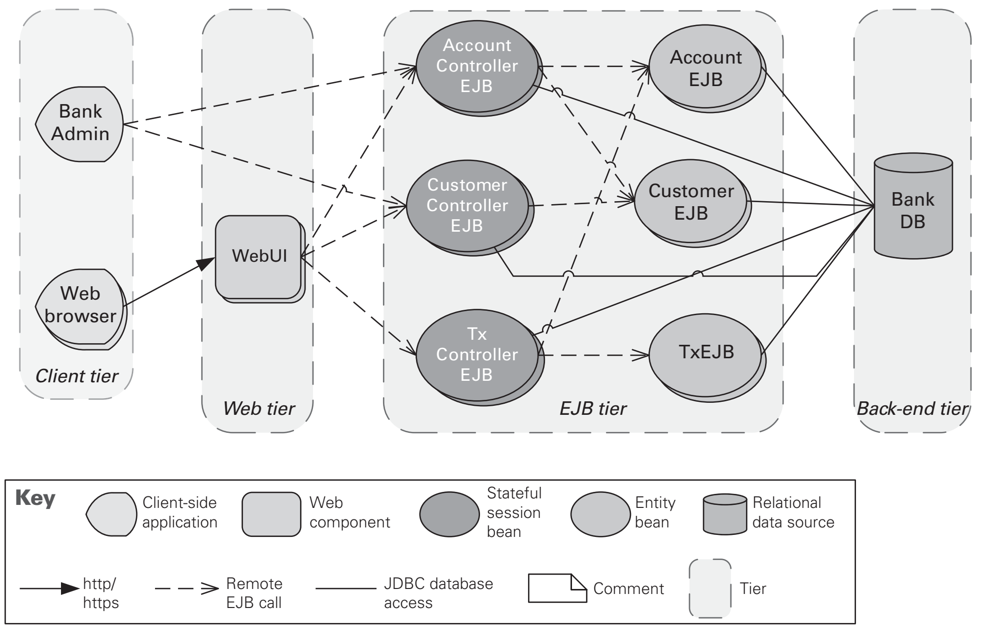


Figure 6.21
The multi-tier client-server view of the Duke's Bank application

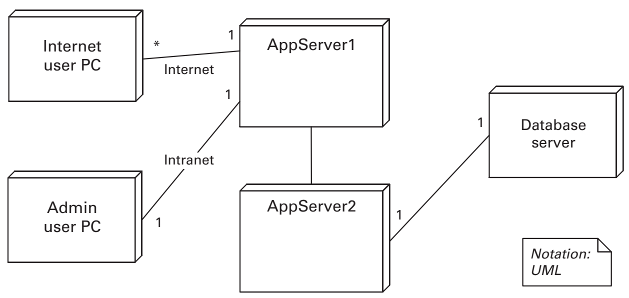


Figure 6.22
A deployment view of the banking system

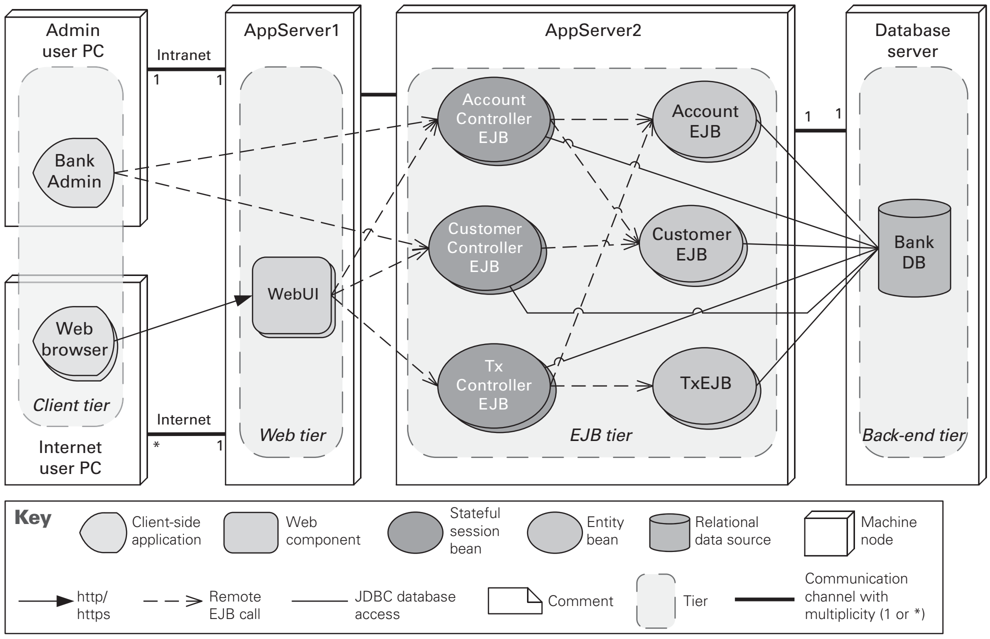


Figure 6.23
A combined multi-tier client-server deployment view

6.7 Summary Checklist

- Refinement, the gradual disclosure of more-detailed information, is a chunking mechanism. Decomposition refinement reveals internal substructure. Implementation refinement replaces elements with different elements showing different element relation types that are closer to the actual realization.
- Documentation may or may not show all elements and relations; when some elements and relations are suppressed, the view documentation should make it clear to the reader that not everything is shown.
- A context diagram shows what's in and what's out of the system under consideration and the external entities with which the system interacts.
- An architecture document does not have a single top-level context diagram, but rather one in each view. Each such diagram shows the interactions with the environment in the vocabulary for that view. All show what's in and what's out.
- Document variation points by describing what elements and relations have been designed with variation mechanisms, and how to exercise those mechanisms. Document the vari-

ation points where they occur, but explain them in a variability guide.

- Showing how views are associated with each other often yields useful insights about the architecture. One alternative to do that is to list, possibly in a table, how elements in one view are associated with elements in the other view. Another alternative is combine different views into one by creating an overlay. Yet another alternative is to produce a view from a hybrid style, which is a combination of two or more styles.
- Views with a high correspondence are good candidates for mapping, and views that complement each other are good candidates for combining.

6.8 Discussion Questions

1. A user invokes a Web browser to download a file. Before doing so, the browser retrieves a plug-in to handle that type of file. Is this an example of a dynamic architecture? How would you document it?
2. Suppose that communication across layers in a layered system is carried out by signaling events. Is event signaling a concern that is part of the layered style? If not, how would you document this system?
3. Consider a shared-data system with a central database accessed by several components in a client-server fashion. What are your options for documenting the two-style nature of this system? Which option(s) would you choose, and why?
4. A bridging element is one that can appear in two separate views. Both views will have room for documenting the element's interface and its behavior. Assuming that we do not wish to document information in two places, how would you decide where to record that information? Suppose that the bridging element is a connector with one role for one style and one role for another. Where would you record the information then?
5. Sketch a top-level context diagram for a hypothetical system as it might appear in the following views, assuming in each case that the view is appropriate for that system: (a) uses, (b) layered, (c) service-oriented, (d) client-server, and (e) deployment.

6.9 For Further Reading

Michael Jackson's book on problem frames has a good chapter on combining multiple problem frames (Jackson 2001). Although it is cast in terms of the problem space, rather than the solution space of architectures, many of the ideas carry over.

A robust community of researchers is interested in the capture and use of architectural knowledge, a generalization of architectural decisions. The Sharing and Reusing Architectural Knowledge (SHARK) series of workshops is a good place to learn more; an online search will turn up current offerings and past results (de Boer and Farenhorst 2008). An important research project in capturing architectural knowledge is the GRIFFIN project (griffin.cs.vu.nl) at VU University in Amsterdam.

The entries in the template for documenting architectural decisions in Section 6.5 are based on IBM's e-Business Reference Architecture Framework (Flurry and Vicknair 2001), where architecture decisions are a key deliverable, and from the REMAP and DRL metamodels (Akerman and Tyree 2005). The template also leverages Kruchten's work on an ontology of software architecture design decisions (Kruchten 2009). Another ontology is given by Komiya (1994). Both are well worth a look.

Documenting Software Interfaces

7

In this chapter, we look at these aspects of interface documentation:

- Standard organization
- Stakeholders
- Conveying syntactic information
- Conveying semantic information
- Examples of interface documentation

7.1 Overview

So far we have emphasized documenting architecture elements and their relations using various kinds of views. More implicitly than explicitly, we have stated that of course all those elements have **interfaces** through which they can interact with each other. Interfaces are supremely architectural, for without them one cannot perform analyses or system building—both activities we want to do with an architecture. Therefore, a critical part of documenting a view includes documenting the interfaces of the elements shown in that view.

An **interface** is a boundary across which two elements meet and interact or communicate with each other.

Modules, as discussed in Chapters 1 and 2, clearly have interfaces. As we said in Chapter 3, components also have interfaces, but they are often called ports. In this chapter, we will not distinguish between module interfaces and component interfaces; the way you document them is the same.

Describing an element's interface means making statements about what other elements can depend on when using this element. Designing an interface means deciding (and documenting with an **interface document**) which services and properties should be externally visible and which should not. Everything

An **interface document** is a specification of what an architect chooses to make publicly known about an element in order for other entities to interact or communicate with it.

that is externally visible becomes a contract, a promise to users that the element indeed will fulfill its obligations. This on the other hand also means that every implementation of the element that does not violate the contract is a valid one.

An element's **actors** are the other elements, users, or systems with which it interacts.

An element is used by **actors**. Actors are other elements, either internal or external to the system documented, that interact with an element through its interface. Those interactions can take a variety of forms, such as function or method calls, Web service requests, remote procedure calls, data streams, shared memory, and message passing. Most involve the transfer of control and/or data. These points of interaction with an element are called **resources**. Thus, an interface consists of one or more resources available for consumption by actors. If the element that provides that interface is a class, the resources are typically called methods.

A **resource** of an interface represents a function, method, data stream, global variable, message end point, event trigger, or any addressable facility within that interface.

An interaction extends beyond functionality and state changes. For example, if element A calls element B, the amount of time that B takes before returning control to A is part of B's interface because it may affect A's behavior.

Let's establish some principles about interfaces.

- *All elements have interfaces.* All software elements described in any view interact with their environment. The architect decides which aspects of the element's interfaces need to be documented.
- *An element's interface is separate from its implementation.* This principle is particularly useful when we want multiple implementations of an element (such as platform-specific implementations) that provide the same interface.
- *An element can have multiple interfaces.* Each interface contains a separate collection of resources that have a related logical purpose, or represent a role that the element could fill, and each collection serves a different class of actors. Multiple interfaces provide a separation of concerns. A specific actor might require only a subset of the resources provided. If the element has multiple interfaces, this subset of resources should be provided by one of the interfaces. Conversely, the provider of an element may want to grant actors different access rights, such as read or write, to prevent resource contention or to implement a security policy. Multiple interfaces support different levels of access.

  Multiple interfaces also support the evolution of elements that are publicly available or used by a large number of actors. If the element's interface changes, it may not be feasible to modify everything that uses the old version. So you can support evolution by keeping the old interface and adding a new one.

- *Elements not only provide interfaces but also require interfaces.* An element interacts with its environment by making use of resources or assuming that its environment behaves in a certain way. Without these required resources, the element cannot function correctly. For example, an element may require Internet connectivity. In this case the element would specify that an actor can use a certain resource it provides only if Internet connectivity is present. Otherwise some error indication will be delivered.

See "Coming to Terms: Provided vs. Required Interfaces" on page 264, in this chapter.

- *Multiple actors may interact with an element through its interface at the same time.* Some interfaces don't allow multiple concurrent interactions because of synchronization and multithreading issues. These restrictions can be made clear by specifying the number of actors that can interact with an element via a particular interface at the same time.
- *Interfaces can be extended by generalization.* Many times, several interfaces you are designing will include a common set of resources. These resources can be placed in a separate interface, and by using a generalization relation, you are indicating that children interfaces contain (and may extend) the common resources. Examples of resources often shared by several interfaces include the following:
  - An initialization operation
  - A set of exception conditions, such as failing to have called the initialization operation
  - A standard way to handle exceptions, such as invoking a named error handler
  - A standard statement of semantics, such as persistence of stored information
- *Sometimes it's useful to distinguish interface types from interface instances in the architecture.* Some components can provide multiple instances of the same interface. Consider for example a component that is an observer in the observer design pattern. This component provides an interface with an operation to be called when an observable component sends a change notification. Thus far we have the interface type. If the component is an observer of multiple different observable components, then it may be useful to represent that in the architecture using multiple instances of the observer interface type.

You can use the same template to document a required interface and a provided interface. The documentation of a required interface may become the documentation of a provided interface once the element that provides it is designed or implemented.

COMING TO TERMS

Provided vs. Required Interfaces

Architecture elements provide services to other elements through one or more interfaces. This concept of an element and its interface is one of the enduring bedrock concepts of software engineering. But architecture elements can, and often do, need specific services from other elements in order to function correctly. To capture this need, architects can document a *required* interface.

The information you need to document about what your architecture element requires is the same as what you should document for what it provides: resources, their syntax and semantics, their error-handling behavior, their quality attribute characteristics, and any variation points they provide. In short, you can use the template given in Figure 7.5, later in this chapter, for documenting an interface that your element provides *and* (separately) an interface that your element requires. You can even fill in the section on rationale and design issues to record your decision-making process for why your element needs what it needs.

Suppose another element provides just the resources you need, or at least resources that are close enough to what you need that you can use them successfully. Would you document a required interface for your element then? Probably not, opting instead to refer to the other element's provided interface, to say essentially, "I need *that*."

But suppose no other element provides the resources your element needs. Maybe there will be one, but its designers are not as far along as you. In that case, it makes much more sense for you to document a required interface for your element, to guide the forthcoming development. Once the element that will provide the interface is designed, your required interface documentation may become the documentation of the new element's provided interface. When the providing element exists but you think its interface is likely to change substantially over time, or if you think that the element itself might become unavailable, then documenting your required interface also makes good sense. It can then be used to guide and constrain the evolution of the other element's provided interface, or to shop for and qualify potential replacement elements. Documenting an element's required interfaces also makes that element's reusability much easier to judge, because you can immediately see what resources it would expect to find should it be moved to a new environment.

Like all architecture documentation, a required interface can be documented to the degree of specificity needed to do the job. You might sketch out some resources and trust the designers of the providing element to fill in the details, to which you can then adapt your element.

Linking up required and provided interfaces (using, for example, UML's socket-and-lollipop notation) can give confidence that in a system build, every element has what it needs to work correctly. In UML a socket-and-lollipop pair symbolizes that the interfaces are "compatible," meaning at least that the provided interface supports a superset of the operations and signals specified in the required interface. That doesn't tell you if the requiring element uses all resources on the provider's interface, or only one or two.

7.2 Interface Documentation

Although an interface comprises all aspects of the interaction an element has with its environment, what we choose to disclose about an interface—that is, what we document in an interface's documentation—is more limited. Writing down every aspect of every possible interaction is not practical and almost never desirable. Rather, you should expose only what users of an interface *need* to know in order to interact with it. Put another way, you choose what information is permissible and appropriate for people to assume about the element.

The interface documentation tells what other developers need to know about an interface in order to use it in combination with other elements. Note that a developer might observe element properties that are an artifact of how the element is implemented but that are not in the interface documentation. Because these are not in the interface documentation, they are subject to change, and developers use them at their own risk.

Also recognize that different people need to know different kinds of information about the interface. You may have to provide separate sections in the interface documentation to accommodate different stakeholders of the interface.

Section 10.1 provides a documentation template for views, which has a section reserved for documenting element interfaces.

Interfaces are documented as part of a view. When a given interface occurs in more than one view, choose one to hold the interface documentation and refer to it in the other. Alternatively, package the interface documentation separately and have all views point to it.

Sometimes interfaces in different views have a direct correspondence but are not exactly the same. For example, the interface of a module in a module view often corresponds directly to the interface of a component in a component-and-connector (C&C) view. In many cases, the module and the C&C interfaces are identical, and documenting them in both places would produce needless duplication. In that case, you should document the interface in the view where the documentation will be more useful and make the other view refer to it. For example, a programmatic interface that offers procedure calls as resources will be most useful for implementers, and they are likely to look for the documentation in module views. On the other hand, an interface that corresponds to a message end point in a system using asynchronous messaging is probably more relevant in a C&C view that describes the runtime interactions, queue capacities, and overall throughput.

In other cases, a module and a C&C interface map to each other but are not identical. For example, a module view of a service-oriented architecture (SOA) system may show a Java class that provides an interface with five different operations. Two of these operations correspond to the interface of a SOAP Web service that is depicted in the C&C SOA view of the same system. The other three operations correspond to the interface of a REST Web service that is provided by a different component in the C&C SOA view. Each of the two Web service interfaces corresponds to only part of the module interface. In addition, the Web service interfaces may expose properties (such as availability or response time) not relevant in the module view. The syntax of the resources and data types may also differ due to the translation from the module implementation language (Java) to the language of the Web service interface (XML). In cases like this, you should document the interfaces separately but also record the mapping between them.

Section 10.2.1 describes how to document the mapping between views, which can contain the mapping between interfaces from different views.

ADVICE

Guidelines for Documenting an Interface

- Focus on how elements interact with their environments, not on how elements are implemented. Restrict the documentation to effects that are externally visible.

- Expose only what users of the interface need to know. Including a piece of information in the documentation is an implicit promise that the information is reliable and stable. Once information is exposed, other elements may rely on it, and changes will have a more widespread effect.
- Keep in mind who will be using the interface documentation and what types of information they will need. Avoid documenting more than is necessary. For example, you probably need less detail in the interface documentation of a module used only by another developer on the team than you need for an interface that is part of a commercially available API. This chapter presents the "maximum" approach, that is, a fully documented interface. Depending on the importance of the interface, you should decrease the amount of information and the effort spent in the interface documentation.
- When a given interface occurs in multiple views, document it in one view and refer to it in the other, or document the interface separately and make the views point to this interface documentation.
- An interface in a module view and its equivalent counterpart in a C&C view should be documented more extensively only in the view where the documentation will be more useful to the stakeholders. When interfaces in different views map to each other but are not identical, you should document them separately and document the mapping as well.
- Be as specific and as precise as you can, remembering that interface documentation that can be interpreted differently by various parties is likely to cause problems and confusion.

See "Coming to Terms: Signature, Interface, API" on page 280, in this chapter.

An interface may or may not have an identity of its own. In the simplest situation, an element A provides a single interface that is not provided by any other element. This interface is implicitly associated with element A and doesn't need a name—it's the interface of A. Documenting the interface of A is part of documenting element A. In another situation, an element provides two or more different interfaces. Then it's probably a good idea to identify the interfaces, as I1 or I2 for example, and document them separately. There's also the situation of a single interface that is provided by two or more elements. In that case, the interface should have an identity so that elements can refer to it, and the interface should be documented independently from the elements.

As in all architecture documentation, the amount of information conveyed in the interface documentation may vary, depending on the importance of the interface and on the stage of the design process when the documentation is updated.

- Early in the design process, the interface might be scarcely specified; for example, an order tracking module provides an operation to locate an order.
- Later, when the responsibilities of the elements become stable, the interface documentation is more fully elaborated; for example, the order-tracking module provides the method `locateOrder(orderId)` with some description about its semantics.
- Some time later, you may even refine the interface documentation with the final syntax for the method: `OrderBean locateOrderById(long orderId)`.

7.2.1 Showing the Existence of Interfaces in Diagrams

The *existence* of interfaces can be shown in the primary presentations by using most graphical notations available for architecture. Figure 7.1 shows an example using an informal notation.

The existence of an interface can be implied even without using an explicit symbol for it. If there is a relation going from element A to B and the relation type involves an interaction,[1] that implies that the interaction takes place through the interface of element B.

Figure 7.2 illustrates how interfaces are shown in UML. A provided interface is depicted as a lollipop, and the socket symbol is used for required interfaces (Figure 7.2(a)). Although it shows the existence of an interface, the lollipop symbol reveals little about the definition of an interface. UML interfaces can be connected to classes, components, and packages.

Sometimes interfaces are depicted by themselves, with or without an associated element. In UML, you can do that by using the classifier box with the <<interface>> stereotype instead of the lollipop. This alternative is particularly useful when multiple elements implement the same interface. Another benefit is that the resources of the interface can be listed in the operations compartment. Figure 7.2(b) shows the provided and required interfaces of class Garage Door as two separate boxes.

1. Examples of relations that don't involve an interaction include *is a subclass of* and *decomposition.*

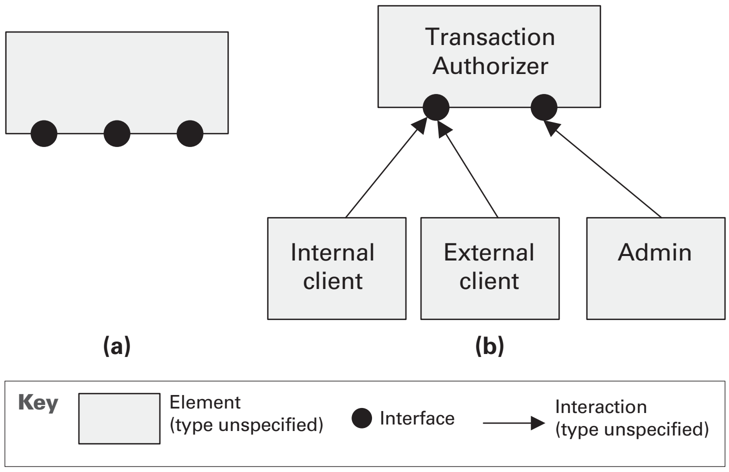


Figure 7.1
Graphical notations for interfaces typically show a symbol on the boundary of the icon for an element. Lines connecting interface symbols denote that the interface exists between the connected elements. Graphical notations like this can show only the existence of an interface, not its definition. (a) An element with multiple interfaces. For elements with a single interface, the interface symbol is often omitted. (b) Multiple actors at an interface. Internal client and External client both interact with Transaction Authorizer via the same interface. This interface is provided by Transaction Authorizer and required by both Internal client and External client.

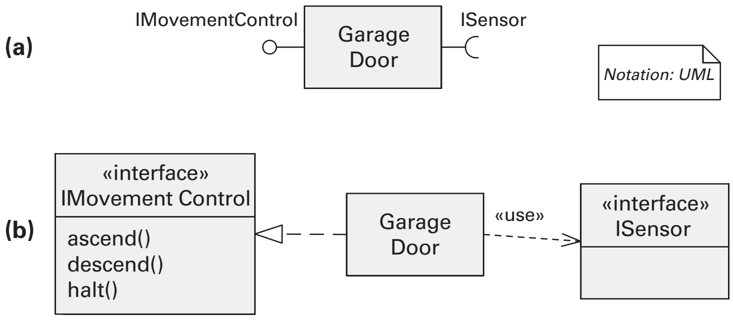


Figure 7.2
UML uses a lollipop to denote a provided interface, which can be appended to classes, components, and packages. Required interfaces are represented with the socket symbol, which is also appended to classes and other types of elements. UML also allows a class symbol to be stereotyped as an interface; a dashed line with a closed, hollow arrowhead shows that an element realizes an interface. The operations compartment of the class symbol can be annotated with the interface's signature information: method names, arguments and argument types, and so on. Thus the diagram in (a) is equivalent to (b) in this figure.

ADVICE

Use an independent box for the interface in your primary presentations if

- You wish to show the operations available in the primary presentation.
- You are making provisions for multiple elements that realize the same interface.

Although it's never wrong to show interfaces explicitly, be careful not to increase the visual clutter of the diagrams.

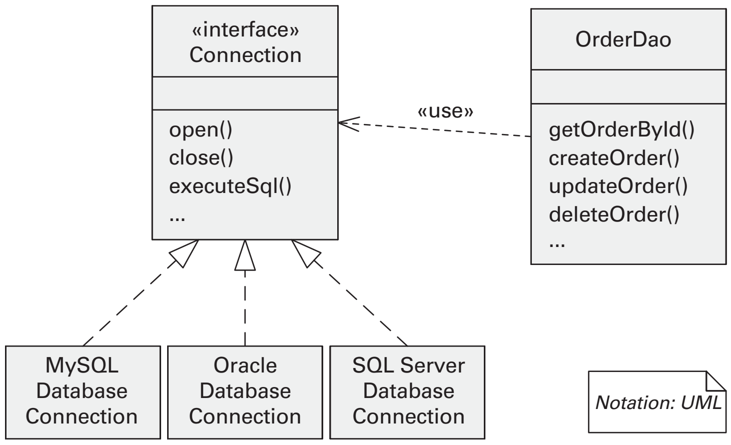


Figure 7.3
An interface can be shown separately from any element that realizes it, thus emphasizing the interchangeability of element implementations. `OrderDao` (and other classes not shown) require an object that implements a database connection, which is represented by the `Connection` interface. Many elements realize this interface, representing the interchangeable alternatives of database connection implementations.

When the diagram shows a module using an independent interface, it indicates that any element implementing the interface can be used. This is a useful means of expressing a particular kind of variability: the ability to substitute realizing elements, as shown in Figure 7.3.

ADVICE

Multiple Interfaces

Elements having multiple interfaces raise some subtle design issues and some important documentation issues. If an element interacts with more than one actor, it's usually best to show interfaces explicitly in your diagrams. If you don't, a diagram such as Figure 7.4(a) can be ambiguous: Does E have one interface or two? Showing the interface symbol, as in Figure 7.4(b) or (c), resolves the ambiguity.

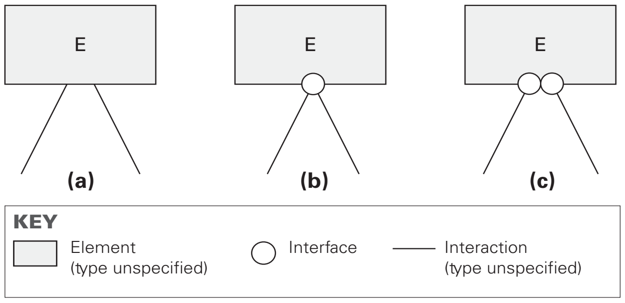


Figure 7.4
(a) Does element E have one interface or two? This diagram makes it difficult to determine at a glance. (b) By using the interface symbol, it's clear that this element has one interface and that (c) this element has two interfaces.

7.3 A Standard Organization for Interface Documentation

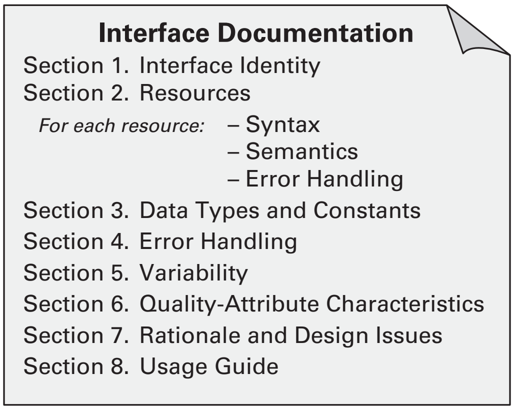
Remember that an important principle for sound documentation prescribes using a standard organization. A standard organization lets you fill in what you know about an interface now and indicate "TBD" for what you don't yet know, thus providing a to-do list for the remaining work. This section suggests a standard organization (that is, a template) for interface documentation (see Figure 7.5).

Like all templates in this book, you may wish to modify the one presented in this section to remove items not relevant to your situation or to add items unique to your business. More important than which standard organization you use is the practice of using one.

The standard organization can be used to document each interface of an architecture element. It consists of the following sections:

1. **Interface Identity.** When an element has multiple interfaces or when the same interface is provided by multiple elements, name the interface. In other cases the identity of the interface is the same as the identity of the element it's associated with. Some programming languages, such as C# and Java, or frameworks, such as COM, even allow these names to be carried through into the implementation. In some cases merely naming an interface is not sufficient, and the version of the interface must be specified as well. For example, in a framework with named interfaces that have evolved over time, it could be very important to know whether you mean v1.2 or v3.0 of the persistence interface.
2. **Resources.** The heart of an interface document is the set of resources provided to its actors. Resources are often operations (such as methods, procedures, and functions), but in a more general notion of interface they can be other things, such as data streams, shared data, and messaging

Figure 7.5
Template for interface documentation



Interface Documentation

Section 1. Interface Identity
Section 2. Resources
For each resource: – Syntax
– Semantics
– Error Handling
Section 3. Data Types and Constants
Section 4. Error Handling
Section 5. Variability
Section 6. Quality-Attribute Characteristics
Section 7. Rationale and Design Issues
Section 8. Usage Guide

end points. In this section you should list the resources and, for each resource, describe the following:

– *Resource Syntax.* This is the resource's signature, which includes any information needed to write a syntactically correct program that uses the resource. The signature includes the name of the resource, names and data types of arguments, if any, structure or data type of return values, if any, and so forth.

– *Resource Semantics.* What is the result of using this resource? What does the resource do from the perspective of the actor invoking it? Semantics come in a variety of guises, including:

  i. Assignment of values to the parameters and returned values, including their purpose and semantics. The value assignment might be as simple as setting the value of a return argument or as far-reaching as updating a database table.

  ii. Changes in the element's externally visible state brought about by using the resource. For example, invoking a resource called `open()` on interface `IConnection` may change the state of the connection to enable it to start exchanging data. Are these changes persistent or transient? If transient, what is the duration or termination condition?

  iii. Events that will be signaled or messages that will be sent as a result of using the resource.

  iv. The side effects on other environmental elements as the result of using this resource. For example, if you ask a resource to destroy an object, trying to access that object in the future through other resources will produce quite a different outcome—an error—as a result.

  v. Humanly observable results. For example, calling a program that turns on a display in a cockpit has a very observable effect: the display comes on.

  vi. Whether the execution of the resource will be atomic or may be suspended or interrupted, and whether the interaction is synchronous or asynchronous, if such a distinction is applicable.

  vii. Usage restrictions. Under what circumstances may this resource be used? Perhaps data must be initialized before it can be read, or perhaps a particular method cannot be invoked unless another is invoked

Consider using preconditions and postconditions for documenting resource usage restrictions and resource semantics. A precondition states what must be true before the interaction is permitted; a postcondition describes any state changes resulting from the interaction.

first. Perhaps there is a limit on the number of actors that can interact via this resource at any instant. Perhaps there is a limit of one actor that has ownership and is able to modify the element, whereas others have only read access. Perhaps the resource is thread safe; that is, it can be invoked simultaneously by multiple actors. Perhaps the resource can be invoked only when the authenticated user belongs to a certain group or has certain access rights. Some restrictions are less prohibitive; for example, Java interfaces can list certain methods as *deprecated,* meaning that users should not use them, as they will likely be unsupported in future versions of the interface. Usage restrictions are sometimes documented by defining *exceptions* that will be raised if the restrictions are violated.

– *Error Handling.* Describe error conditions and exceptions that can be raised by the resource.

See “Coming to Terms: Error Handling” on page 277, in this chapter.

ADVICE

Guidelines for Documenting the Semantics of a Resource

- Write down only those effects that are visible to a user: the actor invoking the resource, another element in the system, or a human observer of the system. Ask yourself how a user can verify what you have said. If your semantics cannot be verified, the effect you have described is invisible, and you haven’t captured the right information.
- Try to define the semantics of invoking a resource by describing ways other resources will be affected. For example, in a stack object, you can describe the effects of `push(x)` by saying that `pop()` returns `x` and that the value returned by `getStackSize()` is incremented by 1.
- If you describe the semantics using prose, be as precise as you can. Be suspicious of all verbs. For every verb in the specification of a resource’s semantics, ask yourself exactly what it means and how the resource’s users will be able to verify it. Eliminate vague words, such as *should*, *usually*, and *may*. For operations that position something in the physical world, be sure to define the coordinate system, reference points, points of view, and so on, that describe the effects.
- Clearly state any assumptions, preconditions, and bound values for parameters. We should expect that users will use a resource in ways the designers did not envision, and we should try to describe what the limits are.
- Avoid giving an example use in place of specifying the semantics. Usage is a valuable part of interface documentation and merits its own section in the

documentation, but it is given as advice to users and should not be expected to serve as a definitive statement of resources' semantics. Strictly speaking, an example defines the semantics of a resource for only the single case illustrated by the example. The user might be able to make a good guess at the semantics from the example, but we do not wish to build systems based on guesswork.

- Avoid giving an implementation in place of specifying the semantics. Do not use code to describe the effects of a resource.

3. **Data Types and Constants.** Sometimes we need to create new data types (such as records, structs, classes, enumerations, or unions) for the data passed to or returned by resources in the interface. These data types may be defined in the scope of the interface and should be described in the interface documentation. For example, in an airline reservation system, interface `IReservation` may provide a resource `makeReservation()` that returns a new data type `ReservationRecord`. This new data type described in the interface documentation may contain flight number, departure date and time, seat assignment, class, fare, and other data elements. If the data type is defined by another element, a reference to the definition in that element's documentation is sufficient. In any case, programmers writing elements using such a resource need to know (a) how to declare and assign values to variables of the data type, (b) what operations and comparisons may be performed on members of the data type, and (c) how to convert values of the data type into other data types, where appropriate.

   Likewise, new constants are sometimes created in interfaces to hold commonly used values and make programming against the interface more convenient. For example, interface `Sequencer` of the Java sound API has an operation `setLoopCount(int count)` to set the number of repetitions of the loop for playback on a MIDI device. For convenience, the interface defines a constant called `LOOP_CONTINUOUSLY` that can be passed as an argument to that operation.

4. **Error Handling.** Often you may want to use an error-handling behavior that is common to all or many resources. In that case, you can use this section to describe common error-handling behavior instead of repeating the behavior for every resource in section 2.

ADVICE

For documenting the error handling for resources in either section 2 or section 4 of interface documentation, do the following:

- If only a few resources have error handling, describe it in section 2.
- If most of the resources follow a common error-handling procedure, describe it in section 4.
- If most of the resources follow a common error-handling procedure but there are resource-specific variations, such as error codes, describe the variations in section 2 and the error-handling procedure in section 4.
- If you are using an error-handling procedure that is common for the whole system, describe the procedure in the rationale section of the "beyond views" part of the documentation (see Section 10.2). Resource-specific information, such as error codes, still needs to be documented with the resource.

When describing error handling, keep in mind that there are different kinds of errors. An architecture-oriented classification of exceptions is summarized in Figure 7.6. In the context of an element's interface, exception conditions are one of the following:

1. *Errors on the part of an actor invoking the resource.*
   a. An actor sent incorrect or illegal information to the resource, perhaps calling a method with a null value parameter that should not be null. Associating an error condition with the resource is the prudent thing to do.
   b. The element is in the wrong state for the requested resource. The element entered the improper state as a result of a previous action or lack of a previous action on the part of an actor. An example of the latter is invoking a resource before the element's initialization method has been called.
2. *Software or hardware events that result in a violation in the element's assumptions about its environment.*
   a. A hardware or software error occurred that prevented the element from successfully executing. Processor failures, network not responding, and inability to allocate more memory are examples of this kind of error condition.

Chapter 10 presents a standard organization for documenting architecture views and the "beyond views" part of the architecture documentation, where you find sections for capturing rationale.

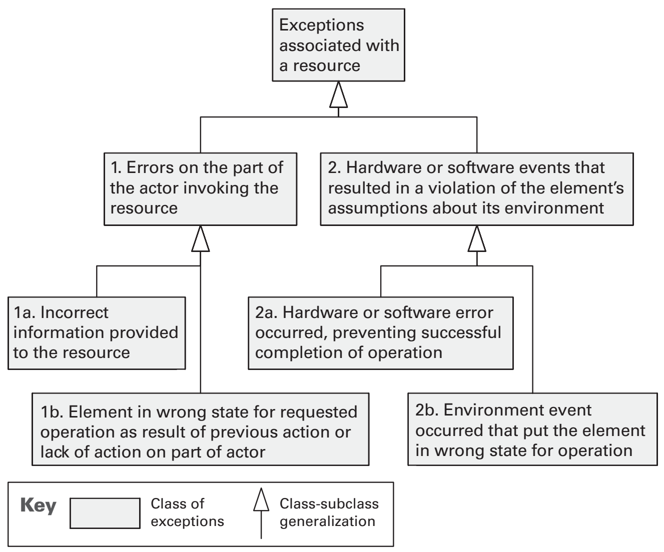

Exceptions associated with a resource
1. Errors on the part of the actor invoking the resource
2. Hardware or software events that resulted in a violation of the element's assumptions about its environment
1a. Incorrect information provided to the resource
2a. Hardware or software error occurred, preventing successful completion of operation
1b. Element in wrong state for requested operation as result of previous action or lack of action on part of actor
2b. Environment event occurred that put the element in wrong state for operation
Key
Class of exceptions
Class-subclass generalization

Figure 7.6
A classification of exceptions associated with a resource on an element's interface

b. The element is in the wrong state for the requested resource. The element's improper state was brought about by an event that occurred in the environment of the element, outside the control of the actor requesting the resource. An example is trying to read from a sensor or write to a storage device that has been taken off-line by the system's human operator.

Variability is discussed in detail in Section 6.4.

5. **Variability.** Does the interface allow the element to be configured in some way? These *configuration parameters* and how they affect the semantics of the interactions in the interface must be documented. Examples of variability include capacities—such as of visible data structures—that can be easily changed. Name and provide a range of values for each configuration parameter, and specify the time when its actual value is bound.

6. **Quality Attribute Characteristics.** You need to document what quality attribute characteristics, such as performance or reliability, the interface makes known to the element's users. This information may be in the form of constraints on implementations of elements that will realize the interface. The qualities you choose to concentrate on and make promises about will depend on the context. If you're devel-

oping an SOA application where services will be available to external service users, a service-level agreement (SLA) may be required. The SLA specifies quality properties for the entire service or specific operations in the service interface. For example, it may specify that certain operations should provide a specific response time, availability level, and capacity in terms of number of concurrent requests.

7. **Rationale and Design Issues.** Like rationale for the architecture or architecture views at large, you should also record the reasons behind the design of an element's interface. The rationale should explain the motivation behind the design, constraints and compromises, alternative designs that were considered and rejected and why, and any insight the architect has about how to change the interface in the future.
8. **Usage Guide.** Section 2 documents the syntax and semantics on a per-resource basis. This sometimes falls short of what is needed. In many cases, it's helpful to complement that information with examples that show the usage protocol for one or more resources of the interface. Code snippets are common in the usage guide, but sequence diagrams and other behavioral diagrams are also good choices, especially when a certain sequence of steps for the resource usage is required. Try to craft some clear and simple examples of the most common ways the interface might be used.

Behavior documentation is covered in Chapter 8.

COMING TO TERMS

Error Handling

When designing an interface, architects naturally concentrate on documenting how resources work in the nominal case, when everything goes according to plan. The real world, of course, is far from nominal, and a well-designed system must take appropriate action in the face of undesired circumstances. What happens when a resource is called with parameters that make no sense? What happens when the resource requires more memory, but the allocation request fails because there isn't any more? What happens when a resource never returns, because it has fallen victim to a process deadlock? What happens when the software is supposed to read the value of a sensor, but the sensor isn't responding or is responding with gibberish?

Terminating the program on the spot seldom qualifies as "appropriate action." More desirable alternatives, depending on the situation, include various combinations of the following:

- Returning a status indicator: an integer code—or even a message—that reports on the resource's execution, describing what, if anything, went wrong and what the result was.

- Retrying, if the offending condition is considered transient. The program might retry indefinitely or up to a preset number of times, at which point it returns a status indicator.
- Computing partial results or entering a degraded mode of operation.
- Attempting to correct the problem, perhaps by using default or fallback values or alternative resources.

These are all reasonable actions that a resource can take in the presence of undesired circumstances. If a resource is designed to take any of these actions, that should simply be documented as part of the effects of that resource. But many times, something else is appropriate. The resource can, in effect, throw up its hands and report that an error condition existed and that it was unable to do its job. This is where old-fashioned programs would print an error message and terminate. Today, they often raise an exception, which allows execution to continue and perhaps accomplish useful work. Modern programming languages provide facilities for raising exceptions and assigning handlers.

The right place to fix a problem raised by a resource is usually the actor that invoked it, not in the resource itself. The element detects the problem; the actor handles it. The actor might handle the exception by raising an exception of its own and bubbling the responsibility back along the invocation chain until the actor ultimately responsible is notified.

7.4 Stakeholders of Interface Documentation

In the prologue, we talked about stakeholders having special needs and expectations of an architecture. Some of the stakeholders of interface documentation and the kinds of information they require are as follows:

- *Developer of an element*, who needs the most comprehensive documentation of the interface the element provides. The developer needs to see any assertions about the interface that he or she will realize in the code. A special kind of developer is the *maintainer*, who makes assigned changes to the element and its interface.
- *Tester of an element*, who needs detailed information about all the resources and functionality provided by an interface. The tester can test only to the degree of knowledge embodied in the interface description. If required behavior for a resource is not specified, the tester will not know to test for it, and the element may fail to do its job.
- *Developer using an interface*, who needs detailed information about the resources provided in the interface to implement elements that will use it. A special case is the *integrator*, who

puts the system together from its constituent elements and has a stronger interest in the behavior of the resulting assembly. In a software product-line context, this stakeholder exploits the variability available in the elements to build different products.

- *Analyst,* whose information needs depend on the types of analyses conducted. For a performance analyst, for example, the interface document should give information that can feed a performance model, such as execution time required by resources.
- *Architect looking for assets to reuse in a new system,* who often starts by examining the interfaces of elements from a previous system. The architect may also look in the commercial marketplace to find off-the-shelf elements that can be purchased and do the job. To see whether an element is a candidate, the architect is interested in the capabilities of the interface resources, their quality attributes, and any variability that the element provides.
- *Project manager,* who is likely to use interface documents for planning purposes. Project managers can apply metrics (such as function-point analysis) to gauge the complexity and then infer estimates for how long it will take to develop an element that realizes the interface. Project managers can also spot special expertise that may be required, and this will assist them in assigning the work to qualified personnel.

7.5 Conveying Syntactic Information

Often architects use a notation they're familiar with or the notation of the target implementation technology when specifying the syntax of operations in an interface. A very common choice is a C-like syntax, for example:

```
Order getOrderById(long orderId)
```

Most programming languages have built-in ways to specify the signature of operations alone. C header (.h) files, and Java and C# interfaces are examples. Some technologies also provide their own syntax for describing the interfaces. The Object Management Group (OMG) Interface Definition Language (IDL) is used in the CORBA technology to specify interfaces' syntactic information. The Web services technology offers the Web Services Description Language (WSDL). However, WSDL is XML-based and would hardly be considered a good alternative to describe the signature of interface operations.

In the architecture interface documentation, it's often a good idea to use a syntax that is close to the syntax that will be used in the implementation. However, these days many interfaces are totally or partially implemented using languages that are suitable for automated parsing and processing but may be cumbersome for human readers. XML and JavaScript Object Notation (JSON) are examples. Avoid using these languages in the architecture documentation.

7.6 Conveying Semantic Information

Natural language is the most widespread notation for conveying semantic information. In many cases, a few sentences suffice to describe what an operation in the interface does and what are the usage restrictions. In other cases, natural language is not enough, and a formal language or notation can prevent future integration errors.

The example of interface documentation in Section 7.7.1 uses preconditions and postconditions to help explain the semantics of each resource.

A relatively simple and effective method for expressing the semantics of a resource in an interface is to write down its preconditions and postconditions. They can be specified using natural language, but Boolean algebra (that is, first-order logic) is sometimes used to enhance precision.

Traces are also used to convey semantic information by writing down sequences of interactions that describe the element's response to a specific use.

Section 8.5 describes behavior notations such as sequence diagrams and statecharts.

Semantic information often includes the behavior of an element or one or more of its resources. In that case, notations for behavior, such as sequence diagrams and statecharts, come into play.

COMING TO TERMS

Signature, Interface, API

Three terms people use when discussing element interactions are *signature*, *API*, and *interface*. Often they use the terms interchangeably, with unfortunate consequences for their projects. We have already defined an interface to be a boundary across which two elements meet or communicate with each other, and we have seen that documenting an interface consists of naming and identifying it, documenting syntactic information, and documenting semantic information.

A signature deals with the syntactic part of documenting an interface. When an interface's resources are invokable procedures, each comes with a signature that names the procedure and defines its parameters. Parameters are defined by giving their order, data type, and, sometimes, whether their value is changed by the procedure. A procedure's signature is the information that you would find about it, for instance, in the element's C or C++ header file.

An API, or application programming interface, is a vaguely defined term that people use in various ways to convey interface information about an element. Sometimes people assemble a collection of signatures and call that an element's API. Sometimes people add statements about programs' effects or behavior and call that an API. An API for an element is usually written to serve developers who use the element.

Signatures and APIs are useful but are only part of the story. Signatures can be used, for example, to enable automatic build checking, which is accomplished by matching the signatures of different elements' expectations of an interface, often simply by linking different units of code. Signature matching will guarantee that a system will compile and/or link successfully. But it guarantees nothing about whether the system will operate successfully, which is, after all, the ultimate goal.

In September 1999, NASA lost a $125-million orbiter when it was about to enter orbit around Mars. An undetected error in a data transfer between the Mars Climate Orbiter spacecraft team in Colorado and the flight navigation team in California caused the loss of the spacecraft. The error was a semantic mismatch in the data for maneuvering the orbiter into Mars orbit: one team used English units and the other used metric units.

A full-fledged interface is written for a variety of stakeholders and specifies the full range of effects of each resource, including quality attributes. Signatures and low-end APIs are simply not enough to let an element be put to work with confidence in a system. A project that adopts them as a shortcut will pay the price when the elements are integrated, if they're lucky, but more than likely after the system has been delivered to the customer.

7.7 Examples of Interface Documentation

Following are a couple of examples of interface documentation.

7.7.1 Zip Component API

The interface documentation that follows is for a hypothetical Windows COM component that provides standard zip archive operations. A client application can call the interface to create a zip file and add files to it, extract files from a zip file, list the files inside a zip file, and delete files from a zip file. The example is inspired by publicly available components that offer similar functionality, such as XZip (xstandard.com/en/documentation/xzip).

SAMPLE INTERFACE DOCUMENTATION

Section 1. Interface Identity

DSAVandBzip: Offers operations to compress, extract, list contents, and delete files from a standard zip file.

Section 2. Resources

```
void Zip(string[] filesToZip, string zipFile,
         bool savePath, int compressionLevel)
```

Compress the specified files and folder and add them to the specified zip file. Does not put a file system lock on files when reading them. If the destination zip file doesn't exist, create it.

Parameters:

- `filesToZip`: array with the names of the files or folders to be zipped. If an item is a folder, all files and folders inside the folder are zipped recursively.
- `zipFile`: pathname to the destination zip file that will hold the zipped content.
- `savePath`: if true, the items in the zip file will keep the original pathname relative to the folder specified in `filesToZip`; if false, path information will be removed.
- `compressionLevel`: varies from 1 (minimum compression, but faster to zip and unzip) to 4 (maximum compression, but slower).

Preconditions:

- Files listed in `filesToZip` exist and are not locked.
- The folder where the specified `zipFile` is located already exists, the current user has write permission on it, and there is enough disk space.

Postconditions:

- On success, the zip file is created and closed. The original files that were zipped are also closed and remain unchanged.

Possible error codes: 201, 203, 206, 211, 215, 252, 300

`void Zip(string[] filesToZip, string zipFile, bool savePath )`

Same as `Zip()` using the default `compressionLevel`. See “Section 5. Variability.”

`void Unzip(string zipFile, string destFolder, bool overwrite)`

Extract and decompress all items inside the specified zip file and save them to the specified destination folder. If a zipped file has a relative path associated to it, the pathname is appended to the destination folder. If the corresponding subfolders don’t exist in the destination, they are created.

Parameters:

- `zipFile`: pathname to the destination zip file that holds the zipped content.
- `destFolder`: pathname to the folder where the zipped files will be extracted to.
- `overwrite`: if true, simply overwrite existing files and folders with the same name in the destination folder.

Preconditions:

- Specified zip file is valid and nonempty.
- The destination folder already exists, the current user has write permission on it, and there is enough disk space.

Postconditions:

- The zip file is closed and its contents unchanged. The extracted files are closed at the end and contain the exact content of the original file prior to compression.

Possible error codes: 201, 206, 207, 252, 300

`ZipItem[] GetItems(string zipFile)`

Get a list of the contents of the specified zip file. Return an array of `ZipItem` objects in the order they were added to the zip file. Each zip item can be a file or a folder. This operation does not involve decompressing the files.

Parameters:

- `zipFile`: pathname to the destination zip file that holds the zipped content.

Preconditions:

- Specified zip file is valid and nonempty.

Postconditions:

- The zip file is closed and its contents unchanged.

Possible error codes: 201, 206, 207

`long ErrorCode`

Global read-only variable that contains the error code of the last operation or zero if the operation was successful. See “Section 4. Error Handling” for more information.

Section 3. Data Types and Constants

- `struct ZipItem`—represents an item (file or folder) inside a zip file. Attributes:
  - `string Name`: name of the file or folder
  - `string Path`: path to the zipped item
  - `DateTime Modified`: last modified on this date/time
  - `long OriginalSize`: size in bytes of original file
  - `long CompressedSize`: size in bytes of compressed file
  - `byte Type`: indicates whether it’s a file or a folder. Use constants `FOLDER` and `FILE`.
- `const byte FOLDER` = 1
- `const byte FILE` = 2

Section 4. Error Handling

Upon failure or when certain preconditions are not satisfied, all operations set the `ErrorCode` global variable. Possible error codes are:

- 201—Zip file is not valid.
- 203—Cannot create zip file.
- 206—Cannot allocate memory.
- 207—Cannot open zip file.
- 211—Cannot open file/folder to zip.
- 215—Zip file is same as the input file.
- 252—Cannot create files for swapping.
- 254—Unknown error when modifying zip file.
- 300—Disk is full or protected.

Section 5. Variability

- The component may be deployed as a Windows service or as a DLL to be loaded by a caller application.
- Windows registry keys are used for configurable properties, which are read by the component at load time:
  - Default compression level
  - Whether a log file is created with results of last operations
  - Location of the log file

Section 6. Quality Attribute Characteristics

The compression level will affect performance and disk space. If the level is higher, the zip or unzip operation will take longer. However, the operation will require less disk space as the resulting compressed file is smaller. The normal compression ratio obtained at compression level 4 is similar to the ratio obtained using commercial data compression tools, such as WinZip or WinRAR.

Operations that create or update a zip file require disk space for temporary files. The amount of space is not bigger than the size of the zip file.

The execution time of zipping a file is log-linear ($n \log n$) proportional to the size of the file.

The operations in the interface are thread safe and can be called by multiple simultaneous users.

Section 7. Rationale and Design Issues

Different compression levels were created to improve the flexibility for users that require maximum compression ratio versus users that need just a simple and fast compression component.

Section 8. Usage Guide

- Example of calling the component to zip some files:

```
DllImport("DSAVandBzip.dll")
Public static extern void Zip(string[] filesToZip, string zipFile, bool
savePath, int compressionLevel);
string[] myFiles = new string[3];
myFiles[0] = "C:\SEI\DSA\Chapter2.doc";
myFiles[1] = "C:\temp\new.css";
myFiles[2] = "C:\SEI\DSA\TOC.docx ";
Zip(myFiles, "C:\SEI\DSA\test.zip", true, 4);
```

7.7.2 Interface to a SOAP Web Service

The example software architecture document accompanying this book online contains the architecture documentation for the Adventure Builder application. See wiki.sei.cmu.edu/sad. The OPC Uses View contains the documentation for the `OpcPurchaseOrderService` and the `OpcOrderTrackingService` interfaces, which are SOAP-based Web services interfaces.

7.8 Summary Checklist

- All elements have interfaces.
- Many notations for interface documentation show only syntactic information. Make sure to include semantic information as well.
- Elements can have provided interfaces and required interfaces.
- An element can have multiple interfaces and multiple actors at each interface.
- An architect must carefully choose what information to put in interface documentation, striking a balance between usability and modifiability. Put information in an interface document that you are willing to let people rely on. If you don't want people to rely on a piece of information, don't include it.
- In graphical depictions, show interfaces explicitly if elements have more than one interface or if you want to emphasize the existence of an interface through which interactions occur. Otherwise, interfaces can be implicit.
- Follow the template given in Figure 7.5 or create your own, making sure to address the needs of the interface documentation's stakeholders.

7.9 Discussion Questions

1. Think about your favorite Web browser. How many interfaces does it have, and what actors are served by those interfaces?
2. Sketch a picture of the Web browser showing its interfaces and its environment.
3. For one of the interfaces you described in question 1, list a set of exceptions that the browser detects or, from your experience, fails to detect but should.
4. What's the difference between an interface and a connector?
5. What's the difference between an interface and a port?
6. Is there a difference between module (as described in Chapter 1) interfaces and component (as described in Chapter 3) interfaces?
7. Why does UML have different symbols for interface and port? In what situation, if any, would you attach an interface to a port of a UML component?
8. Look at an interface description in the Javadoc (or doxygen) documentation for a publicly available library and try to identify the information that corresponds to the information required by the sections of the template presented in Figure 7.5. Is any information missing?

7.10 For Further Reading

An excellent foundation paper on exceptions, which lays the groundwork for separating the concern of detecting an exception from the concern of handling an exception, is the one by Parnas and Wuerges (1976).

Joshua Bloch has delivered at conferences an excellent talk, titled "How to Design a Good API and Why It Matters," which contains practical guidelines regarding the design and documentation of APIs (Bloch 2006).

Mary Shaw has made the observation that we can't have complete interface documentation, because the cast of stakeholders is too numerous and the range of information they need is too broad. And in a world in which we get our components from other sources and know precious little about them, good interface documentation is even more rare. However, she points out that we can and do accomplish useful work with such incomplete knowledge. This is so because we can assign confidence measures to individual units of information that we pick up about a component from various sources. She calls such a unit a "credential," and she assigns it properties such as how we know it and what confidence we have in it (Shaw 1996a, Scaffidi and Shaw 2007).

Interfaces are extremely important in service-oriented solutions in general, and for applications that follow the software as a service (SaaS) model. In SaaS, instead of paying for a software license, customers pay for using the software, which exposes an interface and is available via the Web. In such a scenario, it's common to provide a service-level agreement. Quality properties that are usually expressed in SLAs, notations for SLAs, and mechanisms to monitor quality of service are discussed in the report by Bianco, Lewis, and Merson (2008).

Viewing interfaces as the set of assumptions that two components are allowed to make about each other dates from early work by Parnas (1971), echoed in later work about architectural mismatch (Garlan, Allen, and Ockerbloom 1995).

This page intentionally left blank

8 Documenting Behavior

With George Fairbanks

Documenting behavioral aspects of an architecture provides many benefits both during development of the architecture and during system maintenance. This information can be used to gain understanding of a system, and it can also help stakeholders reason about how a system built to the architecture will be able to meet many of its quality-related goals. For example, behavior documentation can identify potential deadlocks and bottlenecks. Such documentation clarifies to developers the steps and states involved in the operations.

Documenting an architecture requires behavior documentation that complements structural views by describing how architecture elements interact via their structures. Examples of structural diagrams include module, component-and-connector (C&C), and deployment diagrams. Structural relations provide a system view that reflects all potential interactions, few of which will be active at any given instant during system execution. Many notations are available to capture system behavior.

In this chapter, we recommend what aspects of behavior to document, we explain why that behavior documentation can be useful, and we show examples of how this documentation is used during the earliest phases of system development. In addition, we provide overviews and pointers to notations, methods, and tools that are available to help practitioners document system behavior.

8.1 Beyond Structure

Architecture is frozen music.

—Johann Wolfgang von Goethe

Reasoning about characteristics such as a system's potential to deadlock, its ability to complete a task in the desired amount of time, or its maximum memory consumption requires that the architecture description contain information about both the characteristics of individual elements as well as patterns of interaction among them. Behavior documentation adds information that reveals things like the following:

- The ordering of interactions among the elements
- Opportunities for concurrency
- Time dependencies of interactions, such as at a specific time (for example, "At 8 a.m.") or after a period of time (for example, "Every 30 ms.").
- Possible states of the system or parts of the system
- Usage patterns for different system resources (such as memory, CPU, database connections, or network)

Sequence diagrams and statecharts found in UML are examples of notations that support capturing behavioral information.

8.2 How to Document Behavior

Documented behavior supports exploring the range of possible orderings of interactions, opportunities for concurrency, and time-based interaction dependencies among system elements. In this section, we recommend steps you can take to reap these benefits.

There are three things you need to do to capture system behavior: (1) decide what kinds of questions the documentation should answer; (2) determine what behavioral information is available or can be stated as constraints on downstream developers; and (3) choose a notation.

8.2.1 Step 1: Decide What Kinds of Questions You Need to Answer

Determining what kinds of behavior to model depends on the type of system being designed, the stage of development, and the focus of the design effort.

For example, consider a banking system. In such a system, you focus on the order of events: credit, deposit, operation fee, and logging in a money transfer operation. The behavior must ensure that the transaction is atomic and that rollback procedures are in place. On the other hand, in a real-time embedded system, you need to say a lot about timing properties in addition to the order of events.

Early in the development you will want to talk about the elements and how they interact, not about the details of how input data is transformed into outputs. It may also be useful to say something about constraints on the transformational behavior within elements, because that behavior affects the global behavior of the system. Later in the development, the details should also be considered.

At a minimum you should model the stimulation of actions and the transfer of information from one element to another. In addition, you might want to model ordering constraints on these interactions. Restrictions on the order and combinations in which actions must occur should be documented if correct behavior depends on it. Documentation that has more explicit information about the constraints on interactions is more prescriptive for developers and more precise for analysis, and hence more likely to result in an implementation that will exhibit the intended behavior.

As an example of the importance of focus, consider an exploratory robot (or rover). If you're creating a sequence diagram to describe an interaction within the communication subsystem, you may abstract (that is, omit) interactions with the power management subsystem, because the diagram only needs to clarify the interactions concerning the communication submodules. Defining the scope is particularly critical for state-based diagrams. The first thing to ask is "This diagram shows the states of what?" For example, in an ATM system, a statechart can describe the states of the user screen, the money dispenser, the communication channels, a bank account, a bank card, the card reader, and so on. Each of these elements has distinct states and transitions, and they may be related (for example, if the card reader retains a card the user forgot to pick up, an event may be triggered to deactivate the card). If the scope of the state-based diagram is not well defined, you may end up trying to model combined elements that together have too many states.

8.2.2 Step 2: Determine What Types of Information Are Available or Can Be Constrained

Types of Communication

Looking at a structural diagram that depicts two interrelated elements, users of the documentation often ask "What does the line connecting the elements mean? Is it showing flow of data or control?" The answer should be in the diagram key. A behavioral diagram provides a place to describe aspects of the transfer of information and the stimulation of actions from one element to another in more detail than you include in diagram keys.

See "Perspectives: Quivering at Arrows" on page 41, in the prologue.

Table 8.1 shows some common examples of various types of communication. In this table we identify three different important characteristics of a type of communication. The first characteristic is the general purpose of the communication. In

Table 8.1 Types of communication

	Synchronous	**Asynchronous**
Data		*Local*: Shared memory *Remote*: Database
Stimulation	*Local*: Procedure call, semaphore *Remote*: RPC without parameters	*Local*: Interrupt, signal, or event without parameters *Remote*: Signal or event without parameters
Both	*Local*: Procedure call *Remote*: RPC with parameters	*Local*: Message, event with parameters *Remote*: Message, event with parameters

some cases, the primary purpose is to exchange data. In others, the primary purpose is to stimulate another element to signal that a task is completed or that a service is required. Often, however, a combination of the two is the main idea, as is the case when an element stimulates another to deliver data or when information is passed in messages or as parameters of events.

A second characteristic indicates whether elements communicate via synchronous or asynchronous means. Remote procedure call (RPC) is an example of synchronous communication. The sender calls the receiver and is blocked until the receiver responds. Messaging is an example of asynchronous communication. The sender does not concern itself with the state of the receiver when sending a message or posting an event. Right after the message is sent, the sender continues its execution and is not blocked waiting for a response. In fact, the sender and receiver may not be aware of each other's identity.

Consider the telephone and e-mail as examples. If you make a phone call to someone, the person has to be at the phone in order for it to achieve its full purpose. That is synchronous communication. If you send an e-mail message and go on to other business, perhaps without concern for a response, the communication is asynchronous. The distinction between synchronous and asynchronous communication has implications for the behavior of the transaction. An asynchronous call introduces concurrency and is more suitable for loosely coupled elements. The distinction also affects modifiability. Asynchronous interactions are usually more complicated, especially when the transaction needs a callback, which may require establishing a callback end point and a mechanism for correlating the original call to the callback message.

A third characteristic of the type of communication is whether the call is local (within the same container or machine) or remote. If it's remote, the performance is worse, because of the network overhead (even if the remote call reaches a component within the same machine, there's the overhead of going through the stack of network layers). Remote calls are also less reliable. A call or its response may not be delivered, may get corrupted, or may arrive in the wrong order.

Constraints on Ordering

In the case of synchronous communication, you probably want to say more than that there is two-way communication from A to B. For instance, you may want to say whether the target of the original message uses the assistance of other elements before it can respond to the original request.

You may want to be more specific about certain aspects of the way an element reacts to its inputs. You may want to note whether an element requires all or just some of its inputs to be present before it begins calculating. Also, you may want to say whether it can provide intermediate outputs or only final outputs. If a specific collection of events must take place before an action of an element is enabled, that should be specified, as should the circumstances (such as ordering) in which the events or element interactions will be triggered. These types of constraints on interactions provide information that is useful for analyzing the design for functional correctness, as well as for quality attributes.

Time-Based Stimulation

If any activities are specified to take place at specific times or after certain intervals of time, some concept of time needs to be introduced into your documentation. Time can be specified as either a point in time (that is, calendar based) or as a duration (timer based). Duration can be based on either wall time or task time. As an example of using a point in time, you may specify that certain behavior is different on weekends or holidays. As an example of using wall-time duration, you may specify that every five minutes, the system should determine how many people are logged in. As an example of task-based duration, you may specify that a task can use one minute of CPU time before being temporarily interrupted.

8.2.3 Step 3: Choose a Notation

Any language that supports documenting system behavior must include constructs for describing sequences of interactions.

Because a sequence is an ordering in time, it should be possible to show time-based dependencies. Sequences of interactions and the triggered activities are displayed in the order they are supposed to occur after certain stimulus arrives. Examples of stimuli are the passage of time and the arrival of an event. Examples of activities are computing and waiting. Constructs that show time as a point—for example, 8:00 a.m.—and time as duration—such as wait for 10 seconds—are normally also provided. As documentation of behavior implicitly refers to structure and uses structure, the structural elements of a view are an essential part of the language. In most behavior documentation, therefore, you can find representations of the following:

- Stimulus and activity
- Ordering of interactions
- Structural elements with some relations the behavior maps to

Two groups of behavior documentation are available. The languages to support behavior documentation tend to fall into one of two corresponding categories: traces and comprehensive models.

Use trace-oriented documentation if the goal is to describe the sequence of activities in the system in a specific scenario.

- One type of documentation allows you to capture what happens through the structural elements of a system during a scenario as *traces*. Traces are sequences of activities or interactions that describe the system's response to a specific stimulus. Traces are by no means a complete behavioral model of a system. However, the explicit enumeration of all traces would generate a complete behavioral model, although this isn't remotely feasible in most systems. Traces are easier to design and communicate because they have a narrow focus.

Use the comprehensive model type of documentation when a complete behavioral understanding is required, as is the case when performing a simulation or when applying static analysis techniques.

- Another type of documentation, often state based, shows the complete behavior of a structural element or a set of elements. This is called a *comprehensive model* of behavior because it is possible to infer all paths from initial state to final state. Comprehensive behavioral models support documentation of alternatives and repetitions to provide the opportunity of following different paths through a system, depending on runtime values. With this type of documentation, it is possible to infer the behavior of the elements for the arrival of any possible stimulus.

A difference between the two approaches is the focus of the documentation relative to individual elements. Traces are typically scoped to include all the system elements that are involved in a particular scenario. However, as mentioned earlier, only a fraction of the behavior of any given element shows up in any particular trace. Each comprehensive model, on the other hand, is typically scoped to focus on all the behavior of a particular element or group of elements. In order to reason about system-wide behavior, you must look at multiple comprehensive models side by side.

Many languages and notations are available for both types of behavior documentation. These differ in their emphasis on certain aspects of the behavior, such as how ordering is identified, how much support is available for documenting timing, what types of communication are easily modeled, and so on.

8.3 Notations for Documenting Behavior

In the subsections that follow, we provide cursory overviews of several notations to show trace and comprehensive types of behavioral specifications. The discussions are intended to provide a flavor of the particular notations and to motivate their use. There are many ways in which the diagrams we present in this section may be used together to support system understanding. Figure 8.1 shows a reasonable way to combine the strengths of several notations.

8.3.1 Notations for Capturing Traces

Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state. A trace describes a sequence of activities or interactions between structural elements of the system. Although it is conceivable to describe all possible traces to generate the equivalent of a comprehensive behavioral model, it is not the intention of trace-oriented documentation to do so.

In this section, we describe four notations for documenting traces: use cases, sequence diagrams, communication diagrams,

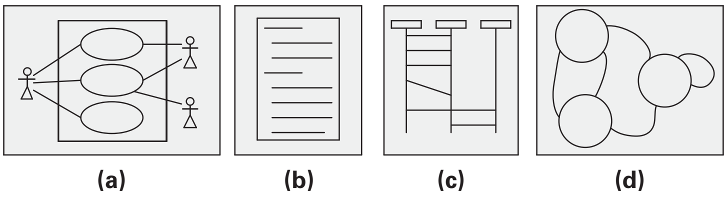


Figure 8.1
Using various types of behavior documentation together. (a) Begin by documenting an overview of the functional requirements as use case diagrams. (b) Then produce use case descriptions to document the events and actions that correspond to performing each use case. (c) Next, for each use case produce either a sequence diagram or a communication diagram to define the messages between envisioned architecture elements. (d) Finally, produce statecharts to complement the behavior documentation of the elements that have elaborate states and state transitions.

and activity diagrams. Communication diagrams were introduced in UML version 2 and are based closely on the collaboration diagram from UML version 1. Although other notations are available, we have chosen these four as a representative sample.

Use Cases

Use cases describe how actors can use a system to accomplish their goals. Use cases are frequently used to capture the functional requirements for a system.

UML provides a graphical notation for use cases, as shown in Figure 8.2, but it does not say how the text of a use case should be written. The UML use case diagram can be used effectively as an overview of the actors and the behavior of a system, but most of your effort should go into producing the textual use case description.

The use case description is textual, and it should contain: the use case name and brief description, actor or actors who initiate the use case (primary actors), other actors who participate in the use case (secondary actors), flow of events, alternative flows, and non-success cases. The use case description can be enhanced with preconditions, postconditions, assumptions, priority, and other information. A use case may include or extend other use cases. Figure 8.3 shows an example of a use case description for making a call in a telephone system.

All interactions in a use case are interactions between the actors and the system; no interactions within the system are shown. Human users are actors, but other computer systems can also play the role of an actor.

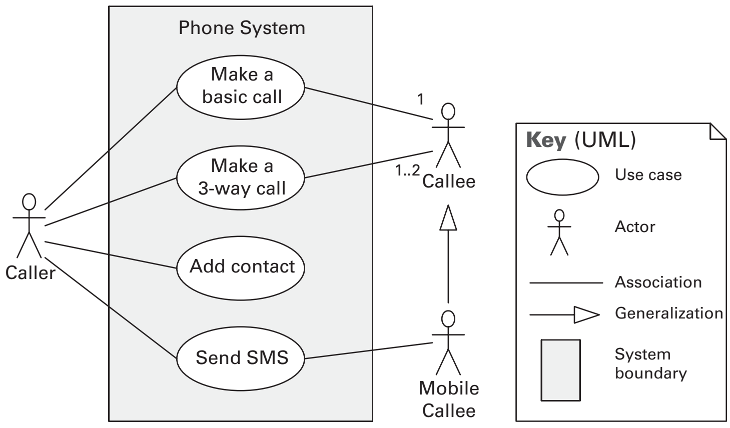


Figure 8.2
The UML use case diagram provides a quick overview of the system, actors, and the required behavior. This example shows some use cases in a telecommunication system.

Name: Make a basic call

Description: Making a point-to-point connection between two phones.

Primary actors: Caller

Secondary actors: Callee

Flow of events:

The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.

Exceptional flow of events:

The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

Figure 8.3
Example use case description for making a basic call in a telephone system. This use case contains a main flow of events and one exceptional flow of events.

Sequence Diagrams

A UML sequence diagram shows a sequence of interactions among instances of elements pulled from the structural documentation. It shows only the instances participating in the scenario being documented. A sequence diagram has two dimensions: vertical, representing time, and horizontal, representing the various instances. The interactions are arranged in time sequence from top to bottom. Figure 8.4 is an example of a sequence diagram that illustrates the basic UML notation. In practice, the notation you'll find in sequence diagrams is often simpler: return messages may not be there, execution occurrence bars may not be drawn, a single type of arrow may be used for all types of messages, and labels on messages may not exist.

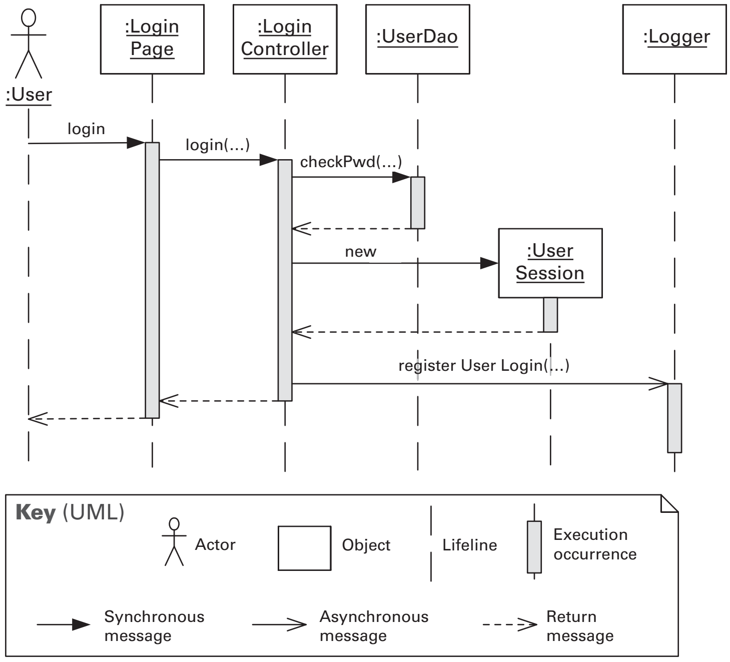
If your trace involves a mix of synchronous and asynchronous messages, use the different types of arrows in the UML sequence diagram notation to differentiate them and add the return messages for the synchronous calls.

Sequence diagrams are not explicit about showing concurrency. If that is your goal, use activity diagrams instead. Although instances in a sequence diagram can be running concurrently, no assumptions can be made about ordering when a sequence diagram depicts an instance sending messages at the "same time" to different instances or, conversely, receiving multiple stimuli at the "same time."

Figure 8.5 shows a more interesting sequence diagram. It demonstrates some features introduced in UML 2.0 that help in communicating a design to developers:

- *Named frame.* The optional frame around the diagram contains the name of the sequence diagram, which in the example is `sdProcessOrder`.

Figure 8.4
A simple example of a UML sequence diagram. Objects (that is, element instances) have a lifeline, drawn as a vertical dashed line along the time axis. The sequence is usually started by an actor on the far left. The instances interact by sending messages, which are shown as horizontal arrows. A message can be a method or function call, an event sent through a queue, or something else. The message usually maps to a resource (operation) in the interface of the receiver instance. A filled arrowhead on a solid line represents a synchronous message, whereas the open arrowhead represents an asynchronous message. The dashed arrow is a return message. The execution occurrence bars along the lifeline indicate that the instance is processing or blocked waiting for a return. Because this sequence diagram explicitly shows the creation of the `UserSession` object, its box is inserted at the point where the creation takes place.



- *Reference.* An existing sequence diagram can be referenced in other diagrams by using the frame with the upper-left label “ref”. The example indicates that all interactions in the referenced sequence diagram, `CreditCardValidation`, take place right after `ProcessOrderRPCService` interacts with `DaoOrder`.
- *Time constraint.* The example specifies that the interaction of `Customer` and `GWTClientApp` should take between one and five seconds.
- *Loop.* The loop frame indicates that the interactions are repeated in a loop. The expression after keyword “loop” in the upper-left label defines the number of iterations. In the example, the interactions in the loop frame are repeated for each order item (it’s not shown in this diagram, but each order contains a collection of order items).
- *Alternatives.* The frame with upper-left label “alt” contains interactions that are executed only if the specified guard condition (in square brackets) is true. The alt frame can be segmented, and each segment can have a guard condition. In the example, the alt frame has two segments, which semantically correspond to an *if-then-else* construct.

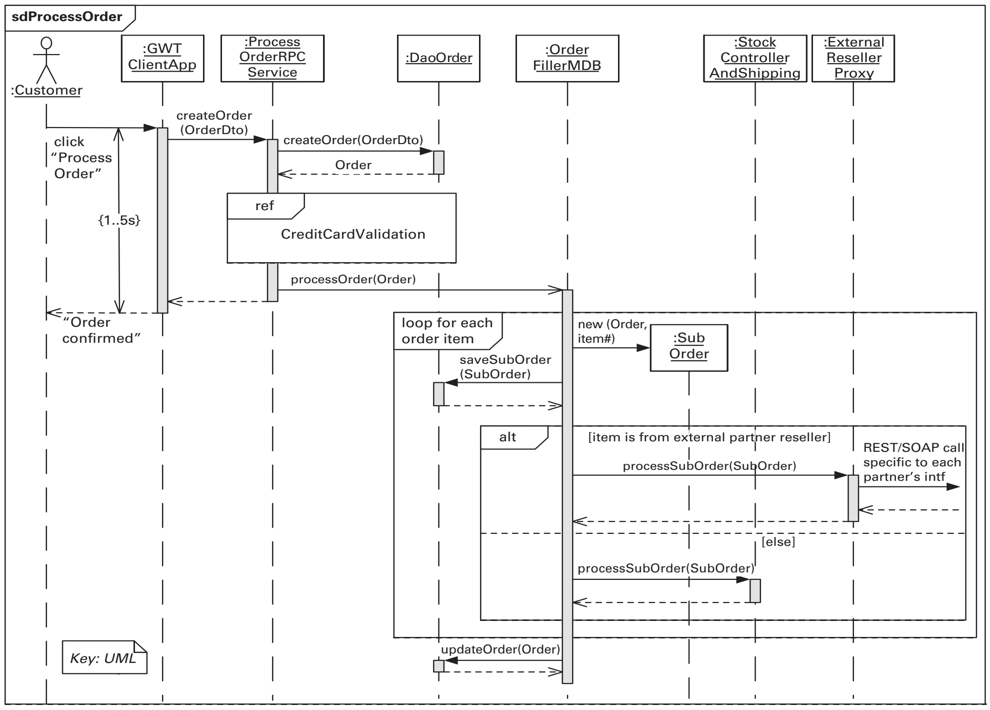


Figure 8.5
Example of a UML sequence diagram for processing a purchase order on a Web store. This example exhibits some powerful features of the sequence diagram notation, such as time constraint, reference to another sequence diagram, loop, and conditional alternative messages.

Communication Diagrams

Like other trace notations, a UML communication diagram shows ordered interactions among elements needed to accomplish a purpose. Whereas a sequence diagram shows order using a time-line-like mechanism, a communication diagram shows a graph of interacting elements and annotates each interaction with a number denoting order. As in sequence diagrams, instances shown in a communication diagram are instances of elements described in the accompanying structural documentation. Communication diagrams are useful when the task is to verify that an architecture can fulfill the functional requirements. The diagrams are not useful if the understanding of concurrent actions is important, as when conducting a performance analysis.

A communication diagram also shows relations among the elements, called links (see Figure 8.6). Links show important aspects of relations among those structural instances. Links between the same instances in different communication dia-

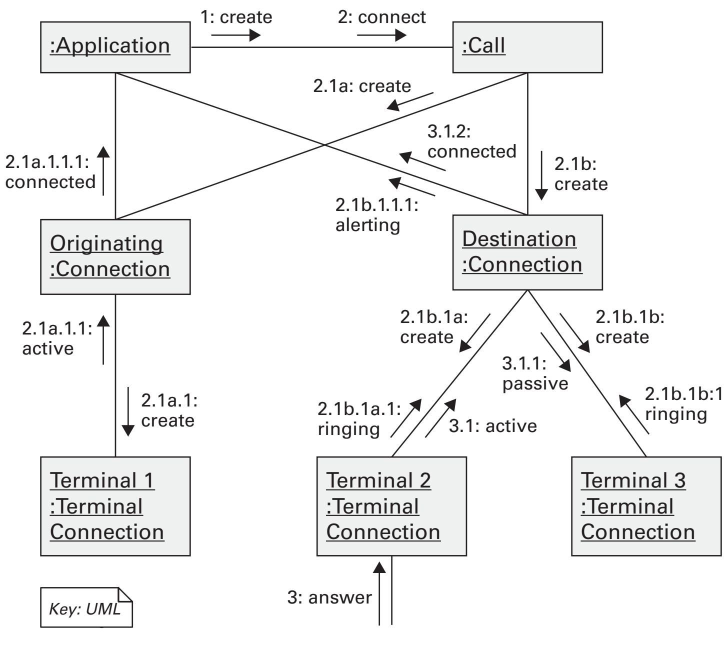


Figure 8.6
A UML communication diagram for placing a three-way call on a telephone system. Interactions are shown by lines between the instances labeled with a sequence number, the name of the resource being called, and an arrowhead that indicates the communication direction. The sequence numbers show which interactions follow which. Subnumbering can be used to show nested stimuli and/or parallelism. For example, the interaction with a sequence number 2.1a is the first stimulus sent as a result of receiving stimulus number 2. The letter *a* at the end means that another stimulus, 2.1b, can be performed in parallel. This numbering scheme may be useful for showing sequences and parallelism, but it tends to make a diagram unreadable.

grams can show different aspects of relations between the same structural elements.

Communication diagrams and sequence diagrams essentially express the same information, though you may choose one or the other based on how they highlight time sequences and element relations. Sequence diagrams show time sequences explicitly, making it easy to see the order in which interactions occur; communication diagrams indicate ordering by using numbers. Communication diagrams can resemble the structural diagram (such as a class diagram) they derive from and also make it easy to see how elements are statically connected; sequence diagrams do not clearly show static connections (such as use dependencies) between elements.

Activity Diagrams

UML activity diagrams are similar to flow charts. They show a business process as a sequence of steps (called actions) and include notation to express conditional branching and concurrency, as well as to show sending and receiving events. Arrows between actions indicate the flow of control. Optionally, activity diagrams can indicate the architecture element or actor performing the actions. One way to do that is by drawing an activity partition (also called a swim lane) for each element and placing the actions performed by that element within the corresponding partition. Figure 8.7 is an example of an activity diagram that has six activity partitions.

Another important feature of activity diagrams is the ability to express concurrency. A fork node (depicted as a thick bar orthogonal to the flow arrows) splits the flow into two or more concurrent flows of actions. The concurrent flows may later be synchronized into a single flow through a join node (also depicted as an orthogonal bar). The join node waits for all incoming flows to complete before proceeding. An alternative is to use a merge node, which is depicted as a diamond with

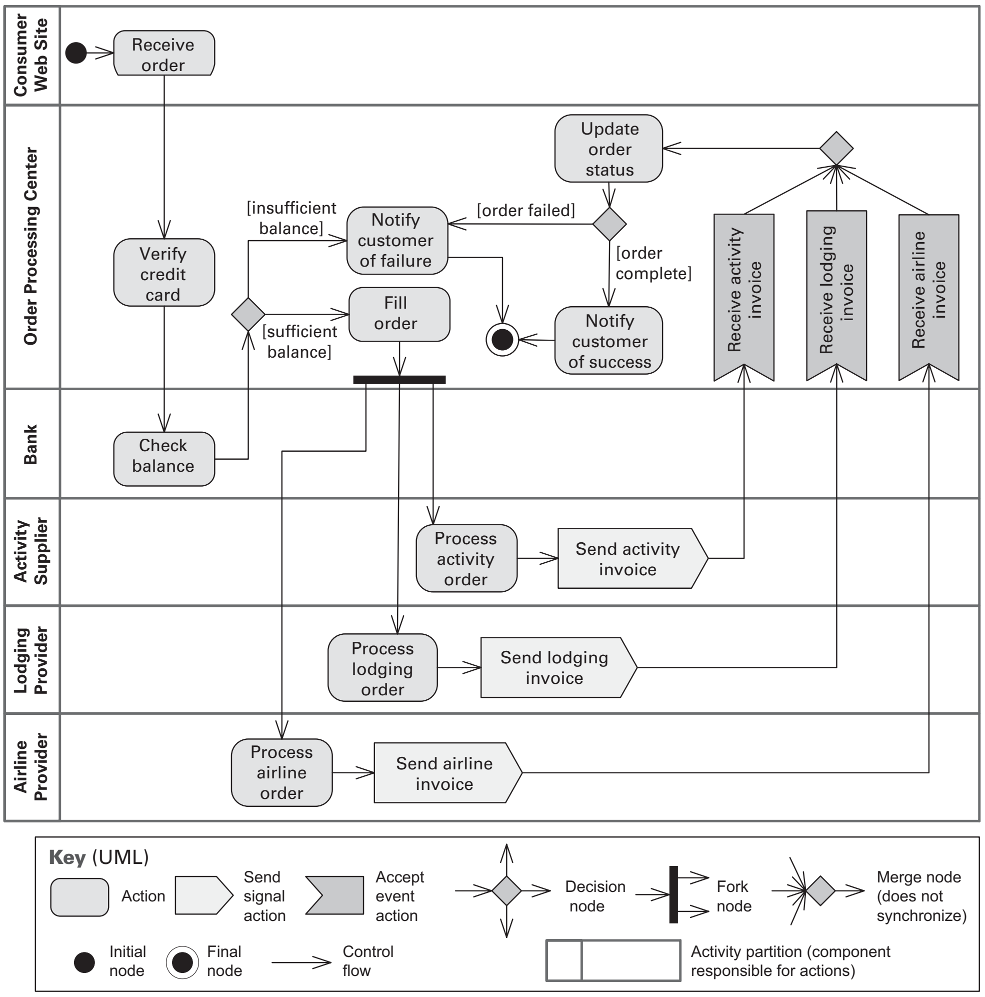


Figure 8.7
Example of an activity diagram for processing an order in the Adventure Builder system (Adventure Builder 2010). The elements listed on the left are components from the top-level service-oriented view of the system.

multiple incoming flows. The merge node does not synchronize the incoming flows; instead, as each flow completes, control is passed to the following action.

Unlike sequence and communication diagrams, activity diagrams don't show the actual operations being performed on specific objects. Activity diagrams are very useful to broadly describe the steps in a specific work flow. Conditional branching (shown with the diamond symbol) allows a single diagram to represent multiple traces, although it's not usually the intent of an activity diagram to show all possible traces or the complete behavior for the system or part of it.

Other Trace-Based Notations

Use cases, sequence diagrams, communication diagrams, and activity diagrams are perhaps the most commonly seen notations for capturing traces, but there are other notations that have specialized purposes:

- A message sequence chart is a message-oriented representation containing the description of the communication between instances. Simple message sequence charts look like sequence diagrams but have a more specific definition and are a more precise notation. The main area of application for a message sequence chart is as an overview specification of the communication behavior among interacting systems, especially telecommunication switching systems. Message sequence charts are often seen in conjunction with Specification and Description Language (SDL), which is a comprehensive behavioral notation. Whereas a message sequence chart focuses on representing the message exchange between elements, such as systems and processes, SDL focuses on documenting what does or should happen in an element. In that respect, message sequence charts and SDL diagrams complement each other.
- UML timing diagrams show state changes of one or more objects along a time line. When two or more objects are shown, the timing diagram can also display the messages exchanged between them, similar to a UML sequence diagram. Timing diagrams resemble digital signal diagrams in that time progresses from left to right, but they have a rich vocabulary of annotations to express timing constraints between the events and message exchanges.
- Business Process Execution Language (BPEL) is a language that supports creation of work flows that consist of interactions among Web services. BPEL is an executable XML-based language and hence not suitable for architecture

design. However, BPEL tool environments usually provide a graphical notation (for example, Business Process Modeling Notation, or BPMN) that can be used to create behavioral diagrams that describe control and data flow through the system.

8.3.2 Notations for Capturing Comprehensive Models

Comprehensive models show the complete behavior of structural elements. Given this type of documentation, it is possible to infer all possible paths from initial state to final state. The state machine formalism is a good candidate for representing the behavior of architecture elements because each state is an abstraction of all possible histories that could lead to that state. State machine languages allow you to complement a structural description of the elements of the system with constraints on interactions and timed reactions to both internal and environmental stimuli.

In this section, we describe UML state machine diagrams. Although other languages are available, we have chosen state machine diagrams because they can describe behavior in a form that captures the essence of what you wish to convey to system stakeholders. State machines are also used in many disciplines of computer science (from compilers to data modeling) and are part of UML, so you are likely to find them in modeling and drawing tools. State machine diagrams are also available in development tools that allow you to design, simulate, and analyze your system, and sometimes generate code.

UML State Machine Diagrams

UML state machine diagram notation is based on the statechart graphical formalism developed by David Harel for modeling reactive systems; they allow you to trace the behavior of your system, given specific inputs. A UML state machine diagram shows states represented as boxes and transitions between states represented as arrows. The state machine diagrams help to model elements of the architecture that have interesting or complex states. Figure 8.8 is a simple example showing the states of a vehicle cruise control system.

Each transition in a state machine diagram is labeled with the event causing the transition. Optionally, the transition can specify a guard condition, which is bracketed. When the event corresponding to the transition occurs, the guard condition is evaluated and the transition is enabled only if the guard is true at that time. Transitions can also have consequences, called actions or effects, indicated by a slash. When an action is noted, it indicates that the behavior following the slash will be

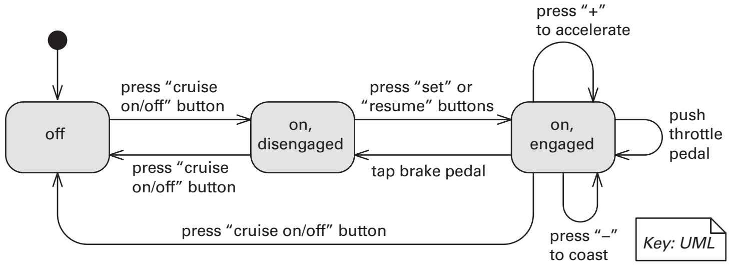

off
on, disengaged
on, engaged
press "cruise on/off" button
press "cruise on/off" button
press "set" or "resume" buttons
tap brake pedal
press "+" to accelerate
push throttle pedal
press "–" to coast
press "cruise on/off" button
Key: UML

Figure 8.8
UML state machine diagram for the cruise control system of a motor vehicle. The transitions correspond to the buttons the driver can press or driving actions that affect the cruise control system.

performed when the transition occurs. The states may also specify entry and exit actions.

UML state machine diagrams also support the nesting of states. The outer state is called the composite state; inner states are called substates. The composite state defines the scope of a new state diagram, and the substates are related by transitions, just as in a finite state machine. When the composite state is entered, the initial state within the composite state is also entered. Grouping substates into a composite state allows common behavior to be expressed concisely. Any behavior indicated at the composite state level—depicted as transitions from the composite state boundary rather than from any specific substate—applies to all substates. A good use of this technique is to indicate common error handling or termination behavior. Figure 8.9 shows an example of this; the "user canceled order" transition out of the "filling order" composite state (top right) expresses that the transition can occur in any of the substates.

In UML state machines, concurrency is represented by dividing a composite state into regions. Each concurrent region contains a state machine that is a grouping of substates. Regions are shown separated by dotted lines. The state machine in Figure 8.9 shows three concurrent regions. When all regions reach their final state, transition "all bookings confirmed" from the composite state is triggered. This transition causes the "e-mail customer" action to be executed. Alternatively, the system can leave the composite state if any of the requests is not satisfied or, as we mentioned before, if the user cancels the order. UML's state machine diagram notation contains many other features not mentioned here, such as means of expressing choice, timing, and history.

The boxes in a state machine diagram are states; they are not components or modules. The arrows are transitions; they are not connectors. A state machine diagram may model the states of the entire system, a component, a collection of components, or an attribute of an object. Be clear as to what you are modeling before creating a state machine diagram.

State diagrams by definition are supposed to show all states and all transitions out of a state. For example, when an ATM is in state "Enter Pin", you should show a transition for the case when the user walks away. Otherwise, the developer may not implement a time-out.

Decide what states your state diagram represents. Choosing a scope that is too broad may result in a diagram that is too big to understand and analyze.

Don't forget to indicate the initial state, and decide whether or not there is a final state.

Other Comprehensive Notations

Formal languages such as Z, CSP, and FSP are popular in niche domains with demanding requirements, such as safety-critical systems. They are mathematical languages based on predicate

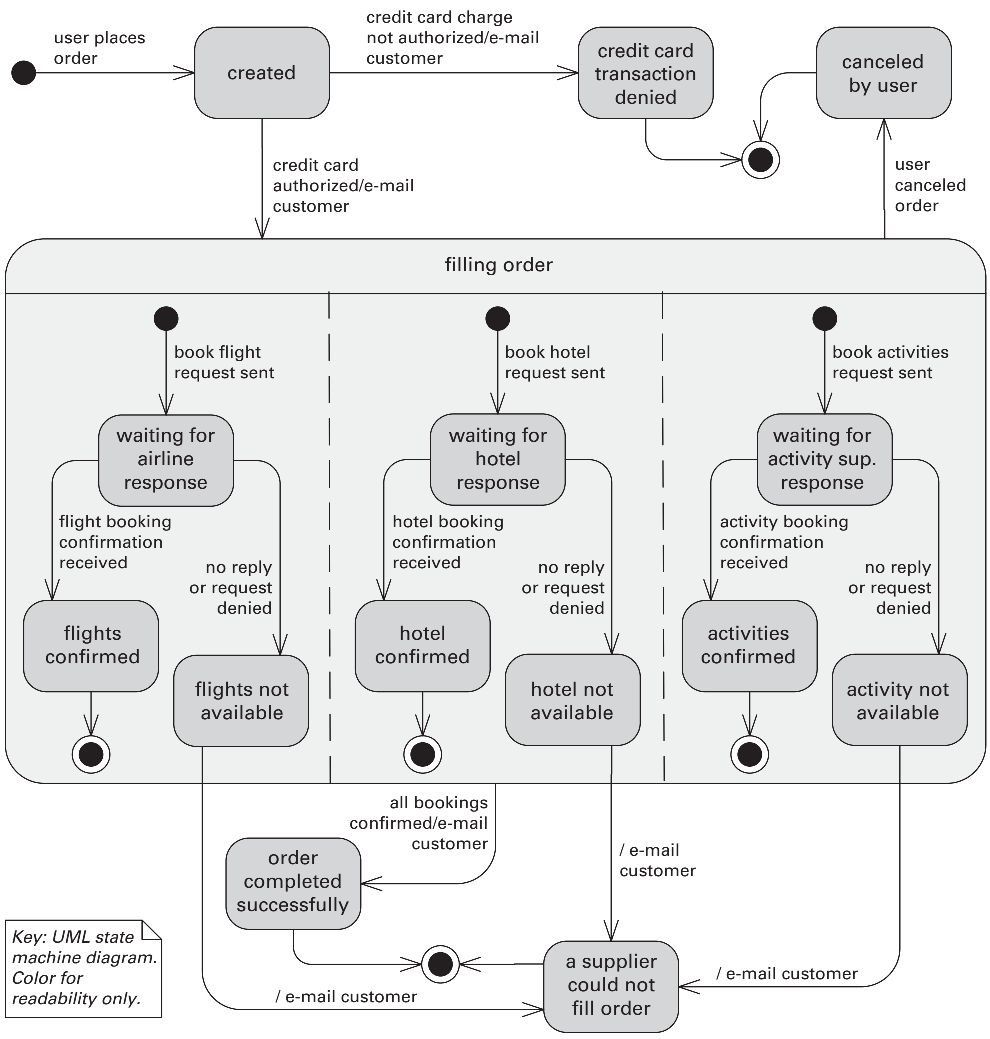

user places order
created
credit card charge not authorized/e-mail customer
credit card transaction denied
canceled by user
credit card authorized/e-mail customer
user canceled order
filling order
book flight request sent
waiting for airline response
flight booking confirmation received
no reply or request denied
flights confirmed
flights not available
book hotel request sent
waiting for hotel response
hotel booking confirmation received
no reply or request denied
hotel confirmed
hotel not available
book activities request sent
waiting for activity sup. response
activity booking confirmation received
no reply or request denied
activities confirmed
activity not available
all bookings confirmed/e-mail customer
order completed successfully
/ e-mail customer
a supplier could not fill order
/ e-mail customer
/ e-mail customer
Key: UML state machine diagram. Color for readability only.

Figure 8.9
A UML state machine diagram showing the states of an order in the Adventure Builder system (Adventure Builder 2010). Once the credit card is authorized, the order moves to the "filling order" composite state. The three regions represent concurrent, independent substates of filling the order.

logic and set theory. These languages can be used to produce precise behavioral models and permit rigorous analyses, such as type checking, model checking, and proofs. However, they include a large set of symbols, and expressions are written in terms of predicate logic, making it difficult for some designers to warm up to.

Other notations exist that are used in various niche areas. Architecture Analysis and Design Language (AADL), which is described in Appendix C, can be used to reason about runtime behavior. Specification and Description Language (SDL) is used in telephony. Koala is an architecture description language designed with product-line architectures in mind; it provides support for variability in component selection and variety of composition binding times.

8.4 Where to Document Behavior

Architects document behavior to show how an element behaves when stimulated in a particular way or to show how an ensemble of elements react to one another. In an architecture documentation package, where behavior is shown depends on what is being shown. For example, in an architecture view:

- *Behavior has its own section in the element catalog.* For a complex transaction, you might describe how elements interact to process requests using a sequence diagram.
- *Behavior can be part of an element's interface documentation.* The semantics of a resource on an element's interface can include the element's externally visible behavior that occurs as a result of using the resource. Or in the usage guide section of an interface document, behavior descriptions can be used to explain the effects of a particular usage pattern, that is, a particular sequence of resources used.
- *Behavior can be used to fill in the rationale section, which includes results of analysis.* Behavior descriptions are often a basis for analysis, and the behaviors that were used to analyze the system for correctness or other quality attributes can be recorded here.

Documenting interfaces is described in Chapter 7.

Documenting rationale is described in Section 6.5.

In the documentation that applies beyond views, the rationale for why the architecture satisfies its requirements can include behavior documentation as part of the architect's justification.

8.5 Why to Document Behavior

Documentation of behavior is most commonly used for communication among stakeholders during development and maintenance activities. It can also be used for system analysis. The types of analyses you perform and the extent to which you check the quality attributes of your system are based on the type of system you are developing.

8.5.1 Driving Development Activities

Behavior documentation plays an important part in architecture's role as a vehicle for communication among stakeholders during system development activities. It's probably safe to say that every architect has drawn a sequence diagram (or some similarly expressive diagram) on the whiteboard during a meeting in order to make concrete their ideas about what components need to exist and the interactions among those components. These diagrams, along with associated rationale, should be captured as part of the architecture's documentation. The process of designing the architecture helps the architect develop an understanding of the internal behavior of system elements and gross system structure, and it improves confidence that the system will be able to achieve its goals.

In the architecting process, system decomposition identifies sets of subelements and defines both the structure and the interactions among the subelements in a way that supports the required behavior of the parent element. In many cases behavior documentation is created to help reason about the interaction of the subelements and their responsibilities, to see if the decomposition is appropriate.

Trace-oriented diagrams, such as sequence diagrams, can be created based on an implementation already in place. In that case, the diagrams can help to spot bottlenecks, memory leaks, and other defects, as well as identify opportunities for performance improvements and refactorings.

If you're spending a long time to find a bug in a specific transaction involving several modules, create a sequence diagram showing the relevant steps. The diagram helps you and others to get a hold of the overall transaction and may expose erroneous interactions. This advice is particularly useful when the bug is not easily reproducible because of concurrency issues.

8.5.2 Analysis

Behavior documentation allows you to reason about the completeness, correctness, and quality attributes of the software system. Once the structure of an architecture view has been identified and the interactions among elements have been constrained, you need to look at whether the proposed system will be capable of doing its job as planned. This is your opportunity to reason about both the completeness and the correctness of the architecture. The behavior of the system can be simulated to help reason about the architecture's ability to support the range of functionality and related quality requirements of the system. Behavior documentation can be built as a model that serves as input to the simulation.

The amount of information in the behavioral models required to perform behavioral analysis varies greatly, depending on the level of certainty and precision required of the result. Therefore, it is generally a good idea to do some type of trade-off comparison to determine the cost/benefit involved with applying certain types of architecture analysis techniques. For any system, it is a good idea to identify and to simulate a set of requirements-based scenarios. If you are developing a safety-critical system, the application of more-expensive, formal analysis techniques, such as model checking, is justified in order to identify possible design flaws that could lead to safety-related failures.

Documenting system behavior supports exploration of the quality attributes of a system early in the development process. Some techniques and tools are available or are being developed that can be used to predict, based on the architecture, that the production of a system will exhibit specific measures related to such quality attributes as performance, reliability, and safety.

Architecture-based simulation is similar to testing an implementation in that a simulation is based on a specific use of the system under specific conditions and with expectation of a certain outcome. Typically, a developer identifies a set of scenarios based on the system requirements. These scenarios are similar to test cases in that they identify the stimulus of an activity and the assumptions about the environment in which the system is running and describe the expected result. These scenarios are played out against documented system models that support relating system elements and the constraints on their interactions. The results of "running the architecture" are checked against expected behavior.

Whereas simulation looks at a set of special cases, system-wide techniques for analyzing the architecture evaluate the overall system: analysis techniques for things like change impact, deadlock, safety, and schedulability. These techniques require information about the behavior of the system and its constituent elements in order to perform the appropriate analyses. Dependencies between and within elements can be used to identify potential execution paths, which are needed to evaluate quality attributes such as performance, and to identify chains of *uses* relations to help evaluate modifiability.

8.6 Summary Checklist

- Documenting behavior adds semantic detail to elements and their interactions that have time-related characteristics. Behavioral models complement structural models by adding information that reveals ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions, such as at a specific time or after a period of time.
- Constraints on the interaction between elements should be documented. Document any ordering constraints on actions or interactions. Document a clock if your system depends on time.

- Most behavioral languages include representations of stimulus and activity, ordering of interactions, and structural elements.
- Trace-oriented models consist of sequences of activities or interactions that describe the system's response to a specific stimulus when in a specific state. They document the trace of activities through a system described in terms of its structural elements and their interactions. Use cases, sequence diagrams, communication diagrams, and activity diagrams are trace-oriented modeling languages.
- Comprehensive models, often state based, show the complete behavior of a structural element or set of elements. UML state machine diagrams are a comprehensive behavior modeling language.
- Behavior can be documented in the element catalog of a view and in interface documentation, and it can be used to fill in the design background section, which includes results of analyses.

8.7 Discussion Questions

1. Consider a car radio with seek, scan, power on/off, and preset station buttons, along with a manual tuning knob and volume control and a digital frequency display. (a) Of the languages and notations for describing behavior presented in this chapter, which ones would be good candidates for describing the behavior of this radio? Why? (b) Using one of the languages you chose in the previous question, sketch the behavior of the car radio.

2. Suppose that you wanted to make sure that your car radio did not exhibit undesirable behavior in unusual circumstances, such as the display going blank when the driver turns the frequency knob while holding down a preset button. What languages would you likely use to help in that case, and why?

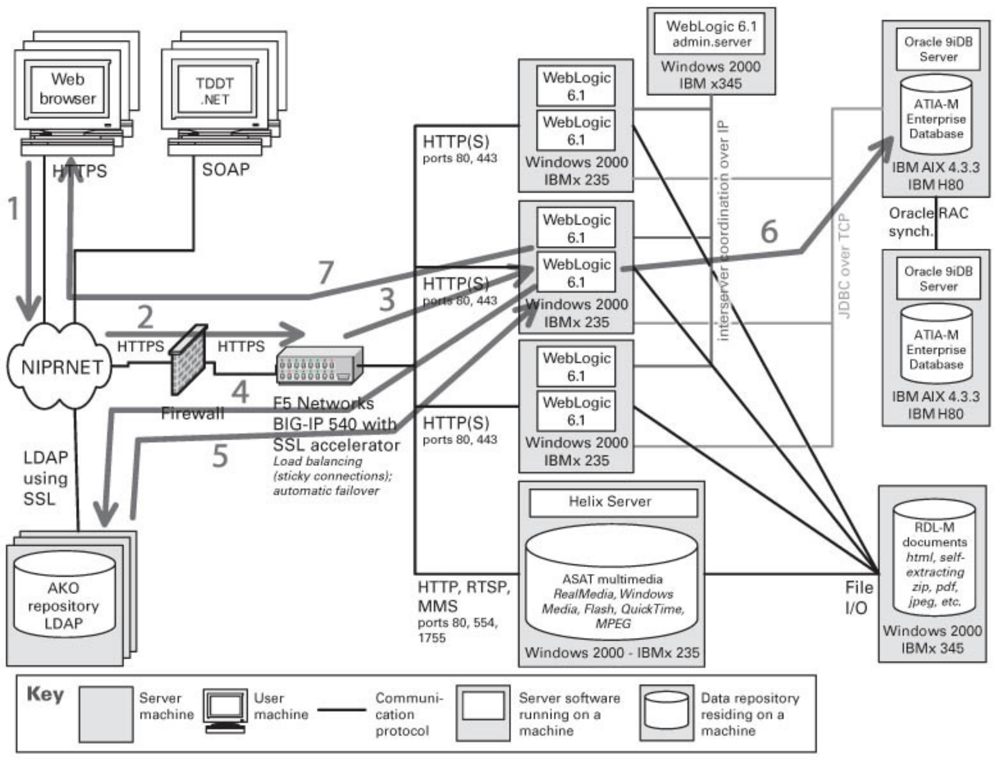
3. When would you choose to document behavior using trace models or using comprehensive models? What value do you get, and what effort is required, for each of them?

4. Draw a statechart for an automatic teller machine for each of the following stakeholders: (a) a customer wanting to deposit or withdraw money from an account; (b) a bank executive wanting to track machine usage across the bank's territory; (c) a service technician sent to repair the machine when it becomes inoperative for any reason; (d) a

security monitor whose job it is to take appropriate action when the machine's money safe is open, the tilt alarm goes off, or the machine stops communicating with the bank. Discuss the differences among the four. If you wanted to combine them to create a single overall statechart for the machine, how would you go about it?

5. Documenting an architecture involves drawing various views that must be consistent with each other. If a behavior model shows a component sending a message to another, there must be a connector between them, but connectors are not shown on behavior diagrams. What other consistency checks between views can you think of? In answering, consider views in each category: module, C&C, and allocation.

6. Is Figure 8.10 a behavioral diagram? Why or why not?



Web browser
TDDT .NET
HTTPS
SOAP
1
2
3
4
5
6
7
NIPRNET
HTTPS
HTTPS
Firewall
F5 Networks BIG-IP 540 with SSL accelerator
Load balancing (sticky connections); automatic failover
LDAP using SSL
AKO repository LDAP
HTTP(S) ports 80, 443
HTTP(S) ports 80, 443
HTTP(S) ports 80, 443
HTTP, RTSP, MMS ports 80, 554, 1755
WebLogic 6.1 admin.server
Windows 2000 IBM x345
WebLogic 6.1
WebLogic 6.1
Windows 2000 IBMx 235
Helix Server
ASAT multimedia RealMedia, Windows Media, Flash, QuickTime, MPEG
Windows 2000 - IBMx 235
interserver coordination over IP
JDBC over TCP
Oracle 9iDB Server
ATIA-M Enterprise Database
IBM AIX 4.3.3 IBM H80
Oracle RAC synch.
File I/O
RDL-M documents html, self-extracting zip, pdf, jpeg, etc.
Windows 2000 IBMx 345
Key
Server machine
User machine
Communication protocol
Server software running on a machine
Data repository residing on a machine

Figure 8.10
The numbered arrows drawn on top of this network diagram show the sequence of steps to process a specific user transaction in terms of communicating elements.

8.8 For Further Reading

A rich source of behavior descriptions can be found in the UML definition that is publicly available from the OMG. At uml.org, you can find the UML specifications, which contain definitions, descriptions, and examples of sequence and communication diagrams, as well as example use cases and state machine diagrams. You can also find several books that explain UML and its usage in detail. Two seminal books that you will find to be valuable references are *The Unified Modeling Language User Guide*, by Booch, Rumbaugh, and Jacobson (2005) and *UML Distilled*, by Martin Fowler (2003), which focuses on the 20 percent of UML that you will use 80 percent of the time.

A good reference for statecharts is *Modeling Reactive Systems with Statecharts: The Statemate Approach*, by Harel and Politi (1998).

The BPEL specification can be found at oasis-open.org/committees/wsbpel. The Object Management Group's home page for BPMN is bpmn.org.

Message sequence charts, especially combined with SDL diagrams, are most commonly used by the telecommunication industry. The Web site of the International Telecommunication Union, at itu.int, has references to resources needed to understand and use message sequence charts and SDL. Additional information and pointers to events, tools, and papers can be found at the SDL Forum Society's Web site, sdl-forum.org.

Many books have been written about use cases. The book from Ivar Jacobson that started the whole use case discussion is *Object-Oriented Software Engineering: A Use Case Driven Approach* (1992). This book can serve as a starting point to understand what was originally meant by use cases and their underlying concepts. Alistair Cockburn's book *Writing Effective Use Cases* (2000) provides practical guidance on avoiding pitfalls, structuring collections of use cases, and organizing use cases into goal levels.

The Z language was originally developed at Oxford University in the late 1970s and has been extended by a number of groups since then. Tools that help create and analyze specifications have been developed by various groups and are available freely over the Internet. A great resource for information and pointers is the Web archive found at formalmethods.wikia.com/wiki/Z_archive. J. M. Spivey's book *The Z Notation: A Reference Manual* (1988) is available online at spivey.oriel.ox.ac.uk/mike/zrm. It provides a good reference in terms of a standard set of features.

AADL is a standard published by the Society of Automotive Engineers (SAE). The SAE AADL team keeps an updated Web site at aadl.info. An overview of AADL and its associated tools is found in the technical note by Feiler, Gluch, and Hudak (2006), as well as in Appendix C of this book.

PART III

Building the Architecture Documentation

Parts I and II covered the kind of information that should appear in architecture documentation. Part I covered styles, with their attendant element and relation types, that architects can use to engineer views. Part II covered other critical information beyond elements and relations that should be documented.

Part III deals more directly with the care and feeding of the architecture documentation itself. Exactly how does an architect decide what views to put into an architecture document? How should the architecture document be organized, laid out, divided into sections, and packaged? How should it be reviewed for quality and fitness for stakeholder use?

These and other topics are the subject of Part III.

- Chapter 9 provides detailed guidance for choosing the set of views to incorporate into a documentation suite, explores examples of sets of views, and gives two short examples for illustrating how to decide which views to use.
- Chapter 10 prescribes templates and detailed guidance for documenting views and documenting information that applies to more than one view.
- Chapter 11 presents a step-by-step approach for reviewing an architecture document. The approach is focused on involving the appropriate stakeholders and asking questions directed at making sure the document satisfies their specific needs and concerns.

This page intentionally left blank

9 Choosing the Views

As we have seen, a large part of designing the architecture for a system consists of choosing and designing software structures, often as described in terms of architecture styles. Choosing, for example, a service-oriented style for your system means putting a service-oriented structure in place and populating it with services and their interconnections. To the extent that you write down that structure, and the interfaces and behavior of the elements, you've created a view of your architecture, because a view is a representation of a structure.

Poetry is a condensation of thought. You write in a few lines a very complicated thought. And when you do this, it becomes very beautiful poetry. It becomes powerful poetry.

—Chen Ning Yang, winner of the Nobel Prize in Physics, 1957 (quoted in Moyers 1989, p. 313)

In other words, documenting your design decisions as you make them (something we strongly recommend) produces views, which are the heart of an architecture document. It is most likely that these views are sketches more than finished products ready for public release; this will give you the freedom to back up and rethink design decisions that turn out to be problematic without having wasted time on cosmetic polish. (In some cases, they might *literally* be sketches—see Figure 11.8 for an example.)

By the time you're ready to release an architecture document, then, you're likely to have a fairly well worked-out collection of architecture views. At some point you'll need to decide which to take to completion, with how much detail, and which to include in a release. You'll also need to decide which views can be usefully combined with others, so as to reduce the total number of views in the document and reveal important relations among the views.

Combined views can be produced by defining a hybrid style, or by making an overlay. These are discussed in Section 6.6.

And that is the topic of this chapter: how an architect decides on the views to include in the documentation package.

We have tried to explain the benefits of each kind of documentation, to help you decide under what circumstances you would want to produce it. Understanding which views to produce at what time and with how much detail can be reached only in the concrete context of a project. You can determine which views are required, when to create them, and how much detail to include in order to make the development project successful if you know the following:

If you can't afford to produce a particular part of the architecture documentation package, at least make sure you understand what the long-term cost will be for the short-term savings. Use the formula in Section P.2.4 in the prologue to help you estimate the cost and benefit.

- What people, and with what skills, are available
- With which standards you have to comply
- What budget is on hand
- What the schedule is
- What the information needs of the important stakeholders are

This chapter is about helping you make those determinations. Once the entire documentation package has been assembled, or at opportune milestones along the way, it should be reviewed for quality, suitability, and fitness for purpose by those who are going to use it.

Chapter 11 covers the review of architecture documents by stakeholders.

9.1 Stakeholders and Their Documentation Needs

All fine architectural values are human values, else not valuable.

—Frank Lloyd Wright

To choose the appropriate set of views, you must identify the stakeholders that depend on software architecture documentation. You must also understand each stakeholder's information needs.

The set of stakeholders will vary, depending on the organization and the project. The list of stakeholders in this section is suggestive but is not intended to be complete. As an architect, one of your primary obligations is to understand who the stakeholders are for your project. Similarly, the documentation needs we lay out for each stakeholder are typical, but not definitive. So take the following discussion as a starting point and adapt them according to the needs of your project and your stakeholders.

Project managers

Project managers care about schedule, resource assignments, and perhaps contingency plans to release a subset of the system for business reasons. To create a schedule, the project manager needs information about the modules to be implemented, with some information about their complexity, such as the list of responsibilities, as well as dependencies that exist to other modules, which may suggest a certain sequence in the implementation. This person is not interested in the design specifics of any element or the exact interface beyond knowing whether those tasks have been completed. But the manager is interested in the system's overall purpose and constraints; its interaction with other systems, which may suggest an organization-to-organization interface that the manager will have to establish; and the hardware environment, which the manager may have to procure. The project manager might create or help create the work assignment view, in which case he or she will need a decomposition view to do it.

As shown in Figure 9.1, a project manager, then, will likely be interested in

- Module views: decomposition and uses and/or layered
- Allocation views: deployment and work assignment
- Other: top-level context diagrams showing interacting systems and system overview and purpose

Members of the development team

Members of the development team, for whom the architecture provides marching orders, are given constraints on how they do their job. Sometimes a developer is given responsibility for an element he or she did not implement, such as a commercial off-the-shelf product. Someone still has to be responsible for that element, to make sure that it performs as advertised and to tailor it as necessary. This person will want to know the following:

- The general idea behind the system. Although that information lies in the realm of requirements rather than architecture, a top-level context diagram or system overview can go a long way to provide the information.
- Which element the developer has been assigned, that is, where functionality should be implemented.
- The details of the assigned element, including the data model with which it must operate.
- The elements with which the assigned part interfaces and what those interfaces are.
- The code assets the developer can make use of.

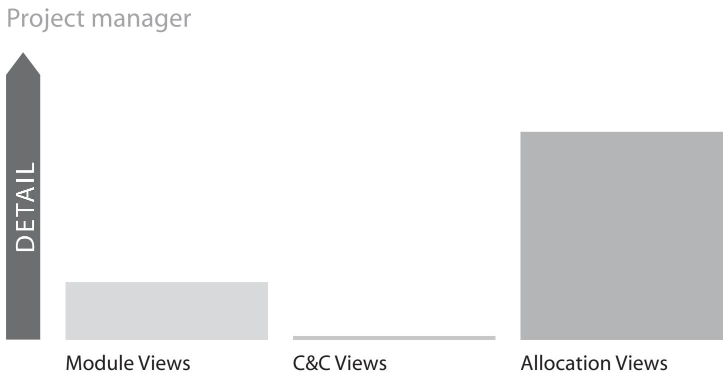


Figure 9.1
A project manager usually creates the work assignments and therefore needs some overview information of the software.

- The constraints, such as quality attributes, legacy systems interfaces, and budget, that must be met.

As shown in Figure 9.2, a developer, then, is likely to want to see

- Module views: decomposition, uses and/or layered, and generalization
- Component-and-connector (C&C) views: various, showing the component(s) the developer was assigned and the components they interact with
- Allocation views: deployment, implementation, and install
- Other: system overview; a context diagram containing the module(s) he or she has been assigned; the interface documentation of the developer's element(s) and the interface documentation of those elements with which they interact; a variability guide to implement required variability; and rationale and constraints

Testers and integrators

Testers and integrators are stakeholders for whom the architecture specifies the correct black-box behavior of the pieces that must fit together. A unit tester of an element will want to see the same information as a developer of that element, with an emphasis on behavior specifications. A black-box tester will need to see the interface documentation for the element. Integrators and system testers need to see collections of interfaces, behavior specifications, and a uses view so they can work with incremental subsets.

As shown in Figure 9.3, testers and integrators, then, are likely to want to see

- Module views: decomposition, uses, and data model
- C&C views: all

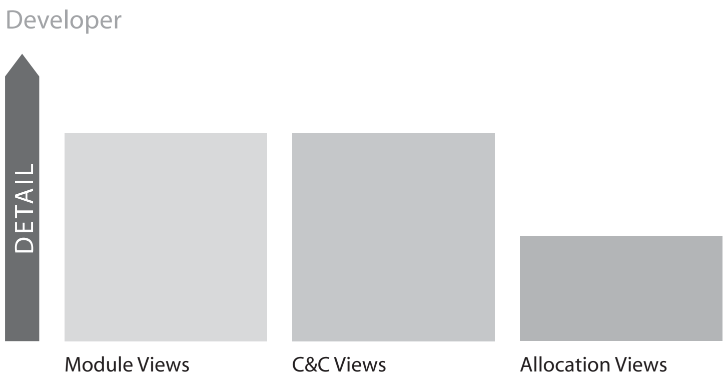


Figure 9.2
Developers have interest mainly in the software itself and therefore create detailed module and C&C views and have some interest in allocation views.

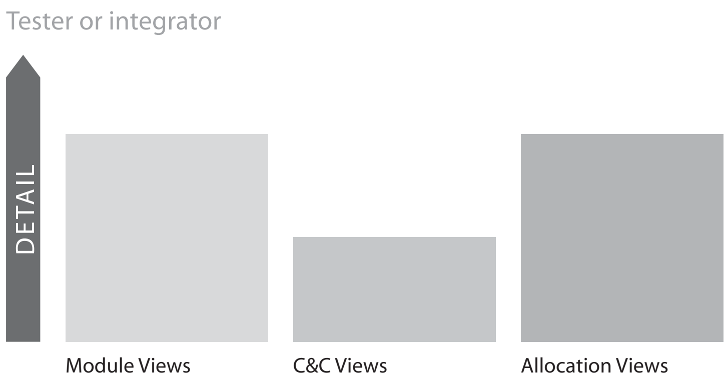


Figure 9.3
Testers and integrators need context and interface information, along with information about where the software runs and how to build incremental parts.

- Allocation views: deployment; install; and implementation, to find out where the assets to build the module are
- Other: context diagrams showing the module(s) to be tested or integrated; the interface documentation and behavior specification(s) of the module(s) and the interface documentation of those elements with which they interact

Designers of other systems

Designers of other systems with which this one must interoperate are stakeholders. For these people, the architecture defines the set of operations provided and required, as well as the protocols for their operation. As shown in Figure 9.4, these stakeholders will likely want to see

- Interface documentations for those elements with which their system will interact, as found in module and/or C&C views
- The data model for the system with which their system will interact

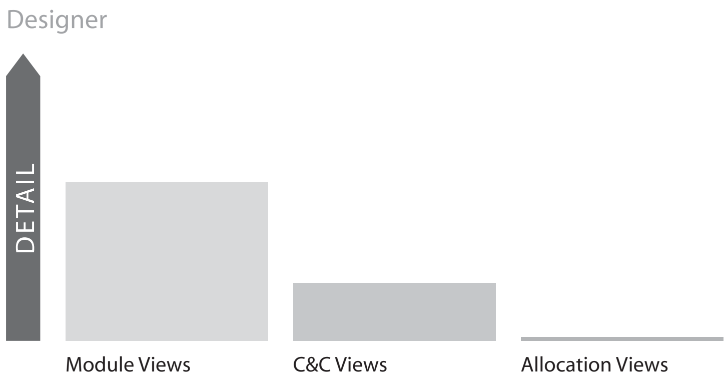


Figure 9.4
Designers of other systems are interested in interface documentation and important system behavior.

- Top-level context diagrams from various views showing the interaction

Maintainers

Maintainers use architecture as a starting point for maintenance activities, revealing the areas a prospective change will affect. Maintainers will want to see the same information as developers, for they both must make their changes within the same constraints. But maintainers will also want to see a decomposition view that allows them to pinpoint the locations where a change will need to be carried out, and perhaps a uses view to help build an impact analysis to fully scope out the effects of the change. Maintainers will also want to see design rationale that will give them the benefit of the architect's original thinking and save them time by letting them see already discarded design alternatives.

As shown in Figure 9.5, a maintainer, then, is likely to want to see the views as mentioned for the developers of a system, with special emphasis on

- Module views: decomposition, layered, and data model
- C&C views: all
- Allocation views: deployment, implementation, and install
- Other: rationale and constraints

Application builders

Application builders in a **software product line** tailor the core assets according to preplanned and built-in variability mechanisms, add whatever special-purpose code is necessary, and instantiate new members of the product line. Application builders will need to see the variability guides for the various elements, to facilitate tailoring. After that, application builders need to see largely the same information as integrators do.

A **software product line** is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way. (Clements and Northrop 2001)

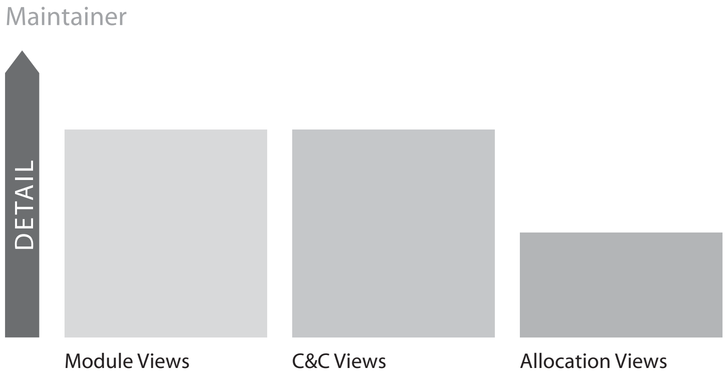


Figure 9.5
A maintainer has the same information needs as a developer but with a stronger emphasis on design rationale and variability.

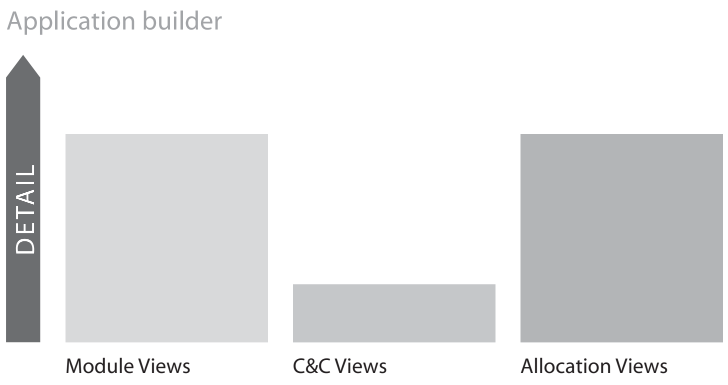


Figure 9.6
An application builder needs to understand what adaptations to make in order to build new products.

As shown in Figure 9.6, a product-line application builder, then, is likely to want to see the views mentioned for an integrator, plus

- A variability guide, as given in module and/or C&C views

Customers

Customers are the stakeholders who pay for the development of specially commissioned projects. Customers are interested in cost and progress and convincing arguments that the architecture and resulting system will meet the quality and functional requirements. Customers will also have to support the environment in which the system will run and will want to know that the system will interoperate with other systems in that environment.

As shown in Figure 9.7, the customer, then, is likely to want to see

- C&C views: the analysis results will be of particular interest

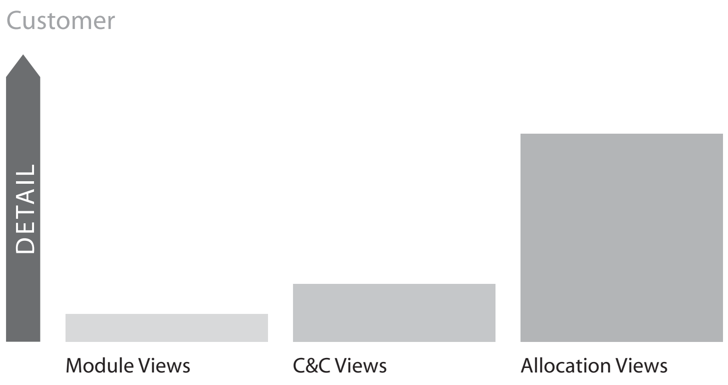


Figure 9.7
A customer is interested mainly in how the software works in the desired environment.

- Allocation views: work assignment view, no doubt filtered to preserve the development organization's confidential information, and a deployment view
- Other: a top-level context diagram in one or more C&C views

End users

End users do not need to see the architecture, which is, after all, largely invisible to them. But they often gain useful insights about the system, what it does, and how they can use it effectively by examining the architecture. If end users or their representatives review your architecture, you may be able to uncover design discrepancies that would otherwise have gone unnoticed until deployment.

To serve this purpose and as shown in Figure 9.8, an end user is likely to be interested in

- C&C views: views emphasizing flow of control and transformation of data, to see how inputs are transformed into outputs; analysis results dealing with properties of interest, such as performance or reliability
- Allocation views: a deployment view to understand how functionality is allocated to the platforms with which the users interact

Analysts

Analysts are interested in the ability of the design to meet the system's quality objectives. The architecture serves as the fodder for architecture evaluation methods and must contain the information necessary to evaluate such quality attributes as security, performance, usability, availability, and modifiability. For performance engineers, for example, architecture provides the model that drives such analytical tools as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, and model checkers. These tools require information about resource consumption, scheduling policies, dependencies, and so forth.

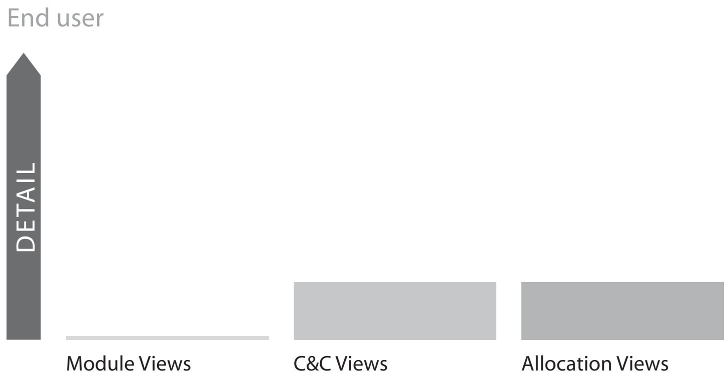


Figure 9.8
An end user needs to have an overview of the software, how it runs on the platform, and how it interacts with other software.

In addition to generalized analysis, architectures can be evaluated for the following and other quality attributes, each of which suggests certain documentation obligations.

- *Performance.* To analyze for performance, performance engineers build models that calculate how long things take. Plan to provide a communicating-processes view to support performance modeling. In addition, performance engineers are likely to want to see a deployment view, behavior documentation, and those C&C views that help to track execution.
- *Accuracy.* Accuracy of the computed result is a critical quality in many applications, including numerical computations, the simulation of complex physical processes, and many embedded systems in which outputs are produced that cause actions to take place in the real world. To analyze for accuracy, a C&C view showing flow and transformation of data is often useful because it shows the path that inputs take on their way to becoming outputs, and it helps identify places where numerical computations can degrade accuracy.
- *Modifiability.* To gauge the impact of an expected change, a uses view and a decomposition view are most helpful. Those views show dependencies and will help with impact analysis. But to reason about the runtime effects of a proposed change requires a C&C view as well, such as a communicating-processes view, to make sure that the change does not introduce deadlock.
- *Security.* A deployment view is used to see outside connections, as are context diagrams. A C&C view showing data flow and security controls is used to track where information goes and is exposed; a decomposition view is used to find where authentication and integrity concerns are handled. Denial of service is loss of performance, and so the security analyst will want to see the same information as the performance analyst.
- *Availability.* A C&C communicating-processes view will help analyze for deadlock, as well as synchronization and data consistency problems. In addition, C&C views show how redundancy, failover, and other availability mechanisms kick in as needed. A deployment view is used to show possible points of failure and backups. Reliability numbers for a module might be defined as a property in a module view, which is added to the mix.
- *Usability.* A decomposition view will enable analysis of system state information presented to the user; help with determination of data reuse; assign responsibility for usability-related operations, such as cut-and-paste and undo; and other things. A C&C communicating-processes view will enable analysis of cancellation possibilities, failure recovery, and so on.

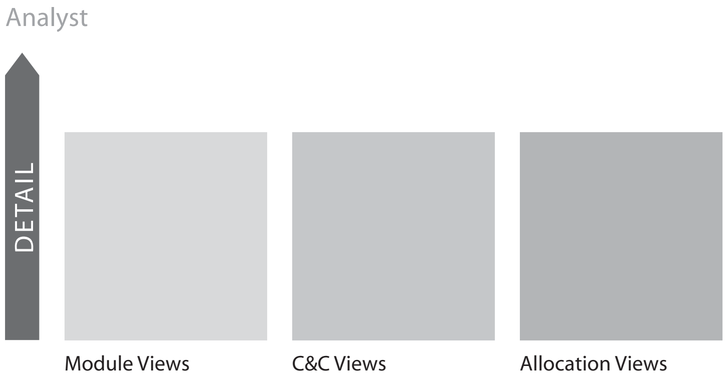


Figure 9.9
An analyst needs information from all views. Depending on the specific analysis, other, more detailed information might be required.

As shown in Figure 9.9, an analyst is likely to be interested in

- Module views: various
- C&C views: various, but especially those showing processes
- Allocation views: deployment

Infrastructure support personnel

Infrastructure support personnel set up and maintain the infrastructure that supports the development, build, and production environments of the system. You need to provide documentation about the parts that are accessible in the infrastructure. Those parts are usually elements shown in a decomposition, C&C, install, and/or implementation view. A variability guide is particularly useful to help set up the software configuration management environment.

As shown in Figure 9.10, infrastructure support people likely want to see

- Module views: decomposition and uses
- C&C views: various, to see what will run on the infrastructure

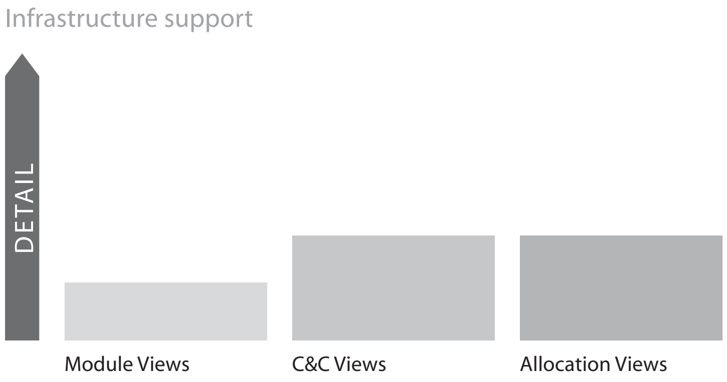


Figure 9.10
Infrastructure support people need to understand the software artifacts produced to provide tool support.

- Allocation views: deployment and install, to see where the software (including the infrastructure) will run; implementation
- Other: variability guides

New stakeholders

New stakeholders will want to see introductory, background, and broadly scoped information: top-level context diagrams, architecture constraints, overall rationale, and root-level views, as shown in Figure 9.11. People new to the system will usually want to see the same kind of information as their counterparts who are more familiar with the system, but new people will want to see it in less detail.

Future architects

Future architects are the most avid readers of architecture documentation, with a vested interest in everything. After the current architect has been promoted for producing the exemplary documentation, the replacement will want to know all the key design decisions and why they were made. As shown in Figure 9.12, future architects are interested in it all, but they will be

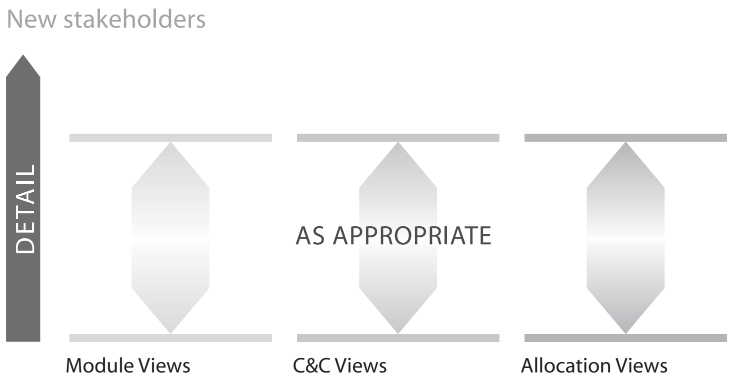


Figure 9.11
New stakeholders need to have the same information as their counterparts.

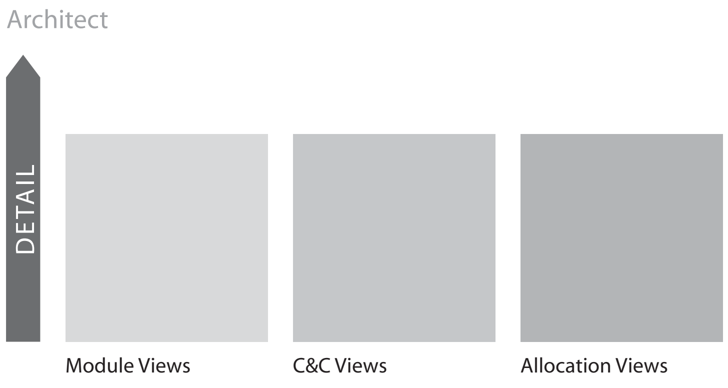


Figure 9.12
A future architect has strong interest in all the architecture documentation.

Table 9.1 Summary of documentation needs

	Module Views					C&C Views	Allocation Views				Other Documentation					
	Decomposition	**Uses**	**Generalization**	**Layered**	**Data Model**	**Various**	**Deployment**	**Implementation**	**Install**	**Work Assignment**	**Interface Documentation**	**Context Diagrams**	**Mapping Between Views**	**Variability Guides**	**Analysis Results**	**Rationale and Constraints**
Project managers	s	s		s			d			d		o				s
Members of development team	d	d	d	d	d	d	s	s	d		d	d	d	d		s
Testers and integrators	d	d	d	d	d	s	s	s	s		d	d	s	d		s
Designers of other systems					s						d	o				
Maintainers	d	d	d	d	d	d	s	s			d	d	d	d		d
Product-line application builders	d	d	s	o	s	s	s	s	s		s	d	s	d		s
Customers							o			o		o			s	
End users						s	s		o						s	
Analysts	d	d	s	d	d	s	d		s		d	d		s	d	s
Infrastructure support personnel	s	s			s	s	d	d	o					s		
New stakeholders	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Current and future architects	d	d	d	d	d	d	d	s	d	s	d	d	d	d	d	d
	Key: d = detailed information, s = some details, o = overview information, x = anything															

especially keen to have access to comprehensive and candid rationale and design information.

Table 9.1 summarizes the documentation needs of the stakeholders presented in this section.

9.2 A Method for Choosing the Views

At a minimum, expect to have at least one module view, at least one C&C view, and at least one allocation view in your architecture document.

This section presents a three-step method for choosing the views.

- **Step 1. Build a stakeholder/view table**. For this step, begin by building a table for your project, like that in Table 9.1.

  Enumerate the stakeholders for your project's software architecture documentation down the rows. Your stakeholder list is likely to be different from the one in Table 9.1; however, be as comprehensive as you can. For the columns, enumerate the views that apply to your system. As discussed in the prologue, some views (such as decomposition, uses, and work assignment) apply to every system, while others (various C&C views, the layered view) apply only to systems designed according to the corresponding styles. That is, you can produce a layered view only if your system is layered; you can produce a client-server view only if you used the client-server style; and so on. For the columns, make sure to include the views or view sketches you already have as a result of your design work so far.

Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail. The candidate view list going into step 2 now consists of those views for which some stakeholder has a vested interest.

Decide for which stakeholders you need to provide architecture documentation. Understand what type of information they need and with how much detail. Use this information to decide what views are needed and how to structure them into view packages to support your stakeholders.

PERSPECTIVES

Listening to the Stakeholders

It is asking a lot of an architect to divine the specific needs of each stakeholder, and so it is a very good idea to make the effort to communicate with stakeholders, or people who can speak for those roles. Talk with them about how they will best be served by the documentation you are about to produce. Practitioners of architecture evaluation almost always report that one of the most rewarding side effects of an evaluation exercise comes from assembling an architecture's stakeholders around a table and watching them interact and build consensus among themselves. Architects seldom practice this team-building exercise among their stakeholders, but a savvy architect understands that success or failure of an architecture comes from knowing who the stakeholders are and how their interests can be served. The same holds true for architecture documentation.

Before the architecture documentation effort begins, plan to contact your stakeholders. This will, at the very least, compel you to name them. For a large project in which the documentation is a sizable line item in the budget, it may even be worthwhile to hold a half-day or full-day roundtable workshop. Invite at least one person to speak for each stakeholder role of importance in your project. Begin the workshop by having each stakeholder explain the kind of information he or she will need to carry out his or her assigned tasks. Have a scribe record each stakeholder's answer on a flip chart for all to see. Then present a documentation plan: the set of views you've chosen, the supporting documentation, and the cross-view information you plan to supplement them with. Stakeholders may not necessarily understand what the views mean that you present. Have some examples ready to show how a specific view looks and what kind of information it will show. Finally, perform a cross-check to find requested but missing information and planned but unneeded documentation. Whether you hold a full-blown workshop or talk to your stakeholders informally, the result will be vastly increased buy-in for your documentation efforts and a clearer understanding on everyone's part of what the role of the architecture and its documentation will be.

The information that stakeholders need will not always align nicely with the information the architect was planning to produce. This is why it's so important to ask the stakeholders what they need. But you also have to listen to the answers.

We once ran a workshop with an architecture team to fix some issues that had arisen during an architecture evaluation. The evaluation revealed what we thought was a very well documented architecture. But during the follow-up workshop, the project manager raised the same issue over and over again: "Give me the data that I need to create a reliable project plan. It doesn't matter how long the project lasts, as long as we can reliably meet our delivery promises."

Each time, the chief architect made the same reply: "The information you need is in the architecture document." That statement was technically true, but not particularly helpful. The project manager needed a uses view including a list of module responsibilities. The architect provided it (using UML output from a tool), but the manager wasn't satisfied. "I cannot find the information in the document," he said. "I don't understand what those symbols mean and I don't have time to spend searching the document." The right documentation for him would have been an extraction of the effort/dependency information of modules from the UML model, rendered in plain text, and packaged separately.

Sadly, about a year later the project was canceled. The customer had lost faith in the company's ability to deliver what they promised.

In another case, we saw an architecture documented using the Kruchten 4+1 view set. The system was a typical three-tiered client-server architecture in which the middle tier was a framework that defined the applications as plug-ins. Once you knew that much, it was straightforward to come up with the views. The customer, on the other hand, knew that he was responsible for the system's maintenance after it was delivered. He always made the same demand: "Tell me what and where I have to change when I want to change the content of a specific Web page." He clearly had a "page-oriented view" in mind. This need was not satisfied by any of the views that had been documented using the very reasonable 4+1 approach. He might have been well served by a view showing which plug-ins or parts of plug-ins contributed to producing a Web page.

Keep an open mind when listening to your stakeholders. They'll tell you what they need. Many times it won't be what you were planning to provide, but many times (as in these two cases) what they need is easily produced from the information you already have at hand. And it might make the difference between success and failure.

—F.B. and P.C.

Section 6.6 discusses how to combine views, and which views are often easy and useful to combine.

- **Step 2. Combine views.** The candidate view list from step 1 is likely to yield an impractically large number of views. This step will winnow the list to manageable size.

  Look for views in the table that require only overview, or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger constituency.

When combining views it is useful to consider the costs associated with producing and maintaining a view. There are at least two sources of the cost. First is the cost required to generate the view, and second is the cost required to maintain it and keep it consistent with other views.

- **Step 3. Prioritize and stage**. After step 2 you should have the minimum set of views needed to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific to your project, but here are some things to consider:
  - Not all the information needs of all the stakeholders must be satisfied to the full extent. Providing 80 percent of the requested information goes a long way, and it might be "good enough" so that the stakeholders can do their job. Check with the stakeholder if a subset of information would be sufficient. They typically prefer a product that is delivered on time and in budget over getting the perfect documentation.
  - You don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best.
  - Some stakeholders' interests supersede others. A project manager, or the management of a company with which yours is partnering, often demands attention and information early and often.
  - If your architecture has not yet been validated or evaluated for fitness of purpose, then documentation to support that activity merits high priority.
  - Resist the temptation to relegate rationale documentation to the "do when we have time" category, because rationale is best captured when fresh.

The decomposition view is a particularly helpful view to release early. High-level decompositions are often easy to design, and with this information the project manager can start to build development teams, put training in place, scour the commercial markets or legacy repositories for modules that fill the bill, and start producing budgets and schedules.

Use view packets (discussed in Section 10.1.3) as a mechanism to let you provide overviews or less-detailed documentation to certain stakeholders.

9.3 Example

This section provides an example of applying the procedure in the previous section to select a set of views for a project.

ECS is a system for capturing, storing, distributing, processing, and making available extremely high volumes of data from a constellation of earth-observing satellites. By any measure, ECS is a very large project. Many hundreds of people are involved in its design, development, deployment, sustainment, and use. Here is how the three-step view selection approach might have turned out, had it been applied to the ECS software architecture.

Step 1: Produce a Candidate View List

Stakeholders for the ECS architecture include the usual suspects: the current and future architect, developers, testers and integrators, and maintainers. But the size and complexity of ECS, plus the fact that it is a government system whose development is assigned to a team of contractors, add complicating factors. In this case, there is not one project manager, but several: one for the government and one for each of the contractors. Each contractor organization has its own assigned part of the system to develop and, hence, its own team of developers and testers. ECS relies heavily on commercial off-the-shelf (COTS) components, so the people responsible for selecting COTS candidate components, qualifying them, selecting the winners, and integrating them into the system play a major role. We'll call these stakeholders COTS engineers.

The important quality attributes for ECS begin with performance. Data must be ingested into the system to keep up with the rate at which it floods in from the satellites. Processing the raw data into more sophisticated and complex "data products" must also be done every day to stay ahead of the flow. Finally, requests from the science community for data and data analysis must be handled in a timely fashion. Data integrity, security, and availability round out the important list of quality attributes and make the analysts concerned with these qualities important architecture stakeholders.

ECS is a highly visible and highly funded project that attracts oversight attention. The funding authorities require at least overview insight into the architecture to make sure the money over which they have control is being spent wisely. Finally, the science community using ECS to measure and predict global climate change also requires insight into how the system works, so they can better set their expectations about its capabilities.

At least five of the component-and-connector views discussed in Chapter 4 and four of the module views of Chapter 2 apply to ECS. It is primarily a shared-data system. Its components interact in both client-server and peer-to-peer fashion. Many of those components are communicating processes. And while the system is not actually built using pipes and filters, the pipe-and-filter style is a very useful paradigm to provide an overview to some of the stakeholders. (Information more detailed than the overview will be in a different view, becoming an implementation refinement of the pipe-and-filter view.)

Table 9.2 shows the stakeholders for the ECS architecture documentation and the views useful to each. At this point, the candidate view list contains 12 views.

Table 9.2 ECS stakeholders and architecture documentation they might find most useful

	Module Views				C&C Views					Allocation Views		
Stakeholder	**Decomposition**	**Generalization**	**Uses**	**Layered**	**Pipe-and-filter**	**Shared-data**	**Client-server**	**Peer-to-peer**	**Communicating-processes**	**Deployment**	**Implementation**	**Work assignment**
Current and future architect	d	d	d	d	s	d	d	d	d	d	s	s
Government project manager	d	o	o	s	o	s	o	o	o	s		d
Contractor's project manager	s	o	s	s	o	s	s	s	o	d	s	d
Member of development team	d	d	d	d	o	d	d	d	d	s	s	d
Testers and integrators	s	s	d	s	o	d	d	d	s	s	d	
Maintainers	d	d	d	d	o	d	d	d	d	s	s	s
COTS engineers	d	s		d		d	d	d	s	d		d
Analyst for performance	d	s	d	s	o	d	d	d	d	d		
Analyst for data integrity	s	s	s	d	o	d	d	d	d	d		
Analyst for security	d	s	d	d	o	s	d	d	d	d	o	o
Analyst for availability	d	s	d	d				s	s	d		o
Funding agency	o				o	o				o		
Users in science community	o				o	o				o		
Key: d = detailed information, s = some detail, o = overview												

Step 2: Combine Views

As usual, the C&C views provided good candidates for combination. In the case of ECS, augmenting the shared-data view with other components and connectors that interact in client-server or peer-to-peer fashion allowed those three views to become one. The communicating processes mapped straightforwardly to components in this combined view, allowing it to be folded in as well. The pipe-and-filter view can be discarded; the combined C&C view plus some key behavioral traces showing the data pipeline from satellite to scientist would provide the same intuitive overview to the less detail-oriented stakeholders.

Similarly, some of the module views were combined. Recording uses information as a property of the decomposition view yields a combination of the decomposition and uses views.

It would have been easy to combine the work assignment and implementation views with decomposition as well. However, because of the large size of this project and the number of different development organizations involved, the work assignment view was kept separate. Also, this view was of key interest to managers and the funding agency, who did not want to see details of the modules. Similarly, because a large number of stakeholders interested in the module decomposition would not be interested in how the modules were allocated to files in the development environment, the implementation view was also kept separate.

After this step, the following views remain:

- Three module views: decomposition/uses, layered, and generalization
- One C&C view: shared-data/client-server/peer-to-peer/communicating-processes
- Three allocation views: deployment, implementation, and work assignment

We entered step 2 with 12 candidate views, too many to be efficiently maintained. Now there are 7.

Step 3: Prioritize

To let the project begin to make progress required putting contracts in place, which in turn required coarse-grained decomposition. Turning out the higher levels of the decomposition and work assignment views received the highest priority, in order to meet these needs.

In ECS, the layering in the architecture was very coarse grained and can be described quickly. Similarly, generalization occurred largely in only one of the three major subsystems, was also coarse grained, and was able to be described quickly. These two views were given next priority.

The combined C&C view and the deployment view followed, nailing down details of runtime interaction only hinted at by the module views. This allowed analysis for performance to begin.

Finally, because the implementation view can be relegated to each contractor's own internal development effort, it received the lowest priority from the point of view of the overall system.

The result was four "full-fledged" views (decomposition, work assignment, the combined C&C view, and deployment) plus three minor ones that are coarse grained or can be deferred.

PERSPECTIVES

How Not to Introduce an Architecture

With John Klein

Several years ago, I was chief architect for a business unit at a large software product development company. My manager, the vice president of engineering for the business unit, approached me one spring day and challenged me to define a single, unified architecture that could be applied to all current products in our portfolio, and that would also support our best guess of future needs. Recognizing the relationship between architecture and organization, he wanted to use this new architecture as the basis for a major reorganization of the 300-person software engineering team, and he wanted to roll out this reorganization at an engineering management meeting planned for late summer. So, my team of five architects had just 90 days to define enough of a system architecture that our VP could build an organization around it.

After a stakeholder analysis, we determined that we needed to produce a number of views. Three of the views are described elsewhere in this book:

- A decomposition view (which we called our “Information Hiding Module Guide)
- A uses view
- A C&C communicating-processes view

In addition to these, we decided to create the following:

- A “technology view,” which would be a type of allocation view that would map modules to implementation technology (programming language and middleware)
- A “design model,” which was an information model for product configuration and customization data
- An “integration view,” to specify the external interfaces of our products
- A “Zoo of Examples,” which was “information beyond views” that showed how to use the architecture to create products

Our documentation plan also included an “Architecture Description Overview,” which contained the stakeholder analysis, descriptions of each view, and a set of roadmaps to help different stakeholders navigate the documentation.

We began by focusing on the Information Hiding Module Guide. We wanted to meet our VP’s need to reorganize based on the architecture; the information hiding decomposition provided a natural basis for structuring the development organization. Also, we recognized that we were developing a software product-line architecture, and we saw the value of structuring the information hiding decomposition to encapsulate the variation points that the architecture would support.

We spent much of our 90-day schedule working on the Module Guide, essentially a detailed decomposition view. We also completed the Architecture Description Overview, which we thought clearly showed our vision for documenting the architecture. Finally, we created a set of "marketecture" diagrams, which we used to communicate the architecture to our executives.

As the deadline approached, the team was feeling proud of our efforts. We felt we had achieved the goal of developing enough architecture to drive the reorganization. Besides, we thought, nobody really expected us to create the entire architecture in just 90 days! We felt we had done enough.

Were we ever wrong!

On the appointed day, we presented the Architecture Description Overview and Module Guide during an "all hands" conference call with the entire software engineering team. There were some polite questions, but we sensed that there was a lot of confusion among the team members. Our organization did not have much exposure to architecture-centric practices. The idea of multiple views of an architecture was understood by some but not all of the staff. Also, the thought that you would incrementally release subsets of the architecture documentation was foreign. Finally, the first view we released showed the information hiding structure, which the staff found difficult to grasp and appreciate.

In retrospect, I realize that the decomposition view is simply a well-organized "parts list" for all of the products in our product line. Of course people were confused—we didn't show which parts go into which products, didn't describe how to assemble the parts into products, and didn't provide a picture of any of the final built products. We confused the development team so much that most of the development staff pointed to the Module Guide as "the architecture document." Here's what we learned.

In most cases, the first release of documentation for an architecture will not be complete and whole. Recommendations in this book, such as "Begin your documentation with a standard outline," will help you and your stakeholders understand the vision and the intended final structure of the documentation, but the first time you release the documentation, people will read the parts you have completed and try to make sense of them.

Stage your architecture design and documentation to deliver coherent subsets to stakeholders. Make each subset internally consistent and complete, and present a chunk of the architecture that will make sense to your stakeholders. Specifically, include some C&C views in early releases so that stakeholders can understand how the system will function at runtime. This is usually a more natural perspective to begin reasoning about the system, as compared to a module view showing design-time structure.

Include in each stage subsets of several views, rather than the approach we took of delivering views sequentially. Provide your stakeholders a complete specification of a subset of the system: a "parts list" (module view), assembly instructions (allocation view and "beyond views"), and a picture of the running system (C&C view). This allows them to make complete sense of each incremental release of the architecture documentation.

Failure to do these things may damage your credibility, stakeholders may lose interest, and your project may fail, as ours eventually did.

9.4 Summary Checklist

- What views you choose depends on who the important stakeholders are, what budget is on hand, what the schedule is, and what skills are available. It also depends on what structures are present in the architecture.
- You should expect to choose at least one of each of the three different types of views: module, component-and-connector, and allocation.
- You should expect to combine some views to reduce the number of views you have to create, keep consistent, and maintain in your architecture document.
- Prioritize and stage your release of views to serve important project needs early.

9.5 Discussion Questions

1. Suppose that your company has just purchased another company and that you've been given the task of merging a system in your company with a similar system in the purchased company. What views of the other system's architecture would you like to see, and why? Would you ask for the same views for both systems?
2. Some architects speak of a "security view" or documentation of a "security architecture." What do you suppose they mean? What might this consist of?
3. How would you make a cost/benefit argument for the inclusion or exclusion of a particular view in an architecture documentation package? If you could summon up any data you needed to support your case, what data would you want?

9.6 For Further Reading

Around 2001, practitioners at Nokia developed the Rapid7 approach to produce high-quality usable documentation in an Agile environment. The central approach to Rapid7 is to hold a stakeholder workshop at each document delivery milestone in the project. The workshop is facilitated to produce a document outline that stakeholders will actually use. For more information, see the paper by Kylmäkoski (2003).

A central theme of the book by Hofmeister, Nord, and Soni (2000) is the coordinated use of separate (in their case, four) views to engineer and document software systems. Their treatment provides an excellent foundation for the philosophy behind choosing the views: providing information to stakeholders, and points of engineering leverage to the architect, based on expected needs of the system being built.

10 Building the Documentation Package

You now have everything you need to begin building the complete documentation package. You have a repertoire of styles from which you can construct views, a method for choosing the most useful views to document, and insights about how to document architecture information beyond structure: context, diagrams, variability, interfaces, and behavior. This chapter shows you how to put it all together.

First, we return once again to our fundamental principle of documenting architectures:

> **Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.**

Rule 4 for sound documentation, given in the prologue, counsels us to use a standard organization for documents. Combining these two foundations, this chapter provides standard document organizations for documenting architecture views, along with the information that transcends views.

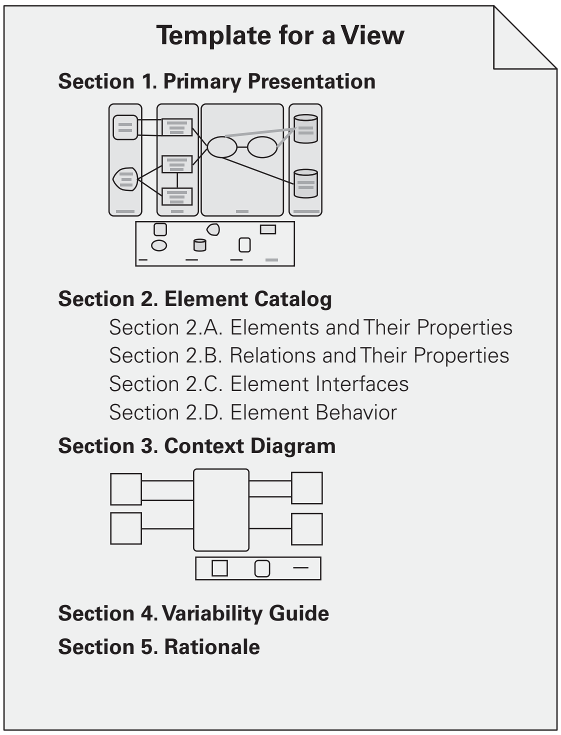
10.1 Documenting a View

Figure 10.1 shows the template for documenting a view.

10.1.1 A Standard Organization for Documenting a View

No matter what the view, the documentation for a view can be placed into a standard organization consisting of these parts.

Figure 10.1
View template



Template for a View

Section 1. Primary Presentation

Section 2. Element Catalog

Section 2.A. Elements and Their Properties
Section 2.B. Relations and Their Properties
Section 2.C. Element Interfaces
Section 2.D. Element Behavior

Section 3. Context Diagram

Section 4. Variability Guide

Section 5. Rationale

Section 1. The Primary Presentation

Drawings help people to work out intricate relationships between parts.
—Christopher Alexander

The *primary presentation* shows the elements and relations of the view. The primary presentation should contain the information you wish to convey about the system—in the vocabulary of that view—first. It should certainly include the primary elements and relations but under some circumstances might not include all of them. For example, you may wish to show the elements and relations that come into play during normal operation but relegate error handling or exception processing to the supporting documentation. What information you include in the primary presentation may also depend on what notation you use and how conveniently it conveys various kinds of information. A richer notation will tend to enable richer primary presentations.

If your primary presentation is graphical, make sure to include a key that explains the notation.

The primary presentation is most often graphical. It might be a diagram you've drawn in an informal notation using a simple drawing tool, or it might be a diagram in a semiformal or formal notation imported from a design or modeling tool that you're using. The figures that illustrate diagrams from example views in Chapters 2, 4, and 5 are all diagrams that would appear in a primary presentation.

Sometimes the primary presentation can be textual, such as a table or a list. If that text is presented according to certain stylistic rules, they should be stated or incorporated by reference, as the analog to the graphical notation key. Regardless of whether the primary presentation is textual instead of graphical, its role is to present a terse summary of the most important information in the view.

An example of a textual primary presentation is shown in Figure 2.4, in Section 2.1.6.

The primary presentation may feature more than one diagram. For example, suppose the system has two separate subsystems, each of which is built using the pipe-and-filter style. A pipe-and-filter view of this system could have two diagrams in its primary presentation. Each would show the pipe-and-filter elements in one of the two subsystems.

To remind us that the primary presentation is only the starting point for documenting a view, we call the graphical portion of the view an **architecture cartoon**. We use the definition from the world of fine art: A cartoon is a preliminary sketch of the final work; it is meant to remind us that the picture, although getting most of the attention, is not the complete description but only a sketch of it.

An **architecture cartoon** is the graphical portion of a view's primary presentation, without supporting documentation.

ADVICE

Use the organization described in this section as the basis for your view template. Modify it as necessary to make it appropriate for your organization's standards and the special needs of the development project at hand. Be cautious about throwing out sections that you think you don't need; the presence of a section in the template can prod you to think about the issue across the system, whereas omitting the section will let you forget about it, perhaps to the detriment of the system. For each section, include a terse description of the contents of that section.

Whatever organization you choose for documenting your views, explain it to your readers. In the Views and Beyond template, this is done in section 2 of the template for Documentation Beyond Views; see Section 10.2.

Even if some items are empty for a given view—for example, perhaps no mechanisms for variability exist or no relations other than those shown in the primary presentation exist—include those sections, marked "none" or "not applicable." Don't omit them, or your reader may wonder whether it was an oversight.

If the view derives from one or more published styles or patterns, let the reader know. It is most convenient to do this by adding an annotation to the primary presentation or (if it's graphical) adding a note in the notation key. Be sure to cite the published source and not just name the pattern or style, since many patterns and styles with common names are described differently by different authors.

See "Not Every View Comes from a Published Style or Pattern," on page 343.

Section 2. The Element Catalog

The *element catalog* details at least those elements depicted in the primary presentation. For instance, if a diagram shows elements A, B, and C, then the element catalog needs to explain what A, B, and C are and their purposes or the roles they play, rendered in the vocabulary of the view. In addition, if elements or relations relevant to this view were omitted from the primary presentation, they should be introduced and explained in the catalog. Specific parts of the catalog include the following:

Section I.3, in the introduction to Part I, discusses how to choose properties to document.

a. *Elements and their properties.* This section names each element in the view and lists the properties of that element. Each style introduced throughout Part I listed a set of suggested properties associated with that style. For example, elements in a decomposition view might have the property of "responsibility"—an explanation of each module's role in the system—and elements in a communicating-processes view might have timing parameters, among other things, as properties. Whether the properties are generic to the style chosen or the architect has introduced new ones, this is where they are documented and given values.
b. *Relations and their properties.* Each view has specific relation type(s) that it depicts among the elements in that view. Mostly, these relations are shown in the primary presentation. However, if the primary presentation does not show all the relations, or if there are exceptions to what is depicted in the primary presentation, this is the place to record that information. Otherwise, this section will be empty.

Documenting interfaces is covered in Chapter 7.

Documenting behavior is covered in Chapter 8.

c. *Element interfaces.* This section documents element interfaces.
d. *Element behavior.* Some elements have complex interactions with their environment. For purposes of understanding or analysis, it is incumbent on the architect to specify element behavior.

Section 3. Context Diagram

Context diagrams are discussed in Section 6.3.

A *context diagram* shows how the system or portion of the system depicted in this view relates to its environment.

Section 4. Variability Guide

Using a variability guide to document architecture variation points is covered in Section 6.4.4.

A *variability guide* shows how to exercise any variation points that are a part of the architecture shown in this view.

Section 5. Rationale

Documenting rationale is covered in Section 6.5.

Rationale explains why the design reflected in the view came to be. The goal of this section is to explain why the design is as it is and to provide a convincing argument that it is sound. The use of a pattern or style in this view should be justified here.

Items 2–5 are called the *supporting documentation* and explain and elaborate the information in the primary presentation.

PERSPECTIVES

From Context Diagrams to a Context View

With Nick Rozanski and Eoin Woods

We always include a system context diagram in any architecture description we produce, although sometimes that's just a reference to a context diagram defined elsewhere. In our experience a good context diagram is an essential part of an effective architecture document. The Views and Beyond approach extends that basic piece of good practice to provide a different kind of context diagram in every view.

However, it wasn't until long after our book *Software Systems Architecture* (2005) went to press that we realized that we were asking the same sorts of questions when creating our context diagrams that we were asking when creating the architecture views, namely:

- Who are the stakeholders interested in the context diagram?
- What are their concerns?
- How can we document the system context in a way that illustrates how these concerns are addressed by the architecture?

We realized that these questions, and their answers, are important enough to merit full consideration on their own, rather than just being implicitly answered in one or more context diagrams. We have therefore come to the conclusion that we need to add an additional viewpoint to our viewpoint set, namely the *system context viewpoint*.

Using a system context view gives us the opportunity to explain how key concerns will be addressed by the system, and to set out the decision-making processes we have gone through to finalize the scope. It often also enables us to identify some key system-wide principles, constraints, and risks at an early stage.

The purpose of any sort of context diagram is to define what is in scope and out of scope for the system, and how the system relates to its environment. In theory all of this should be clearly understood and written down before the development starts, but in practice this is often not the case. Scope often changes during development, and these changes must be reflected in the context diagram. The scope may not be documented anywhere, or if it is, this may be vague or inconsistent. Even worse, there may be a number of different versions of the "implied" scope of the system, which is a recipe for disaster. And the environment may be far from fixed and understood.

It is this reality that the system context viewpoint is designed to address, and as such a context view is usually one of the first that the architect should produce.

Chapter 9 tells us that the first question we must answer when choosing our views is "Who are the stakeholders?" For a system context view, the answer is somewhat uncomfortable: "Pretty much all of them." Obviously the acquirer (sponsor) and users are interested in the scope, since this defines the extent of the delivered system and the functionality it will and will not provide. Developers need to know what the system comprises and, in particular, which external systems and organizations it will need to interact with.

Many other stakeholders also have concerns that will be addressed in a system context view. For example, operational staff will need to understand the other systems that this system interacts with, so that they can begin to plan the processes and tools they will need for its monitoring and support. Testers will need to understand the inbound and outbound data flows so that they can start to think about integration and preproduction testing, and plan the creation of appropriate stubs and test harnesses. And key stakeholders in the other systems that interact with this one may need to start work on changes to their interfaces, or make improvements in their system's scalability, availability, and response time.

To meet these concerns, a system context view should document:

- The key responsibilities of the system
- The identity and key responsibilities of external entities
- The main external interdependencies, including the expected external interactions, the nature of the external connections, and high-level definitions of the external interfaces

Within a system context view, context is modeled in a single (or sometimes several) context model, which resemble "traditional" context diagrams, as described in this chapter. These illustrate the system, the external entities, and the connections between them. The system is usually represented as a "black box," without any of its internal details exposed—these may not be known at this early stage anyway. Models may be annotated with logical or physical details of the external systems and interfaces. Supplementary information is often produced as well. This may include a list of the key functional capabilities that are in and out of scope, a description of the key information flows, or a description of some key interaction scenarios.

Because of the wide range of stakeholders and concerns, it is tempting to overload the context view models with as much information as possible: systems, interfaces, hardware and software, organizational boundaries, constraints, and more. However, this flies in the face of everything we know about good architecture documentation, and it is why the concept of views was introduced in the first place. You should strive for a context view that has an even and consistent level of focus, brings out all the key dependencies and interconnections, and gets the right balance between brevity and accuracy. This is by no means easy, but using multiple models—that is, multiple context diagrams—to convey different kinds of context information is a good way to avoid overloading a single model.

The system context view is produced at a very early stage in the development, when there are many unanswered questions. There may also be a significant amount of political positioning, maneuvering, and horse-trading of scope, requirements, and plans going on. If it is used effectively, it is a very valuable way to communicate architecture decisions and plans in a way that is meaningful to a broad community of stakeholders and provides a solid foundation for the architecture, design, and build.

ADVICE

Not Every View Comes from a Published Style or Pattern

Until now we have written about views as though they are nothing more than a published style or pattern applied to a system: Take the element and relation types defined in the style or pattern, make a bunch of instances, wire them together following the restrictions of the pattern or style, and there you are.

And that works. Many very, very useful architecture views are exactly that (or as we saw in Section 6.6, combinations of exactly that). Views such as layered, service-oriented, client-server, peer-to-peer, and a host of others are found in real-world architecture documents, and they derive precisely from the corresponding styles or patterns in published references.

But not every view enjoys such a formal pedigree. First, real architects often make specializations of “standard” or published styles and patterns to suit their needs. For example, they may impose a particular protocol on client-server, or layered callback, or service interactions over and above what is called for in a published pattern. Second, real architects may use common element types and wire them together using bare-minimum architecture mechanisms to suit their needs. For example, a view showing a system’s fail-over policy might show components designated as “primary” and “secondary” connected by a simple call connector to a heartbeat monitor—even if that’s not a published style.

What is the documentation obligation in this case? You could define a new style or pattern that is exactly what (and as specialized as) you need, then cite it. More convenient, however, is to describe the specializations you’ve made to a published form or (if you’re really working from a new style) define the element types and relation types in situ. A convenient place to do this is the element catalog of the view in which your specialized form appears. The diagram in the primary presentation should make it clear (either through annotation or by introducing new graphical elements that you’ll make sure to put in the key) which elements or relations are your specializations. And you should explain your choices in the rationale section.

10.1.2 Useful Variations in the Standard Organization for a View

The standard organization for documenting a view presented in the last section serves well in most cases. However, there are some useful variations that may serve better in others. These include the following:

Variation 1: Divide the View into View Packets

Views of large software systems can contain hundreds or even thousands of elements with arbitrarily deep levels of nesting. Showing these elements in a single presentation, along with the relations among them, can result in a blizzard of information that is indecipherable. Also, many stakeholders aren't interested in the entire view, just their little part of it. Or just the broad picture bereft of much detail. In some organizations, you might want to impose access control on parts of a view you don't want your subcontractors to look at.

A **view packet** is the smallest bundle of view documentation you would show an individual stakeholder, such as a developer assigned to implement a small portion of the system or a customer interested in an overview.

If you need a way to present a view's information in smaller "chunks," break it up into **view packets**. Each view packet can show a fragment of the system with great depth of detail, or broad areas of the system with shallower detail. The documentation for a view, then, can consist of a set of view packets.

View packets make an excellent way to help the architect carry out and record refinements as they are made as part of the journey through the spectrum of design.

Refinement and the spectrum of design are discussed in Chapter 6. For an example of using view packets to record more and more detailed architectural decisions, see the sidebar "Using View Packets to Record Architecture Design Steps," on the next page.

The same standard organization we used for a view also works for a view packet. Just remember:

- The primary presentation shows the elements and relations that populate the portion of the view shown in this view packet, rather than the whole view.
- The supporting documentation (element catalog, context diagram, variability guide, and rationale) all explain just the part of the architecture shown in the primary presentation. The "environment" shown in the context diagram may well be other elements that are internal to the overall system whose architecture we're documenting.
- As an aid to readers' navigation among view packets, it helps to add a pointer to a view packet's parent view packet as well as its sibling and child view packets. (A "child" view packet is one that shows a decomposition refinement of one or more elements in its "parent" view packet.)

Decomposition refinement is discussed in Section 6.1.1.

If you divide a view into view packets, preface the set with an explanation of what view packets are provided and what part of the system each one shows. One way to do this is to show the context diagram for each included view packet, with parent/child links among the diagrams, to help a reader navigate to and identify the view packet he or she wants to see.

ADVICE

Using View Packets to Record Architecture Design Steps

Throughout this book, we have tried to make the point that architecture documentation is not just a necessary afterthought of architecture design, but an important contributor to the design process itself. View packets make an excellent vehicle for storing architecture decisions as they are made, making architecture design and documentation go hand in hand.

We illustrate this concept using version 2 of the Attribute-Driven Design (ADD) method. ADD is a step-by-step architecture design method that relies on iteratively choosing a part of the system to design, and then choosing appropriate architecture styles, patterns, and tactics to satisfy the architecturally significant requirements for that part. The result of each ADD iteration can be recorded in its own view packet.

Since ADD is a sequential, step-by-step method, you can also record the chronology of your design—what decisions came before and after what other decisions. This will be helpful if you need to change a design decision. You can easily see what design decisions you made after the one in question, to determine if they need to change as well.

Below are the rough steps of ADD, along with what you should record in a view packet when you carry out each one.

Step of ADD Method	Information to Record in a View Packet
Step 1: Confirm there is sufficient requirements information.	None.

continues

Don't try to record all the information in pristine, ready-for-prime-time fashion. For one thing, ADD includes a back-up-and-try-again option in step 4. (Perhaps the design concept you chose three iterations ago unwittingly precluded meeting the requirements you're handling in the current iteration. You'll have to back up and try again.) So don't waste time making the information beautiful. Instead, make it comprehensible. You can shine it up when you have an architecture you have confidence in.

Step of ADD Method	**Information to Record in a View Packet**
Step 2: Choose an element of the system to design (for the first iteration of ADD, this "element" may well be the whole system).	Start with a new, blank view template. In the rationale section, explain why you chose this element of the system. In the related view packets section, point to this element's parent (if any), and chronological predecessor (if any). Create a context diagram for the element based on what you know about its interactions with entities external to it.
Step 3: Identify candidate architecture drivers.	Record the drivers in the rationale section.
Step 4: Choose a design concept that satisfies the architecture drivers.	Describe the design concept—typically a choice of architecture patterns or styles augmented with tactics—in the rationale section, and say why you chose it.
Step 5: Instantiate architecture elements and allocate responsibilities.	Capture the instantiation in the primary presentation. Describe the instantiated elements, relations, and element behavior in the element catalog.
Step 6: Define interfaces for instantiated elements.	Record preliminary interface definitions in the element catalog.
Step 7: Verify and refine requirements and make them constraints for instantiated elements.	None. When you turn your design attention to one of those instantiated elements, its requirements and constraints will yield a set of drivers that you'll record as step 3 of that future iteration.
Step 8: Repeat steps 2 through 7 for the next element of the system you wish to decompose.	None. The method terminates when all requirements and constraints have been allocated to (and satisfied by) architecture elements.

Notice how using view packets to hold design decisions as you go obviates the question of what views to use during architecture design. Your choice of architecture patterns and styles binds you to a choice of views. If you choose a service-oriented style when designing a system (or portion of a system), you'll document a service-oriented view to capture its instantiation; if you choose a layered style, you'll document a layered view to capture its instantiation; and so forth. Later, when you have a collection of view packets representing a collection of views, you can assemble them into collections that make sense using the Choosing the Views approach of Chapter 9.

Variation 2: Add a Section to Document the Behavior of the Whole Architecture Shown in the Primary Presentation of a C&C View

The primary presentation of a C&C view shows a group of architecture elements (components and connectors) and their runtime interactions. The element catalog contains the behavior of those elements. But you'll almost certainly want to document the behavior of the group as a whole somewhere. Where? Alternatives include the following:

- *The behavior section of the view's element catalog.* This section is primarily intended to capture the behavior of individual elements. You could add a special entry at the end to capture the behavior of everything working together.
- *A new section in the standard organization for a view.* Architects often show structure and the behavior of that structure next to each other, affording equal status to both. Giving the behavior its own section makes that easier.
- *If you use view packets, you don't need to change the template.* The group of components and connectors shown in a view packet might well be a specialization of a single component or connector shown in a parent view packet. In that case, its behavior will be documented in the element catalog of the parent view packet.

Variation 3: Combine the Primary Presentation and Context Diagram

If you combine the primary presentation with context, make sure to indicate the system boundary. Either use a clear distinguished bounding symbol, and put that symbol in your key, or indicate clearly which elements are external to the system.

Stripped to its essentials, a context diagram shows the system being described along with external elements with which it interacts or is related. As shown in Figure 10.2, the system is depicted as a monolithic entity, starkly bounded, with no internal structure: a black box.

Showing the internal structure is the job of the primary presentation. While this represents a useful separation of concerns in many cases, sometimes the primary presentation can be more expressive if it contains external entities. Especially with C&C views, combining the primary presentation with the context diagram lets you see where the arrows begin and end, which can help you ensure that none of the ties between system internals and externals is overlooked.

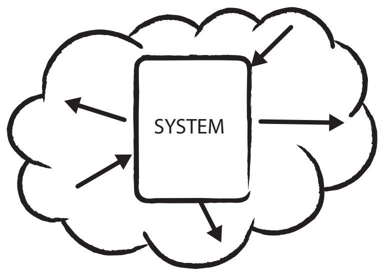


Figure 10.2
A pure context diagram shows the system, with no internal structure shown, and its relations to entities in its environment.

It's common to see external entities in primary presentations, but architects usually don't bother indicating the fact that some of the elements they're showing are external. This can be confusing. Figure 10.3 is an example of a cartoon that combines a primary presentation with a context diagram.

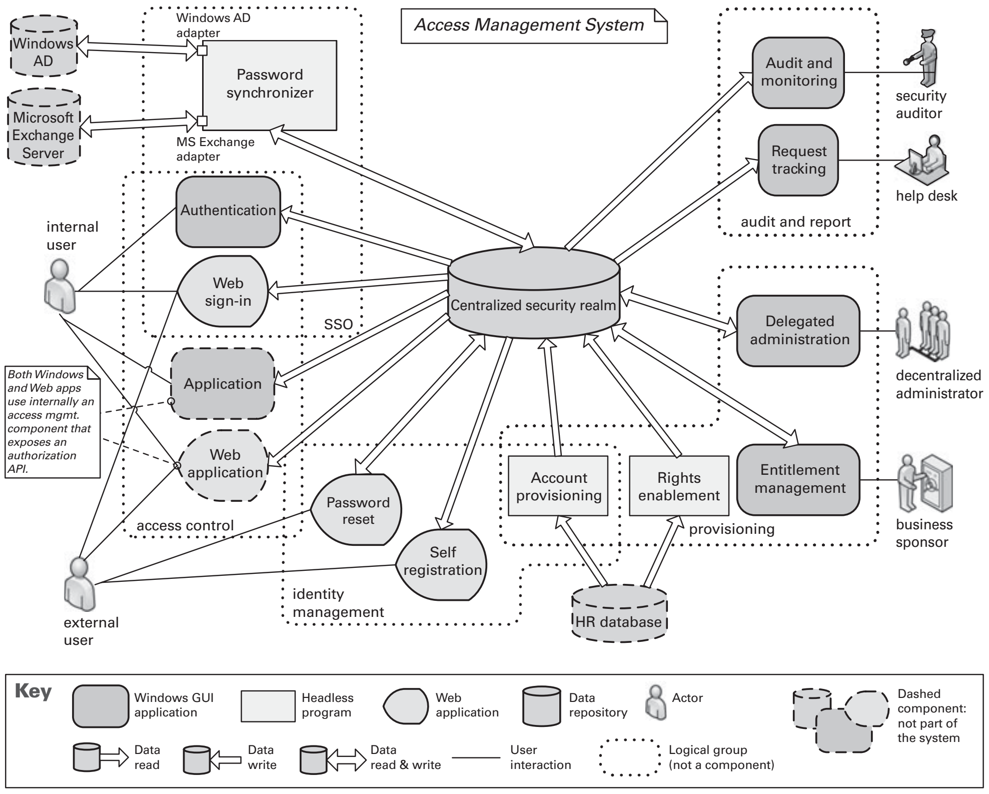


Figure 10.3
A context diagram and a primary presentation combined. Here, the external entities are denoted by symbols identified in the key.

Variation 4: A View with a Multi-part Primary Presentation

Recall that the point of view packets was to keep from having to present a massive (and massively complex) single diagram of interest to almost no one. If for some reason your stakeholders are not well served by dividing the view into view packets, you can document the whole view using a series of diagrams in its primary presentation—the diagrams that would have populated the view packets. The supporting documentation in sections 2–5, then, will explain those diagrams taken as a whole. If you take this option, you'll have to explain how the diagrams relate to each other and/or how to navigate among them.

10.1.3 Avoiding Unnecessary Repetition Across Views or View Packets

Using the view or view packet template naively might result in information being repeated in more than one place, violating our injunction in Section P.5 to avoid unnecessary repetition. Cases include the following:

A module or a component appears in more than one view. For example, the same module might appear in a decomposition, uses, and generalization view. Rather than give its definition, properties, interface, and behavior in the element catalog of every view in which it appears, you can do the following:

- Pick the view in which it seems most appropriate to capture this information, and have the element catalogs in the other views simply refer to it.
- Package the potentially redundant information separately and have all views refer to it or automatically incorporate it.
- In the case of online documentation, have every view's element catalog link to the information.

The context diagram of a child view packet looks like the primary presentation of its parent. Suppose a view packet shows an element without internal substructure, but you create another view packet to show the decomposition refinement of that element—that is, to reveal its internal substructure. Then the context diagram for the second view packet is going to look a lot like the primary presentation for the first. In that case, make the context diagram a simple pointer to the first view packet's primary presentation.

Decomposition refinement is covered in Section 6.1.1.

Global policies apply to many elements. Architects often make decisions that apply to all the elements in a view, such as "All components must write a human-readable message to a

Rozanski and Woods call information like this part of the "common design model." See Section E.3.

log after the start and finish of every transaction." To document information like this, you can do any of the following:

- Add an annotation to the view showing the affected elements.
- Add an entry at the beginning of the element catalog.
- Add an entry to the behavior documentation.
- Explain the global policy in the architecture background section of the documentation beyond views (see Section 10.2).

If you use view packets, you can document global policies in either of two places:

- In the same place that lists the view packets in a view, you can also put information common across all view packets, as a way of "factoring out" commonality and putting it in one place to avoid repetition.
- In a view packet with the greatest scope and least depth. Then all other view packets can "inherit" the common information.

10.2 Documentation Beyond Views

QUOTE

It may take you months, even years, to draft a single map. It's not just the continents, oceans, mountains, lakes, rivers, and political borders you have to worry about. There's also the cartouche (a decorative box containing printed information, such as the title and the cartographer's name) and an array of other adornments—distance scales, compass roses, wind-heads, ships, sea monsters, important personages, characters from the Scriptures, quaint natives, menacing cannibal natives, sexy topless natives, planets, wonders of the ancient world, flora, fauna, rainbows, whirlpools, sphinxes, sirens, cherubs, heraldic emblems, strapwork, rollwork, and/or clusters of fruit.

—Miles Harvey, *The Island of Lost Maps: A True Story of Cartographic Crime* (2000, p. 98)

In many ways, an architecture is to a system what a map of the world is to the world. Thus far, we have focused on capturing the various architecture views of a system, which tell the main story. In the words of Miles Harvey, they are the "continents, oceans, mountains, lakes, rivers, and political borders" of the map that we are drawing. But we now turn to the complement of view documentation, which is capturing the information that applies to more than one view or to the documentation package as a whole. Documentation beyond views corresponds to the adornments of the map, which complete the story and without which the work is inadequate.

10.2.1 A Standard Organization for Documenting Information Beyond Views

Documentation beyond views can be divided into two parts:

1. *Information about the architecture documentation.* How the documentation is laid out and organized so that a stakeholder of the architecture can find the information he or she needs efficiently and reliably.
2. *Information about the architecture.* Here, the information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of the system, the way the views are related to one another, an overview of and rationale behind system-wide design approaches, a list of elements and where they appear, and a glossary and an acronym list for the entire architecture.

Figure 10.4 summarizes documentation beyond views.

Document Control Information

List the issuing organization, the current version number, the date of issue and status, a change history, and the procedure for submitting change requests to the document. Usually this is captured in the front matter. Change-control tools can provide much of this information.

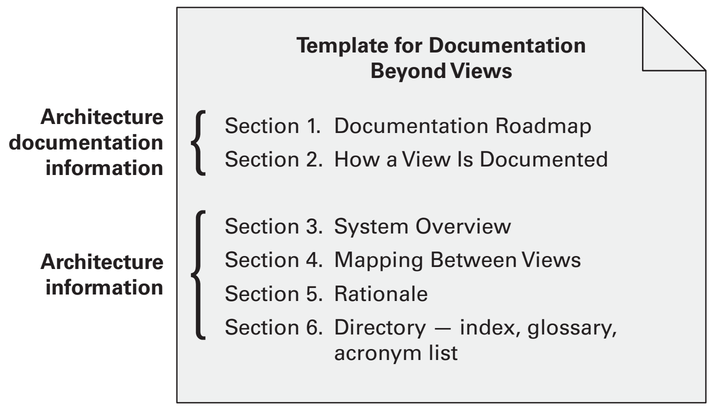

Template for Documentation Beyond Views
Architecture documentation information
Section 1. Documentation Roadmap
Section 2. How a View Is Documented
Architecture information
Section 3. System Overview
Section 4. Mapping Between Views
Section 5. Rationale
Section 6. Directory — index, glossary, acronym list

Figure 10.4
Summary of documentation beyond views

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to," said the Cat.

"I don't much care where," said Alice.

"Then it doesn't matter which way you go," said the Cat.

—Lewis Carroll, *Alice in Wonderland*

Section 1. Documentation Roadmap

The documentation roadmap tells the reader what information is in the documentation and where to find it.

A roadmap consists of four sections:

1. *Scope and summary.* Explain the purpose of the document and briefly summarize what is covered and (if you think it would help) what is not covered. Explain the relation to other documents (such as downstream design documents, or upstream system engineering documents).
2. *How the documentation is organized.* For each section in the documentation, give a short synopsis of the information that can be found there. An alternative to this is to use an annotated table of contents. This is a table that doesn't just list section titles and page numbers, but also gives a synopsis with each entry. It provides one-stop shopping for a reader attempting to look up a particular kind of information.
3. *View overview.* The major part of the roadmap describes the views that the architect has included in the package. For each view, the roadmap gives
   i. The name of the view and what style it instantiates.
   ii. A description of the view's element types, relation types, and property types. This lets a reader begin to understand the kind of information that is presented in the view.
   iii. A description of language, modeling techniques, or analytical methods used in constructing the view.
4. *How stakeholders can use the documentation.* The roadmap follows with a section describing which stakeholders and concerns are addressed by each view; this is conveniently captured as a table. This section shows how various stakeholders might use the documentation to help address their concerns. Include short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification. The maintainer consults the decomposition view to understand the responsibilities of each module in order to identify the modules likely to change. The maintainer then consults the uses view to see what modules use the affected modules (and thus might also have to change)."

To be compliant with ISO/IEC 42010:2007 (see Section E.1), you must consider the concerns of at least users, acquirers, developers, and maintainers.

Section 2. How a View Is Documented

Adopting a standard organization means using it, but also explaining it. This is where you explain the standard organization you're using to document views—either the one described in this chapter or one of your own. It tells your readers how to find information in a view.

If your organization has standardized on a template for a view, as it should, then you can simply refer to that standard. If you are lacking such a template, then text such as that in Section 10.1.2 should appear in this section of your architecture documentation.

Section 3. System Overview

This is a short prose description of the system's function, its users, and any important background or constraints. The purpose is to provide readers with a consistent mental model of the system and its purpose.

The system overview is, strictly speaking, not part of the architecture—that is, it is not part of the designed solution. However, it is indispensable for understanding the architecture. If an adequate system overview exists elsewhere, such as in the overall project documentation, you can incorporate it by reference.

Section 4. Mapping Between Views

Because all the views of an architecture describe the same system, it stands to reason that any two views will have much in common. Helping a reader understand the associations between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the association by providing mappings between views is key to increasing understanding.

The associations between elements across views in a particular architecture are in general many-to-many. For instance, each module may map to multiple runtime elements, and each runtime element may map to multiple modules. Sometimes runtime elements of the system do not exist as code elements at all, such as when they are imported at runtime or incorporated at build or load time. Sometimes modules, such as layers, do not appear at runtime. In general, parts of elements in one view correspond to parts of elements in another view.

There are three ways to document a mapping between views.

1. State a rule that lets a reader know how to look at two views and see the association between elements in each. Naming conventions often provide convenient rules for mappings. The simplest rule is that if an element with the same name appears in different module views, or two different C&C views, it's the same element.
2. View-to-view associations can be conveniently captured as tables, such as the one in Figure 10.5, taken from the

Figure 10.5
An excerpt from a mapping between views

Element in C&C View X	Element in Module View Y
BankAdmin	com.sun.ebank.appclient com.sun.ebank.util stubs from com.sun.ebank.ejb
Web browser	—
WebUI	web com.sun.ebank.web com.sun.ebank.web stubs from com.sun.ebank.ejb
AccountControllerEJB	com.sun.ebank.ojb com.sun.ebank.util
AccountEJB	com.sun.ebank.ojb com.sun.ebank.util
...	...

Duke's Bank is an example application used in Sun's online Java tutorial. See java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html.

Duke's Bank example. List the elements of the first view in some convenient lookup order. The table itself should be annotated or introduced with an explanation of the association that it depicts; that is, what the correspondence is between the elements across the two views. Examples include "is implemented by," for mapping from a component-and-connector view to a module view; "implements," for mapping from a module view to a component-and-connector view; "included in," for mapping from a decomposition view to a layered view; and many others.

Allocation views (discussed in Chapter 5) also show mappings. They map between software structures and nonsoftware structures in the system's environment.

3. The mapping can be shown graphically. An example is shown in Figure 10.6.

For which views should you provide an explicit mapping? (Mappings using naming conventions are implicit.) Begin with these rules of thumb:

- Provide a mapping between the decomposition view and every C&C view.
- Ensure at least one mapping between a module view and a component-and-connector view.
- If your system uses more than one module view, map them to each other.

Use the guidelines in Section 6.5 to help you capture the key architecture decisions.

Section 5. Rationale

This section documents the architectural decisions that apply to more than one view. Prime candidates include documentation of background or organizational constraints or major

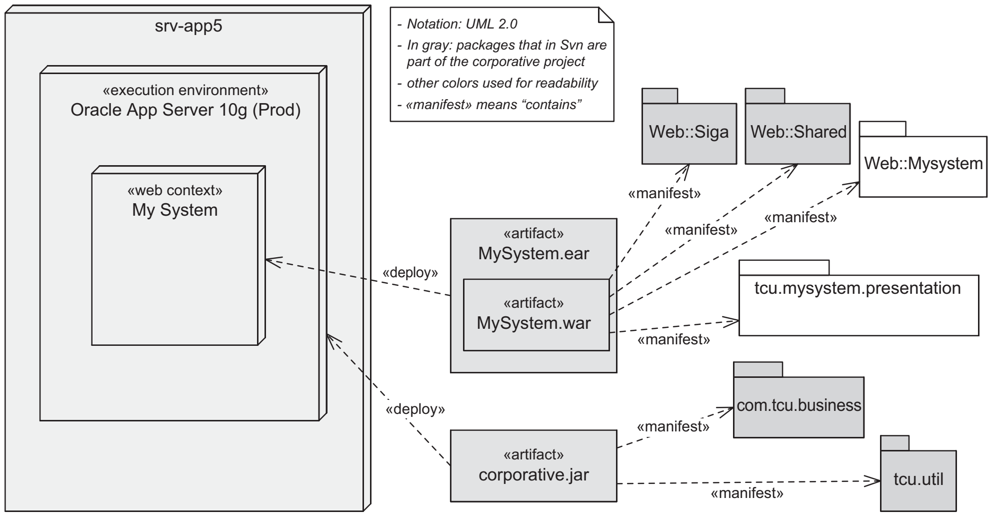


Figure 10.6
A graphical mapping between views. On the far right are UML packages that correspond to modules from a decomposition view. The <<manifest>> relations show the different JAR files (UML artifacts) inside which the modules are bundled for deployment. The <<deploy>> relations show how the JAR files (MySystem.ear, MySystem.war, and corporative.jar) are deployed to the production platform. The platform elements on the left are represented as UML nodes and come from the deployment view of the architecture.

requirements that led to decisions of system-wide import. The decisions about which fundamental architecture patterns or styles to use are often described here.

We are searching for some kind of harmony between two intangibles: a form which we have not yet designed and a context which we cannot properly describe.
—Christopher Alexander

Section 6. Directory

The directory is a set of reference material that helps readers find more information quickly. It includes the following:

- *Index.* Include an index of the elements, relations, and properties that appear anywhere in the architecture documentation. The index should also distinguish between pages where a term is used and the page where it is first defined. A convenient way to do this is to embolden the page number on which the term is defined. (An online search capability may obviate this need.)
- *Glossary.* The glossary defines terms used in the architecture documentation that have special meaning. Often there is a system-wide glossary; if that exists and suffices, it can be incorporated by reference.
- *Acronym list.* Make sure to define the important acronyms you use in the architecture documentation. Again, projects often keep a system-wide acronym list, which can be incorporated by reference.

Include terms in the glossary that your stakeholders won't necessarily know, or terms whose meaning might not be the same among stakeholders.

- *Referenced material.* This is the place to put references to material cited throughout the architecture documentation.

10.2.2 Useful Variations in the Standard Organization for Documentation Beyond Views

See "Coming to Terms: Product-Line Architectures" on page 234, Chapter 6.

The idea for using audiovisual media to document architecture comes from Markus Voelter, coauthor of the (German) book *Software Arkitektur*. He was interviewed during the OOPSLA 2007 conference. You can find the interview at infoq.com/interviews/MarkusVoelterabout-SoftwareArchitecture-Documentation.

Variation 1: Document How to Use the Architecture

You may wish to document "use cases" for the architecture—that is, how to use the architecture to build applications. This is especially helpful if the architecture is meant to be a product-line architecture. People usually learn by internalizing examples, so give a few. Start small; show how to build your application's equivalent of "hello, world!" and then work up. Audiovisual media such as a video or a podcast can be useful in demonstrating the use cases.

Variation 2: Document the Major Design Approaches Taken

Architectures often have dominating "motifs" or design approaches, and elegant ones almost always do. These approaches often take the form of well-known architecture styles or patterns, but other overarching motifs are possible as well. For example, your architecture may dictate that elements implementing new functions all have certain programs on their interfaces, or share data or handle errors in a particular way.

QUOTE

[Architectural p]atterns are a means of documenting software architectures. They can describe the vision you have in mind when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying the system's code. For example, if you know that a system is structured according to the Model-View-Controller pattern, you also know how to extend it with a new function: keep core functionality separate from user input and information display.

—Buschmann et al. (1996, pp. 6–7)

Variation 3: Make a Single Element Catalog for the Whole Architecture

Because the same element might appear in more than one view, there is a danger that its element catalog entries will be redundant. An option is to take all of the element catalogs and merge them into a single one for the whole architecture. This "supercatalog" would belong in the "documentation beyond views" part, because it obviously contains information common to more than one view.

Be careful if you take this option. Describing an element in separate views tends to reinforce the specific role the element plays in the architecture in each view, and that's helpful. For example, a module (in a decomposition view) is usually described in terms of what kinds of changes it encapsulates against, whereas the same module (in a uses view) is described in terms of what it uses and the role it plays in incremental development. The same element, showing up as a component in some C&C view, would be described in terms of its interactions with other elements and its runtime quality attribute properties. Just giving an element a single element catalog entry runs the risk of overlooking what it contributes to each view in which it appears.

Variation 4: Add a Section to Record Open Questions

This is particularly helpful during early development. It provides a "to do" list for the architect, and it informs stakeholders of the major unknowns (and hence possible areas of instability) still in the architecture.

10.3 Documenting a Mapping to Requirements

In many projects, showing how the architecture satisfies requirements is an important part of the documentation. This helps to validate the architecture by showing that

- No requirement was forgotten.
- No requirement was contradicted.
- Every architectural decision is either predicated on at least one requirement or legitimately within the discretion of the architect. (Not every architectural decision satisfies a stated requirement.)

To facilitate validation, the architect records a mapping between architectural decisions and requirements. Anyone interested in a particular requirement should be able quickly to find where in the architecture it is handled.

A detailed mapping to requirements often changes quite frequently during a project's life cycle. Consider capturing the mapping in a database rather than a static document, to facilitate updates and to allow queries and searches to be run.

The mapping can be as detailed as the requirements themselves. Projects with formal requirements documents generally have detailed mappings, whereas projects with informal or fluid requirements (especially Agile projects) will have less detailed mappings. Mappings are conveniently recorded in tables such as in Figure 10.7.

Use Case	UI Screen	Architecture Solution and Architecture Notes
UC1 Create project	New ArchE Project dialog box Navigator view	Project is an inherent abstraction in Eclipse and so is the navigator view. "Garage Door System" would be a project. For each project, a completely independent instance of "Jess rule engine (ArchE core)" is created (see *Fig. d*). All the data for a given project is stored in file "Persisted fact base .txt)" (*Fig. d*). There is one such file per project. Also, the exported design (".xml file" in *Fig. d*) is one per project. There will be a user command to create a project triggered by a menu option, which will activate a specific action handler (*Fig. e*). This action handler will open the New ArchE Project dialog box, which is one of the dialog boxes also in *Fig. e*.
UC2 CRUD scenarios	Scenarios table view Scenarios—static filter dialog box Scenario dialog box Scenario Responsibility Mapping table view	Scenarios table view is one of the views and editors in *Fig. e*. When the user selects the option to create a scenario, a specific action handler (see *Fig. e*) will be activated. This will open the Scenario dialog box, which is one of the dialog boxes in *Fig. e*. The action handler ultimately makes the call to "ArchE core façade," which updates the core. The sequence of steps described for component "ArchE core façade" in Section G takes place.
UC8 Export design	Main menu option: File | Export Design Save File As dialog box	There is an action handler (*Fig. e*) that processes this user command. It uses the Save File As dialog box to ask for the name of the file. Then it calls "Design export" (*Fig. e*), which creates the "exported design" file. The external design tool is activated manually.
Generic (not specific to a use case)	Question to User dialog box Questions table view	A question in the Questions view corresponds to a "QA_" fact in the core. When the user double clicks an entry, there is an action handler responsible for processing the user command (*Fig. e*). It opens the "Question to User" dialog box, which is one of the dialog boxes in *Fig. e*. Answering a question causes the action handler to store a fact in the core.

Figure 10.7
This figure shows an excerpt of the mapping between functional requirements (here, use cases) and architecture for the ArchE system. Because ArchE is a GUI desktop application, most of the use cases are mapped to one or more UI screens (second column). The third column describes how each use case is handled in the architecture. The figures mentioned in the description are primary presentations of the view(s) where the referenced element is defined. (Sections 2.3.6 and 6.6.4 have more information about the ArchE tool.)

You can document a mapping to requirements in any of the following ways.

1. *Put the mapping in a single place in the documentation, a new section in the documentation beyond views.* This option is good for projects that have informal or fluid requirements or that do not require fine-grained accounting of each requirement. Putting the information in one place makes it is easy to update and convenient for validation, and it doesn't clutter the documentation with information that is needed only for a short while by just a few stakeholders. This option most often takes the form of a table that maps a requirements reference to an architecture element, decision, or section of the architecture document. An example is shown in Figure 10.7.
2. *Distribute the mapping throughout the architecture documentation.* You could add a separate section to each view. Or you could overlay every place in the architecture with a tag or adornment that reflects a requirement—every primary presentation, element catalog, context diagram, or variability mechanism. This option is good for projects with fine-grained requirements that map to fine-grained architectural decisions. The architect can record the requirements addressed in the same place and at the same time as the architecture decisions are made. It's also reasonably convenient if the documentation is in an electronic form that allows us to switch the adornments on and off or (even better) automatically extract, collect, and index them to produce an all-in-one-place summary.
3. *Capture the mapping to requirements in a view of its own.* This option is explored in the sidebar "The Requirements Viewpoint." Where might such a view belong in the Style Zoo?
   - You could consider the requirements a "structure" in the software's environment as real as the organizational, development, or execution structures. Thus, a mapping to requirements could be considered a new kind of allocation style, and documented as a kind of allocation view.
   - You could consider the requirements as a set of concerns that crosscut the architecture elements you've designed. Thus, a mapping to requirements could be considered a kind of aspect view. This option is good for projects with fine-grained requirements that map to multiple architectural decisions or elements.

ADVICE

The Requirements Viewpoint

With Peter Eeles

Beyond the approaches for capturing a mapping to requirements outlined in this chapter, there are several precedents for treating the requirements that influence the architecture as a more first-class citizen in terms of an architecture description. For example, there is Kruchten's "Plus One View" of architecture, whose scenarios "are in some sense an abstraction of the most important requirements" (Kruchten 1995). In "The Process of Software Architecting" (Eeles and Cripps 2009), the authors take this thinking further by introducing a more comprehensive requirements **viewpoint**. A requirements view, based on this viewpoint, describes those requirements that have shaped the architecture, and may include functional requirements, quality attribute requirements, and constraints.

You can read about the 4+1 approach in Section E.2.

ISO 42010 defines **viewpoint** as a work product establishing the conventions for the construction, interpretation, and use of architecture views and associated architecture models (ISO/IEC 42010:2007). ISO 42010 is described in Section E.1.

The value of a requirements view, however, is not confined to the identification of the subset of requirements that are deemed to be architecturally significant; the architecture description as a whole should explicitly define how the architecture addresses each of these requirements. Such "traceability" from architecture to requirements can be particularly useful during architecture reviews when the architect needs to justify their decisions, or when the architect needs to remind themselves of the rationale for their decisions.

One place this traceability might be captured is in the rationale section of your documentation. Documenting rationale is described in Section 6.5.

The architecturally significant requirements that you capture in a requirements view may be defined within the current project that is responsible for developing the system, or they may come from outside the project (such as an enterprise architecture or an industry body defining mandatory regulations). The solution architecture is derived from both sets of requirements, as shown in Figure 10.8, with the outermost ring representing requirements defined outside the project, the inner ring representing those requirements defined within the current project (and that align with the requirements defined outside the project), and the center representing the solution architecture that is shaped by both sets of requirements.

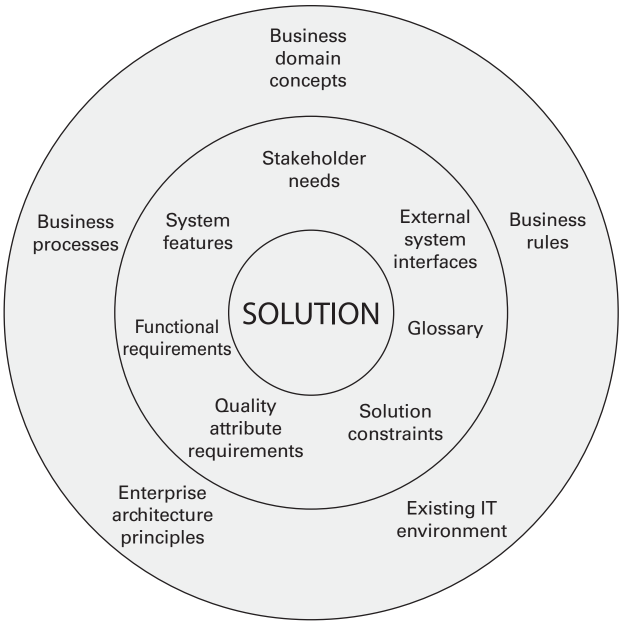


Figure 10.8
Elements of a requirements view (Eeles and Cripps 2009)

Elements defined within the project may include stakeholder needs, system features, interfaces between the system and external entities, functional requirements, a glossary of terms, quality attribute requirements, and any constraints on the solution. Elements defined outside the project, but that also influence the architecture of the system, may include a definition of the key concepts in the business domain, business processes, business rules, principles that inform and guide the way in which the system will be created (such as "buy versus build"), and a description of existing elements that comprise the current IT environment and that may be used by, or constrain, the system under development. These elements constitute the contents that you should capture and document in a requirements view.

PERSPECTIVES

A Mapping to Requirements: You Might Already Have It

Although a mapping between architecture and requirements has important uses, I've observed over many years that it's seldom produced unless contractually required. During early stages of the architecture, too much is in flux, and keeping the mapping consistent is impractical. Toward the end, when the architecture is more stable, nobody has the time (or desire) anymore to put the mapping in. What's a practitioner to do?

Existing products of the architecting process can be used to help define a mapping between requirements and architecture. Let us consider separately requirements for quality attributes and requirements for functionality. Quality attribute requirements are the main drivers of the architecture. Architecture documentation can be thought of to a large extent as describing how quality attribute requirements are supported by the architecture. So the necessary information is there, but it needs to be organized in a way that easily shows which structures and/or behaviors apply to which quality attribute requirement. You can attach a kind of container to each quality attribute requirement. Depending on the tool support you have, the container might contain links to the supporting diagrams (behavioral or structural) or other relevant sections of the architecture document.

What about functional requirements? Usually an architecture document is not deemed complete if it does not contain some description of how at least the "essential" requirements are supported by the architecture, where *essential* means those requirements that are the primary purpose of the system. So, for example, if the system is a communication system, then one of the essential functional requirements would be to establish connections. An essential functional requirement of a sensor system would be to capture data and make it usable for end users or other systems. Usually, even a large system does not have very many essential requirements. In many cases architects will document those essential functional requirements as use cases. To these you can attach some behavioral descriptions, such as sequence diagrams or collaboration diagrams, that describe how the architecture elements with assigned responsibilities interact with each other to provide the functionality needed.

—F.B.

10.4 Packaging the Architecture Documentation

10.4.1 Packaging Schemes

You can use the templates in this chapter to create architecture documentation structured in a variety of ways. Which option you choose will depend on the size of the system, how you wish to package it for its stakeholders, and your organization's standards and practices.

Produce All One Package

Here is a suggested ordering for producing a single architecture document:

1. Document control information
2. Documentation roadmap
3. How a view is documented
4. System overview
5. Views
6. Mapping between views
7. Rationale
8. Directory

Produce Separate Documents

If a single document seems too unwieldy, there are a number of ways to divide the documentation into more manageable chunks. One way is to break the views out into their own document. If you do that, then it makes sense to put the "How a view is documented" section with them. Both documents should have document control information.

Other arrangements are possible, such as putting every view in its own document, grouping views by category (module, C&C, allocation), or breaking out the mapping to requirements into its own document. You may also wish to divide the documentation along architectural lines—a document per subsystem, for example. Consult your stakeholders to find out what would work best for them.

Produce Documentation Packages from Different Views

A view is a representation of a set of element types and relation types applied to a system. If you break a view into view packets, then every view packet in that view shares the same underlying type. This is not always the most convenient documentation package to give to a stakeholder. Stakeholders are often interested in, for example, shallow overviews of the whole system, or holistic (that is, multi-view) insight and detail about a particular subsystem or layer.

To serve stakeholders like these, you can put together a package of information including view packets from different views. For example, you can assemble a package that provides a broad overview of the architecture by showing high-level view packets from various views. This can be followed by view packets that show deeper and deeper levels of the architecture, again across views. The documentation roadmap for documentation like this must tell the reader how to navigate through the view packets.

ADVICE

Building an Architecture Overview Presentation

Sooner or later, every architect has to give an oral overview of an architecture, backed up by slides. Once built, the presentation is likely to be used often, introducing the architecture to managers, developers, sponsors, evaluators, customers, and even visitors. What should such a presentation contain? The goal is to help the audience gain an appreciation of the problem, see the solution(s) chosen, understand why they were chosen, and gain confidence that the architecture is the right one for the job.

Here's an outline for a five-part, one-hour overview containing anywhere from 20 to 35 slides.

1. *Problem statement: 2–3 slides.* State the problem the system is trying to solve. List driving architecture requirements, the measurable quantities you associate with them, and any existing standards/models/approaches for meeting them. State any technical constraints, such as a prescribed operating system, hardware, or middleware.
2. *Architecture strategy: 2 slides*. Describe the major architecture challenges. Describe the architecture approaches, styles, patterns, or mechanisms used, including what quality attributes they address and a description of how the approaches address those attributes.
3. *System context: 1–2 slides*. Include one or two whole-system context diagrams that clearly show the system boundaries and other systems with which yours must interact.
4. *Architecture views: 12–18 slides*. Use the views you've chosen as the backbone of the presentation. For each view, include the top-level (that is, system-wide) primary presentation and, depending on the amount of detail you want to include, perhaps a few refined primary presentations as well. Naturally, each should include a notation key.

   An overview presentation is the one case for which a cartoon does not have to be accompanied by the supporting documentation, but you will want to have it available for answering questions.

   For each slide showing a primary presentation, make a couple of accompanying slides that explain (a) how the architecture shown supports the functionality and achieves the system qualities that reside with that view and (b) the rationale for choosing that design. You may wish to annotate or color some of the cartoons to show programmatic information about the elements, such as which elements are provided by third parties, the state of an element's development, the amount of risk posed by an element, or the scheduled delivery or other milestone of an element. You need not include every view in the presentation, but you should include at least one module view, at least one C&C view, and at least one allocation view.

Where views can be straightforwardly mapped to each other, include slides that do so. This will be very useful in conveying the overall picture.

5. *How the architecture works: 3–10 slides*. Trace up to three of the most important use cases. If possible, include the runtime resources consumed for each use case. You should be able to extract the traces from your behavior documentation in the form of, for example, sequence diagrams or statecharts.

   Show the architecture's capacity for growth with a trace of up to three of the most important change scenarios. If possible, describe the change impact—estimated size/difficulty of the change—in terms of the changed elements, connectors, or interfaces.

   Depending on the importance of each item, consider tracing a scenario that illustrates any of the following: concurrency, failure recovery, error propagation, or key end-to-end data flows. Again, you should be able to extract this information from your behavior documentation.

You may wish to have the following slides available to answer questions or to help discussion but not make them part of the standard presentation:

- The set of stakeholders for the documentation and a sketch of the concerns and information needs of each (2–3 slides)
- Glossary (1–2 slides)

Preface the whole package with a title slide, sprinkle outline slides throughout to let the audience follow the outline of the presentation, end with a "for further information" slide, and you're done.

A good presentation can help an architect in many ways. Recorded on video, it can free the architect from having to brief new hires or low-ranking visitors. It can be handed to junior designers as a way to groom them for technical leadership positions. And it helps establish a consistent vision of the architecture throughout an organization, which makes every architect's life easier.

10.4.2 Online Documentation, Hypertext, and Wikis

These days, Web-based documentation is becoming the norm. Hyperlinking your documents can provide easy navigation in and among them, as well as instant access to related documents, definitions, catalogs, and external references. Hyperlinking also relieves you of all the problems associated with keeping multiple copies of documents around: You make one copy and link to it wherever the information contained in it is needed. (Recall the second rule of sound documentation: Avoid unnecessary repetition.)

Prepared using a Web-based documentation tool, a document can be structured as linked Web pages. Compared with documents written with a text-editing tool, Web-oriented documents typically consist of short pages (created to fit on one screen) with a deeper structure. One page usually provides some overview information and has links to more-detailed information. When done well, a Web-based document is easier to use for people who just need to have some overview information. On the other hand, it can become more difficult for people who need detail. Finding information can be more difficult in multi-page, Web-based documents than in a single-file, text-based document, unless a search engine is available.

Shared documentation, in a Web-based environment, allows you to increase collaboration among stakeholders and avoid unnecessary repetition.

Using readily available tools, it's possible to create a shared document that many stakeholders can contribute to. The hosting organization needs to decide what permissions it wants to give to various stakeholders; the tool used has to support the permissions policy. In the case of architecture documentation, we would want all stakeholders to comment on and add clarifying information to the architecture, but we would want only architects to be able to change it, or at least provide architects with a "final approval" mechanism.

In a shared document environment, where every user is allowed to (and is encouraged to) contribute, the workload is distributed—an effect that is typically seen as very positive. The concepts of author (one who creates and maintains the document) and reader (one who only reads the document) are diminished. Readers feel more empowered, and hence have a stronger stake in the documentation. A special kind of shared document is a wiki. A wiki is a collection of Web pages designed to enable anyone with access to contribute or modify content.

A **wiki** is a collection of Web pages designed to enable anyone with access to contribute or modify content, using a simplified markup language. (Wikipedia 2010c)

COMING TO TERMS

Wiki

A **wiki** is a Web site that allows users freely to create and edit Web-page content using any Web browser. A wiki offers an alternative to using an editing tool paired with a configuration management tool. A wiki, however, is not an alternative to modeling or drawing tools.

Everyone who can use a Web browser and fill out Web-based forms can view and edit the content of a wiki page. A wiki supports hyperlinks and has a simple text syntax for creating new pages and links between pages "on the fly." If you can suppress your desire for fancy formatting, wiki is a fast-to-learn, easy-to-use, and intuitive editing environment. It allows novice users to produce fairly nice-looking Web pages that are immediately available to all other users. A wiki also allows the reorganizing of content. Pages can be reordered, and new pages can be created to show the existing content in a different order. When anyone makes changes to a page, everyone can see what was changed.

ADVICE

If you are going to use a wiki as the repository of your software architecture, there are some practical considerations and guidelines that may help. Here is a list of recommendations for the configuration and day-by-day use of your wiki-based architecture document.

- The first step is to create a new wiki or define a page for the architecture document in an existing wiki. It is possible to automatically enforce a specific structure for a wiki page, but not for an entire wiki. But it is highly advisable to follow a standard organization like the ones in this chapter, enforced by convention. Create the initial page of the architecture documentation as a list of links to the main topics.
- Create one wiki page for each architecture view, and a template for that page. Follow a convention to name the views, so that it is easy to remember the names when creating links (the view and its wiki page should share the same name).
- Create one wiki page for each mapping between views, so that each mapping can be edited independently.
- If you are using a drawing tool, such as Visio or PowerPoint, create one file for each diagram or one file for each architecture view. Prefix the file with the name of the view, replacing spaces with a standard character.
- A wiki does not provide an editorial feature similar to the Track Changes option in Word. The wiki option is to add comments to the discussion page. An alternative that has proven to be effective when reviewing a wiki page is this process:

1. Copy the wiki page to a blank Word document.
2. Activate the Track Changes option.
3. Edit the Word document and add comments as needed.
4. Send the Word document to the author of the wiki page, who then can change the wiki page based on the edits and comments in the review.

- It is very common for an element in the architecture to appear in more than one view. Create the description of that element in a separate page and include it by reference in the element catalog of all pages that contain that element.
- If you already have documentation created in Word and want to migrate it to a wiki, there are macros/scripts that can help. To find them, do a Web search for "Word2Wiki" or "WordToWiki."

The process of including a description by reference is called *transclusion*.

10.4.3 Configuration Management

What book on documentation would be complete without stressing the importance of keeping your documentation complete and up to date? Recall the sixth rule of sound documentation: "Keep documentation current but not too current." Nothing is worse than opening a set of architecture documentation and trying to figure out if it represents the most recent version of the system.

Documents should be dated and versioned. If someone is looking at several figures, it should be obvious at a glance which figures are from the same version of the system.

You probably think of software configuration management systems more in terms of keeping track of the code associated with your project, but we recommend that you think of the documentation that you are creating as software too, and treat it just as carefully as you do the code that derives from it.

In fact, the versions of the documentation and the code should refer to each other. When looking at code, it should be easy to determine which version of the architecture it reflects.

10.4.4 Follow a Release Strategy

Your project's development plan should specify the process for keeping the important documentation, including architecture documentation, current. The architect should plan to issue releases of the documentation to support major project milestones, which usually means far enough ahead of the milestone to give developers time to put the architecture to work.

Projects follow a rhythm, a drumbeat of incremental milestones leading to eventual full release, and then entry into maintenance and sustainment. Early in the life cycle, the drumbeat tends to be much faster than after the system is released or brought to market. Plan your releases of architecture documentation to support the next beat of the drum. For example, the end of each iteration or sprint or incremental release could be associated with providing revised documentation to the development team.

PERSPECTIVES

Presentation Is Also Important

Throughout this book, we focus on telling you what to document. We do not spend much, if any, time on how it should look—but not because form is unimportant. Just as the best-designed algorithm can be made to run slowly by insufficient attention to detail during coding, so too the best-designed documentation can be made difficult to read by insufficient attention to presentation details: for example, the style of writing, fonts, types and consistency of visual emphasis, and the segmenting of information.

We have omitted these issues not because we think they are unimportant but because presentation details are not our field of expertise. Universities offer master's degrees in technical communication, in information design, and in other fields related to the presentation of material. We have been busy being software engineers and architects and have never been trained in presentation issues. Having denied expertise, however, I am now free to give some rules of thumb.

- *Adopt a style guide for the documentation.* The guide should specify such particulars as fonts, numbering schemes, conventions with respect to acronyms, captions for figures, and other such details. The guide should also describe how to use the visual conventions discussed in the next several points.
- *Use visually distinct forms for emphasis.* Word processors offer many techniques for emphasis. Words can be bold, italic, large, or underlined. Using these forms makes some words more important than others.
- *Be consistent in using visual styles.* Use one visual style for one purpose, and do not mix purposes. That is, the first use of a word might be italicized, and a critical thought might be expressed in bold, but do not use the same style for both purposes, and do not mix styles.
- *Do not go overboard with visuals.* It is usually sufficient to use one form of visual emphasis without combining them. Is bold less arresting to you than bold red italic? Probably not.
- *Try to distinguish different types of ideas with different visual backgrounds.* In this book, we attempted to put the main thread of discussion in the body of

the book, with ancillary information as sidebars. We also made the sidebars visually distinct, so that you would know at a glance whether what you were reading was in the main thread or an ancillary thread.

The key ideas with respect to presentation are consistency and simplicity.

- Use the same visual language to convey the same idea: consistency.
- Do not try to overwhelm the user with visuals; you are documenting a computer system, not writing an interactive novel: shoot for simplicity.

The goal of the architecture documentation, as we have stressed throughout this book, is to communicate the basic concepts of the system clearly to the reader. Using simple and consistent visual and stylistic rules is an important aspect of achieving this goal.

—L.B.

PERSPECTIVES

Tooling Requirements

The benefits of having architecture documentation need to outweigh the costs of producing it, or it won't be produced. Throughout this book we have argued forcefully for the benefits but have thus far paid little attention to the costs. The lower the cost of the documentation, the more activities for which it becomes worthwhile to produce documentation.

Although the cost of documentation is primarily a human cost, the cost is strongly related to the existence of appropriate tools that support the humans in the production process.

What would an ideal tool to support the documentation process look like? If such a tool were to exist, the cost of producing documentation would be much lower than it is today. This sidebar will discuss requirements for a tool that reduces the human cost of producing documentation.

There are two primary requirements for an ideal documentation tool.

1. The tool must generate documentation at the push of a button from information already in the tool as a result of design or other project activities. No information necessary for the documentation should need to be added to the information already necessary for project activities.
2. As the system evolves, the documentation simultaneously evolves.

The requirement that the tool generates documentation at the push of a button means that the tool must have information about each of the views in a variety of different granularities and must have the beyond views information as well. Being able to make connections between views requires that the tool have a sophisticated association capability. That is, given any two entities in the tool, the architect can link them together with an appropriate annotation in a matter of one or two button pushes or drags.

Limiting the number of user actions necessary means that the documentation tool must be very flexible in terms of when a user of the tool can make a linkage. This in turn means that no particular process should be imposed, because a user may be in the middle of one activity when a stray thought arrives about another linkage that should be made. Making the linkage and returning to the original activity should require only one or two button pushes.

How would the tool help with specific parts of architecture documentation? Here are some examples.

- *Rationale*. One portion of the documentation template is the rationale for particular decisions. If the rationale results from an automated analysis, then linking the documentation tool to the analysis tool will make the necessary information available. Otherwise, the rationale would need to be entered manually or linked from an existing document.
- *Mapping to requirements*. Another set of information necessary for the documentation is linkage to requirements fulfillment. The requirements information should also be available to the user of the documentation tool with one or two button pushes.
- *Elements and properties*. Entities within the tool should have a collection of attributes that include the views of which they are a portion and the properties needed for analysis. Hierarchies of entities will allow for viewing a design at different points in the spectrum of design. Good navigation and search capabilities are a must, and will allow for many different organizations of the entities.

When the system is updated, it raises the possibility that the documentation may need to be updated. The updating of the documentation should be as painless as possible. Work-flow techniques can be used to determine whether the architecture documentation needs to be updated to reflect changes in the system and to alert the person(s) responsible for the documentation of a particular portion of the architecture.

Because the documentation should be available at the push of a button and because it will be evolving, the documentation tool will need to construct the view packets dynamically. Different views at different granularities will require different subsets of the information available. Information should be self-contained with links to context and related information but at a size that will localize changes to affect only the necessary information. For example, a module's responsibility might be decomposed into smaller responsibilities assigned to submodules. A responsibility can then be linked to its parent and to its children in a decomposition of responsibilities. It can also be linked to the modules where it is realized, but it should exist as an independent entity, so that changes to its description will have the fewest ripples.

Finally, the tool should support multi-user access and editing. Developers will need to access the tool to understand the architecture. Different people will be responsible for modifying different portions of the documentation. Some people will need only a simple drawing tool, and so the tool should either provide that or be able to digest and process diagrams from such tools. Development is more and more a global matter, and so the tool should support access from around the globe.

As may be apparent by now, the ideal documentation tool will be just one portion of an integrated design, project management, requirements, analysis, and documentation tool, because these are roles that some of the consumers of the documentation will fulfill. There may be additional roles that require integration with other functions that occur during the development of the system.

The type of tool we described here does not currently exist, although the ideas are drawn from various existing tools or prototypes. We hope the description here will help speed the introduction of such a tool.

—L.B.

10.5 Summary Checklist

- A complete architecture documentation package consists of a set of views, along with documentation of the information that applies to more than one view.
- Document the views, and documentation beyond views, using the templates in this chapter (tailored for your own use if necessary) or one of your own making.
- A view consists of a primary presentation, an element catalog, a context diagram, a variability guide, and rationale. The part after the primary presentation is called supporting documentation.
- Documentation beyond views consists of document control information, a documentation roadmap, a view template, a system overview, mapping between views, rationale, and a directory.
- Document the mapping between views by using a table showing how elements of one view correspond to elements of another. You can also show the mapping graphically.
- A view packet is a portion of a view that you would want to show to a single stakeholder. A view packet includes a primary presentation depicting a part of the system, and supporting documentation that explains the primary presentation.
- Choose a scheme for capturing the mapping to requirements based on the nature of the requirements and stakeholder need.

10.6 For Further Reading

To read more about documenting architectures using a wiki, see the technical note “Experience Using the Web-Based Tool Wiki for Architecture Documentation,” by Felix Bachmann and Paulo Merson (2005). You can also search for “wikis for software engineering” to see the results of workshops and research in this area.

11 Reviewing an Architecture Document

With David Emery and Rich Hilliard

QUOTE

The iconic American poet Emily Dickinson craved pointed reviews of her work. Here is how she asked for one from a literary confidant in 1862 (Weeks and Flint 1957):

Mr. Higginson, Are you too deeply occupied to say if my verse is alive?

The mind is so near itself it cannot see distinctly, and I have none to ask.

Should you think it breathed, and had you the leisure to tell me, I should feel quick gratitude.

If I make the mistake, that you dared to tell me would give me sincerer honor toward you.

I inclose [*sic*] my name, asking you, if you please, sir, to tell me what is true?

That you will not betray me it is needless to ask, since honor is its own pawn.

The prologue presented seven rules for sound documentation. The rules concluded with this prescription:

> **Review documentation for fitness of purpose.** Only the intended users of a document will be able to tell you whether it contains the right information presented in the right way. Enlist their aid. Before a document is released, have it reviewed by representatives of the community or communities for which it was written.

This chapter describes a procedure for doing just that. Like all prescriptions in this book, you should use just as much of it as you think will be beneficial, given the realities and circumstances of your organization and project. For example, Scrum projects often require a complete product (including requirements, design, code, and test results) every 30-day sprint, with a planning session at the beginning and an evaluation at the end. The question sets given in this chapter could serve as a quick checklist for evaluating the documentation products along the way.

To be clear, we are not discussing how to evaluate an *architecture*; there are several existing methods for that already. Rather, we are evaluating the *documentation* of an architecture (one purpose of which may be to support an architecture evaluation exercise).

11.1 Steps of the Procedure

This is a six-step procedure. The first step establishes the "why, when, and who" of the review. Subsequent steps provide the "what" and "how."

Step 1: Establish the purpose of the review. An architecture document (AD) review establishes whether the AD is fit for some specific purpose by a set of identified stakeholders. Stating that purpose will focus the review participants and direct the review. The questions you'll ask about the document will be different depending on the purpose you have in mind. The sidebar "Why Review an Architecture Document?" provides some examples of why the AD might be reviewed.

Choose one or more of these purposes or craft your own. A review purpose can be stated as a scenario that describes how a particular stakeholder can successfully use the AD to carry out part of his or her job.

It is likely that any AD will need to be fit for more than one purpose, and hence the review will be multi-faceted. The alternative is several smaller reviews, each with a single purpose.

Knowing the "why" will help you identify the "who." As part of establishing the purpose, identify the stakeholders of the AD who should be represented in the review.

Knowing the "why" will also tell you the "when." No matter what life-cycle process you're using, various review purposes will align with certain project stages or milestones. To give an idea of this, Table 11.1 shows a loosely defined set of broadly

Table 11.1 Typical life-cycle phases and the AD reviews that are appropriate for each

Project Phase	Typical Activities	Review AD for . . .
Concept	• Identifying stakeholders' needs • Exploring concepts, propose viable solutions • Analyzing alternative architectures • Preparing an architectural concept (such as when assembling a bid for a contract) • Communicating between acquirers and developers as a part of contract negotiations	• Capturing the right stakeholders and concerns • Support for proposal

Table 11.1 Typical life-cycle phases and the AD reviews that are appropriate for each (*continued*)

Project Phase	Typical Activities	Review AD for . . .
Development	• Refining system requirements • Creating solution description • Building system or systems • Verifying and validating system	• Support for conformance to a normative specification • Support for evaluation • Support for development • Support for input to generation and analysis tools • Support for judging implementation conformance to architecture • Support for project planning
Utilization	• Operating the system to satisfy users' needs	• Support for help in tracking down operational errors
Support	• Providing sustained system capability	• Support for system evolution in concert with the architecture and the associated business planning for evolution

applicable project phases, the typical activities in each phase, and what you might wish to review the AD for in each case. Of course, the particular life-cycle model your project uses will lead to different phases, activities, and reviews. Carry out the review with enough spare time to allow the AD to be modified after the review to serve its purpose.

ADVICE

Why Review an Architecture Document?

- *Review the AD for conformance to a normative specification.* This kind of review is intended to discover if the AD conforms to some normative specification that has been imposed on it. The focus is on the AD itself; the architecture it describes is deemphasized. For example, the AD may be required (or claim) to conform to ISO/IEC 42010:2007, the U.S. Department of Defense Architecture Framework (DoDAF), The Open Group Architecture Framework (TOGAF), the Federal Enterprise Architecture Framework (FEAF), or other standards, guidelines, or templates mandated by the developing organization. A conformance review will see if it does.
- *Review the AD for its ability to support use of the architecture for its intended purpose.* This kind of review is carried out to see if stakeholders of the architecture can use the AD to do their jobs. The focus is on how well the AD describes the architecture. Understandability and usability of the AD are important review criteria. Examples include the following:

  - Can the AD support downstream software design, development, and evolution? Can the AD enable effective communications among organizations involved in the development, production, fielding, operation, and maintenance of a system? Here, important concerns are comprehension and completeness, as well as the precise conveyance of global design concepts (and their rationale) so that all groups have the same mental model of the architecture.
  - Can the AD support project planning, budgeting, and scheduling? Here, the emphasis is on the ability to predict the size, complexity, risk, reuse opportunities, and requirements for specific expertise.
  - Can the AD support the development of a group of systems sharing a common set of features and built from a common set of core assets? Here, the emphasis may be on the specification in the AD of commonalities, points of variation, and variation mechanisms built into the architecture.
  - Can the AD support preparation of acquisition documents (such as requests for proposals and statements of work)? Can the AD support communications between acquirers and developers as a part of contract negotiations? Here, the important concern is comprehension, so that all groups have the same understanding of the architecture plan and the architecturally significant requirements.
- *Review the AD for its suitability to support architecture evaluation or analysis.* This kind of review is carried out to see if the AD provides sufficient information to be able to predict system qualities by examining or analyzing the architecture. Examples include the following:
  - Can the AD support an architecture evaluation using a method such as the SEI Architecture Tradeoff Analysis Method (ATAM)? Here, important concerns are attention to quality attributes required of and provided by the architecture, as well as evidence of feasibility—namely, that the architecture can in fact be built under the budget and schedule allotted.
  - Can the AD support analysis of alternative architectures? The AD must have the qualities necessary to evaluate an architecture by itself but also include sufficient rationale to provide in-depth qualitative insight about whether the architecture is well suited to take the organization into the future, so it can be compared with other candidates.

Chapter 9 describes how much information of various kinds is usually needed by different kinds of stakeholders.

Step 2: Establish the subject of the review. This step involves identifying the types of artifacts, the versions of the artifacts, their sources, and the degree of completeness of the artifacts necessary to conduct the review. Obviously, the AD needs to be available. Use the purpose(s) laid out in step 1 to establish the artifact collection required and then gather them for the review. For example, if the AD is being reviewed for conformance to a standard or to a framework, the normative requirements of the standards/framework should also be available. In all cases, make sure that all reviewers are working from the same version(s) of the artifact(s).

Step 3: Build or adapt the appropriate question set(s). This step involves identifying the questions that your review will put to the AD. If you already have a set of questions that meets the purpose of your review, you can use it (perhaps with some modification). If not, you will have to construct it. Organizing questions as question sets allows them to be reused by providing contextual information about the purpose and stakeholder concerns that need to be addressed, as well as guidance for obtaining and interpreting the results. Later in this chapter we present a number of example question sets, each one designed to serve a review purpose. If you choose to use existing question sets, they need to be tailored for the purposes of the review. Questions that are not relevant can be omitted. General questions can be made more specific according to the technology of the project (for example, references to data persistence may be replaced by references to an Oracle database). The question set(s) that you pick will suggest a particular approach, and the questions need to be formulated appropriately. For example, will you use the active design review technique, a questionnaire or checklist given to stakeholders, some sort of automated or measurement-based analysis, or some other approach?

A **question set** groups questions that collectively address a narrowly focused purpose for an AD review. Besides the questions themselves, a question set contains information to allow a user to ensure the question set is appropriate and to use it effectively. This information includes the name, purpose, stakeholders and concerns, respondents, expected answers, criticality, and advice.

Active design reviews are explained in the "Coming to Terms" sidebar on page 380, in this chapter.

Step 4: Plan the details of the review. Planning involves setting a date for the review, as well as deciding on the time frame and the basic format of the review. The time frame might allow as much time as needed to answer questions or only a limited amount of time, in which case the questions need to be prioritized. Time and resources will affect the format and "weight" of the review. How the results will be communicated needs to be determined and could affect the format and weight of the required answers.

This step also involves identifying the actual review participants (not just abstract stakeholder roles) and securing their participation. An initial assignment of questions to the reviewers responsible for asking them and the stakeholders responsible for supplying the answers can be made at this time. As the review is conducted, the initial priorities and stakeholder assignments may change as a deeper understanding of the documentation is gained and the reviewers probe further into applicable areas.

There's no limit on what can be inspected, so inspections should be limited to those items where the benefit is likely to be worth the cost. Consider the context (rigor vs. scope vs. resources vs. time vs. costs) and be practical. A less formal walk-through process may be adequate.

—Watts S. Humphrey (1989, p. 172)

This step also involves handling the logistics for the review: time and place of meeting(s), paying for everyone's time, providing read-ahead materials, and so on.

Step 5: Perform the review. Performing the review involves posing the questions to the stakeholders involved in the review and gathering their answers. Depending on the specific approach chosen, this might involve an individual objective review, where stakeholders also play the role of the reviewer and pose questions to themselves; or an inspection, where a separate review team poses questions to the stakeholders. Inspections could take the form of an all-hands gathering, a number of one-on-one meetings, or something in between; the meetings could be face to face, or distributed and remote, using (for example) online virtual meetings. After the results are gathered, the evaluation considerations and criteria are applied, as defined by the chosen question set(s). Although the reviewers can make some preparations, not all the important issues can be known beforehand. These issues need to be determined in the initial part of the review and will influence the questions and artifacts used as the reviewers dig deeper in these areas.

Step 6: Analyze and summarize the results. The intent of this step is to aggregate the answers to the questions and then make a qualitative determination of the overall impact of the AD against the stakeholders and concerns. Results are not likely to be a simple pass/fail but rather a more nuanced conclusion concerning specific problems in specific parts of the AD.

COMING TO TERMS

Active Design Reviews

In an active design review, reviewers are actively engaged to exercise the artifact they are reviewing, not just look it over and scan for defects. Here is what David Weiss, one of the creators of the active design review technique, has to say about them:

Starting in the early 1970s I have had occasion to sit in on a number of design reviews, in disparate places in industry and government. I had a chance to see a wide variety of software developers conduct reviews, including professional software developers, engineers, and scientists. All had one thing in common: the review was conducted as a (usually large) meeting or series of meetings at which designer(s) made presentations to the reviewers, and the reviewers could be passive and silent or could be active and ask questions. The amount, quality, and time of delivery of the design documentation varied widely. The time that the reviewers put in preparation varied widely. The participation by the reviewers varied widely. (I have even been to so-called reviews where the reviewers are cautioned not to ask embarrassing questions, and have seen reviewers silenced by senior managers for doing so. I was once hustled out of a design review because I was asking too many sharp questions.) The expertise and roles of the reviewers varied widely. As a result, the quality of the reviews varied widely. In the early 1980s Fagin-style code inspections were introduced to try to ameliorate many of these problems for code reviews. Independently of Fagin, we developed active design reviews at about the same time to ameliorate the same problems for design reviews.

Active design reviews are designed to make reviews useful to the designers. They are driven by questions that the designers ask the reviewers, reversing the usual review process. The result is that the designers have a way to test whether or not their design meets the goals they have set for it. To get the reviewers to think hard about the design, active reviews try to get them to take an active role by requiring them to answer questions rather than to ask questions. Many of the questions force them to take the role of users of the design, sometimes making them think about how they would write a program to implement (parts of) the design. In an active review, no reviewer can be passive and silent.

We focus reviewers with different expertise on different sets of questions so as to use their time and knowledge most effectively. There is no large meeting at which designers make presentations. We conduct an initial meeting where we explain the process and then give reviewers their assignments, along with the design documentation that they need to complete their assignments.

Design reviews cannot succeed without proper design documentation. Information theory tells us that error correction requires redundancy. Active reviews use redundancy in two ways. First, we suggest that designers structure their design documentation so that it incorporates redundancy for the purpose of consistency checking. For example, module interface documentation may include assumptions about what functionality the users of a module require. The functions offered by the module's interface can then be checked against those assumptions. Incorporating such redundancy is not required for active design reviews but certainly makes it easier to construct the review questions.

Second, we select reviewers for their expertise in certain areas and include questions that take advantage of their knowledge in those areas. For example, the design of avionics software would include questions about devices controlled or monitored by the software, to be answered by experts in avionics device technology, and intended to insure that the designers have made correct assumptions about the characteristics, both present and future, of such devices. In so doing, we compare the knowledge in the reviewers' heads with the knowledge used to create the design.

I have used active design reviews in a variety of environments. With the proper set of questions, appropriate documentation, and appropriate reviewers, they never fail to uncover many false assumptions, inconsistencies, omissions, and other weaknesses in the design. The designers are almost always pleased with the results. The reviewers, who do not have to attend a long, often boring, meeting, like being able to go off to their desks and focus on their own areas of expertise, with no distractions, on their own schedule. One developer who conducted an active review under my guidance was ecstatic with the results. In response to the questions she used she had gotten more than 300 answers that pointed out potential problems with the design. She told me that she had never before been able to get anyone to review her designs so carefully.

Of course, active reviews have some difficulties as well. As with other review approaches, it is often difficult to find reviewers who have the expertise that you need and who will commit to the time that is required. Since the reviewers operate independently and on their own schedule, you must sometimes harass them to get them to complete their reviews on time. Some reviewers feel that there is a synergy that occurs in large review meetings that ferrets out problems that may be missed by individual reviewers carrying out individual assignments. Perhaps the most difficult aspect is creating design documentation that contains the redundancy that makes for the most effective reviews. Probably the second most difficult aspect is devising a set of questions that force the reviewer to be active. It is really easy to be lured into asking questions that allow the reviewer to be lazy. For example, "Is this assumption valid?" is too easy. In principle, much better is "Give 2 examples that demonstrate the validity of this assumption, or a counterexample." In practice, one must balance demands on the reviewers with expected returns, perhaps suggesting that they must give at least one example but two are preferable.

Active reviews are a radical departure from the standard review process for most designers, including architects. Since engineers and project managers are often conservative about changes to their development processes, they may be reluctant to try a new approach. However, active reviews are easy to explain and easy to try. The technology transfers easily and the process is easy to standardize; an organization that specializes in a particular application can reuse many questions from one design review to another. Structuring the design documentation so that it has reviewable content improves the quality of the design even before the review takes place. Finally, reversing the typical roles puts less stress on everyone involved (designers no longer have to get up in front of an audience to explain their designs, and reviewers no longer have to worry about asking stupid questions in front of an audience) and leads to greater productivity in the review.

11.2 Sample Question Sets for Reviewing the Architecture Document

Posing and answering questions in a review is, of course, the heart of the matter. This section discusses what is involved in the formation of question sets—groups of questions that, together, address a narrowly focused purpose for an AD review. Besides the questions themselves, a question set must also contain information to allow a user to make sure the question set is appropriate and use it effectively, as shown below:

1. **Question Set Name.** As an artifact to be reused, give the question set a name by which it can be referred.
2. **Purpose.** What review purpose does the question set address?
3. **Stakeholders and Concerns.** Who are the stakeholders, and which of their concerns are being addressed by the questions? Making stakeholders and concerns a first-class dimension of an AD review effectively elaborates the purpose of the question set and informs the formulation of the questions. (While we can't expect all of an architecture's stakeholders to participate in a review, we want to make sure that all of the important stakeholder *roles* are represented.)
4. **Questions.** This section contains the questions that constitute the question set. For each question, give the following information:
   a. **Respondents.** To whom should each question be posed? The questions might be addressed to the person speaking for the AD. Usually this will be the architect. The questions might be addressed to reviewers checking the understandability of the AD by using it to answer questions about the architecture it describes. For instance, if the AD should support project planning (a purpose) and is being reviewed for such (using a "project planning" question set), the respondents would include those concerned with project planning—technical managers. If the AD should support development and is now being reviewed for that, the respondents will certainly include key developers. Questions about the AD itself can be answered by examining the AD or analyzing it with a tool (for example, automatically checking to make sure that every cross-reference is defined).

      The person(s) to whom a question is posed may or may not be the same as the stakeholder(s) whose concern the question addresses. Review participants may be proxies for stakeholders.

   b. **Expected Answers.** What answer(s) are we looking for? A question set will also involve formulating a set of considerations and criteria to help the reviewers evaluate the AD based on the answers they receive. For example, they might wish to understand not just the answers given by the reviewers but also how much difficulty the reviewers had coming up with those answers. They might wish to understand the criteria the stakeholders used for why they answered "Yes, we're happy" or "No, we're not happy."

A concise statement of the purpose can often be useful to capture in the name; for example, "Ready to support development."

The respondents should not be shown the expected answers, to avoid biasing their answers.

c. **Criticality.** How critical is each question? The "wrong" answer to some questions might halt a project until it's resolved, whereas the "wrong" answer to other questions might merely be something to watch over time. The questions should come with guidance (perhaps a weighting) to help establish their importance.

5. **Advice.** Provide additional useful information on how and when the review should be conducted. You might relate experience gained through using the question set in a prior review.

Figure 11.1 provides a sample template that can be used when constructing a question set.

Following are a few example question sets to serve specific AD review purposes. (Some questions might apply to more than one question set.) They are written in different styles to illustrate the ways a question set may be used. For example, the example question set for capturing the right stakeholders is written in the active design review style, and the questions are really directions to stakeholders to use the AD for some purpose. The other example question sets are written as if an interviewer is questioning a stakeholder. These could be adapted to an active design review style or for the purposes of an individual objective review. Some questions that can be answered yes or no are serving as filters, and when the answer is yes, it is appropriate to ask follow-up questions of the form, "How do you know?"

Figure 11.1
Template for a question set

1. Question Set Name

2. Purpose

3. Stakeholders and Concerns

4a. Questions (*organized by respondents*)

4b. Expected Answers

4c. Criticality

5. Advice

11.2.1 Example Question Set for Capturing the Right Stakeholders and Concerns

The Views and Beyond approach to architecture documentation uses the explicit identification of stakeholders and their concerns to determine which views to include in the AD. Explicitly identifying stakeholders and concerns is also a requirement of ISO/IEC 42010:2007. Therefore, a useful review of the AD examines its choice of stakeholders and concerns to ensure that the important ones are accounted for. Such a review could be usefully carried out quite early, when the stakeholders and concerns are documented but before the rest of the AD is created.

See Section 9.1 for more information about stakeholders and their documentation needs.

The questions in the example question set below are formulated using the active design review technique.

1. Question Set Name: Capturing the right stakeholders and concerns

2. Purpose

The purpose of this question set is to gauge the appropriateness of the architect's list of stakeholders and concerns and to review how well the stakeholders believe their interests and concerns have been captured.

3. Stakeholders and Concerns

All those with a substantial stake in the architecture should be involved or have their roles and concerns represented.

4a. Questions

Respondents: All stakeholders

1. State your stakeholder role. List the set of concerns you have that pertain to the architecture whose AD is being reviewed.
2. Find and record all places in the AD where your stakeholder role is listed as being covered.
3. Find and record all places in the AD where your concerns are listed as being addressed.
4. Find and record all places in the framework used (if any) where your stakeholder role is listed as being addressed.
5. Find and record all places in the framework used (if any) where your concerns are listed as being addressed.
6. Record all concerns you have that are not listed as being covered in either the AD or any framework being used or that are listed in an unclear fashion. For each, state the impact of this omission or misunderstanding on project success.
7. For each of your concerns as a stakeholder, find and record the places in the AD where that concern is addressed (not just listed). Explain why you do or do not believe that the concern will be satisfied by the architecture.
8. Find and record the place in the AD that prioritizes the concerns. Explain why you do or do not agree with it.
9. Record important stakeholders that you are aware of that are not listed and whose concerns are not represented in the AD.
10. State how you know that the architecture satisfies the concerns of the missing stakeholders and where this information can be found in the AD.

Respondents: Architect

11. Show where in the AD the generic stakeholders and concerns required by the framework in use (if any) have been listed and addressed.
12. State how you produced the list of stakeholders and their concerns.

4b. Expected Answers

Each stakeholder should be able to find where in the AD (and framework, if any) their role and concerns are listed and their concerns are addressed. Every relevant stakeholder and concern should be covered; missing ones should be noted. All concerns should be tied to at least one stakeholder. The architect should provide a convincing argument that the process for identifying stakeholders and their concerns was adequate.

In addition to producing satisfactory answers, the respondents should also note the ease or difficulty in using the AD to answer the questions.

4c. Criticality

Questions revealing missing stakeholders or missing concerns are the most critical.

5. Advice

This question set is especially appropriate for an active design review, in which an all-hands meeting is not required. Individual reviewers representing different stakeholder roles and concerns can be engaged separately, perhaps even by telephone or electronic mail, to make sure their concerns are addressed in the AD.

By contrast, however, a similar review was carried out as a two-day all-hands workshop for a large U.S. defense project. The first half-day was used to present ISO/IEC 42010:2007 terms and approaches. This was a long review because the project is large. Some 30–40 people were involved, and even then some stakeholder communities were overlooked.

On a small distance-learning project, a review for this purpose took 6 hours with a dozen people: 6 architects and 6 stakeholders. The agenda devoted 2–3 hours to the procedure and 3 hours to concerns.

11.2.2 Example Question Set for Supporting Evaluation

When an architecture is subjected to a comprehensive evaluation, the AD is the vehicle for communicating the architecture to the reviewers, or at least substantiating the architect's presentation of the architecture. Therefore, it is useful to review the AD before an architecture evaluation takes place to see if it contains the necessary information to allow the evaluation to go forward. By extension, such a review determines whether the architecture is ready (complete enough) to be evaluated.

1. Question Set Name: Supporting evaluation

2. Purpose

The purpose of this question set is to determine whether the architecture is ready to be evaluated. This helps ascertain whether evaluation stakeholders have sufficient information to do their job and know when their job is completed. The emphasis is on the artifacts needed for analysis.

3. Stakeholders and Concerns

The business manager is the spokesperson for the business goals the system is meant to support. These goals include what the customer wants to build and the objectives of the organization building the system. The business manager is concerned with how the technical solution supports the business goals.

The architect is concerned with whether the AD supplies sufficient information for analysis and how usable the AD is in supporting an evaluation. The architect would like to use the AD to determine whether one alternative is better than another in terms of technical considerations, difficulty, and risk.

The team preparing to conduct an architecture evaluation is concerned with knowing what to evaluate and whether the AD supplies sufficient information for analysis.

4a. Questions

Respondents: Business manager, Architecture evaluation team

1. Are the business goals the system must satisfy clearly articulated and prioritized?
2. Is it clear how the business goals determine the requirements? Is there a mapping between business goals and requirements? Are the requirements prioritized according to business importance?
3. Is there traceability between the business goals and the technical solution? That is, can you navigate from business goals to architecturally significant requirements (ASRs), to technical decisions and associated risks, and finally back to implications on achieving the business goals?
4. What criteria are used to determine whether the architecture is supporting the business goals?
5. How might the system change over its lifetime of deployment (including retiring the system)?

Respondents: Architect, Architecture evaluation team

6. Is the context of the system (or subsystem) clearly defined?
7. Have the stakeholders and their concerns been clearly defined?
8. Have the requirements, constraints, standards, and quality-assurance policies been clearly defined?
9. Are the ASRs which the system must satisfy clearly articulated and prioritized according to their impact on the architecture?
10. Are the ASRs clear and unambiguous? Are they “testable”? Have they been prioritized?
11. Is it clear which techniques the architect used to achieve the ASRs? Have alternatives that were considered but not chosen been documented?
12. Is it clear how the architecture fulfills the other requirements that are not ASRs?
13. Has the AD identified the key decisions? If so, where are they?
14. Has the AD captured the rationale for key decisions? If so, where?
15. Can you describe the runtime resources consumed for each concern that affects the operation of the system?
16. Can you describe the change impact (estimated size/difficulty of the change) for those modifiability concerns that lead to changed design elements?
17. Can you determine the views necessary to analyze each ASR? Does the AD provide the views necessary to cover the ASRs?
18. Within each view, are its models clear? Are its models well-defined by the viewpoint? Do the models address the ASRs? Which ASRs are addressed by the models in this view (to the extent that the model provides enough information to determine whether the ASRs have been satisfied)?

Viewpoints, models, and correspondences are concepts in the ISO/IEC 42010 standard, discussed in Section E.1.

19. Are all ASRs addressed by either one or more models or one or more correspondences among models?
20. Have the architects done any preliminary analysis? Have these results (including architecture issues and risks) been articulated? Where?
21. How will the architecture be introduced and retired within the business?
22. Is the current document complete in the sense that all the information is documented? If not, are there placeholders for what has yet to be documented along with descriptions of what still needs to be worked out?
23. Can you navigate through the material during the evaluation to show the decisions made to address stakeholders' concerns?

Respondents: Architecture evaluation team

24. Are the concepts and notations underlying the AD clearly explained (for example, is there a glossary of terms, key for diagrams)?
25. Have the scope and the objectives of the evaluation been clearly defined?
26. Is the context of the system (or subsystem) to be evaluated clearly defined?
27. Have the stakeholders and their concerns for the system (or subsystem) to be evaluated been clearly defined?
28. For each view, do you understand how to evaluate its contents?
29. For correspondences across views, do you understand how they are represented and how to evaluate them for accuracy and completeness?
30. Are the views sufficiently complete to support the intended analysis? Can you work around gaps identified by the architects?

4b. Expected Answers

The business manager and the architect should provide a convincing argument that the documentation captures the important analysis artifacts that allow one to navigate from business goals to architecturally significant requirements, to technical decisions and associated risks, and finally back to their implications on achieving the business goals.

The evaluation team should have a clear understanding of the objectives and scope of the evaluation. That understanding will determine what AD artifacts are needed and to what degree.

4c. Criticality

Questions revealing missing analysis artifacts (for example, architecturally significant requirements, architecture decisions) are the most critical.

Questions indicating incompleteness or ambiguity in conducting the analysis are also critical.

5. Advice

Depending on the scope of the evaluation, there could be some overlap with the "Question set for supporting development." Analysis could include "buildability" or "feasibility in building the system as the customer describes it." There is no overlap when evaluation is more narrowly scoped in the sense of identifying decision points and the rationale for selecting alternatives. In this case, the AD is treated as a sketch that shows alternatives rather than a blueprint from which to build the system.

If the AD uses frameworks and viewpoints, then a question set for reviewing the choice of framework and viewpoints could be created and used in conjunction with this review. If the AD does not use these concepts explicitly, some of the questions could still be used to understand the documentation.

The business manager and the architect share their answers to the questions with the evaluation team. The evaluation team may answer the questions separately to varying degrees of detail in order to validate the results.

The set of questions will be tailored according to the scope and objectives of the evaluation (any combination of the system, stakeholders, ASRs, views, and decisions).

11.2.3 Example Question Set for Supporting Development

Architecture has value by driving a conforming implementation—that is, that the developers can follow the specifications and constraints of the architecture. The purpose of a review for supporting development is to determine whether there is enough information in the architecture for the development stakeholders to do their jobs. A closely related task is to determine if the AD is sufficient to determine whether a system's implementation actually conforms to the architecture described in the AD. The emphasis there is on the ability of the AD to identify *conformance points* for the implemented system, with the expectation that a subsequent review or audit will actually determine conformance of the system to the architecture (described by the AD).

1. Question Set Name: Supporting development

2. Purpose

The purpose of this question set is to determine whether the AD contains enough information to "drive" a conforming implementation. This helps ascertain whether development stakeholders have sufficient information to do their job and know when their job is completed. The focus is less on analysis and more on comprehension and completeness of the AD.

3. Stakeholders and Concerns

Architects are concerned that their AD is ready to pass to developers.

Designers and implementers are concerned with knowing what to build—that is, what they must do in order to implement the architecture.

Software managers are concerned with estimating and/or predicting needed development resources (budget, schedule).

Developers are concerned with when to enter test.

Testers are concerned with whether the AD supplies sufficient information to enable architecture-based testing and to determine when to exit test.

QA stakeholders are concerned with whether the AD supplies sufficient information to enable quality assurance and to know when they are done. A special kind of QA stakeholder is the "conformance checker," concerned with how to tell whether an implementation conforms to the architecture.

Integrators are concerned with whether the AD supplies sufficient information to plan integration.

Fielders are concerned with whether the AD supplies sufficient information to plan deployment.

Customers and program managers have indirect concerns about whether the AD is usable by developers and how the architecture is constrained by existing components.

4a. Questions

Respondents: Software manager

1. Can you identify the full set of implementation units (elements to be implemented)?
2. Can you determine which units require development (and integration and test) resources?

3. For each unit requiring development, can you make predictions in terms of use of development resources, variance, and risk?
4. Can you determine development dependencies between implementation units?
5. Can you identify runtime dependencies between units?
6. Can you lay out a schedule for this development?
7. Can you lay out a schedule for an architecture prototype?
8. Can you tell if you have enough development resources?
9. Does the AD overconstrain the stakeholders (such as developers, integrators)?
10. Does the AD identify opportunities for parallel development? Can you identify units that can be implemented in parallel?

Respondents: Designers and implementers (including unit testers)

11. Can you identify the allowed and prohibited dependencies between implementation units?
12. Can you identify applicable architecture constraints, rules, principles, styles, patterns, and so on, on units or their aggregation?
13. Can you navigate from an implementation unit to its associated requirements (formal, derived, quality, performance, and design constraints)?
14. Can you determine a test approach for the set of implementation units?
15. Can you determine approaches for error handling, resource management, human-computer interaction, data management and persistence, variation and variability (for example, across a product line or evolution over time), and so on?
16. Can you determine what is likely to change and how it impacts your design?
17. Can you tell how solid each decision is?
18. Can you tell what needs to change as the result of entering a new cycle?
19. Do you understand how conformance to the AD will be determined?
20. Does the AD identify opportunities for parallel development? Can you identify units that can be implemented in parallel?

Respondents: Integrators and fielders

21. Can you identify what units must be integrated?
22. Can you determine the resources needed to operate the unit?
23. Can you determine the integration test obligations?
24. Can you identify runtime (such as load, elaboration) dependencies between units?
25. Do you understand how conformance to the AD will be determined?

Respondents: Testers (not unit testing, but rather architecture-based testing)

26. Can you determine which units can be cost-effectively tested in isolation?
27. For each unit, can you determine what is needed (for example, data, special hardware, other units) to test it?
28. For each unit, can you determine what constitutes test success criteria?
29. Can you test the system as a whole?

Respondents: QA stakeholders

30. Is the AD baselined?
31. Is there a history of changes to the AD?
32. Does the AD identify key decisions?
33. Does the AD capture the key decisions and design rationale?

34. Does the AD articulate “open decisions” deferred to implementation?
35. Are inconsistencies known and documented?
36. Are there known associations between each view’s models and developed/delivered artifacts? (For example, if we have a “deployment view” in the architecture, do we have a “packing list” for the system?)
37. Are specific conformance points identified in the views? For each such point, do we know which view and model captures this information and which artifact/artifacts must conform? Is there a documented method for checking conformance (for example, inspection, developer test, formal qualification test)?
38. What questions, concerns, or issues have the developers raised during their work? How are these captured/resolved in the AD? How has the AD changed in response to these concerns?
39. Are the test approaches and artifacts consistent with the AD? (Could include formal trace or an informal assessment. This is particularly associated with “use case” kinds of views, where you want the testers to test the known use cases that the architecture should have addressed.)
40. Is there a formal process for establishing conformance?
41. Does the content of the AD support this process?

Respondents: All stakeholders

42. Can you identify open, partially resolved, or unresolved issues in the AD?
43. Can you identify where automated tools will be used? Does the AD have the right content that is in a format that can be processed by the tools?

4b. Expected Answers

In all cases, the stakeholders should provide a convincing argument that the documentation captures the important artifacts that allow one to implement the architecture.

In addition to producing satisfactory answers, the respondents should note the ease or difficulty in using the AD to answer the questions.

4c. Criticality

Questions revealing incompleteness or misunderstanding of artifacts are the most critical. In this case, the AD is treated as a blueprint from which to build the system or to which the built system must conform.

5. Advice

This question set might overlap with a question set that reviews the AD for its ability to support an architecture evaluation, in that the evaluation could analyze for “buildability” or “feasibility in building the system as the customer describes it,” which are, of course, among the developer concerns addressed here.

A subset of the question set may be used in a more specialized review for supporting planning.

11.2.4 Example Question Set for Reviewing for Conformance to ISO/IEC 42010

This review assesses whether the AD conforms to the requirements of ISO/IEC 42010, *Systems and Software Engineering—Architecture Description.*

FOR MORE INFORMATION

ISO/IEC 42010:2007 is the ISO adoption of ANSI/IEEE 1471-2000, and it is identical to that earlier standard. At the time of this book's publication, a joint revision of ISO/IEC 42010 and ANSI/IEEE 1471 was ongoing. The questions in this question set reflect the expected form and content of the ISO/IEC 42010 revision, including new topics such as architecture frameworks and model correspondences.

1. Question Set Name: Reviewing for conformance to ISO/IEC 42010

2. Purpose

This question set is used to assess the conformance of the AD to the requirements of the international standard ISO/IEC 42010. Conformance to the standard may be a prerequisite to acceptance of the AD as a deliverable or to other reviews.

3. Stakeholders and Concerns

Architects, acquirers, and architecture analysts all have the following concern: Does my AD meet all of the conformance points of the standard? Can conformance be verified?

4a. Questions

Respondents: Architects

1. Does the AD contain the appropriate administrative and overview data (date of issue, version status, issuing organization, change history, summary, scope, context, glossary, and references)?
2. Does the AD contain architecture documentation required by the using organization?
3. Who are the specific stakeholders for this AD? Is there evidence the architect has given consideration to these stakeholder classes: users of the system, system acquirers, system developers, and system maintainers?
4. Are the stakeholders' concerns captured? Does the AD show evidence of having considered the purposes of the system; the suitability of the architecture to achieve those purposes; the feasibility of constructing and deploying the system; the potential risks of system to its stakeholders throughout its life cycle; and the maintainability and evolvability of the system?
5. Is every stakeholder and every concern covered by at least one viewpoint?
6. Is each viewpoint identified? Is there a definition for each viewpoint used in the AD? Does each viewpoint definition include: viewpoint name; identification of the stakeholders addressed by that viewpoint; the architectural concerns framed by that viewpoint; and the model kinds used by the viewpoint? For each model kind, are the conventions, including any notations, languages, modeling techniques, and analytical methods, defined?
7. If the viewpoint comes from an external source, is it fully defined and identified in that source? Is there an association between that viewpoint and the stakeholders' concerns? Are models/modeling techniques identified? Does the viewpoint contain analysis techniques, rules, or constraints?
8. Is there a view for each viewpoint? Does the view correctly use/implement the models required by its viewpoint? Does the view cover the system under review? Is the view-viewpoint relationship one-to-one?

9. Does each view contain an identifier, introductory information, configuration information as defined by the using organization, and one or more models?
10. Are any known inconsistencies between views documented?
11. Are there correspondence rules? For each such rule, is there at least one correspondence satisfying each rule?
12. Does the AD cite an existing architecture framework? Is each viewpoint in the framework used in the AD? Does the AD capture all of the framework's correspondence rules?
13. Does the AD contain the rationale for its architectural decisions, such as
    - Selection of viewpoints and models/modeling techniques?
    - Correspondence rules?
    - Key decisions captured within each view?

Respondents: Acquirers and architecture analysts

14. Is the set of stakeholders and concerns complete?
15. Is the set of viewpoints both complete and minimal?
16. Is the set of correspondence rules (if used) appropriate?
17. Are the views complete? Do they communicate the key decisions?
18. Is the set of correspondences complete?
19. Does the rationale capture sufficient information to assist reviewers and architecture analysts in understanding the architecture and its decisions?
20. Do the set of viewpoints and/or the selected architecture framework match contractual requirements and/or institutional practices?

4b. Expected Answers

Positive answers are expected, as well as the ability of the participants to point out specific places in the AD to justify their positive answers.

4c. Criticality

For the purpose of ascertaining conformance, all requirements in ISO/IEC 42010 are of equal importance, and all are mandatory. (There are no tailoring options in the standard.)

5. Advice

"Complete" here is expected to be a value judgment in the review, rather than any formally determined property. The stakeholders need to understand the context (including resource constraints) as part of evaluating "completeness." Generally, "complete" should be interpreted as "good enough to meet our expectations for this system within the context in which we are developing it." These rules should not be required as having the architecture description account for every (software equivalent of a) nail in the structure.

Each item chosen above directly maps to conformance points in ISO/IEC 42010:2007. However, the terms in this section are taken directly from ISO/IEC FCD 42010:2010.

11.3 An Example of Constructing and Conducting a Review

This section shows an example of constructing and carrying out an AD review. The review was conducted to see if a project's AD was sufficient to support an architecture evaluation.

- **Step 1: Establish the purpose of the review.** The purpose was to evaluate an AD to see if it was sufficiently complete and consistent to support a formal evaluation of the architecture. The chosen architecture evaluation method was the Architecture Tradeoff Analysis Method (ATAM), which uses a trained evaluation team to assess the consequence of architecture decisions in light of quality attribute requirements and business goals. The evaluation team interacts with the project's architect and senior designers, as well as important architecture stakeholders. The purpose of the AD review is to ensure that those analysis artifacts (architectural decisions, quality attributes, and business goals) are well documented.

The "why" establishes the "who," and in this case the architecture evaluation team became the architecture documentation review team.

The "why" also establishes the "when," and in this case, we conducted the review in time for everyone to present their results at the evaluation kick-off meeting. This meeting is a standard part of the ATAM, in which the evaluation team meets, discusses the architecture, agrees on team roles, and makes the go/no-go decision.

- **Step 2: Establish the subject of the review.** The ATAM requires the client to provide a presentation of the architecture as well as the architecture documentation before the evaluation exercise commences. In fact, these are used to make a go/no-go decision: If the architecture is not sufficiently mature, it cannot be reliably evaluated. In this case, the client provided the ATAM team leader with copies of both the presentation and the architecture document (in this case called a "software design document") one month before the scheduled beginning of the evaluation.
- **Step 3: Build or adapt the appropriate question set(s).** The ATAM go/no-go criteria led us to select the question sets for (1) capturing the right stakeholders and concerns and (2) supporting evaluation (see Section 11.2). If a framework such as TOGAF had been used, the question set for reviewing the choice of the framework and associated viewpoints would have been included as well. We did not involve the business manager or the architect in our review, but we did have available a viewgraph presentation from each of them that described the business drivers and architecture, respectively.
- **Step 4: Plan the details of the review.** The evaluation team leader did double-duty as the review team leader. That involved making sure all team members had the appropriate artifacts: the architecture documentation and presentation and the right question sets. Since our evaluation team

was geographically distributed, with members in five different U.S. cities, we arranged the review process so that members could work independently, at their own pace and schedule. Everyone was asked to report their findings at the kick-off meeting.

- **Step 5: Perform the review.** The reviewers checked the AD to make sure that the following are documented: a list of the stakeholders' roles and concerns, the criteria the architect used to produce that list, and how the architecture satisfies the concerns. Each reviewer applied the questions against the AD and recorded their answers. They were e-mailed to the review leader before the kick-off meeting, so that he could get a sense of the findings and make a preliminary judgment as to the suitability of the AD.
- **Step 6: Analyze and summarize the results.** Each member of the evaluation team provided answers to the team leader. It took each person anywhere from one to four hours to complete the question set. The evaluation leader examined the answers, and in less than an hour was able to glean a team consensus that the AD, while not perfect from the perspective of the question set, was sufficient to support an evaluation. This impression was confirmed during a subsequent telecon. The team leader, satisfied that the AD was sufficiently developed to support an evaluation, decided to proceed. During the evaluation itself, 12 scenarios were analyzed. In every case, the architect was able to use architecture information from the AD (in the form of a viewgraph presentation) to walk through the scenarios and explain how the architecture did or did not support them. In two cases, the evaluation team asked where a particular piece of key information was documented, and the architect was able to show its location in the AD. In all cases, the AD supported the analysis, suggesting that the team's conclusion as to its suitability was well founded.

11.4 Summary Checklist

- Review architecture documentation to ensure that the architecture is effectively captured in a form that allows stakeholders to understand and use the architecture in the way it was intended.
- Choose questions based on the purpose of the review; three examples of why the AD might be reviewed include: conformance to some normative specification, suitability to support use of the architecture for its intended purpose, and suitability to support architecture evaluation or analysis.

- Organize questions as question sets so they can be reused by providing contextual information about the purpose and stakeholder concerns that need to be addressed, as well as guidance for obtaining and interpreting the results.

11.5 Discussion Questions

1. Suppose that you have been asked to review architecture documentation for conformance to an architecture framework such as DoDAF or TOGAF. Given this purpose, whom among the stakeholders would you invite and when in the life cycle would you hold the review? What questions or question sets (if any) from those in this chapter would you reuse? What additional questions would you ask?
2. Discuss the advantages and disadvantages of conducting an AD review as a separate method or as a procedure that is part of an existing method. For a project you have in mind, which would you choose and why?
3. Knowing that you intend to review the architecture documentation, how might this influence your choosing the views and building the documentation package? What criteria would help you decide whether to incorporate reviews as part of the documenting process or to conduct a separate review activity upon completion of the documentation?

11.6 For Further Reading

An active design review (Parnas and Weiss 1985) is a technique for carrying out guided documentation-based reviews. Some of the example question sets provided in this report use the active design review approach.

SEI Active Reviews for Intermediate Designs (ARID) (Clements, Kazman, and Klein 2002) is a method for performing a scenario-based stakeholder-centric review of a portion of architecture. The review is focused on whether the design is sufficient for the software developers who will use it. ARID is based on active design reviews and the ATAM. The elements of the ARID method could be focused on documentation to create a method to review documentation in line with the approach described in this chapter. The question set for supporting development is especially relevant. Active design reviews are a most promising starting point. For example, active design reviews call for recruiting different kinds of reviewers for different kinds of reviews. Support staff is often used, for instance, to review for document consistency and completeness and for conformance to a template. Active design reviews naturally go with the idea of a spectrum of review purposes, either as separate reviews or as multiple purposes of a single review.

Architecture-centered software project planning (ACSPP) (Paulish 2002) is another approach (like ARID) where a portion of the architecture documentation is given to the developers who are asked to use it. In this case, they are asked to take four hours to sketch an initial design of the subsystem they are tasked with developing and to fill out a sheet of metrics documenting the time and resources needed for the development effort. The question set for supporting development would be relevant for that part of the effort that involves understanding the architecture.

The SARA report (SARA 2002) presents a useful generic model for evaluating software architectures, and it is a good starting point for reading on this subject. Particular methods, such as the SEI's ATAM, can be thought of as special cases of the SARA model.

Epilogue: Using Views and Beyond with Other Approaches

The word architecture goes back through Latin to the Greek for "master builder." The ancients not only invented the word, they gave it its clearest and most comprehensive definition. According to Vitruvius—the Roman writer, whose Ten Books on Architecture is the only surviving ancient architectural treatise—architecture, is the union of "firmness, commodity, and delight"; it is, in other words, at once a structural, practical, and visual art. Without solidity, it is dangerous; without usefulness, it is merely large-scale sculpture; and without beauty . . . it is not more than utilitarian construction.

—Marvin Trachtenberg and Isabelle Hyman, *Architecture: From Prehistory to Post-Modernism/The Western Tradition* (1986, p. 41)

This book has presented guidance, which we call the Views and Beyond approach, for assembling a package of effective, usable documentation for a software architecture. Using a basic set of concepts (views and styles) and an organizing principle (module views, component-and-connector (C&C) views, allocation views), we have shown how to document a wide range of architecture-centric information: from structure to behavior to interfaces to rationale. The book stands on its own as a complete handbook for documentation.

But the book does not exist in a vacuum. Other writers, on their own or under the auspices of large organizations or standards bodies, have prescribed specific view sets or other approaches for architecture. There is now an ISO standard for architecture documentation. Many people are writing about how to document an "enterprise architecture." It may not be clear whether the advice in this book is in concert or in conflict with these other sources. In some cases, it isn't clear whether there's a relationship at all.

The purpose of this chapter is to answer the following questions:

How do I use the Views and Beyond approach if I want to produce software architecture documentation that . . .

1. **. . . is compliant with the ISO standard for architecture documents?**
2. **. . . adheres to the Rational Unified Process 4+1 approach to documentation?**
3. **. . . uses the Rozanski/Woods viewpoint set?**
4. **. . . supports an Agile development project?**

Over and above these software-oriented variations, this chapter also covers the U.S. Department of Defense Architecture Framework (DoDAF), which is not intended for software architectures but nevertheless is sometimes pressed into service in that way.

E.1 ISO/IEC 42010, *née* ANSI/IEEE Std 1471-2000

With Rich Hilliard and David Emery

E.1.1 Overview

What ISO 42010 calls an *architecture description* is what the Views and Beyond approach calls an *architecture document*. See "Coming to Terms: Specification, Representation, Description, Documentation" on page 10, in the prologue, for why we chose the term we did. In this section, we will defer to the ISO's terminology.

ISO/IEC 42010 (or "eye-so-forty-two-ten" for short) is the ISO standard, *Systems and software engineering—Architecture description*. The first edition of that standard was published in 2007. It was the fast-track adoption by ISO of IEEE Std 1471-2000, which was developed by an IEEE working group drawing on experience from industry, academia, and other standards bodies between 1995 and 2000. ISO 42010 is centered on two key ideas: a conceptual framework for architecture description and

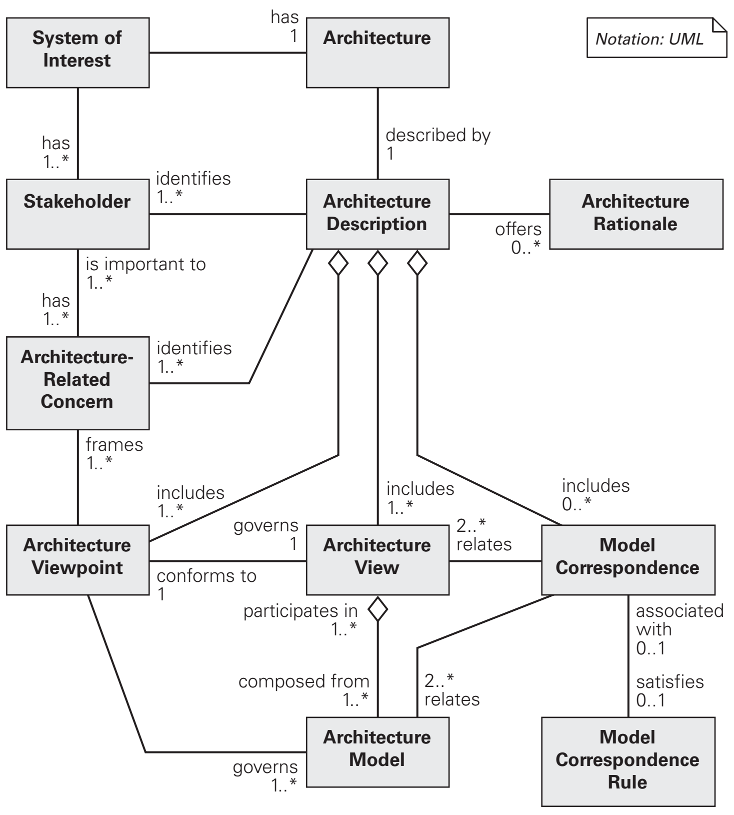


Figure E.1
Core concepts of ISO/IEC 42010:2007

a statement of what information must be found in any ISO 42010-compliant architecture description.[1]

Under ISO 42010, as in the Views and Beyond approach, **views** have a central role in documenting software architecture. The architecture description of a system includes one or more views.

ISO 42010 defines a **view** as a "work product representing a system from the perspective of architecture-related concerns."

Figure E.1 illustrates the core concepts of architecture description in the standard:

Under ISO 42010, an *architecture description* is a *work product*—a concrete artifact (which could be a document or repository) that documents the architecture of a **system of interest**. A system of interest exists in some *environment* (containing other

ISO 42010 defines **system of interest** as encompassing "individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest."

1. Now that ISO has adopted the IEEE standard, the two organizations will undertake a coordinated update to both the ISO and IEEE standards. This section describes the standard as reflected by Committee Draft 1 (CD1), dated January 2009 (ISO/IEC CD1 42010 2009). The material in this draft had undergone substantial technical review within the working group but had not been formally balloted at the time this chapter was written.

systems, humans, and so on), which motivates, constrains, and interacts with the system of interest. ISO 42010 requires that an architecture description contain the following:

- Identification of the stakeholders for the architecture and the system of interest
- Identification of the architecture-related concerns of those stakeholders
- A set of architecture **viewpoints** defined so that all of the stakeholder concerns are covered by that set of viewpoints
- A set of architecture views, such that there is one view for each viewpoint
- A set of architecture models from which the views are composed
- Architecture rationale to record key decisions

ISO 42010 defines **viewpoint** as a work product establishing the conventions for the construction, interpretation, and use of architecture views and associated architecture models.

ISO 42010 is based on the following tenets:

1. Architecture is an abstraction; the standard deals with the work product used to capture an architecture, namely, the architecture description.
2. An architecture description is inherently multi-view. No single view is sufficient to capture an architecture because architecture is multi-disciplinary, with multiple stakeholders and multiple architecture-related concerns that the architect must deal with.
3. It is useful to separate viewpoints (perspectives on the architecture) from views (what is captured in the description of a specific architecture from the perspective of a viewpoint for a system of interest). This distinction was motivated by the body of existing practice that defines viewpoints for a number of architecture-related concerns. (However, the term *viewpoint* was introduced in the standard for this notion.)
4. There should be a viewpoint for each view. Just as every map should have a legend, each view should have a viewpoint explaining the conventions being used in that view. Figure E.2 illustrates one possible template for a viewpoint.
5. An architecture description is driven by stakeholders' architecture-related concerns, because these reflect the issues the architect must deal with. Viewpoints are selected for use in an architecture description to ensure coverage of the identified architecture-related concerns.

Viewpoint Name	The name for the viewpoint, and any synonyms for the viewpoint.
Overview	An abstract or brief overview of the viewpoint and its key features.
Concerns	A listing of the architecture-related concerns framed by this viewpoint. This is crucial information for the readers, because it helps them decide whether this viewpoint will be of use to them.
Anti-Concerns	*Optional.* It can be useful to document the kinds of issues a viewpoint is *not appropriate for.* Articulating anti-concerns may be a good antidote for certain overused notations.
Typical Stakeholders	*Optional*. The typical audiences for views prepared using this viewpoint. Who are the usual stakeholders for this kind of view?
Model Types	Identify each type of model used by the viewpoint.
Model Languages	For each type of model used, describe the language, notation, or modeling techniques to be used. Each model language is a key modeling resource that the viewpoint makes available. Model languages provide the vocabularies for constructing the view. ISO/IEC 42010 does not specify how a modeling language is documented. It could be by reference to an existing modeling language (such as SADT or UML) or technique (for example, M/M/4 queues from queuing theory); by providing a metamodel for the language to define the language's core constructs; via a template that users fill in; or by some combination of these methods.
Viewpoint Metamodels	*Optional*. A metamodel presents the conceptual entities, their attributes, and the relations that comprise the vocabulary of a type of model. There are different ways of representing ontologies (such as entity-relation diagrams, class diagrams). Any metamodel should capture: • *Entities.* What are the major sorts of elements present in this type of model? • *Attributes*. What properties do entities in this type of model possess? • *Relationships*. What relations are defined among entities within this type of model? • *Constraints.* What kinds of constraints are there on entities, attributes, or relations within this type of model? Entities, attributes, relations, and constraints are all architecture elements in the sense of ISO/IEC 42010.
Conforming Notations	Identify an existing notation or model language to be used for this type of model.
Model Correspondence Rules	The viewpoint may specify model correspondence rules. Each one may be documented here.
Operations on Views	Operations define the methods that may be applied to views and their models. Operations can be divided into categories: • *Creation methods* are the means by which views are prepared using the viewpoint. These could be in the form of process guidance (how to start, what to do next), work product guidance (templates for views of this type), heuristics, styles, patterns, or other idioms. • *Interpretive methods* provide the means by which views are to be understood by readers and system stakeholders. • *Analysis methods* are used to check, reason about, transform, predict, apply, and evaluate architecture results from this view. • *Implementation methods* capture how to realize or construct systems using information from this view.
Examples	*Optional*. This section provides examples for the reader.
Notes	*Optional*. Any additional information users of the viewpoint may need.
Sources	What are the sources for this viewpoint, if any? This may include author, history, literature references, prior art, and more.

Figure E.2
Template for a viewpoint

The Rational Unified Process and Kruchten's "4+1" approach are discussed in Section E.2.

DoDAF is discussed in Section E.5.

One of the goals for the joint revision of ISO/IEC 42010:2007 was to align with existing ISO architecture efforts, specifically GERAM (ISO 15704 2000) and RM-ODP (ISO/IEC 10746-2 1996). The use of these standards, and existing architecture approaches such as Kruchten's "4+1" approach (Kruchten 1995), Zachman's Architecture Framework (Zachman 1987), and even the DoD Architecture Framework (DoDAF 2007), underscores the fact that many (if not most) practicing architects operate within an architecture framework. Each of these approaches could be considered as defining a set of viewpoints, and in fact it was the existence of such approaches that motivated the separation of viewpoint from view.

ISO 42010 defines an **architecture framework** as "conventions and common practices for architecture description established within a specific domain or stakeholder community."

The standard also establishes requirements for creating and documenting **architecture frameworks**. In the terms of the standard, an architecture framework specifies a set of stakeholders, a set of concerns, and viewpoints covering those concerns.

E.1.2 42010 and Views and Beyond

If you want to use the Views and Beyond approach to produce an ISO 42010-compliant architecture document, you certainly can. The main additional obligation is to choose and document a set of viewpoints and (to a lesser degree) address ISO 42010's required information content. Table E.1 summarizes the information required by the ISO 42010 standard and how the Views and Beyond approach addresses each one.

ADVICE

In ISO 42010, it is natural to talk about (for example) a "safety view" or a "security view." These are views (following from viewpoints) addressing the safety and security concerns, respectively, of various stakeholders. In the Views and Beyond approach, it is more natural to talk about a "service-oriented view" or a "layered view." In the Views and Beyond approach, you can put together a package of documentation for a specific set of stakeholders based on their needs by choosing the most applicable views, or even the most applicable view packets from within the most applicable views, and assembling those. The "What It's For" section of the corresponding style guides will help you choose. You can physically package those together to produce, say, a safety or security documentation "view."

To satisfy the obligation of ISO 42010 of documenting a set of viewpoints, use the information in the "What It's For" section of the style guide. See the introduction to Part I.

To satisfy the information content required by ISO 42010, use the templates described in Chapter 10.

Table E.1 ISO 42010 information requirements and how we address them

ISO 42010 Information Requirement	Views and Beyond Location
Identification and overview information, as appropriate to stakeholder, project, and organization needs. For example: summary, context, glossary, references, and change history.	Several items in this category amount to good bookkeeping. Context is addressed in the context diagrams; the other items are prescribed in the standard organizations of Chapter 10.
Stakeholders and concerns. Identify architecturally relevant stakeholders. At a minimum consider customers, users, operators, acquirers, suppliers, developers, and maintainers. Identify their architecture-related concerns. At a minimum consider system purposes, suitability of architecture to meet purposes, feasibility of construction, potential risks throughout life cycle, maintainability, deployability, and evolvability.	The documentation roadmap called for in Section 10.2 captures information about stakeholders and their concerns—specifically, how they will use the documentation package. For ISO 42010 compliance, make sure the stakeholders and concerns include those named in the left-hand column.
Viewpoints. For each viewpoint, the following must be specified: • The viewpoint name • The subset of identified architecture-related concerns (from above) framed by this viewpoint • The identification of each type of architecture model used by this viewpoint • For each type of model: the languages, notations, rules, constraints, modeling techniques, analytical methods, or operations to be used in creating and interpreting the view • Rationale for selection of the viewpoint • Any additional information, such as completeness and correctness checks, evaluation criteria, heuristics, or guidelines	We define several commonly used module, C&C, and allocation styles. Each style guide defines the concepts—elements, relations, and properties—that should be used in documenting a system in accordance with the style. It contains information about useful notations and modeling techniques for that style. Each style guide also contains a section noting what it's for, which should help users in deciding what concerns will be addressed by the style. All of this information in a style guide constitutes an implicit viewpoint definition, but the standard requires including an explicit set in your document, either directly or by reference. You can easily accommodate this requirement by adding a section for viewpoint definitions to the "documentation beyond views" template in Section 10.2. There, you can reproduce or refer to the specific style guide information as needed.
Views. Each view must include: • A view identifier • Overview and configuration information as required by project or organization • One or more architecture models covering the whole system from the viewpoint	Chapter 10 discusses the information that should be documented for a view.
A record of all inconsistencies among views, preferably accompanied by an analysis of consistency among all views.	In Chapter 6, we discuss techniques for documenting relations among views, which is then recorded in the "documentation beyond views" part of the package, as detailed in Chapter 10.
Rationale for the key architectural decisions made, preferably accompanied by evidence of alternatives considered and rationale for the choices made.	Reserved spots for rationale are provided in each view, in the documentation beyond views, and in interface documentation.

E.2 Rational Unified Process/Kruchten 4+1

The Rational Unified Process (RUP) introduces a five-view approach to documenting software architectures, based on Kruchten's 4+1 approach.

1. The logical view contains the most important design classes.
2. The implementation view captures the architectural decisions made for the implementation.
3. The process view documents the tasks—processes and threads—involved.
4. The deployment view documents the various physical nodes for the most typical platform configurations.
5. The use case view or "plus-one view" contains use cases and scenarios of architecturally significant behavior.

The RUP describes the use case view as a representation of an architecturally significant subset of the use case model, which documents the system's intended functions and its environment. The use case view serves as a contract between the customer and the developers and represents an essential input to activities in analysis, design, and test. It also serves as a design check on the other views: It is incumbent upon the architect to show how each of the other views correctly supports the use cases in the use case view. If they do, then this suggests that they are correct and consistent with each other.

E.2.1 RUP/4+1 and Views and Beyond

If you want to use the Views and Beyond approach to document a 4+1 architecture, you can easily do so.

The decomposition style is covered in Section 2.1.

The uses style is covered in Section 2.2.

The generalization style is covered in Section 2.3.

- Documenting a logical view of the RUP can be done by using certain module or C&C styles. A union of the decomposition style, the uses style, and the generalization style allows you to represent the structural part of the logical view by using such elements as subsystems and classes, whereas a C&C style (which one depends on the design you chose) allows you to represent the runtime aspects by using components and ports.
- An implementation view can be represented by using a combination of the decomposition style, the layered style, the uses style, and the generalization style. The implementation view represents implementation elements, such as implementation subsystems and components. The RUP distinguishes between a design and an implementation model to separate general design aspects from implementation aspects introduced by the use of a specific programming language.

To describe the relations between elements of the design model and the implementation model, the mapping should be documented. To show how the implementation elements are stored in a file system during development, use the Views and Beyond implementation view.

The implementation style is discussed in Section 5.5.

- The RUP process view provides a basis for understanding the process organization of a system, illustrating the decomposition of a system into processes and threads and perhaps also showing the interactions among processes. The process view also includes the mapping of classes and subsystems onto processes and threads. To accommodate the process view, define a style that uses components such as those defined in the C&C communicating-processes style—task, process, thread—and specific refinements of the communication connectors, such as RPC or broadcast. To describe the relations between processes and elements, such as subsystems and classes, the mapping among them should be documented.

The C&C communicating-processes style is covered in Section 4.6.1.

- A RUP deployment view describes one or more physical network—hardware—configurations on which the software is deployed and runs. This view also describes the allocation of processes and threads—from the RUP process view—to the physical nodes. The deployment style is a good match for the RUP deployment view. The RUP deployment view also allows you to assign deployment units to nodes. A deployment unit consists of a build—an executable—documents, and installation artifacts. It is a packaging of implementation elements for selling and/or downloading purposes. To achieve this, you can define a style showing implementation elements—subsystems/classes—and how they are packaged as deployment units.

The deployment style is covered in Section 5.2.

Finally, use cases are a vehicle for describing behavior, and behavior is a part of every view's supporting documentation. Consequently, you can document use cases as behavior documentation for the system or parts of it. You can also document the use case view in the mapping to requirements.

Behavior documentation is covered in Chapter 8.

Table E.2 reconciles the prescribed Rational Unified Process views with our advice in this book.

The mapping of an architecture to its requirements is covered in Section 10.3.

Beyond its five views, RUP does not prescribe other kinds of documentation, such as interface documentation, rationale, or behavior of ensembles. It doesn't call for a documentation roadmap, a mapping between views, view templates, or style guides. But it certainly does not rule these things out, either, so don't forget to add them.

Table E.2 Relating Views and Beyond to RUP

To Achieve This RUP View	Use This Views and Beyond Approach
Use case view	Adopt use cases to specify behavior, either associated with any of the views or as part of the documentation beyond views.
Logical view	Use a module style that shows generalization, uses, and decomposition for structural aspects, and a C&C style for the runtime aspects.
Implementation view	Use a module style that contains implementation elements. Use an implementation view to show allocation to development files.
Process view	Use a style such as the communicating-processes style.
Deployment view	Use the deployment style, one of the allocation styles.

You are free to consider additional views that may be important in your project's context, and you should do so. You should augment the primary presentation of each view with the supporting documentation called for in Section 10.2.1, and you should complete the package by writing the documentation that applies beyond views, as described in Section 10.2. The result will be a RUP-compliant set of documentation having the necessary supporting information to complete the package.

E.3 Using the Rozanski and Woods Viewpoint Set

With Nick Rozanski and Eoin Woods

In 2005, the two coauthors of this section, Nick Rozanski and Eoin Woods, wrote a very useful book on the design and documentation of software systems architecture (Rozanski and Woods 2005). In it, they prescribed a useful set of six viewpoints (in the ISO 42010 sense) to be used in documenting software architectures. The six viewpoints, based on an extension of the Kruchten 4+1 set, are shown in Figure E.3.

The views specified by their viewpoint set are the following:

- The *functional view* documents the system's functional elements, their responsibilities, interfaces, and primary interactions. A functional view is the cornerstone of most architecture documents and is often the first part of the documentation that stakeholders try to read. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on. It also has a significant impact on the system's quality properties, such as its ability to change, its ability to be secured, and its runtime performance.

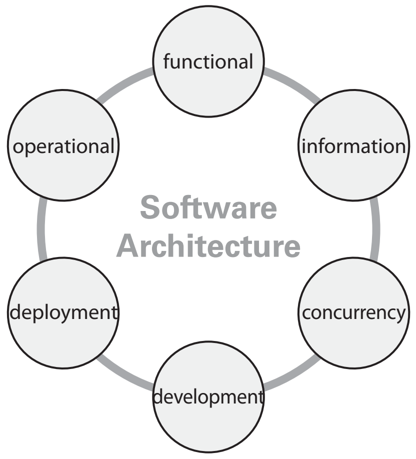


Figure E.3
The Rozanski and Woods viewpoint set (from Rozanski and Woods 2005, p. 213)

- The *information view* documents the way that the architecture stores, manipulates, manages, and distributes information. The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but broad view of static data structure and information flow. The objective of this analysis is to answer the important questions around content, structure, ownership, latency, references, and data migration.
- The *concurrency view* describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation.
- The *development view* describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.
- The *deployment view* describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment. This view captures the hardware environment that the system needs, the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.

- The *operational view* describes how the system will be operated, administered, and supported when it is running in its production environment. For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time. The aim of the operational view is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

An **architecture perspective** is "a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views" (Rozanski and Woods 2005).

COMING TO TERMS

Architecture Perspectives

An **architecture perspective** defines a number of activities, tactics and guidelines for a set of related quality properties. For example, a resilience perspective might include activities such as confirming availability requirements and schedule; estimating the availability of individual components; and deriving the overall platform and service availability. It might include tactics for achieving high availability, such as use fault-tolerant hardware; use clustering and load balancing: and use software availability solutions such as redundant logging and up-to-the-minute data restoration. Notice that a perspective is somewhat more restrictive than a viewpoint as defined in ISO 42010.

Architecture perspectives formalize an activity that good architects do as a matter of course, namely, ensuring that a system exhibits the right quality attribute properties, such as resilience, scalability, security, or extensibility.

This typically requires consideration of the system across a number of its architecture views. For example, achieving good performance requires consideration of the system's functional and concurrency structures, the way it manages and accesses information, and how it is deployed on physical hardware and software.

Having started to design the architecture of the system, and documented the architecture in a number of views, the architect therefore *applies* the perspective to the views to assess its capabilities against those quality properties. Applying a perspective does not result in a new view, but rather, it may result in a number of modifications to existing views to help address stakeholder concerns.

See Section 9.1 for more information about stakeholders and their documentation needs.

Applying perspectives enables the architect to identify any weaknesses or omissions in the architecture, and to suggest enhancements or extensions to it. This leads to insights into the architecture (for example, understanding better where its single points of failure are), improvements to it (such as adding redundant hardware or software components to reduce the likelihood of catastrophic failure), and artifacts (such as service availability models).

If you wish to document perspectives prescribed by the Rozanski and Woods approach, you can do so by supplementing your documentation as follows:

The template for documentation beyond views includes a documentation roadmap. Supplement this roadmap with a description of the perspectives applied to the architecture. These descriptions can be found in the perspective catalog and can be included in the roadmap directly or by reference. Complement the mapping between views with a mapping between perspectives and views. Stakeholders wishing to understand how their concerns are met can look at the applicable perspectives to see which views are involved.

See Figure 10.4 in Section 10.2 for the template for documentation beyond views.

The template for a view packet is the place to record more-detailed information. Capture the explanation for design decisions that resulted from the application of the perspective under rationale, and capture references to the concern that motivated the perspective under other information.

See Figure 10.1, in Section 10.1, for the template for a view.

E.3.1 Rozanski and Woods Viewpoints and Views and Beyond

This set of viewpoint definitions is not prescriptive about the notations or modeling approaches that should be used in each view. Instead, the viewpoints define the type(s) of models expected in each view and the information that should be captured in each, suggesting *possible* modeling approaches for each. Therefore, it is perfectly possible to use this viewpoint set in conjunction with the documentation approaches described by the Views and Beyond approach.

- A *functional view* contains a functional structure model, comprising a set of functional elements, interfaces offered by the elements, connectors between the elements, and external entities that the system's elements interact with. Such a functional view can be documented using a C&C

style, using components and ports to model the functional elements and their interfaces and connectors to link them together.

- An *information view* may contain a wide variety of models related to the information in the system, including static data structure models, information flow models, information life-cycle models, and data ownership models. Here, the data model view directly applies. Data flow can also be documented as a C&C or module style.

Data model views are described in Section 2.6.

- A *concurrency view* may contain a system-level concurrency model, showing architecturally significant process and thread structures, and a state model, showing the valid states and transitions of any system elements with complex life cycles. The concurrency model may contain processes, process groups, threads, and interprocess communication mechanisms. The state models contain the familiar state machines, made up of states, transitions, events, and actions. The concurrency model can be documented using the C&C communicating-processes style, and as mentioned earlier when discussing documenting behavior, state models can be naturally captured as state machines.
- A *development view* may contain a module structure model (showing how the implementation modules are organized), common design models (describing system-wide design conventions), and codeline models (explaining how the source code is organized and built). Of these, the module structure model can be very naturally captured using the module decomposition, uses, or layered styles, while the allocation implementation style may well be helpful in representing a codeline model. The common design model (dealing with functions such as initialization, termination and restart, and message logging) can be captured in the architecture background section of the view packet template under assumptions that pertain to the development environment. These assumptions place design constraints on the developers to maximize commonality across element implementations. These constraints might be recorded in textual form or in the form of design patterns using more specific notations (such as UML).
- A *deployment view* may contain a runtime platform model, showing how the system is deployed to production; a network model, showing its networking requirements; and technology dependency models, showing the requirements that the system has on its runtime environment. Of these,

the runtime platform model and network model are both naturally documented using the allocation deployment style. The technology dependency models simply record the technology dependencies of each part of the deployment environment (that is, required libraries, middleware, and so on). These can be captured using a uses style (represented as a simple table).

- Finally, an *operational view* can contain models relating to system installation, system migration strategy, operational configuration management approach, administration, and system support. These models capture requirements of the operating environment that influence the architecture. Like any other requirements, they can be part of the documentation beyond views. Solutions can be captured in one or more existing views such as the allocation install style, a C&C repository style, or the uses style, showing guidelines for monitoring and message logging.

Table E.3 summarizes the discussion.

Table E.3 Relating Views and Beyond to the Rozanski and Woods viewpoint set

To Achieve This R&W View	Use This Approach
Functional	One or more C&C styles.
Information	Data model style; data flow can be documented as a C&C style.
Concurrency	C&C communicating-processes style.
Development	Decomposition or layered style (to represent the structure model). Implementation style (to represent the codeline model). Documentation of assumptions (to represent the common design model).
Deployment	Deployment style (runtime platform and network models). Uses style (technology dependency model).
Operational	Install style; operational requirements can be part of the documentation beyond views, and solutions can be associated with any of the views.

The data model style is covered in Section 2.6.

The communicating-processes style is covered in Section 4.6.1.

The layered style is covered in Section 2.4.

The implementation style is covered in Section 5.3.

Documentation of assumptions is part of rationale. See Section 6.5.

The deployment style is covered in Section 5.2.

Documentation beyond views is covered in Section 10.2.

E.4 Documenting Architecture in an Agile Development Project

E.4.1 Overview

“Agile” refers to an approach to software development that emphasizes rapid and flexible development and deemphasizes project and process infrastructure for their own sake. Figure E.4 shows the “manifesto” for Agile software development that has served since 2001 as the movement’s *Desiderata.*

There are many different methodological instantiations of the Agile approach. These include Extreme Programming (Beck and Andres 2004), Scrum (Schwaber 2001), Feature-Driven Development (Palmer and Felsing 2002), and Crystal Clear (Cockburn 2004). Practices that show up in one or more of the Agile methods include the following:

- *User stories.* Text specifies functional requirements describing the actions of people.
- *Test-driven development.* Developers create automated tests at the same time they write the tested code.
- *Short iterations.* The development plan consists of short iterations (a few weeks); also called sprints.
- *Pair programming.* Developers work in pairs, where one is typing the code and the other reviews the code looking for defects and ways to improve the design.
- *Refactoring.* As part of the implementation cycle, code is refactored to improve the internal structure and maintainability without altering the externally visible behavior.

For some, agility is used as an excuse to avoid disciplined development. The Dilbert cartoon in Figure E.5 represents this early view of the Agile world.

Figure E.4
The Manifesto for Agile Software Development (Agile Alliance 2002a)

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

– ***Individuals and interactions over processes and tools***
– ***Working software over comprehensive documentation***
– ***Customer collaboration over contract negotiation***
– ***Responding to change over following a plan***

That is, while there is value in the items on the right, we value the items on the left more.

WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
© Scott Adams, Inc./Dist. by UFS, Inc.

Figure E.5
Agile, as some imagined it. (DILBERT: © Scott Adams / Dist. by United Feature Syndicate, Inc.)

In fact, saying that Agile development is antithetical to documented development is simply not true, if indeed it ever was.

A recent survey shows that Agile teams are more likely to build models than traditional teams (Ambysoft 2008).

A related misconception is that in Agile, coding starts on day one of the project. In practice, the first iteration can go by with no production code written at all. This happens because the team is sorting out design alternatives and conducting technical experiments with different frameworks, platforms, or technologies.

The key goal of design and modeling in Agile projects is not to avoid designing, but to avoid "big design up front" (BDUF). Broad and far-reaching architecture strategies are worked out up front, but many other design decisions can be deferred until needed. They can be written down whenever they are made.

Use a standard organization in order to employ documentation as a receptacle to hold the results of design decisions as they are made.

Documented design decisions in Agile projects *tend to be* (but are not always) fewer in number and coarser in granularity than design decisions documented in traditional projects. This comes about because Agile developers are expected to have design skills, and Agile designers and architects are expected to have coding skills. So the communication of design decisions is shorter and denser; it's rather like telling a story to a member of your family as opposed to a complete stranger.

E.4.2 Agile Development and Views and Beyond

The Views and Beyond and Agile philosophies agree strongly on a central point: If information isn't needed, don't document it. All documentation should have an intended use and audience in mind, and be produced in a way that serves both. One of the fundamental principles of technical documentation is "Write for the reader." That means understanding who will read the documentation and how they will use it. If there is no audience, there is no need to produce the documentation.

The Seven Rules for Sound Documentation are given in Section P.5. of the prologue

Architecture view selection is an example of applying this principle. The Views and Beyond approach prescribes producing a view if and only if it addresses the concerns of an explicitly identified stakeholder community.

View selection is covered in Chapter 9.

Another central idea to remember is that documentation is not a monolithic activity that holds up all other progress until it is complete. The view selection method given in Chapter 9 prescribes producing the documentation in prioritized stages to satisfy the needs of the stakeholders who need it now.

Cockburn expresses a similar idea this way: "The correct amount of documentation is exactly that needed for the receiver to make her next move in the game. Any effort to make the models complete, correct, and current past that point is a waste of money" (Cockburn 2002). The trick is knowing who the receivers are and what moves they need to make. Remember that the receiver might be a maintainer whose job begins long after the system is first fielded and the development team is disbanded.

With that in mind, the following is the suggested approach for producing Views and Beyond-based architecture documentation using Agile principles:

1. Adopt a template or standard organization to capture your design decisions.
2. Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.
3. Fill in the sections of the template for a view, and for information beyond views, when (and in whatever order) the information becomes available. But only do this if writing down this information will make it easier (or cheaper or make success more likely) for someone downstream doing their job.

See the formula for the economics of documentation in Section P.2.4 of the prologue.

Actually, this three-step approach is the entire Views and Beyond approach in a nutshell: Have a template. Fill it in as you go. Only write down what's worth writing down. For an Agile project, the emphasis shifts to the guidance about *not* doing things, which is implied by the "only if" clauses.

Beyond this strategic guidance, you can also use the following advice:

Using a wiki to capture an architecture is discussed in Section 10.4.2.

- *Stop designing as soon as you feel you're ready to start coding.* Don't worry about creating an architectural design document and then a finer-grained design document. Produce just enough design information to allow you to move on to code. Capture the design information in a format that is simple to use and simple to change—a wiki, perhaps. In the next sprint, you can expand the existing design as needed in order to capture design decisions required to implement the features listed for that sprint.
- *Don't feel obliged to fill up all sections of the template, and certainly not all at once.* We still suggest you define and use rich tem-

plates because they may be useful in some situations. But you can always write "N/A" for the sections for which you don't need to record the information (perhaps because you will convey it orally).

Using a view template such as the one in Section 10.1, the ultimate simplification is to add the primary presentation and leave all other sections marked as "N/A". In Agile teams, modeling sometimes happens as brief discussions by the whiteboard. In your view, the primary presentation may have a digital picture of the whiteboard and nothing more. Further information about the elements (element catalog), rationale discussion (architecture background), variability mechanisms being used (variability guide), and all else will be communicated verbally to the team—at least for now. Later on, if you find out that it's useful to record a piece of information about an element, a context diagram, rationale for a certain design decision, or something else, you can replace the "N/A" with the corresponding piece of information.

The fourth principle of sound documentation in the prologue tells us that it's better to write "N/A" or "TBD" than leave sections blank. You shouldn't remove the section headers either; otherwise, your document will end up with a different structure than the template.

- *If it's not worth updating the design, throw it away.* As an example, suppose you created a sequence diagram that became part of the architecture documentation. In the implementation, you started off following what's in the sequence diagram. However, you found better ways to implement that transaction and the end result turned out to be fairly different from the sequence diagram. The original diagram fulfilled its primary purpose by guiding the initial implementation. What should you do with the diagram now? You can:
  - *Leave it as is.* This is the worst option, because now the documentation will be at odds with the implementation. Nothing makes a reader flee from documentation faster than the discovery that it is out of date, and now the reader won't trust any other part of the architecture documentation, either.
  - *Update the diagram.* This is the ideal option, given you have time for that. The updated diagram will help maintainers who will need to understand that part of the implementation.
  - *Remove or cross off the diagram.* This option is the realistic choice in many projects. The diagram is out of date; you're better off removing it or marking it as out of date or no longer authoritative (Figure E.6 shows an example) so it won't mislead readers of the documentation. In Agile projects, code, code comments, and associated unit tests often serve as the authoritative documentation for local (element-specific) designs.

Sequence diagrams can be used to document behavior. Sequence diagrams are covered in Section 8.3.2.

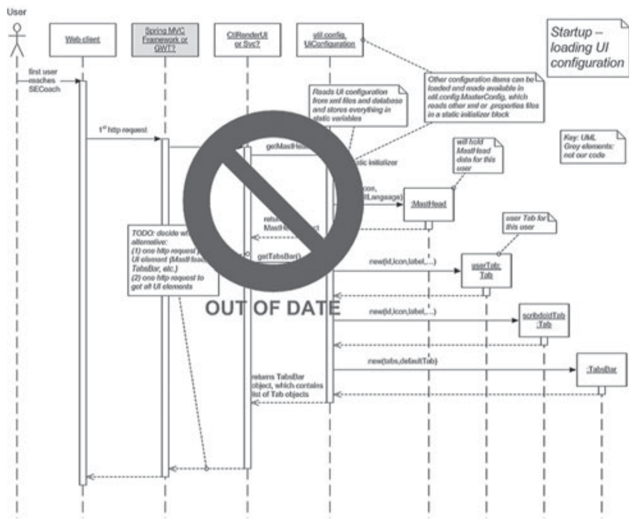

User
Web client
Spring MVC Framework or GWT?
CtlRenderUI or Svc?
util.config. UIConfiguration
Startup – loading UI configuration
first user reaches SECoach
1st http request
Reads UI configuration from xml files and database and stores everything in static variables
Other configuration items can be loaded and made available in util.config.MasterConfig, which reads other xml or .properties files in a static initializer block
will hold MastHead data for this user
Key: UML Grey elements: not our code
:MastHead
user Tab for this user
TODO: decide which alternative: (1) one http request per UI element (MastHead, TabsBar, etc.) (2) one http request to get all UI elements
getTabsBar()
new(id,icon,label,...)
userTab: Tab
OUT OF DATE
scribdoidTab :Tab
new(tabs,defaultTab)
:TabsBar
returns TabsBar object, which contains list of Tab objects

Figure E.6
The architect decided not to update this diagram, but he didn't want to delete it either. So he marked the diagram to prevent others from consuming out-of-date information.

- *Many times, sketches are all you need.* Don't spend time crafting the neatest diagram using the latest and richest notation available. Don't spend money on sophisticated modeling tools if you just need to draw simple diagrams. In many Agile projects, especially the ones with small, collocated teams, the true value of design diagrams comes from drawing them, which forces you to think through the issues; once the issues are solved, the documentation can be refined. Many times the design is represented as a sketch on a whiteboard or piece of paper. Figure E.7 shows an example.

If a sketch successfully conveys the design to the development team, you can use it as the primary presentation in an architecture view.

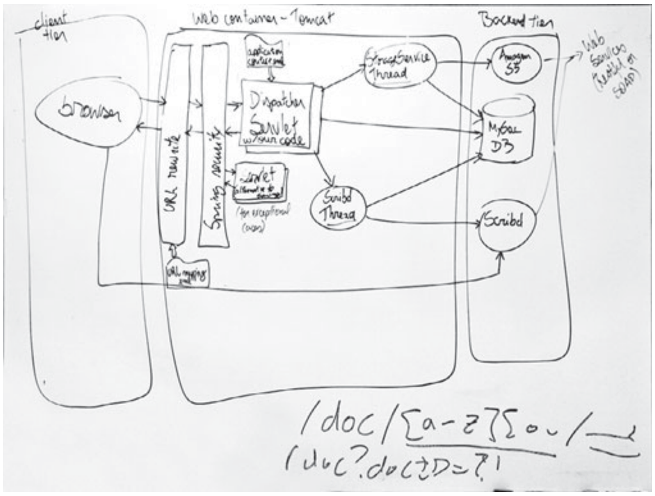


Figure E.7
Sketch of a C&C view on the whiteboard

E.5 U.S. Department of Defense Architecture Framework

With Don O'Connell

E.5.1 Overview of DoDAF

The DoDAF is the U.S. Department of Defense's framework standard on how to document an architecture. According to the DoD:

> The DoDAF provides the guidance and rules for developing, representing, and understanding architectures based on a common denominator across DoD, Joint, and multinational boundaries. It provides insight for external stakeholders into how the DoD develops architectures. The DoDAF is intended to ensure that architecture documentation can be compared and related across programs, mission areas, and, ultimately, the enterprise, thus, establishing the foundation for analyses that supports decision-making processes throughout the DoD.
>
> The DoDAF defines a set of products that act as mechanisms for visualizing, understanding, and assimilating the broad scope and complexities of an architecture description through graphic, tabular, or textual means. These products are organized under four views: [operational view (OV), systems and services view (SV), technical standards view (TV), and all-view (AV)]. Each view depicts certain perspectives of an architecture as described below.
>
> The OV captures the operational nodes, the tasks or activities performed, and the information that must be exchanged to accomplish DoD missions. It conveys the types of information exchanged, the frequency of exchange, which tasks and activities are supported by the information exchanges, and the nature of information exchanges. . . .
>
> The SV captures system, service, and interconnection functionality providing for, or supporting, operational activities. DoD processes include warfighting, business, intelligence, and infrastructure functions. The SV system functions and services resources and components may be linked to the architecture artifacts in the OV. These system functions and service resources support the operational activities and facilitate the exchange of information among operational nodes. . . .
>
> The TV is the minimal set of rules governing the arrangement, interaction, and interdependence of system parts or elements. Its purpose is to ensure that a system satisfies a specified set of operational requirements. The TV provides the technical systems implementation guidelines upon which engineering specifications are based, common building blocks are established, and product lines are developed. It includes a collection of the technical standards, implementation conventions, standards options, rules, and criteria that can be organized into profile(s) that govern systems and system or service elements for a given architecture. . . .

In this section, all quoted material and figures come from DoDAF 2007 (online at www.defenselink.mil/cio-nii/docs/DoDAF_Volume_I.pdf).

The U.K. Ministry of Defence employs a similar framework called MoDAF.

> [The AV captures the] overarching aspects of an architecture that relate to all three views. The AV products provide information pertinent to the entire architecture but do not represent a distinct view of the architecture. AV products set the scope and context of the architecture. The scope includes the subject area and time frame for the architecture. The setting in which the architecture exists comprises the interrelated conditions that compose the context for the architecture. These conditions include doctrine; tactics, techniques, and procedures; relevant goals and vision statements; concepts of operations (CONOPS); scenarios; and environmental conditions.

The relations among these views are shown in Figure E.8.

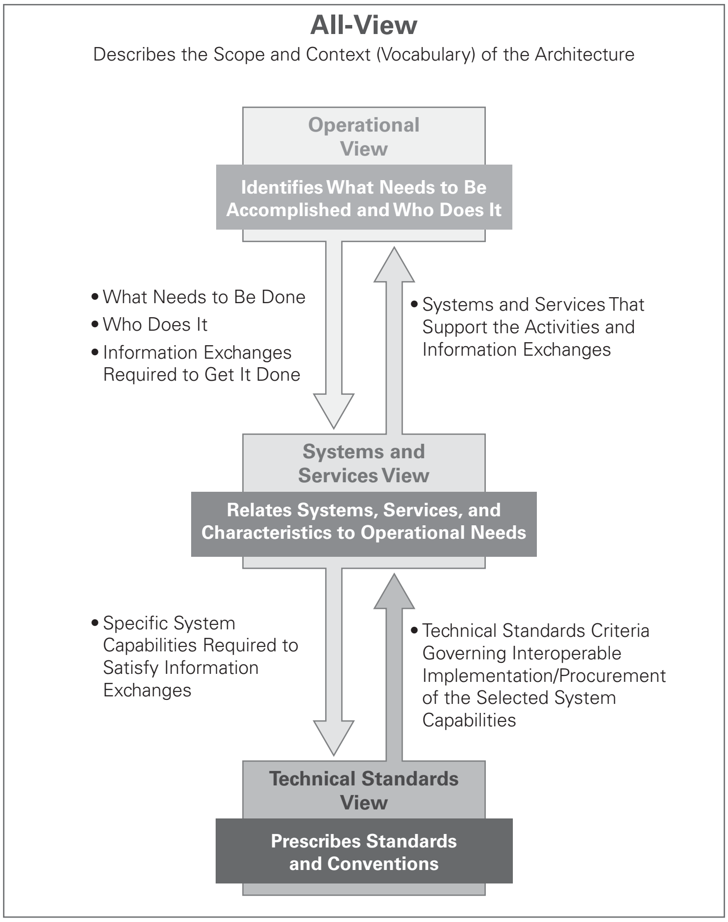

All-View
Describes the Scope and Context (Vocabulary) of the Architecture
Operational View
Identifies What Needs to Be Accomplished and Who Does It
• What Needs to Be Done
• Who Does It
• Information Exchanges Required to Get It Done
• Systems and Services That Support the Activities and Information Exchanges
Systems and Services View
Relates Systems, Services, and Characteristics to Operational Needs
• Specific System Capabilities Required to Satisfy Information Exchanges
• Technical Standards Criteria Governing Interoperable Implementation/Procurement of the Selected System Capabilities
Technical Standards View
Prescribes Standards and Conventions

Figure E.8
All-view describes the overall context of the system. Operational views are largely contextual, concept-of-operations, and capability diagrams and tables. Systems and services views are largely the nodes and interconnectivity, with numerous products showing various functions and behaviors. Technical standards views are about current and future technical standards.

E.5.2 DoDAF and Software Architecture

Although DoDAF is quite reticent about pronouncing what kind of architecture it was intended to capture—software? system? enterprise?—it is quite clear that it was not intended to capture software architectures.

Generally speaking, DoDAF views provide the following relevant software architecture information.

1. Context and scope of the architecture
2. Key capabilities provided by the system, the key mission threads, the operational nodes and their mission activities
3. The system nodes and primary data flows, the networks that connect those nodes, and the allocation of functionality to those system nodes
4. Optionally, performance, availability, information assurance, and interoperability behaviors
5. Deployment views, showing major software components and where they reside
6. Services, along with their capabilities and constraints

E.5.3 DoDAF and Views and Beyond

Table E.4 shows all of the DoDAF products, arranged by type of view. DoDAF is not a particularly suitable framework for software architecture; nevertheless, if you need to produce DoDAF documents, the rightmost column tells you the place in Views and Beyond documentation where you can record it.

Table E.4 DoDAF (v1.5) products and how the Views and Beyond approach can be used to capture their information

View	Framework Product	Framework Product Name	General Description	Views and Beyond Equivalent
All View	AV-1	Overview and Summary Information	Scope, purpose, intended users, environment depicted, analytical findings	This is accounted for in the documentation roadmap and system overview given in the documentation beyond views section, as well as the analytical findings to support rationale for major design decisions. Documentation beyond views is covered in Section 10.2. Documenting rationale is covered in Section 6.5.
	AV-2	Integrated Dictionary	Architecture data repository with definitions of all terms used in all products	Glossary

continues

Table E.4 DoDAF (v1.5) products and how the Views and Beyond approach can be used to capture their information (*continued*)

View	Framework Product	Framework Product Name	General Description	Views and Beyond Equivalent
Operational	OV-1	High-Level Operational Concept Graphic	High-level graphical/ textual description of operational concept	This makes a good part of the system overview given in the documentation beyond views. Documentation beyond views is covered in Section 10.2.
	OV-2	Operational Node Connectivity Description	Operational nodes, connectivity, and information exchange need lines between nodes	Context diagram in a view packet whose scope is the node. Context diagrams are discussed in Section 6.3.
	OV-3	Operational Information Exchange Matrix	Information exchanged between nodes and the relevant attributes of that exchange	C&C view showing information exchange
	OV-4	Organizational Relationships Chart	Organizational, role, or other relations among organizations	A work assignment view is similar. Showing relations among organizations in a work assignment view is analogous to showing the relations among hardware nodes in a deployment view. Work assignment views are covered in Section 5.4. Deployment views are covered in Section 5.2.
	OV-5	Operational Activity Model	Capabilities, operational activities, relations among activities, inputs, and outputs; overlays can show cost, performing nodes, or other pertinent information	
	OV-6a	Operational Rules Model	One of three products used to describe operational activity—identifies business rules that constrain operation	
	OV-6b	Operational State Transition Description	One of three products used to describe operational activity—identifies business process responses to events	These are all descriptions of required behavior and not of architecture constructs. Documenting behavior is covered in Chapter 8.
	OV-6c	Operational Event-Trace Description	One of three products used to describe operational activity—traces actions in a scenario or sequence of events	
	OV-7	Logical Data Model	Documentation of the system data requirements and structural business process rules of the Operational View	

Table E.4 DoDAF (v1.5) products and how the Views and Beyond approach can be used to capture their information (*continued*)

View	Framework Product	Framework Product Name	General Description	Views and Beyond Equivalent
Systems and Services	SV-1	Systems Interface Description, Services Interface Description	Identification of systems nodes, systems, system items, services, and service items and their interconnections, within and between nodes	C&C views showing systems, services, and interconnections
	SV-2	Systems Communications Description, Services Communications Description	Systems nodes, systems, system items, services, and service items and their related communications laydowns	
	SV-3	Systems-Systems Matrix, Services-Systems Matrix, Services-Services Matrix	Relations among systems and services in a given architecture; can be designed to show relations of interest, e.g., system-type interfaces, planned vs. existing interfaces, etc.	Mapping between views; specifically a mapping between C&C views showing systems and C&C views showing services. Documenting a mapping between views is covered in Section 10.2.
	SV-4a	Systems Functionality Description	Functions performed by systems and the system data flows among system functions	Functions performed by the system and its services can be documented in a decomposition view. Decomposition views are covered in Section 2.1.
	SV-4b	Services Functionality Description	Functions performed by services and the service data flow among service functions	
	SV-5a	Operational Activity to Systems Function Traceability Matrix	Mapping of system functions back to operational activities	Mapping to requirements Mappings to requirements are discussed in Section 10.3.
	SV-5b	Operational Activity to Systems Traceability Matrix	Mapping of systems back to capabilities or operational activities	
	SV-5c	Operational Activity to Services Traceability Matrix	Mapping of services back to operational activities	
	SV-6	Systems Data Exchange Matrix, Services Data Exchange Matrix	Provides details of system or service data elements being exchanged between systems or services and the attributes of that exchange	C&C views showing the systems and services, and their information exchange and performance characteristics

continues

Table E.4 DoDAF (v1.5) products and how the Views and Beyond approach can be used to capture their information (*continued*)

View	Framework Product	Framework Product Name	General Description	Views and Beyond Equivalent
Systems and Services (*conitnued*)	SV-7	Systems Performance Parameters Matrix, Services Performance Parameters Matrix	Performance characteristics of Systems and Services View elements for the appropriate time frame(s)	
	SV-8	Systems Evolution Description, Services Evolution Description	Planned incremental steps toward migrating a suite of systems or services to a more efficient suite, or toward evolving a current system to a future implementation	Rationale supporting architecture decisions made to prepare for the evolution Documenting rationale is covered in Section 6.5.
	SV-9	Systems Technology Forecast, Services Technology Forecast	Emerging technologies and software/hardware products that are expected to be available in a given set of time frames and that will affect future development of the architecture	
	SV-10a	Systems Rules Model, Services Rules Model	One of three products used to describe system and service functionality—identifies constraints that are imposed on systems/services functionality due to some aspect of systems design or implementation	Behavior documentation, part of the C&C views showing the elements whose behavior is being documented Behavior documentation is covered in Chapter 8.
	SV-10b	Systems State Transition Description, Services State Transition Description	One of three products used to describe system and service functionality—identifies responses of a system/service to events	
	SV-10c	Systems Event-Trace Description, Services Event-Trace Description	One of three products used to describe system or service functionality—identifies system/service-specific refinements of critical sequences of events described in the Operational View	
	SV-11	Physical Schema	Physical implementation of the Logical Data Model entities, e.g., message formats, file structures, physical schema	Data model view Data model views are covered in Section 2.6.

Table E.4 DoDAF (v1.5) products and how the Views and Beyond approach can be used to capture their information (*continued*)

View	Framework Product	Framework Product Name	General Description	Views and Beyond Equivalent
Technical Standards	TV-1	Technical Standards Profile	Listing of standards that apply to Systems and Services View elements in a given architecture	These standards are primarily intended to address interoperability among systems in the architecture. Using Views and Beyond, you can list standards in the view(s) in which they apply. The "relations" part of the element catalog is a good spot for this.
	TV-2	Technical Standards Forecast	Description of emerging standards and potential impact on current Systems and Services View elements, within a set of time frames	

Generally, the following parts are missing from DoDAF to support a software architecture documentation:

1. Business environment and business drivers.
2. Architecture requirements in the form of quality attributes, plus customer inputs and prioritization of these attributes.
3. Architecture patterns and tactics, and the requirements that they address.
4. Module views showing the build-time relations and dependencies.
5. The SV views show a functional view of the architecture design. What is missing are the following notions:
   a. Infrastructure (messaging, system management, failure detection and recovery, and so on).
   b. Design patterns and other approaches to accomplish the quality attribute requirements.
   c. Dynamic nature of deployments.
   d. The OV-5 views are about mapping operational needs to functions. Software is often not built to these one-to-one mappings; thus, the mapping is not really possible. This mapping can be misleading.
6. Detailed software component interfaces. These are typically missing if the DoDAF views are describing a system of systems. These are also typically missing if DoDAF views are not constructed by software architects.
7. C&C view showing processes and threading of the software components. The notion of threads, interthread communications, multiple processes, and protected data is not supported.

E.5.4 A Strategy to Use DoDAF to Document Software Architecture

DoDAF, for all its attention to architecture, is a poor choice to represent software architecture. Its views were not created to support software architecture, and so unsurprisingly, they do a poor job of it. DoDAF simply speaks a different language, the language of systems and system-of-systems design. It is possible to shoehorn DoDAF into use by replacing its notion of "system" with the software architecture notion of "component," but if you do that, make sure that all of your readers are in on the trick.

A charitable thing to say is that while DoDAF is certainly not sufficient for software architecture, some DoDAF products are useful in representing software architectures. So here's a broad strategy:

- Include system-level behavior documentation as part of the DoDAF operational architecture view, concentrating on use cases that depict information exchange. Include this documentation in the "operational activity sequence and timing descriptions" products.
- Include element-level behavior documentation as part of the DoDAF systems architecture view. Include this documentation in the "systems activity sequence and timing descriptions" products.
- Include allocation views as part of the DoDAF system architecture view, where "physical resources" are documented.
- Include various module and C&C views as part of the DoDAF technical architecture view, appealing to it as the repository of "rules governing the arrangements, interaction, and interdependence of system parts" and "the criteria that describe compliant implementations."
- For the information contained in the beyond views part of the documentation, DoDAF provides slots for overview and summary information and a dictionary. Use the former to hold the documentation roadmap, the view template, the system overview, and system-wide rationale. The latter can be home to the mapping between views, the element directory, and the glossary.

Specific DoDAF products that are useful for software architecture include the following:

- SV-5, which might be the starting point for a 4+1-style logical view.
- OV-2 and OV-3, where information exchange is covered.
- AV-1 and OV-1, which provide contextual views, and those are useful for software.
- OV-7 and SV-11, which show the logical data model and implementation of the data model.

DoDAF 2.0

As this book was going to publication, DoDAF version 2.0 was on the verge of release. Its goal, according to the DoD, is

> to include further guidance on planning, developing, managing, maintaining, and governing architectures through a coherent semantic and structural metamodel. This version will place greater emphasis on a "data-centric" approach that facilitates the use of architecture by a wider variety of decision makers and will include additional information on federation for improved enterprise decisions.

Figure E.9, taken from the DoDAF 1.5 definition document, shows the evolution of DoDAF.

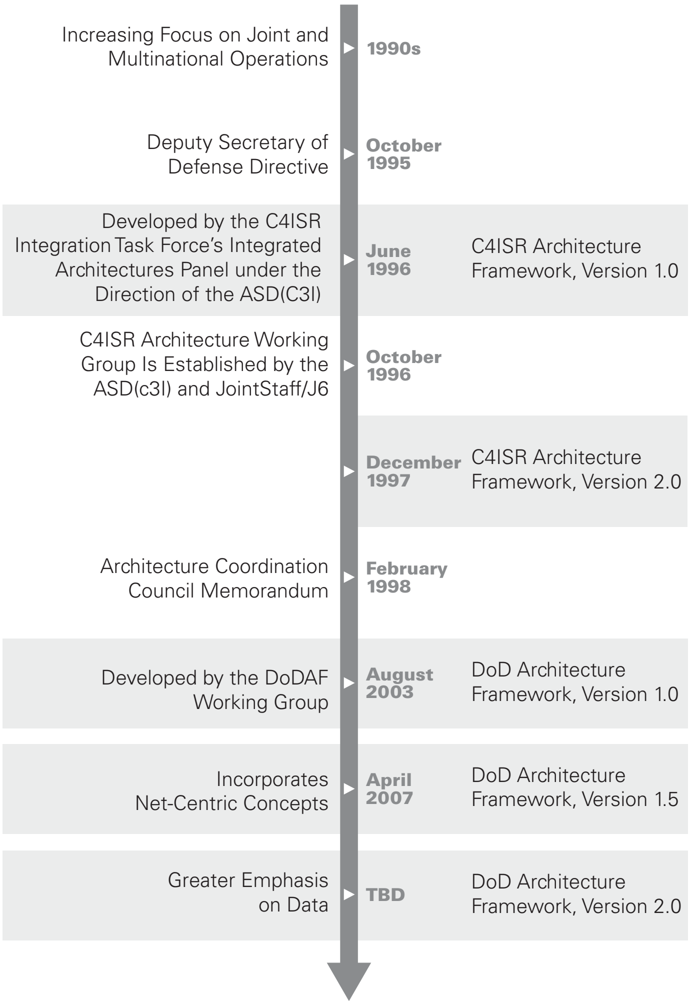

Increasing Focus on Joint and Multinational Operations
1990s
Deputy Secretary of Defense Directive
October 1995
Developed by the C4ISR Integration Task Force's Integrated Architectures Panel under the Direction of the ASD(C3I)
June 1996
C4ISR Architecture Framework, Version 1.0
C4ISR Architecture Working Group Is Established by the ASD(c3I) and JointStaff/J6
October 1996
December 1997
C4ISR Architecture Framework, Version 2.0
Architecture Coordination Council Memorandum
February 1998
Developed by the DoDAF Working Group
August 2003
DoD Architecture Framework, Version 1.0
Incorporates Net-Centric Concepts
April 2007
DoD Architecture Framework, Version 1.5
Greater Emphasis on Data
TBD
DoD Architecture Framework, Version 2.0

Figure E.9
Progression culminating in DoDAF version 2.0

E.6 Where Architecture Documentation Ends

Early in this book, we examined the question of where architecture ends and nonarchitectural design begins. A related question is where architecture documentation ends and other documentation issues begin. Architectures of all stripes exist. Security architectures, enterprise architectures, reference architectures, installation architectures: the list is endless.

Some of the terms are clearly in scope. Reference architectures, for instance, appeared in our discussion of documenting variation points in Chapter 6; the essence of a reference architecture is its ability to be tailored to the needs of any of a family of systems. Security architectures, although not addressed as such, are covered by making sure that a security specialist can find information of analytical use in one or more of the "normal" styles, such as those presented in Part I.

But some writers undoubtedly incorporate into architecture some aspects of system documentation that are outside the scope of this book. We completely agree with Boehm et al. (1999) that architecture is not an island and should be related to other important system development documents; however, all the organizations, templates, and guidelines in the Views and Beyond approach were created to capture *software architectures*. The artifacts we've prescribed let you capture "the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both," quoting from our definition of software architecture we gave at the outset.

How does the guidance in this book relate to architectures that occupy outlying regions of the topic area? To the extent that these "architectures" depend on architecture structures as captured by styles and views, the principles in this book hold. But writing down system installation procedures, for example, is not architectural. Nevertheless, the principles for sound documentation extend well beyond the realm of "mainline" architectures. Involvement of stakeholders, letting the uses of documentation guide its contents, controlling repetition, using a standard organization, avoiding ambiguity: these and other principles form the foundation of a high-quality documentation task.

Many other topics in software engineering are related to documenting software architecture. Chief among them is the general topic of software architecture. Other topics that you want to be aware of but that are outside the scope of this book are architecture description languages, commercial components, hypertext documentation, and configuration management.

E.7 A Final Word

Helping practitioners do their job more effectively is the goal of this book. We wanted to help an architect answer the question, "What do I do now?" Communicating the architecture is as important a task as creating it, for without effective communication, the architecture is nothing.

Architectures are too complex to be communicated all at once, just as a high-dimension object cannot be seen or grasped in its entirety in our three-dimensional world. As a way to divide and conquer complexity, views are by far the most effective means of architecture communication that we know. Styles and views establish a specialized and shared vocabulary, allow reuse of technical knowledge and practice from one system to the next, and facilitate analysis and prediction. Relating the views to one another and making the documentation accessible to its stakeholders completes the communication obligation to the present stakeholders. Capturing the rationale and why things are the way they are completes the communication obligation to the future.

That is the essence of documentation: recognizing and discharging the architect's obligations to the community of stakeholders, present and future, whose needs the architecture is intended to serve. We hope that we have provided guidance that will lead to high-quality products and that is also practical and flexible enough to be useful in the resource-constrained, never-enough-time environments in which all architects labor.

And we look forward to discovering what's on the horizon.

E.8 For Further Reading

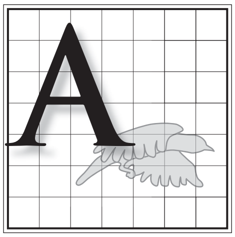
The Internet contains a wealth of information about RUP, DoDAF, and ISO 42010. A good starting point for RUP is Philippe Kruchten's original paper proposing the 4+1 approach for architecture; it is still the best introduction to that concept (Kruchten 1995).

The Web site Agile Modeling Practices, at agilemodeling.com/practices.htm, is a good repository for information about Agile practices. Other foundation works for Agile development include the following:

- Resources produced by the Agile Alliance:
  - "Manifesto for Agile Software Development," at agilemanifesto.org (Agile Alliance 2002a)
  - The Agile Alliance Web site: agilealliance.org (Agile Alliance 2002b)

  - "Principles Behind the Agile Manifesto," at agilemanifesto.org/principles.html (Agile Alliance 2002c)
- Kent Beck and Cynthia Andres's book *Extreme Programming Explained: Embrace Change* (2nd edition) (Beck and Andres 2004).
- Alistair Cockburn's books:
  - *Agile Software Development* (Cockburn 2002)
  - *Crystal Methodologies* (Cockburn 2001)
  - *Crystal Clear: A Human-Powered Methodology for Small Teams* (Cockburn 2004)
- Stephen Palmer and John Felsing's book *A Practical Guide to Feature-Driven Development* (Palmer and Felsing 2002).
- Ken Schwaber's book *Agile Software Development with Scrum* (Schwaber 2001).
- The Rozanski and Woods viewpoint set is described in detail in their book *Software Systems Architecture* (Rozanski and Woods 2005).

UML—Unified Modeling Language



A.1 Introduction

The Unified Modeling Language (UML) is a standardized visual language for modeling software designs. Originally created to merge a number of similar-but-different notations for object-oriented modeling, UML has grown to become the de facto standard for representing software designs in systems of all kinds. The purpose of this appendix is to show how UML should be used to describe different kinds of information found in software architecture documentation: module views, component-and-connector (C&C) views, allocation views, behavior documentation, and interfaces.

The appendix should work as a quick refresher to the UML diagrams and symbols that you may use or may find in architecture documentation. It's not intended to be a UML tutorial. It's assumed that you are familiar with basic UML concepts such as classes, packages, dependencies, and messages.

For a more in-depth explanation of the different UML diagrams, consult the UML books referenced in Section 2.9.

UML retains many of the characteristics that trace back to its object-oriented origins, but object-oriented abstractions are not always the best tools for describing software architectures. For example, UML has no notation for a layer, context diagram, or rich connector. Many changes were incorporated in the 2.0 revision of UML, some motivated by a need for improved architecture abstractions. Language elements such as connectors and ports were introduced to address some problems. Other elements were enriched to improve their suitability; for example, UML components now share many features with classes, such as the ability to add interfaces and behavioral descriptions.

The result is that today's 2.*x* versions of UML are better suited to documenting architectures than earlier versions, but there are still some gaps between UML and architecture abstractions, Section 3.4.3 discusses the problems with using UML concepts to represent C&C abstractions.

particularly for C&C views. This appendix focuses on guidance for documenting software architectures using UML, independent of whether UML is the best choice for each architecture documentation piece.

UML provides 14 types of diagrams divided into two categories: structure diagrams and behavior diagrams. Figure A.1 shows the hierarchy of UML 2.2 diagram types. For each concrete diagram type, the small icons indicate the kinds of information the diagram is better suited to convey.

ADVICE

You probably won't find anyone who uses all 14 types of diagrams to document a software system. It is not a goal to try. Pick a subset chosen to match the modeling tasks you have at hand. Try to use UML diagrams that your readers are familiar with and express the right meaning. Avoid any temptation to show off your UML knowledge by using uncommon UML symbols. Otherwise you may fail to communicate the design.

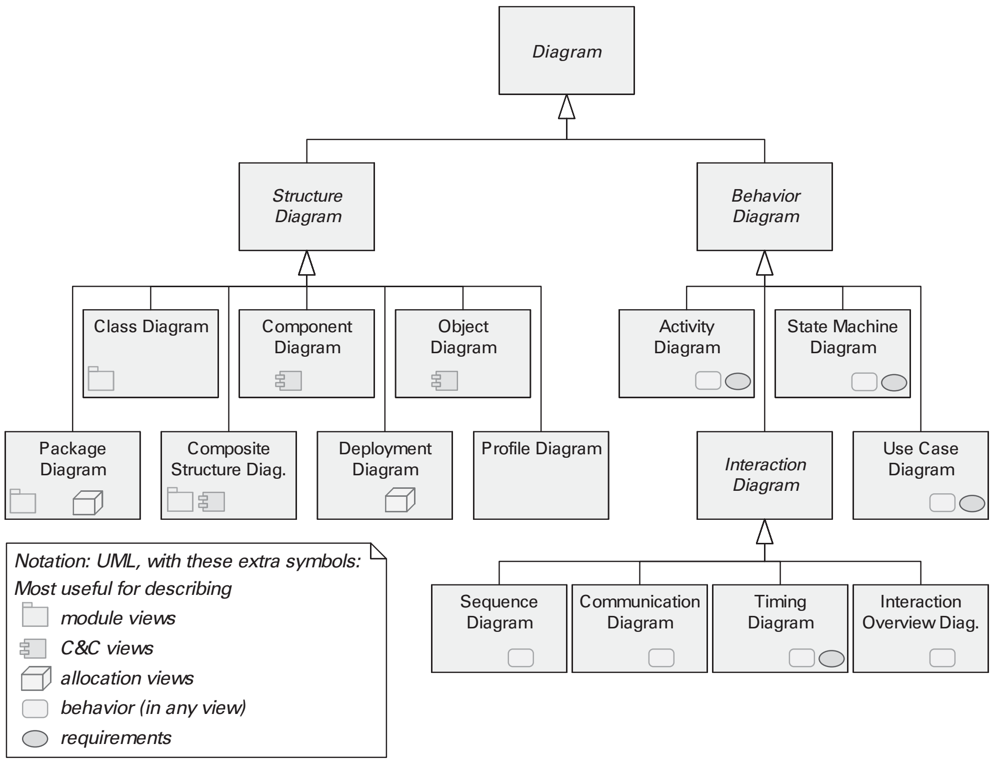


Figure A.1
UML 2.2 diagram types

The meaning of any UML symbol can be further specialized by using stereotypes. A stereotype is a domain-specific or technology-specific label shown within *guillemets* (also known as "angle brackets") that can be applied to existing UML elements and relations. The diagrams in this appendix use standard UML stereotypes where possible. However, several stereotypes are introduced to represent types of elements or relations specific to a style (such as <<layer>>).

The UML standard stereotypes are listed in Annex C of the UML specifications (version 2.2). The UML standard is maintained by the Object Management Group (OMG), and the specifications can be downloaded from uml.org.

A.2 Documenting a Module View

Module views show architecture structures where the elements are implementation units, or modules. Modules should be represented in UML as packages, classes, or interfaces. The following subsections describe how UML should be used to document different module styles and which UML symbols are most appropriate for showing modules and their relations in each style.

A.2.1 Decomposition Style

Modules, as described in Chapter 1, are typically represented in UML as packages or classes. In UML, decomposition of modules in submodules is shown by nesting packages, classes, or interfaces inside packages. Figure A.2 shows an example.

The module decomposition style is described in Section 2.1.

A.2.2 Uses Style

This style describes usage dependencies among modules. In UML, dependencies are shown using the dependency arrow.

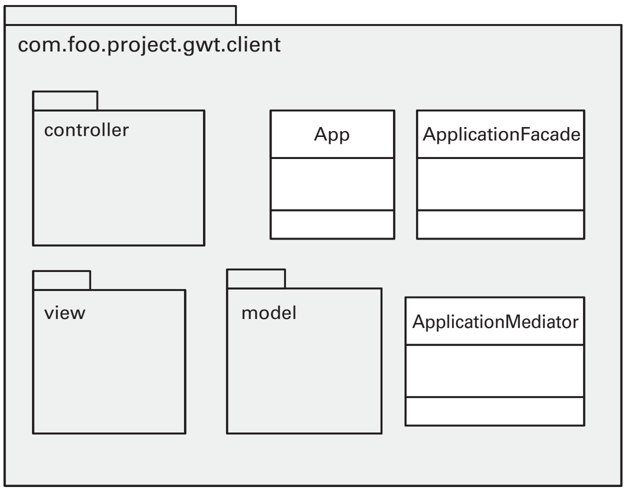
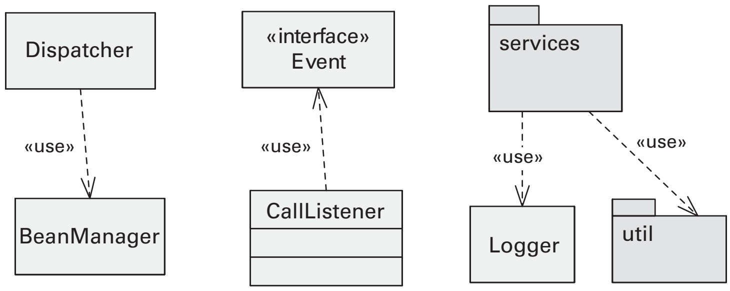


Figure A.2
UML packages and classes are used in decomposition views.

Figure A.3
UML dependencies are used in module uses views.



The uses style is described in Section 2.2.

Use the UML usage dependency (<<use>>) to show usage between packages, classes, or interfaces in a uses view. Figure A.3 shows examples of *uses* relations in UML.

A.2.3 Generalization Style

The generalization style is described in Section 2.3.

Modules in a generalization view should be represented using classes and interfaces. Generalization (*is-a* relation) between modules is shown in UML using the generalization (class inheritance) arrow. Another form of *is-a* relation, interface realization, is shown using the interface realization arrow. Figure A.4 shows an example.

A.2.4 Layered Style

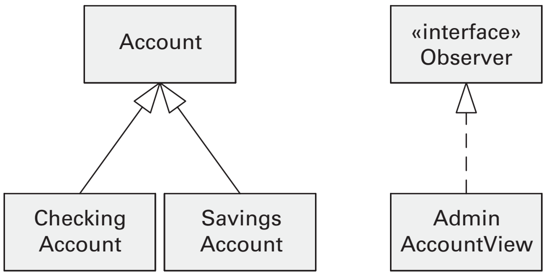
The layered style is described in Section 2.4.

UML doesn't have a built-in notation for layers. Because a layer is a grouping of modules, the natural alternative is to use packages, stereotyped as <<layer>>. The *allowed-to-use* relation between layers should be shown as a stereotyped UML dependency. Figure A.5 shows an example.

A.2.5 Aspects Style

In aspect-oriented software development, a module that is responsible for a crosscutting concern (such as international-

Figure A.4
UML class inheritance (on the left) is used in generalization views. UML interface realization (on the right) is also a kind of *is-a* relationship.



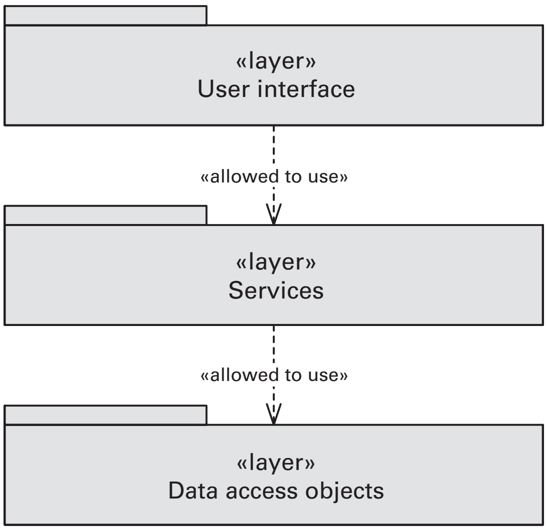


Figure A.5
Stereotyped packages can represent layers in UML.

ization) is called an aspect. UML doesn't have a built-in notation for aspects. You should use classes, stereotyped as <<aspect>>, to represent aspects. The *crosscuts* relation from aspects to classes, packages, and other aspects can be shown graphically using a stereotyped dependency arrow. However, because *crosscuts* relations are often numerous, a less-cluttered alternative is to use annotations to define what each aspect crosscuts. Figure A.6 shows an example.

The aspects style is described in Section 2.5.

A.2.6 Data Model Style

You should document a data model in UML using a class diagram. Classes should have the standard <<entity>> stereotype.

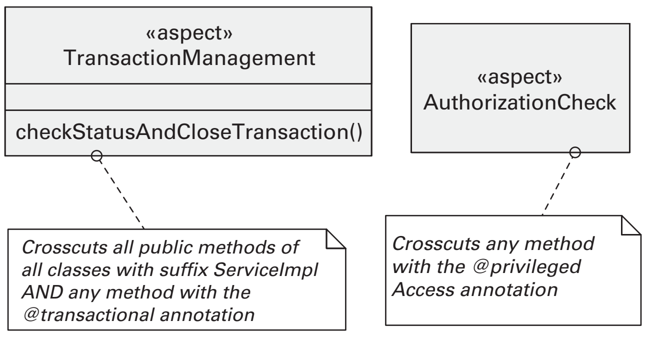

«aspect»
TransactionManagement
checkStatusAndCloseTransaction()
«aspect»
AuthorizationCheck
Crosscuts all public methods of all classes with suffix ServiceImpl AND any method with the @transactional annotation
Crosscuts any method with the @privileged Access annotation

Figure A.6
Aspects can be represented in UML as stereotyped classes. The *crosscuts* relation can be shown graphically or (to reduce visual clutter) with annotations.

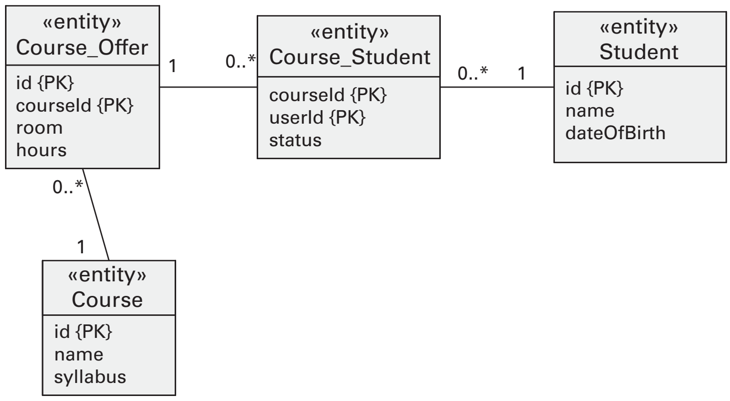


Figure A.7
UML classes and their associations can be used for data modeling.

The data model style is described in Section 2.6.

A special constraint can be used to indicate the attributes that form the primary key (PK) of an entity. Figure A.7 shows an example.

PERSPECTIVES

UML Class Diagrams: Too Much, Too Little

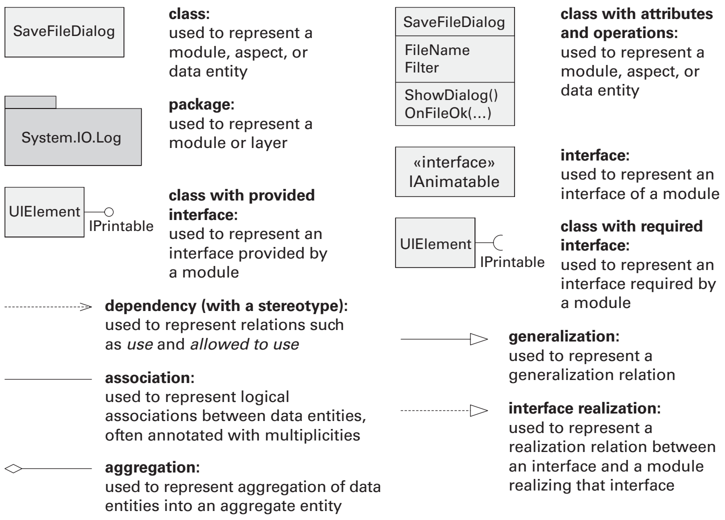
You may have noticed that UML class diagrams can be used in nearly all of the module styles covered in Chapter 2, and you might conclude that a single class diagram can represent all your module views, and maybe more.

In fact, it can. UML class diagrams are a veritable semantic smorgasbord, able to show generalization, dependency, module decomposition, general entity-relationship information, aspect modules, and interface realization. Figure A.8 compiles the UML symbols for the elements and relations usually found in class diagrams.

Good, right? Class diagrams sound like the Rosetta Stone of architecture diagrams. What else do we need?

Well, plenty. First of all, using a single class diagram to represent all possible information undercuts the primary usefulness of views. Views give us different perspectives on the various architecture structures of a system, and one of the greatest sources of confusion in architecture diagrams is the unplanned, haphazard amalgamation of various kinds of information in the same diagram.

Of course, not every view needs to be primitive or stand by itself. A source of great clarity and insight in architec-



class: used to represent a module, aspect, or data entity

package: used to represent a module or layer

class with provided interface: used to represent an interface provided by a module

dependency (with a stereotype): used to represent relations such as *use* and *allowed to use*

association: used to represent logical associations between data entities, often annotated with multiplicities

aggregation: used to represent aggregation of data entities into an aggregate entity

class with attributes and operations: used to represent a module, aspect, or data entity

interface: used to represent an interface of a module

class with required interface: used to represent an interface required by a module

generalization: used to represent a generalization relation

interface realization: used to represent a realization relation between an interface and a module realizing that interface

Figure A.8
A summary of UML symbols used in module views.

ture documentation comes when a small number of views are carefully and consciously chosen to be wed, showing various kinds of information at once and how they overlap and interplay.

Section 6.6 explains how to choose and to document combined views.

But what is produced by using all the class diagram's relations in a single view? The result would be the "*inherits/depends-on/uses/data model/realizes/decomposition*" view, which—unless your system were very small—would probably be too busy to read and too bewildering to understand. Instead, try to document the module views separately, using the restricted forms of class diagrams dictated by the module styles. Combine two views only if it makes sense to do so.

Are class diagrams rich enough to give us all we need in any module view? No. If your architecture is object-oriented, it's natural to think of it first and foremost in those terms: a collection of classes instantiated as objects that interact at runtime. You might be wondering whether you really need to document your module view as anything but that. Maybe, you think, when push com'es to shove, the only thing you give your architecture stakeholders is a set of UML class diagrams. But you need more.

First, trying to represent behavior with a class diagram is out of the question. You'll need sequence diagrams, activity diagrams, state machine diagrams, or other behavior diagrams. Second, class diagrams—even as rich as they are—are fundamentally about attributes and operations of classes, and code relations. Class diagrams have no way to represent temporal information and are not suitable to capture design rationale, variability, data flow, context (what modules are external to the system), and other important information that should be recorded in a module view.

UML class diagrams are a foundational piece of notation for module-based views. But like all good tools, they aren't for every job.

—D.G. and P.M.

A.3 Documenting a Component-and-Connector View

C&C components should be represented using UML component instances in object diagrams or component diagrams; C&C component types should be represented using UML components in a component diagram. Component types and instances should not be represented in the same diagram.

Component types and instances are distinguished in UML by the same convention used to distinguish classes and objects: names that do not include a colon (":") are types, and names that include a colon are instances, with the instance name appearing to the left of the colon and the type name appearing to the right of the colon. Anonymous instances can be documented by not including an instance name to the left of the colon and are typically used when there's only one instance of that type or the instance name is not significant.

To avoid ambiguity, always add UML ports to explicitly represent a component's points of interaction. You should label those ports. If you think representing the ports in the diagram is not necessary (perhaps because each component has only one port), it's OK to omit the UML ports and attach connectors directly to components. But use this simplification with caution, and consider mentioning your convention with a phrase in the diagram's key.

C&C component ports should be represented using UML ports. UML provided and required interfaces can be attached to ports to provide additional information, but this is usually done on component types, not instances. Ports should have an identifier and may have a multiplicity indicator.

Figure A.9 shows examples of representing a component type and instances. Components should be stereotyped to indicate the name of the corresponding component type from the style guide used for the view being documented. For example, in Figure A.9, the Catalog component type that is represented is a subtype of the server type defined in the client-server style in Chapter 4.

C&C connectors can be represented in a few ways in UML, largely depending on the amount of information you want to document in UML (as opposed to prose) or the degree to which you want to convey the connector's semantics. The two

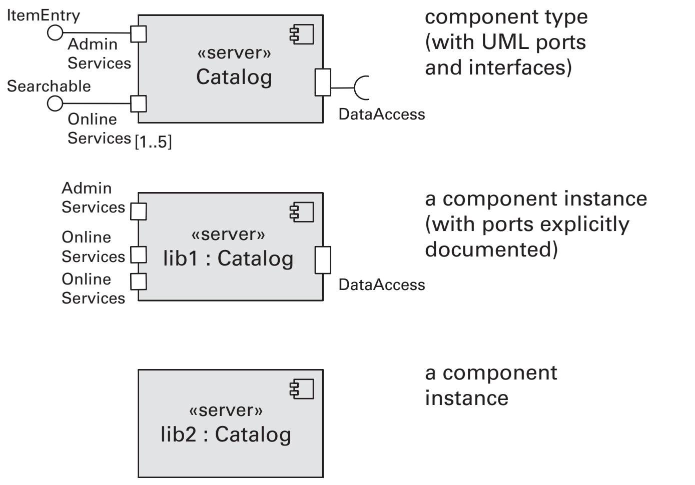


Figure A.9
Component type and instances represented using UML components. The type, `Catalog`, is a subtype of server from the client-server style. It uses UML ports and provided and required interfaces to document its ports. The `Online Services` port includes a multiplicity, constraining how many instances of that port may be on any instance of the `Catalog` component type. `lib1` and `lib2` are instances of the `Catalog` component type. `lib1` includes explicit documentation of component ports, for example, specifying that it has two instances of the `Online Services` port. `lib2` does not explicitly document its ports, leaving the number of instances of `Online Services` to be documented elsewhere.

primary options for representing a C&C connector are a UML connector and a UML component, as shown in Figure A.10.

1. A UML connector is an undecorated line. The connector's type should be denoted by adding a stereotype that identifies it. Unfortunately, UML connectors cannot have substructure, properties, or behavioral descriptions, limiting what can be documented using UML. For example, because formal interfaces (like UML interfaces or ports) cannot be added, connector roles cannot be represented. Their presence can be indicated by labeling the connector ends.

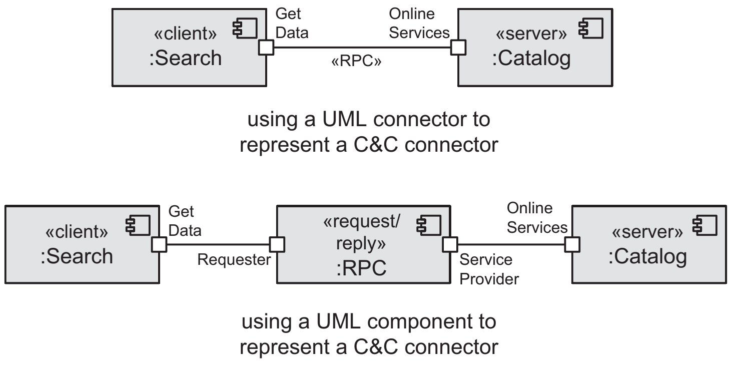


Figure A.10
C&C connectors represented using a UML connector and a UML component. In the top portion of this figure, a C&C connector is represented using a UML connector, with the type of the connector identified by the <<RPC>> stereotype. In the bottom portion, the same C&C connector is represented using a UML component. The type of C&C connector is identified in this case by the anonymous instance's type name (:RPC), which is a subtype of the style guide provided <<request/reply>> type. The UML component version allows the connector's roles to be explicitly represented using UML ports.

Alternatively, their presence can be inferred when attached to explicit component ports that unambiguously match a connector role.

2. A UML component, unlike a UML connector, can have substructure, properties, and behavioral descriptions, making it a better choice when such information needs to be documented for a C&C connector. UML ports are used to represent connector roles, just as they are used to represent component ports.

There are two variations on the UML connector strategy that can be useful in particular situations; these options are shown in Figure A.11.

See "Perspectives: Quivering at Arrows" on page 41, in the prologue.

- A navigable end (an arrowhead) can be shown on one end of a UML connector to identify a direction associated with an interaction. The documentation should identify the meaning of such arrowheads, as multiple interpretations are often possible (for example, does it represent the initiation of an interaction or the direction in which data is passed?). This option is less useful when connectors represent bidirectional interactions, such as protocols. Tool support, however, is not always available for this option (connector tool support in general is inconsistent). Instead, to use this option, you may have to use a UML association rather than a UML connector in order to add a navigable end.
- A UML assembly connector can be used in place of a simple connector. Assembly connectors are drawn using a ball-and-socket notation (the explicit connection of the provided and required interface symbols). This representation maps naturally to connectors between simple provided and required interfaces (such as simple call-return connectors). This option

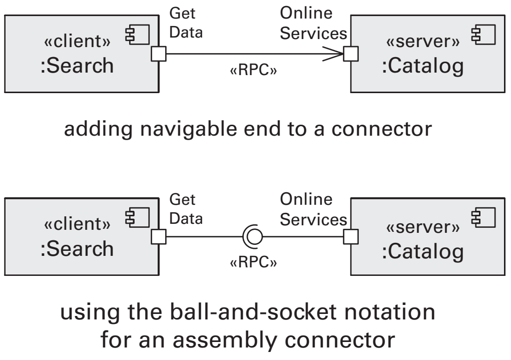


Figure A.11
Two variations of using a UML connector to represent a C&C connector. The top variation uses a navigable end (the arrowhead) to convey the general direction of the interaction. The bottom variation uses the ball-and-socket notation for an assembly connector to convey the attachment to provided and required interfaces of component ports.

is less useful when connectors do not match a provided/required intuition (for example, when an input port of a filter in a pipe-and-filter architecture does not clearly map to either a provided or required interface) or when connectors represent bidirectional interactions.

Simple representations, based on UML connectors, are good options when a connector has well-known semantics and implementations, such as procedure calls or data-read operations. When you need to do more than simply identify a connector type, a UML component representation is a good option. This option allows the explicit representation of connector roles, behavior, and substructure. However, some important properties can be represented without resorting to the use of UML components.

Tagged values can be used, as shown in Figure A.12, to associate attribute values with a UML connector. To use this approach, you create a stereotype for the connector type and define attributes that become associated with the stereotype. These attributes are called tagged values in UML and are shown in a comment box. Some UML tools allow you to create stereotypes (such as <<JMS>>) and define their attributes with name and data type (for example, `queueID : String`; `capacity : integer`; `persistent : Boolean`). Then each time the stereotype is used, the tool allows you to select the stereotyped element or relation and enter the values of the attributes in a properties box. In such cases, comment boxes will not appear in the diagram.

Component or connector substructure should be represented in UML using nested UML components and UML delegation connectors, as shown in Figure A.13. The UML components representing the substructure of a component (or connector) are nested within that UML component. The ports of the outer UML component are associated with the corresponding ports of the inner UML components using UML delegation connectors. A UML delegation connector is shown as a solid line with an open arrowhead. The arrowhead on a UML delegation connector should point inward when relating

The relationship expressed by UML delegation connectors is called interface delegation and is described in Section 3.2.3. Nesting and interface delegation are how UML represents decomposition refinement, which is discussed in Section 6.1.1.

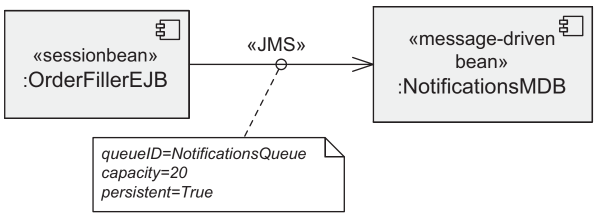
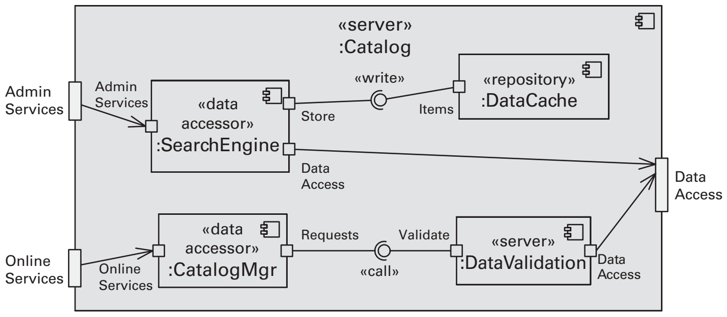


Figure A.12
C&C connectors can be represented as stereotyped UML connectors with tagged values. In this example the connector <<JMS>> represents the use of the Java Message Service, which allows you to define different kinds of message queues.

Figure A.13
Substructure of a UML component. Delegation connector arrows associate the external ports with ports of internal components.



ports that are both "provides" ports, and point outward when relating ports that are both "requires."

The pipe-and-filter style is described in Section 4.2.1.

When documenting a specific C&C view in UML, you should use a stereotype to identify the type of each component and connector, ensuring a clear relationship to the component and connector types defined in the style guides used to create the view. If application-specific subtypes of these types have been defined, those types should be identified in the names of the instances (appearing to the right of the colon). For example, Figure A.14 shows a UML diagram of a pipe-and-filter view. Filters are represented as UML components with the <<filter>> stereotype, and pipes are represented as UML connectors with the <<pipe>> stereotype. These stereotypes associate each instance with its type from the pipe-and-filter style guide. Each

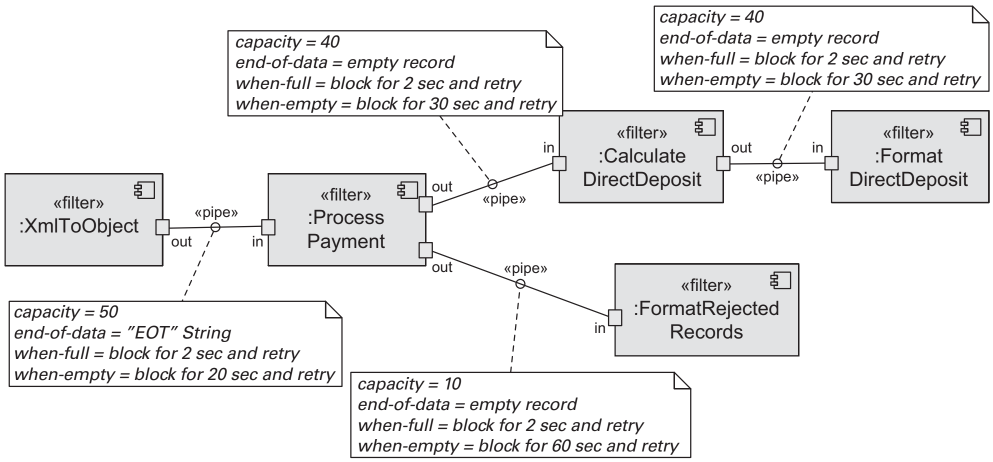

capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry
capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry
«filter»
:XmlToObject
out
«pipe»
in
«filter»
:Process Payment
out
«pipe»
in
«filter»
:Calculate DirectDeposit
out
«pipe»
in
«filter»
:Format DirectDeposit
out
«pipe»
in
«filter»
:FormatRejected Records
capacity = 50
end-of-data = "EOT" String
when-full = block for 2 sec and retry
when-empty = block for 20 sec and retry
capacity = 10
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 60 sec and retry

Figure A.14
UML diagram of a pipe-and-filter view. Filters are shown as stereotyped UML components, and pipes are stereotyped UML connectors. Four tagged values ("capacity", "end-of-data", "when-full", and "when-empty") indicate important properties of each pipe.

filter is further associated with an application-specific subtype of filter, such as `XmlToObject` or `Process Payment`. The <<pipe>> stereotype defines four tagged values to indicate important properties of pipes.

A.4 Documenting an Allocation View

Allocation views present mappings between software elements (from module or C&C views) and environmental elements. Environmental elements are nonsoftware elements (such as hardware nodes) that are somehow associated to the software elements of the system being designed. This section provides guidance on how to document environmental elements in UML and the mappings between software and environmental elements.

When documenting a software element in any allocation view, be sure to use a UML representation that is consistent with how you represented that same element in another view. For example, if you document module TransactionMgr in a uses view with a class, do not represent the TransactionMgr module in an implementation view with a package.

A.4.1 Deployment Style

The environmental elements of a deployment view are hardware elements, such as processors, memory, and network elements. These elements can be represented in UML deployment diagrams using UML nodes. A node is a computational resource, such as a laptop computer, a server machine, a router, or a mobile device. Figure A.15 shows examples of nodes.

You should add a stereotype to each node to identify different categories of computing hardware, communication appliances, and other devices, as shown in Figure A.15.

To document a deployment view of your architecture, map the software elements (elements from a C&C view) to the nodes representing hardware elements in a UML deployment diagram. You can connect UML component instances to nodes using a nonstandard stereotyped dependency (such as <<allocated to>>), as illustrated in Figure A.16.

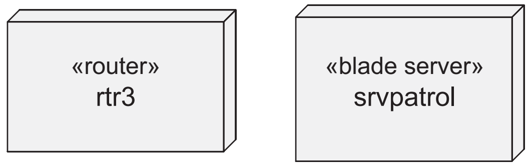


Figure A.15
UML nodes are used to represent hardware elements in a deployment view.

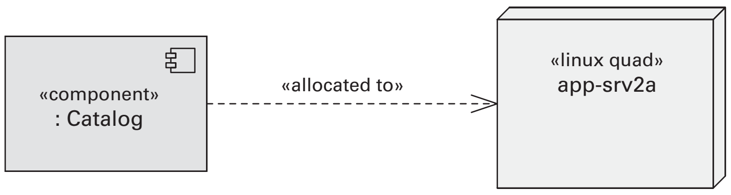


Figure A.16
Using a stereotyped dependency to show that a component is allocated (that is, deployed) to a specific node

In many cases your software system will have a large number of components deployed to the same hardware node. Trying to draw all these UML components connected to the nodes may clutter your UML deployment diagram. In that case, you may have to resort to documenting the complete allocation of components to nodes in a diagram annotation, in the element catalog of the view, or in a table that maps nodes to components.

Another alternative in representing the allocation of components to nodes is explicitly to represent the packaging of components into files (such as zip, setup, or jar files) for deployment. These files should be represented in a UML deployment diagram as *artifacts*. A UML <<artifact>> is a stereotyped class that typically represents a file, such as a script, executable, configuration file, bundle file, source file, XML file, or PDF document. The standard <<manifest>> stereotyped dependency indicates that a given element (such as a component) is manifested in an <<artifact>>; that is, the artifact contains the concrete physical representation of that element. UML also provides a stereotyped dependency, <<deploy>>, to indicate that an artifact is deployed to (that is, installed on) a node. Thus, we can show that a component is allocated to a node using an artifact as an intermediary. Figure A.17 shows three equivalent ways to repre-

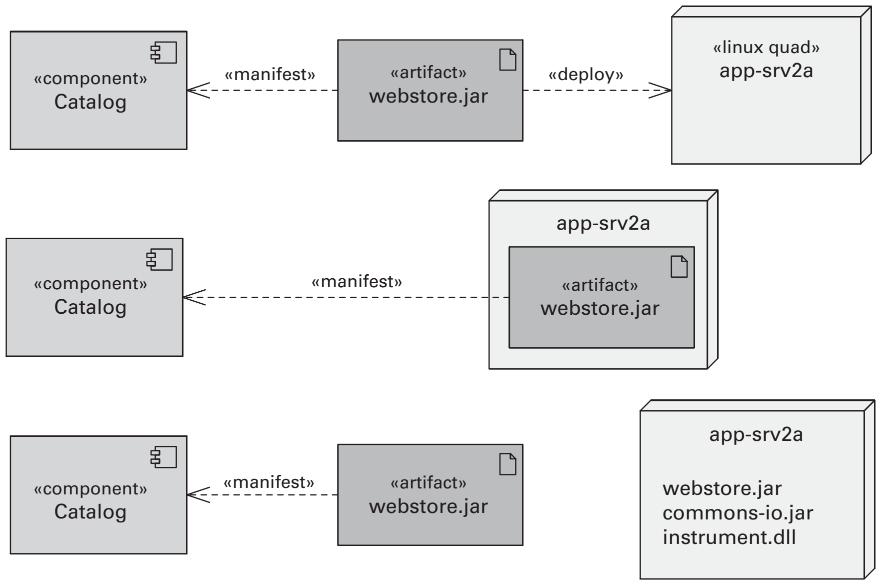


Figure A.17
Three UML alternatives that show a component that is packaged into an artifact and the node to which the artifact is deployed. In the first alternative (top), the <<manifest>> relation shows what components are encapsulated in an artifact, and the <<deploy>> relation shows what node an artifact is deployed to. In the second alternative (center), the association of components to artifacts is shown via <<manifest>>, and deployment of an artifact to a node is shown by nesting. In the third alternative (bottom), deployment of artifacts is shown separately by listing the names of artifacts inside a node (this notation is a shorthand for nesting artifacts inside nodes, but it is not supported by some UML tools).

sent that allocation in a UML diagram. The criteria to choose among these options include graphical convenience and support in the UML tool being used.

Using the alternatives shown in Figure A.17 will lead you to a view that combines the deployment and the install styles. This is because a UML deployment diagram includes some information you would typically find in an install view. For example, a deployment file in a UML deployment diagram is not an element of the deployment style as described in Section 5.2, but rather the install style, as described in Section 5.3.

Section 6.6 discusses combined views.

Deployment views often also show the communication channels between hardware elements. In a UML deployment diagram, nodes are connected to each other by communication paths, as shown in Figure A.18. These paths can have stereotypes to distinguish different kinds of communication channels (such as Internet, LAN, wireless, HTTP). Multiplicity can be used to indicate the number of instances of the node at each end of the communication path.

Figure 5.3, in Section 5.2.4, is an example of a UML deployment diagram.

A.4.2 Install and Implementation Styles

The environmental elements that are the focus of the install and implementation styles described in Chapter 5 are files and directories. These elements can be represented by UML artifacts, which are found in UML deployment diagrams. In an install view, the software elements mapped to the UML artifacts will typically be UML components in a C&C view. In an implementation view, the software elements will typically be classes or packages that are the modules in a module view of the architecture.

To show that a given software element is mapped to a UML artifact, we use the same <<manifest>> stereotyped dependency discussed for the deployment style. This <<manifest>> relationship represents that the artifact contains the concrete physical representation of the software element.

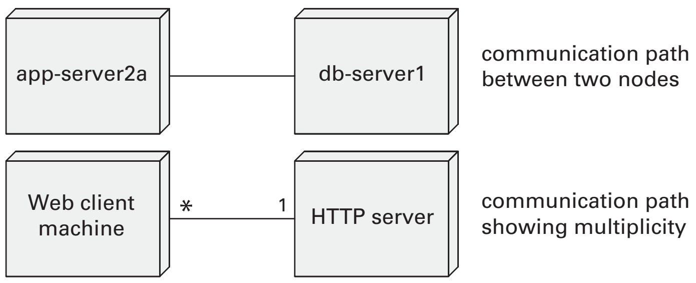

app-server2a
db-server1
communication path between two nodes
Web client machine
*
1
HTTP server
communication path showing multiplicity

Figure A.18
UML nodes are connected by communication paths that can optionally show multiplicity.

The implementation style is described in Section 5.5. The install style is described in Section 5.3.

The containment relation that exists for files and directories in the install and implementation styles can be shown for UML artifacts by nesting the artifacts.

Implementation views typically show the tree structure of files and folders in the development environment; install views show the tree structure of the installed application. A UML artifact naturally represents a file, but not a folder (directory), in the file system. An alternative is to create a stereotype to specialize the standard UML artifact to represent a file system directory. Figure A.19 is an example of an install view using regular artifacts for files and stereotyped <<dir artifacts>> for folders.

A.4.3 Work Assignment Style

In the work assignment style, the environmental elements are people or organizational units. The software elements are modules. UML doesn't have a diagram type that is intended to show work assignment information. However, if you choose UML, you should represent a work assignment view with a package diagram, using actors and packages. Figure A.20 shows a simple example. The packages represent modules from a

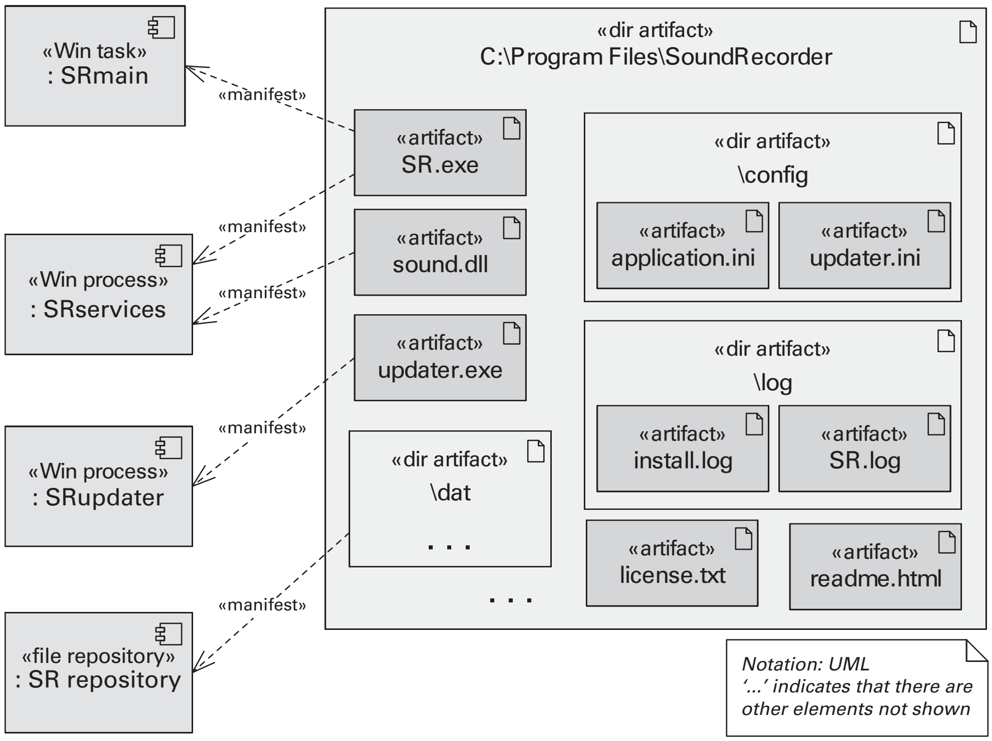


Figure A.19
Simple example of a UML diagram for an install view

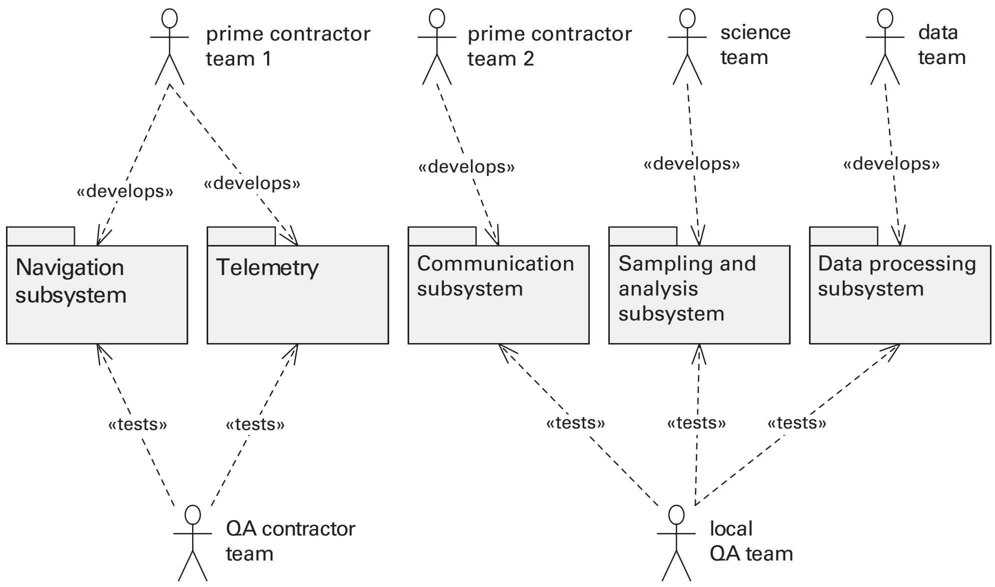


Figure A.20
Simple example of work assignment shown with UML symbols

module view, and the actors are the organizational units you're assigning the work to. Stereotyped dependencies indicate the activities being assigned.

ADVICE

Avoid UML Ambiguity Traps

UML offers some very wide-ranging modeling constructs, many of which have semantics that are open to broad interpretation. That makes it easy to create UML diagrams that, while correct, fail to convey your architectural decisions with precision or (worse) convey the wrong notion altogether. Four cases warrant special admonition:

- *Overusing class diagrams*. This topic is covered separately in the sidebar "UML Class Diagrams: Too Much, Too Little," in Section A.2.6.
- *Using dependency arrows in a C&C diagram*. It is possible to use a UML dependency to represent a connector with an arrow. This is a bad idea! Dependency is a relation typically found in a module view to depict a static relation between code elements, not in a C&C view, where relations represent runtime interactions. A dependency arrow in a C&C view may cause confusion by making your view look like a combination (an unintentional one) of C&C views and module views. Plus, *depends on* is usually just the wrong concept for a C&C view. Architects tend to use this arrow when they want to

imply directionality, as in a pipe-and-filter view, to show flow of information. But the whole point of that style is to create an architecture where the filters are independent of each other. *Depends on* is exactly the wrong thing to say (and draw) in a pipe-and-filter view.

- *Careless use of associations*. In a module view of the architecture, you may find modules represented as UML classes with UML associations between them. A navigable association from X to Y usually means that X and Y can interact in some way, and/or the state of an X object contains one or more (depending on the multiplicity) references to Y objects. Figure A.21 is an example of a UML class diagram with associations. We have observed that architects sometimes use association (improperly, in our opinion) to signify a *uses* relation. Before you use an association to connect classes in a module view, ask yourself whether the association represents just a *uses* relation. If it does, represent it using a <<use>> dependency instead. If it doesn't, make sure it's clear to the stakeholders what the associations represent.

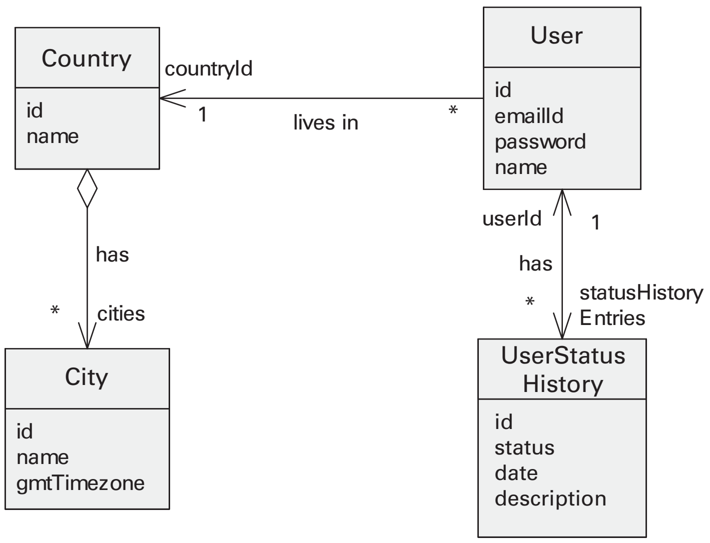


Figure A.21
Examples of UML associations between classes. Cardinality (multiplicity) is indicated by a numeric label at the association end ("*" represents "many"). The hollow diamond indicates an aggregation association, which is a logical part-whole relation. An association may imply a usage dependency relation in the direction of the navigability arrow.

- *Using types instead of instances*. Figure A.22 shows a component type (`Catalog`) being deployed to a hardware node. Although UML allows this, what does it mean? It might mean that all instances of the type are deployed to the node, or any one instance, or one particular instance, or something else. You can use a type name as shorthand for one or more instances—as long as you explain it. If you take this option, add an explanation to the diagram's key to say what you mean.

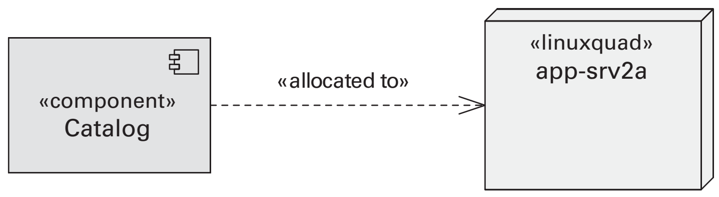


Figure A.22
Allocating a component type to a node

Ambiguities like these should be avoided wherever possible for the benefit of all stakeholders of the documentation.

A.5 Documenting Behavior

UML offers a wide variety of diagram types to model system behavior. Many of them are mentioned in Chapter 8 in this book. Behavior diagrams complement the structure diagrams found in module, C&C, and allocation views. For instance, a UML class diagram showing classes and packages and their usage dependencies can be the primary presentation of a uses view. A sequence diagram can describe the behavior of the modules (classes in the class diagram) when executing a specific trace or scenario. Table A.1 summarizes the types of behavior diagrams available in UML and when to use each one.

Table A.1 UML behavior diagram types

UML Diagram	Definition
Activity diagram	Use to describe a work flow of the system as a sequence of actions. It can show branch conditions and concurrent actions.
Sequence diagram	Use to show the explicit sequence of messages between architecture elements and participants of a specific trace. It can show conditional segments of the trace, loops, and parallel segments.
Communication diagram	Use to show the sequence of messages between architecture elements in a specific trace.
Timing diagram	Use to capture state changes along a strict time line, as well as timing constraints. Particularly useful to model real-time systems.
Interaction overview diagram	Use to compose workflows following the activity diagram notation, where the actions are themselves interaction diagrams (such as sequence diagrams or activity diagrams).
State machine diagram	Use to model the behavior of architecture elements by specifying their states and all possible transitions between states.
Use case diagram	Use to show actors and the use cases that they can perform. Use cases represent functionality of a system or parts of it.

The following subsections give a brief overview of each type of UML behavioral diagram and show the most useful symbols used in each diagram.

A.5.1 Activity Diagram

UML activity diagrams are flow charts. You should use them to describe the sequence of actions performed in a given business process of the system. They are particularly useful to describe business flows that involve concurrency (that is, actions executed in parallel). Figure A.23 shows symbols commonly used in UML activity diagrams.

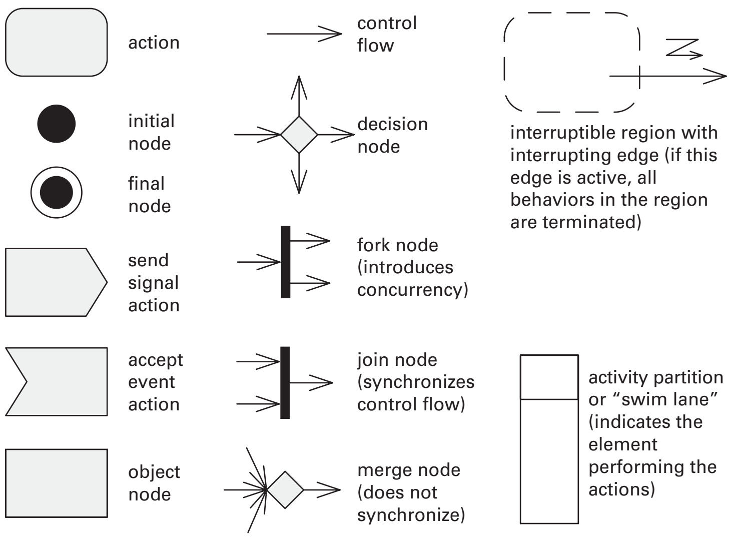
Section 8.3.2 describes UML activity diagrams as a behavioral notation.

When using an activity diagram to describe the behavior of the system, you can indicate which architecture element performs each action using activity partitions ("swim lanes"). If there is an interaction between two swim lanes, there should be a relation or connector between the corresponding architecture elements in the primary presentation where these elements are defined. Figure A.24 shows an example of an activity diagram. In this example, `Depth Meter`, `Dive Tracker`, and `Thermometer` could be modules from a module view or components from a C&C view.

A.5.2 Sequence Diagram

The UML sequence diagram should be used to describe graphically the sequence of interactions among architecture elements

Figure A.23
Symbols used in UML activity diagrams


action
control flow
interruptible region with interrupting edge (if this edge is active, all behaviors in the region are terminated)
initial node
decision node
final node
send signal action
fork node (introduces concurrency)
accept event action
join node (synchronizes control flow)
activity partition or "swim lane" (indicates the element performing the actions)
object node
merge node (does not synchronize)

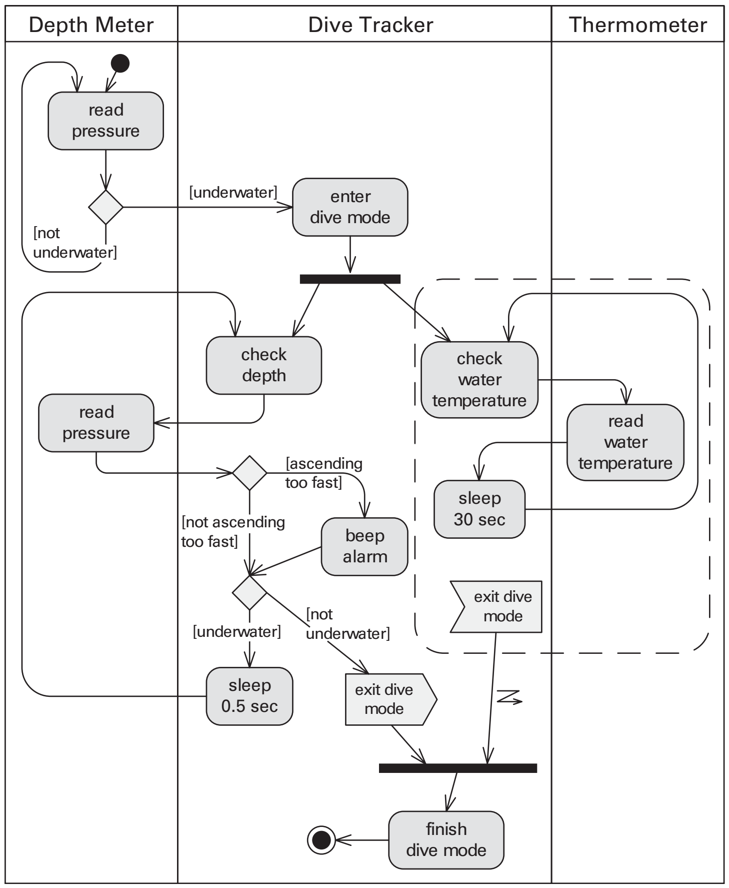


Figure A.24
Example of a UML activity diagram. It shows the flow of activities performed by a simple diving computer that registers depth of the diver (based on the water pressure) and the water temperature. `Depth Meter`, `Dive Tracker`, and `Thermometer` are architecture elements.

in a particular trace or scenario of the system. The participants in a sequence diagram are UML objects. These participants may be instances of UML classes that are modules in a module view, or UML component instances from a C&C view. If the sequence diagram shows a message from one participant to another, there should be a <<use>> dependency or a connector between the corresponding classes or components in the module or C&C view, respectively.

The basic notation for sequence diagrams is shown in Figure A.25. There are also different types of frames that can be used to organize the diagrams and express conditional flows and loops. Figure A.26 shows some of the different kinds of frames available. Figure A.27 shows the notation for timing constraints, parallel traces, and coregions, which are useful to describe behaviors in systems with strict deadlines and concurrent tasks.

Chapter 8 has some examples of sequence diagrams (see Figures 8.4 and 8.5). Another example can be found in the software architecture document that accompanies this book online at wiki.sei.cmu.edu/sad/index.php/Workflowmanager_Module_Uses_View.

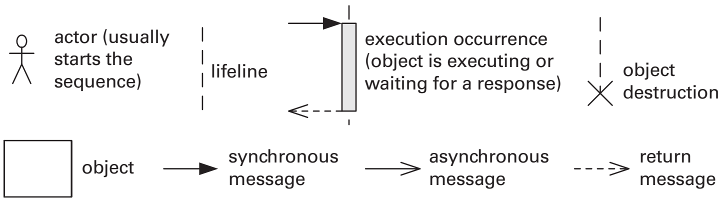

actor (usually starts the sequence)
lifeline
execution occurrence (object is executing or waiting for a response)
object destruction
object
synchronous message
asynchronous message
return message

Figure A.25
Basic notation for UML sequence diagrams

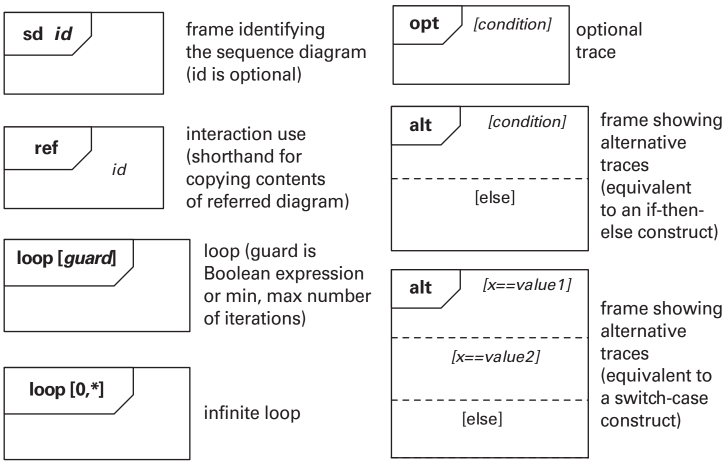

sd id
frame identifying the sequence diagram (id is optional)
opt [condition]
optional trace
ref id
interaction use (shorthand for copying contents of referred diagram)
alt [condition] [else]
frame showing alternative traces (equivalent to an if-then-else construct)
loop [guard]
loop (guard is Boolean expression or min, max number of iterations)
alt [x==value1] [x==value2] [else]
frame showing alternative traces (equivalent to a switch-case construct)
loop [0,*]
infinite loop

Figure A.26
Some of the frames available in the UML sequence diagram notation

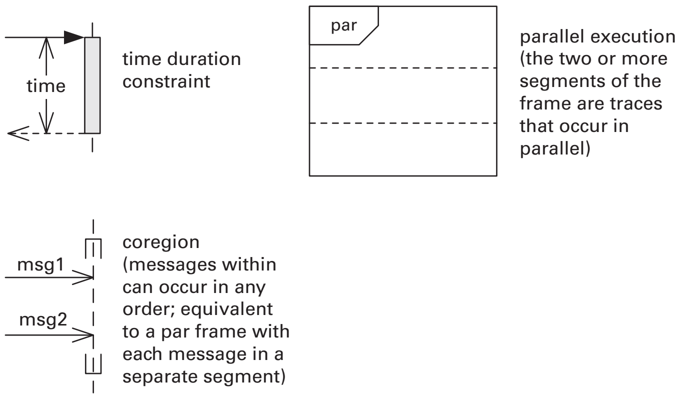

time
time duration constraint
par
parallel execution (the two or more segments of the frame are traces that occur in parallel)
msg1
msg2
coregion (messages within can occur in any order; equivalent to a par frame with each message in a separate segment)

Figure A.27
UML sequence diagram notation for timing constraint, parallel traces, and coregions

Sometimes an object receives a call when it's already executing another call. This reentrant call is represented by an overlapping execution occurrence bar, as shown in Figure A.28.

A special case of a reentrant call is when the object makes a call to itself. The notation for self calls is not defined in the

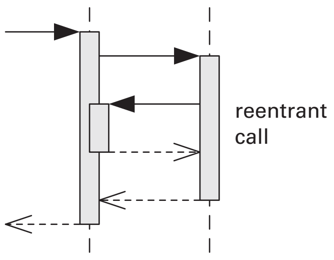


Figure A.28
Showing reentrant calls in a UML sequence diagram

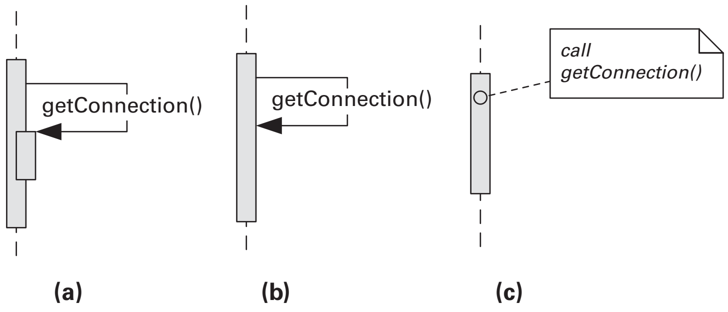


Figure A.29
Options to show self calls in a UML sequence diagram

UML specifications. A common alternative is to use an overlapping execution occurrence and a self message (see Figure A.29(a)). Showing the new execution occurrence is especially useful if you want to indicate other calls that are made within that execution. A valid simplification is to show the self message but omit the overlapping execution occurrence bar (Figure A.29(b)). The third alternative, also valid, is to simply indicate in a comment box that an internal call takes place at that point (Figure A.29(c)).

A.5.3 Communication Diagram

Akin to sequence diagrams, communication diagrams should be used to describe the sequence of interactions among architecture elements in a specific trace or scenario. The architecture elements may be objects (instances of classes from a module view) or component instances from a C&C view. The notation for UML communication diagrams is straightforward, as shown in Figure A.30.

Section 8.3.1 contains a subsection about communication diagrams, with an example.

A communication diagram shows a particular trace. There is a line between two objects if they interact in that trace. The line is labeled with an arrow, an operation name, and a number 1, 2, 3, and so on, to indicate the order of the interactions. In

Figure A.30
Basic notation for UML communication diagrams

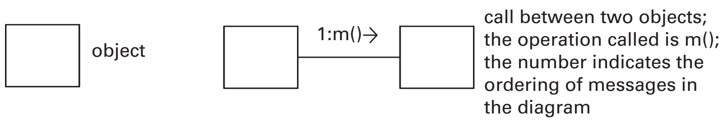
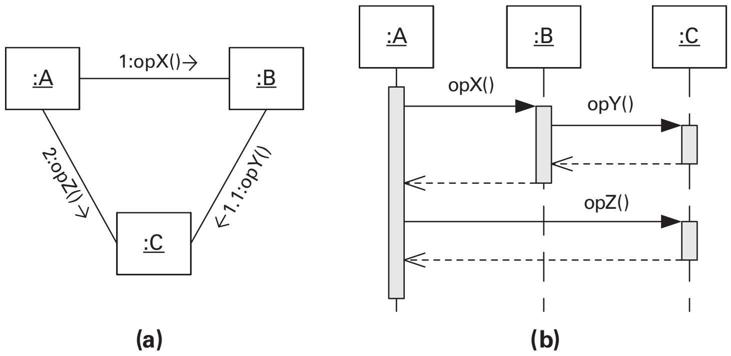

object
1:m()→
call between two objects; the operation called is m(); the number indicates the ordering of messages in the diagram

Figure A.31
Notional example of communication diagram and corresponding sequence diagram that illustrates how calls are numbered in a communication diagram



reality the numbering is not that simple. If an operation call is number *n* in the sequence and if the execution of that call triggers another call, this new call will be numbered *n*.1. If there's a third nested call, it will be *n*.1.1, and so forth successively. Once all nested calls within the execution of *n* are complete, call *n*+1 takes place. Figure A.31 illustrates this idea, showing a simplistic communication diagram on the left and the equivalent sequence diagram on the right.

A.5.4 Timing Diagram

Section 8.3.2 describes the difference between trace-oriented and comprehensive model notations.

A UML timing diagram is particularly useful when you need to describe how the architecture elements interact and change state along a strict time line, as in real-time systems. A timing diagram is a trace-oriented notation; that is, each diagram depicts the behavior of the architecture for a particular trace or scenario.

A timing diagram shows the state changes of one or more objects along a horizontal time scale. These objects may represent modules from a module view or component instances from a C&C view. If the diagram shows multiple objects, in addition to state changes, the timing diagram can display the messages between objects that cause state changes. The diagram can also display duration constraints to emphasize particular timing restrictions. Figure A.32 gives an example of a timing diagram.

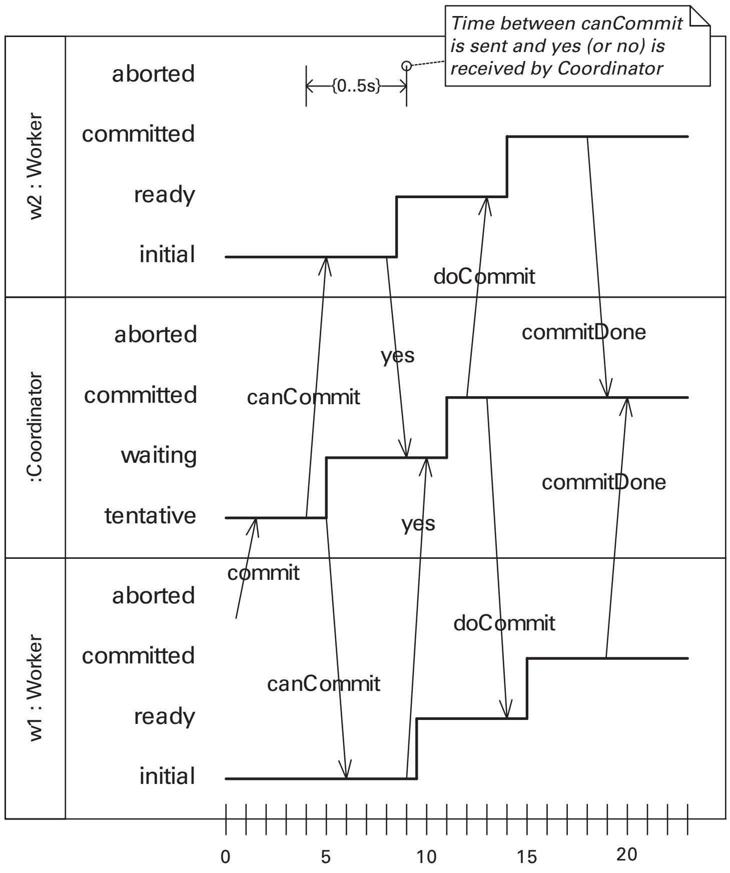


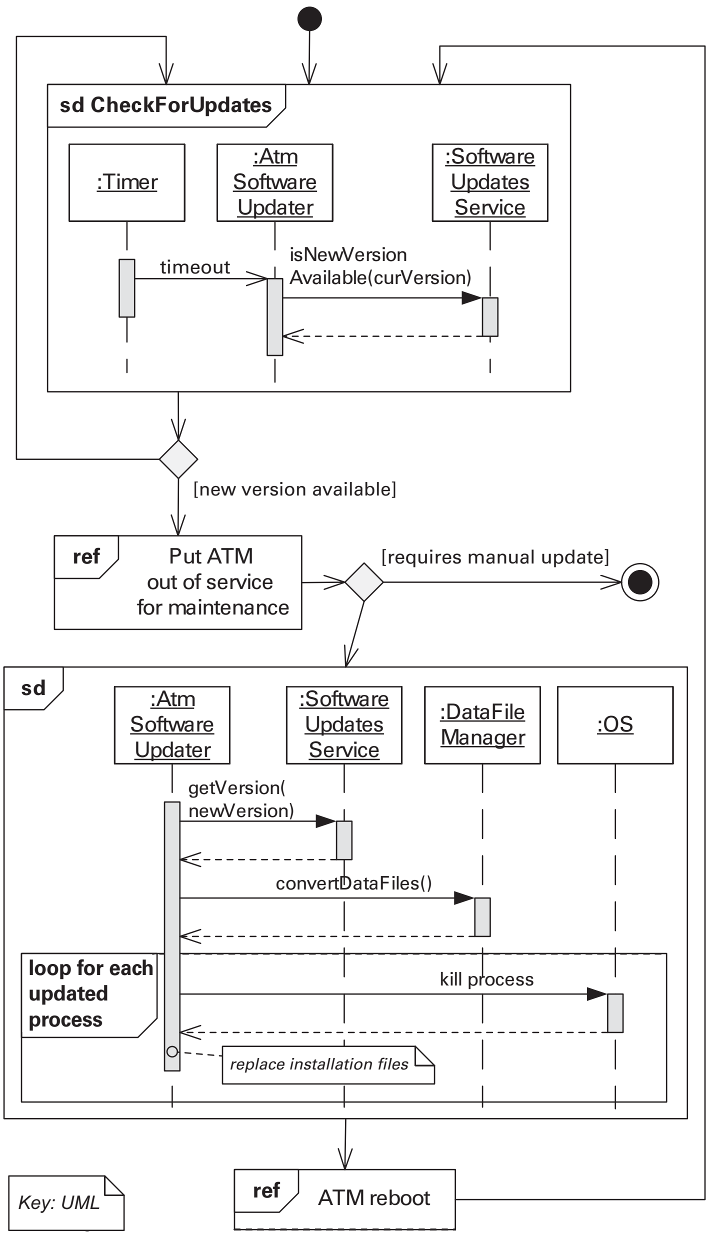
Figure A.32
Example of UML timing diagram showing the state changes and messages for a successful "commit" transaction in the two-phase commit protocol when there are two participants (workers). A duration constraint indicates the coordinator can wait up to 5 seconds for the "yes" (or "no") response from a worker.

A.5.5 Interaction Overview Diagram

UML interaction overview diagrams can be used to describe behavior in architecture views that show interactions of large-scale elements. They are useful to compose existing sequence diagrams, communication diagrams, and other interaction diagrams.

The interaction overview diagram uses the basic notation of activity diagrams to show a composition of work flows; the actions in an interaction overview diagram are replaced with interaction diagrams or references to interaction diagrams (defined elsewhere in the documentation). An interaction diagram (see Figure A.1) can be a sequence diagram, a communication diagram, a timing diagram, or an interaction overview diagram. Thus, an interaction overview diagram can have decision diamonds, initial and final nodes, and fork and merge nodes for concurrency. However, instead of rounded rectangles for actions, we have frames that either define an interaction diagram inline or reference an existing one. Figure A.33 is an example of an interaction overview diagram where two interaction diagrams (sequence diagrams in this case) are shown inline, and two other interaction diagrams are referenced.

Figure A.33
Example of an interaction overview diagram for the automatic updates feature of an ATM



A.5.6 State Machine Diagram

State machine diagrams should be used to model the behavior of architecture elements or groups of elements that go through multiple states and transitions that are clearly identifiable. A state machine can describe possible states and transitions for modules from a module view, components from a C&C view, hardware elements or communication channels from a deployment view, and so on. The UML notation for state diagrams is very rich. In addition to the basic symbols for states and transitions, the notation allows the representation of other useful information, such as the following:

- Initial and final (pseudo-) states.
- Composite states, which are states that have one sub-state machine or multiple concurrent sub-state machines (multiple regions).
- A history (pseudo-) state that represents the fact that a sub-state machine "remembers" its last state when control comes back to it. A history state has a transition to the "default" state that becomes active when the sub-state machine is entered for the first time.
- Guard constraints on transitions. When the event that fires a transition occurs, the transition is enabled only if the guard constraint evaluates to true.
- Entry and exit actions on states, which represent behavior that is executed when the state is entered or exited, respectively.
- Effect on a transition, which is behavior executed when the transition fires.

Chapter 8 discusses UML state machines as a notation for behavior documentation. Figures 8.8 and 8.9 in that chapter show other examples of UML state machine diagrams.

Figure A.34 shows the basic elements of the UML notation for state machine diagrams. Figure A.35 is an example of a state machine diagram.

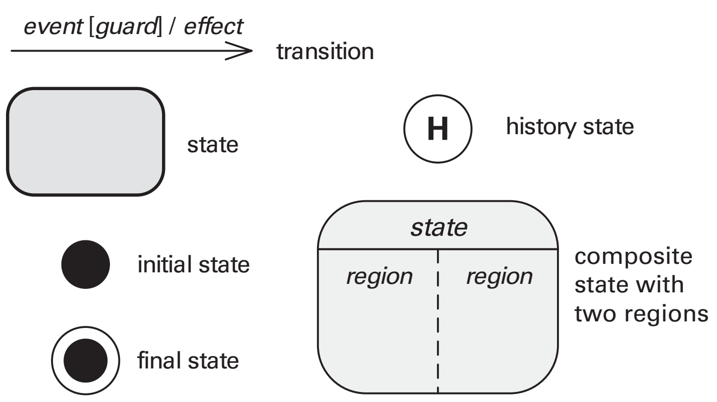


Figure A.34
Notation for UML state machine diagrams

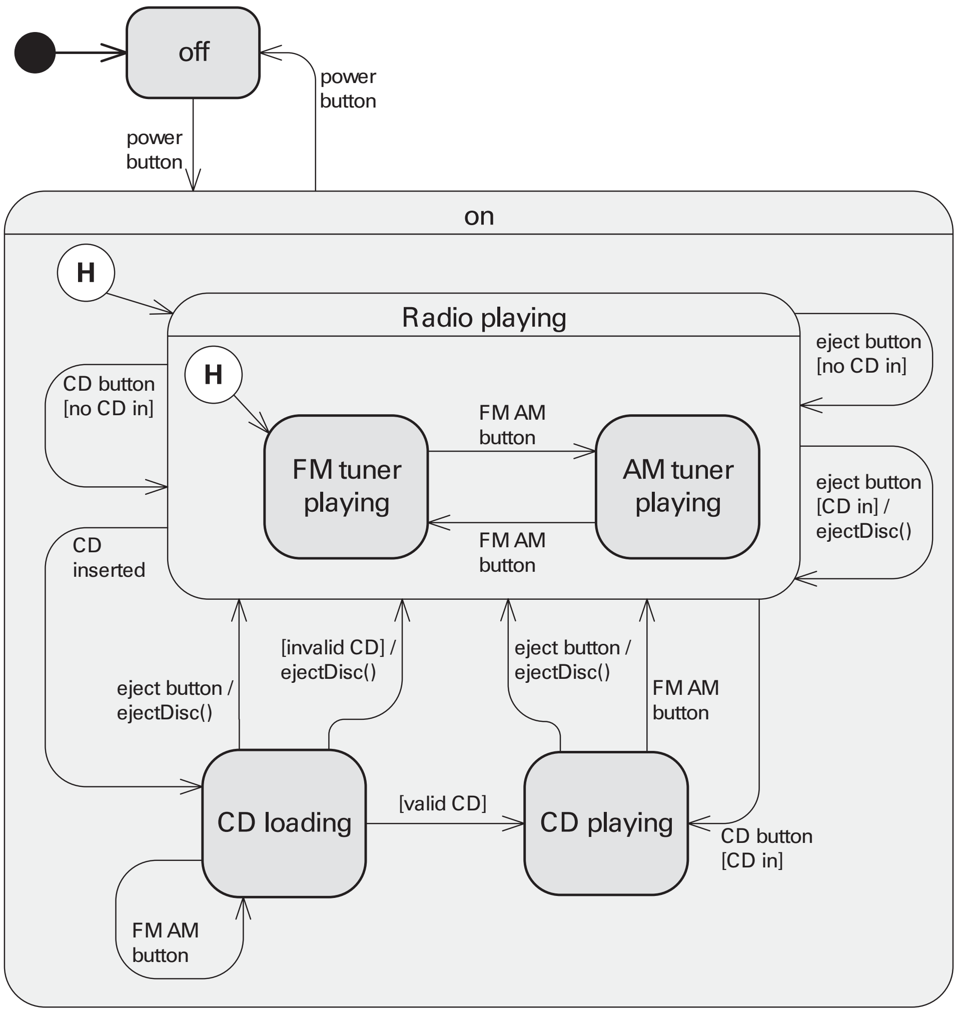


Figure A.35
UML state machine diagram for a car stereo that has an AM/FM tuner and a CD player. The events correspond to the user action of pressing the power, eject, "FM AM," or "CD" button, or inserting a disc. The history states tell that the FM tuner is activated when the stereo is turned on for the first time, and from then on the system will remember whether the radio (FM or AM) or the CD was playing last.

Section 8.3.1 discusses use cases as a notation for behavior documentation. Figure 8.2 is an example of use case diagram.

A.5.7 Use Case Diagram

You should create use case diagrams to specify the features, operations, or actions available in the system, that is, what the system is supposed to do. The actors involved in each use case are also indicated. Actors are human or nonhuman entities outside the system. A typical use case does not show architecture elements but rather an overview of the behavior the system provides. Thus, use cases frequently capture the functional requirements for a system.

The basic notation for use case diagrams consists of use case ovals and actors, and straight lines to show the associations of actors to use cases. You can draw a rectangle around a group of use cases to demarcate the functionality of a *subject* (a system or subsystem). It's also possible to use generalization to show hierarchies of actors or use cases. Figure A.36 shows the basic notation for use case diagrams.

Two relations that can be specified between use cases are these:

- *Extend.* If use case A extends use case B, the behavior specified by A is conditionally inserted into B. Imagine that use case B has an extension point where use case A can be "plugged in." If a certain condition—often specified in a comment note—is true, use case A is executed. Use case B remains independent of A. Figure A.37 shows an example.
- *Include.* The behavior of the included use case is inserted into the including use case(s). The included use case is not optional and the including use case depends on it. An included use case can be used to factor out behavior that can be reused by multiple use cases. Figure A.37 shows an example.

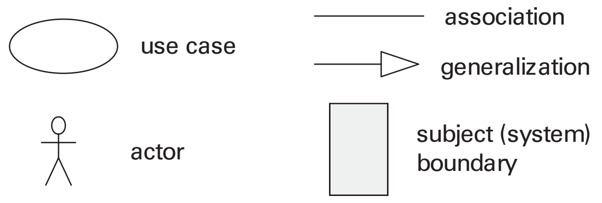


Figure A.36
Symbols used in UML use case diagrams

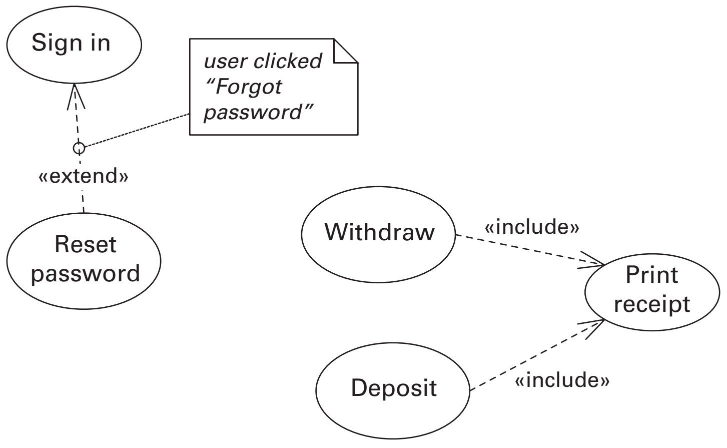


Figure A.37
On the left is an example of the extend relation in a UML use case diagram. The behavior in "Reset password" is conditionally inserted into an appropriate spot in "Sign in," but "Sign in" remains independent of "Reset password." On the right is an example of the include relation. Behavior in "Print receipt" is inserted into the behavior of "Withdraw" and "Deposit"—they depend on the execution of "Print receipt."

A.6 Documenting Interfaces

Architectural elements of different kinds have interfaces across which they interact and communicate with each other. Interfaces of modules and components are represented differently in UML.

Chapter 7 discusses the documentation of software interfaces. Section 7.2.1 provides advice on how to represent interfaces in diagrams, including UML diagrams.

Module interfaces should be documented using UML provided and required interfaces. When a module provides (that is, realizes or implements) an interface, this should be depicted as a provided interface in UML (a lollipop symbol). An interface can also be represented in UML as a stereotyped class, which makes it easier to see operations and attributes of the interface. A realization arrow is used to indicate that a given module provides that interface. Figure A.38 shows both alternatives for depicting provided interfaces.

To indicate that a module requires an interface, you should use a UML required interface (a socket symbol) attached to the class representing the module. It is common to avoid documenting required interfaces as sockets; instead, a provided interface can be represented by drawing a <<use>> dependency from the module requiring the interface to that interface. Figure A.39 shows both options.

Section 3.4.3 has an advice box about representing components, ports, and connectors in UML.

OCL is an OMG standard, and the specifications can be found at omg.org/spec/OCL.

Interfaces in C&C views are called ports (component interfaces) and roles (connector interfaces). Component ports should be represented using UML ports, optionally augmented with UML interfaces (both provided and required, as for modules). A port can include any number of provided and required interfaces, in any combination. UML interfaces can be attached to a port when you want to indicate the operations or attributes provided or required at that port. Ports can also include a multiplicity (typically only on component types), restricting how many occurrences of that port can be found on any corresponding instance.

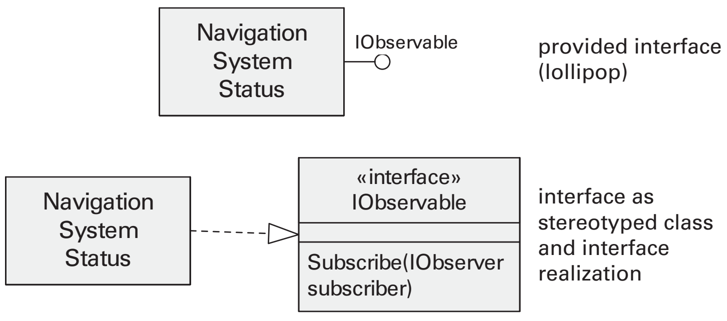

Navigation System Status
IObservable
provided interface (lollipop)
Navigation System Status
«interface» IObservable
Subscribe(IObserver subscriber)
interface as stereotyped class and interface realization

Figure A.38
Two alternatives for showing in UML that an interface (IObservable in this example) is provided (that is, realized or implemented) by a class (NavigationSystemStatus)

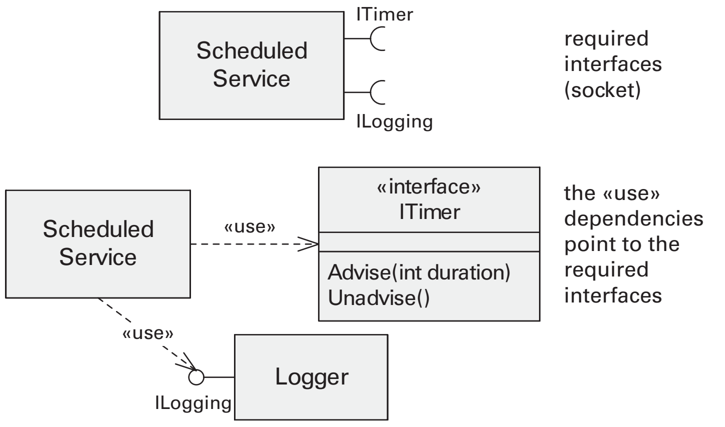


Figure A.39
Two alternatives for showing in UML that an interface is used (that is, required) by a class

Representing connector roles is more difficult in UML. When connectors are represented using UML connectors, UML ports cannot be used. Instead, roles can be at best identified by labeling the connector ends. When connectors are represented using UML components, however, UML ports can be used to represent roles (just as for component ports).

ADVICE

UML interfaces describe the syntax of operations and attributes. To capture semantics, error conditions, and quality attributes of the interface resources, you can use comment boxes in the diagram or the element catalog of your architecture view. Semantics and usage constraints on an interface can also be documented using the Object Constraint Language (OCL). OCL is a formal declarative language that operates on UML models.

PERSPECTIVES

UML Tools

The landscape of UML tools is populated with a wide range of commercial and free tools. When I teach software architecture to practitioners, I'm often asked what the best UML tools are. I always reply with the usual answer: "It depends." And it really does. UML tools these days do much more than create UML models and diagrams. Some tools offer:

- Reverse engineering
- Code generation

- Model-driven architecture (MDA) compliance
- Compiling and debugging code
- Requirements mining
- Project management aid
- Designing aid
- Support to software development processes (such as the Rational Unified Process)
- Code complexity analysis and automatic refactorings
- Modeling using other languages (such as Business Process Modeling Notation, or BPMN, and entity-relationship diagrams)
- Impact analysis
- Calculation of winning lottery numbers (No, not that.)

I've been involved in the evaluation of UML tools several times. As a practitioner, I work with UML tools on a daily basis. Currently I work with three or four different tools in different projects, and I can't help thinking at times how much I wish I had that other tool in front of me. Extra time you and your peers spend because you are not using the best tool for the job usually costs far more than the tool itself. So, choosing wisely may spare you a lot of pain and cash in the long run.

There are basically two categories of UML tools (or software design tools in general): modeling tools and drawing tools. A UML modeling tool will allow you to draw UML diagrams and will catalog in a model all elements and relations that you define in the context of a project. Thus, when you add a message from object ":A" to ":B" in a sequence diagram, the tool can prompt you to choose one of the operations you previously defined for class "B" in a class diagram. On the other hand, a drawing tool or diagramming tool will let you draw UML diagrams without creating a model underneath. The whiteboard or piece of paper where you sketch design diagrams is the simplest form of drawing tool. A sophisticated one is, for example, Microsoft Visio with Pavel Hruby's UML 2 stencil (available at softwarestencils.com/uml).

Many organizations apply a lot of effort to adopt a UML tool. Some of them buy a powerful UML modeling tool, configure the tool on everybody's machine, train the people, and then what happens? Months later they realize that only 10 or 15 percent of what the tool offers is used, or most people simply use the tool as a drawing tool. The first step to choose a UML tool is to define the evaluation criteria, which should be based on well-thought-out requirements. Here are some recommendations for your next quest for the right UML tool:

- The requirement can't be just "I need a good UML tool." The tool should have the features you need. Examples: you may need a tool that does both reverse engineering and code generation (round-trip engineering); you may be looking for a UML tool that has timing diagrams—not all UML tools support all UML diagrams.

- The requirements should come from the people who are going to use the tool. Sometimes management buys a tool to "help out" without consulting with the target tool users.
- The tool should match the skill set of the people who need to use it.
- Consider the geographic distribution of your team. Some tools have better support for distributed teams.
- If you have a software development process in place, the tool should support the process. It's much harder to try to adapt the process to fit the tool.
- Think about the cost of tool support. For free tools, a popular product with a large user base represents greater hopes of finding solutions for the problems you may encounter.
- Don't blindly trust tool advertisements and published tool rankings. The evaluators ranked the tools against their criteria, not yours.

As you may have suspected, for several reasons I'll close this sidebar without expressing my preference for any UML modeling tool. The fun is in finding the right one . . . for you. Just remember that the right tool is the one that makes your job easier.

—P.M.

This page intentionally left blank

B SysML—Systems Modeling Language

With John D. McGregor

Although not intended as a dedicated architecture description language, the Systems Modeling Language (SysML) provides sufficient constructs to meet many of the needs of a systems engineer. The engineer can represent the topology of the hardware and allocate software units to those hardware units. It is possible to represent the various architecture views needed to document a software architecture and particularly to show combined views of hardware and software.

SysML is a general-purpose systems-modeling language intended to support a broad range of analysis and design activities for systems-engineering applications. Systems engineers begin with a general problem statement, evolve toward a more specific problem statement, and eventually allocate portions of the problem to various solution elements. SysML is defined so that sufficient detail can be specified to support a variety of automated analysis and design tools.

SysML is a standard maintained by the Object Management Group (OMG) and was developed by OMG in cooperation with the International Council on Systems Engineering (INCOSE). SysML was developed as a profile of the Unified Modeling Language (UML). Being a profile means that SysML reuses much of UML, but it also provides the extensions necessary to meet the needs of systems engineers. The extensive overlap facilitates the interactions between systems engineers writing in SysML and software engineers writing in UML. SysML retains the extensibility of UML by including the UML elements necessary to define the SysML constructs.

The SysML standard defines several diagram types, shown in Table B.1. The first column lists those UML diagram types that SysML reuses unchanged. The second column lists those UML

Table B.1 SysML diagram types

As Is	Modified	Unique
Sequence (sd)	Activity (act)	Requirements (req)
State (stm)	Block (bdd)	Parametric (par)
Use case (uc)	Internal block (ibd)	
Package (pkg)		

diagram types that have been modified. The third column lists those diagram types unique to SysML. As a convention, SysML diagrams have an enclosing frame with a diagram type designator. The two or three letters following each name in the table form the designator used in the enclosing frame of each diagram to identify the diagram type. A SysML model is composed of several diagram instances, usually from several different diagram types.

B.1 Architecture Documentation

A SysML model is an aggregation of diagram instances, which together completely describes the target. SysML can be used to construct an ISO/IEC 42010-compliant description, or it may describe the architecture of the system in one model. A SysML model is typically organized using packages, each of which defines a namespace. A package contains a set of diagrams and may import diagrams from other packages. SysML supports the standard definitions of viewpoint and view and has stereotypes for each one. In SysML, a view is represented as a package that contains information conforming to a specific viewpoint. Figure B.1 shows an example of a viewpoint and three conforming views described in a SysML block diagram.

B.2 Requirements

SysML provides a means of establishing traceability among requirements and from requirements to their implementation as described in the architecture. Figure B.2 illustrates these relations. Requirements are related to each other for a variety of reasons, including one requirement being derived from another. A requirement can be related to the elements that satisfy the requirement through the *satisfy* relationship. This technique links the requirements to the architecture; for example, Payment is represented in both Figure B.2 and Figure B.3.

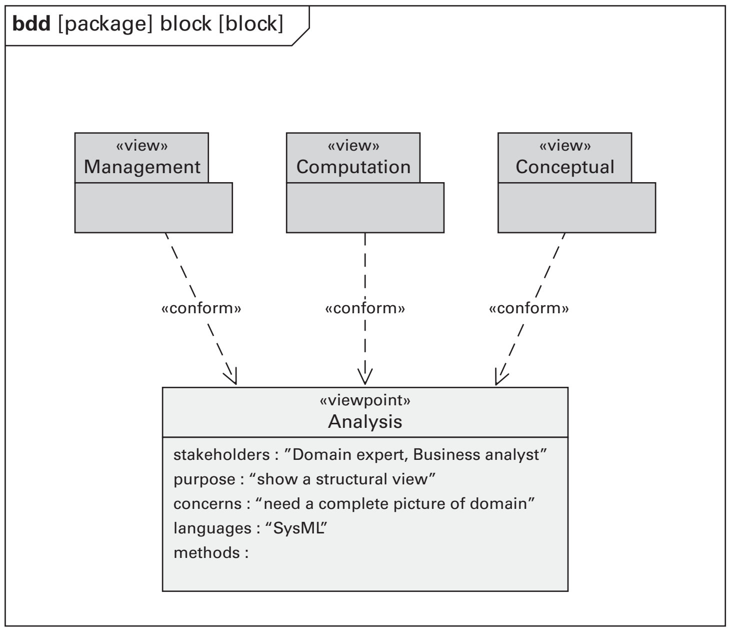

bdd [package] block [block]
«view» Management
«view» Computation
«view» Conceptual
«conform»
«conform»
«conform»
«viewpoint» Analysis
stakeholders : "Domain expert, Business analyst"
purpose : "show a structural view"
concerns : "need a complete picture of domain"
languages : "SysML"
methods :

Figure B.1
SysML block diagram

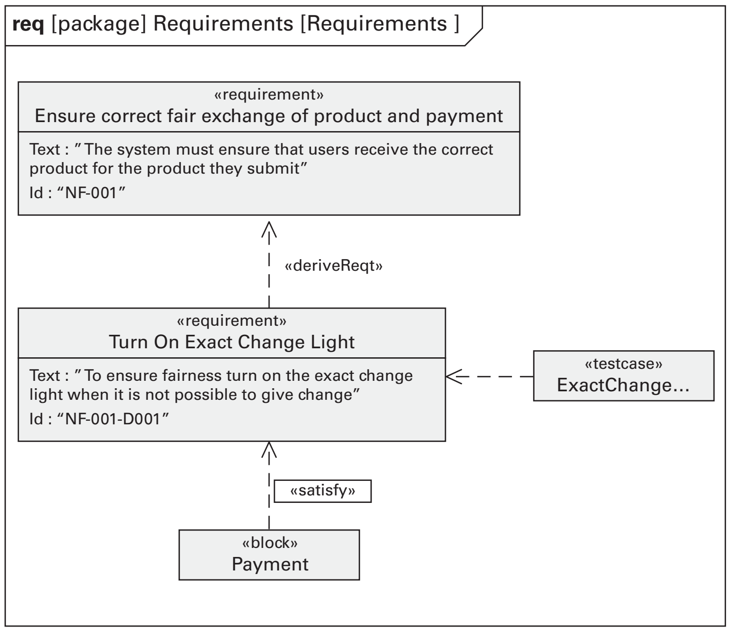

req [package] Requirements [Requirements]
«requirement» Ensure correct fair exchange of product and payment
Text : "The system must ensure that users receive the correct product for the product they submit"
Id : "NF-001"
«deriveReqt»
«requirement» Turn On Exact Change Light
Text : "To ensure fairness turn on the exact change light when it is not possible to give change"
Id : "NF-001-D001"
«testcase» ExactChange...
«satisfy»
«block» Payment

Figure B.2
SysML requirement diagram

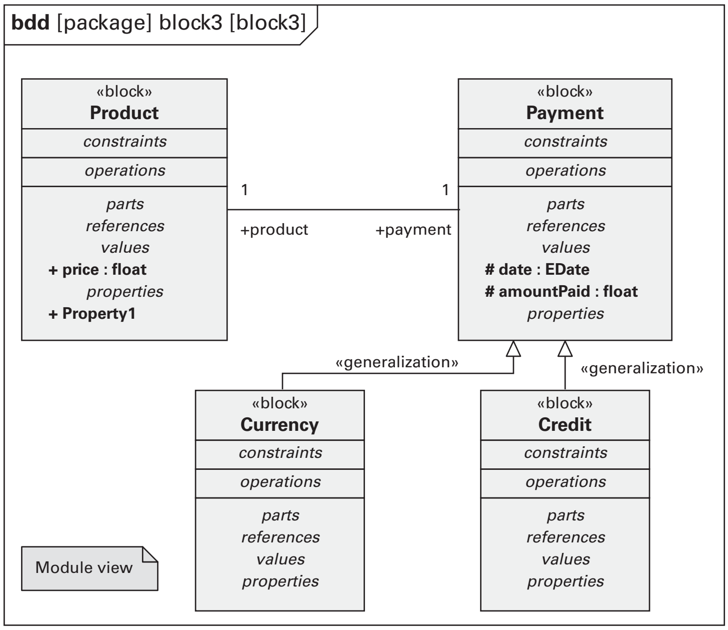
B.3 Documenting a Module View

Table B.2 provides a mapping of the requirements of a Views and Beyond module view to the SysML block diagram. The block is a SysML model element that is similar to a class in UML. The block diagram illustrates the relations among a set of blocks, including the usual *is a*, *depends on*, and *is part of*. This is basic structural information that will be referenced by other views. Figure B.3 shows an *is-a* module view using the SysML block diagram.

Table B.2 Mapping the concepts of a module view to SysML

Module	SysML
Module	Block
Is-a, *is-part-of*, *depends-on* relations	All these relations
Name, responsibilities, implementation information	Name, operations, and properties
Properties of relations	Name, visibility, and numerous other properties, plus the possibility of defining additional ones

Figure B. 3
Generalization view in SysML



B.4 Documenting a Component-and-Connector View

The block and internal block diagrams can be used together to provide a C&C view. The block diagram is used to define the component types and their relations. The internal block diagram is used to represent the component instances and their connections. Table B.3 shows the mapping of the requirements of a C&C view to SysML. Figure B.4 shows a C&C view of the blocks in Figure B.3.

Table B.3 Mapping the concepts of a C&C view to SysML

C&C	SysML
Principal processing units and data stores	Blocks and parts
Interaction mechanisms	
Attachments	Flow ports and item flows
Connector	Connector; connectors connect out port to an in port
Name of component, type, and other properties	Name, type, and any properties from the block
Name of connectors, type, and other properties	Name only; other properties can be added either by a comment or by a stereotype
No fixed topology	Topology to fit the problem

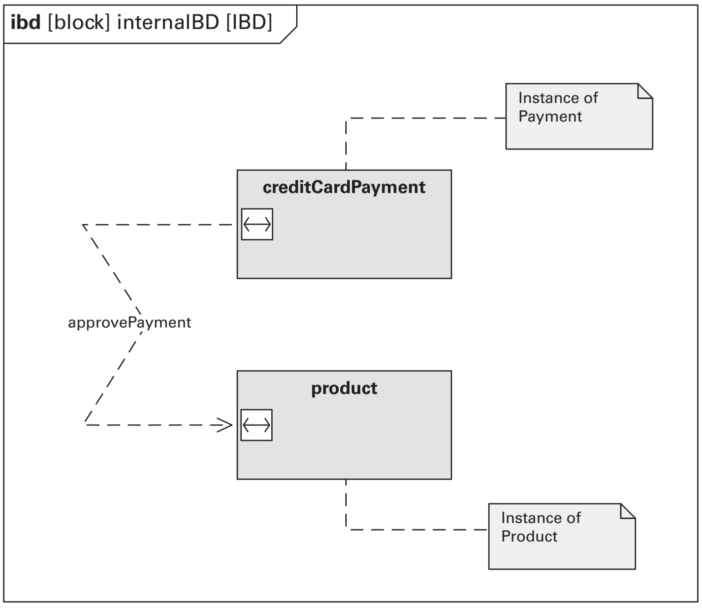


Figure B.4
Internal block diagram

B.5 Documenting an Allocation View

Systems engineers use allocation relations to associate many different types of information. SysML has several ways to show various types of allocation. The most common allocation view allocates the software to hardware. Table B.4 gives a mapping of the allocation view requirements onto a SysML internal block diagram.

Table B.4 Mapping of allocation view concepts to SysML

Allocation	**SysML**
Software element, environment element	A block is stereotyped to represent hardware; a description of the software allocated to that hardware is added.
Allocated-to relations	For this allocation the relation is *runs on*.
Software element has required properties. Environmental element has provided properties.	Both the software element and the hardware element have more complete descriptions in other diagrams that provide this information.
Properties depend on style.	The properties are defined elsewhere.
Topology varies by style.	Pairwise match of “from” and “to” elements

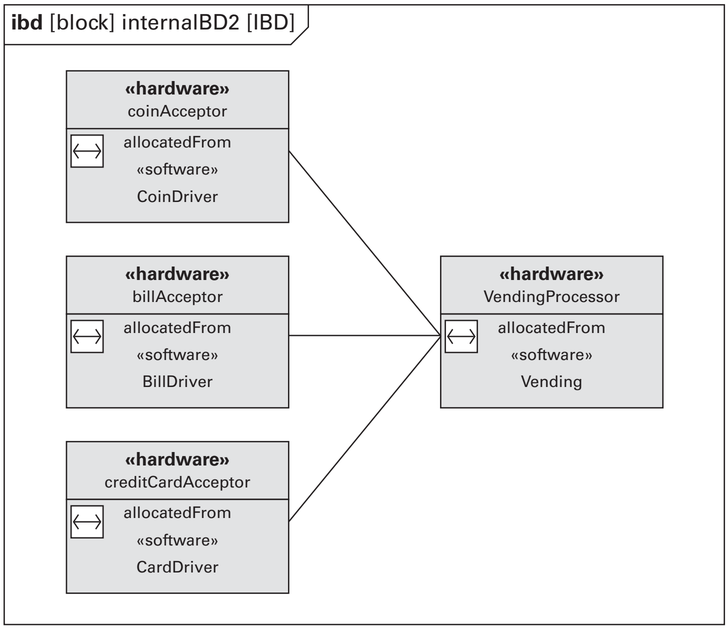


Figure B.5 Allocation of software to hardware in SysML

Table B.5 Table view of allocation

Type	Name	End	Relation	End	Type	Name
Activity	CoinDriver	From	Allocate	To	Block	coinAcceptor
Activity	BillDriver	From	Allocate	To	Block	billAcceptor

Figure B.5 shows the "allocatedFrom" partition in a block definition. It is also possible to have an "allocatedTo" partition, giving two-way traceability. Allocations can also be specified on a large number of other model elements. SysML adds a table style for allocation; see Table B.5 for an example.

B.6 Documenting Behavior

SysML has two ways to model behavior: the sequence and activity diagrams. The sequence diagram usually portrays a single path or scenario. It is unchanged from the UML definition. The activity diagram, in both SysML and UML, can represent a complete algorithm, but the SysML activity diagram has added a number of extensions that support describing a broader range of behaviors more accurately. These additions include the ability to represent inputs and outputs at various points along the paths of the diagram, and the ability to model an activity as a first-class entity that can appear in a class diagram and can participate in specification/generalization relations. Figure B.6 shows a small activity diagram.

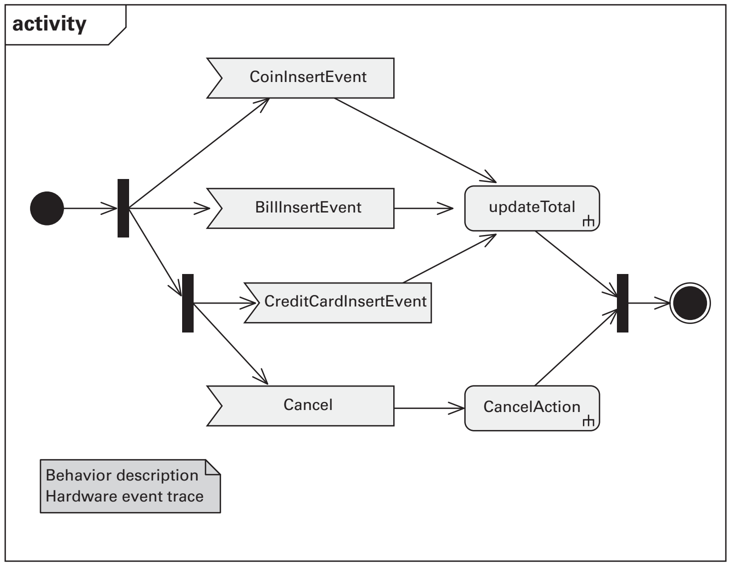


Figure B.6
A SysML activity diagram

B.7 Documenting Interfaces

SysML provides an interface model element in the block diagram type. The interface can also have associated constraints. As with all of the SysML diagram types, constraints may be added to any of the elements in a diagram, usually to specify the semantics of the element to which the constraint is attached. In Figure B.7, a constraint is used to capture the semantics of one dependency of the View interface.

B.8 Summary

A number of commercial and open-source tools support SysML. The Topcased project (topcased.org) provides editors for both the graphical and XML-based syntaxes of SysML. Commercial tools such as Rhapsody, MagicDraw, and Enterprise Architect support SysML.

At this writing, SysML version 1.2 is the latest release. As the use of SysML expands, expect that many change requests will be submitted and the language will evolve to more fully meet the needs of the systems-engineering community. Changes to UML may also be reflected in SysML, because they share a large portion of their metamodels.

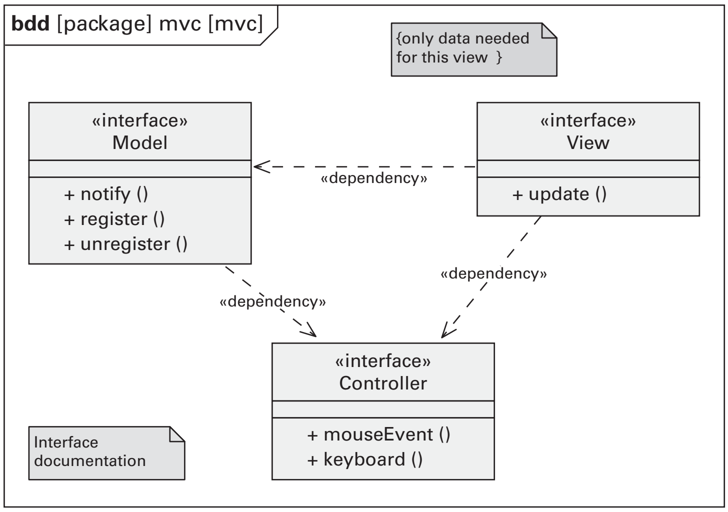


Figure B.7
Interface documentation in SysML

AADL—The SAE Architecture Analysis and Design Language

With Peter Feiler

C.1 Introduction

The Architecture Analysis and Design Language (AADL) (SAE AADL 2010) was developed as an SAE International industry standard with participation from European and U.S. avionics, aerospace, automotive, and medical device industry. SAE International is the largest standards provider for the avionics and automotive industry. AADL was first approved by more than 20 member organizations and published in November 2004 (SAE 2004/2009). In January 2009 a revision was published as SAE document AS5506A, based on feedback from industrial experience with AADL.

The AADL standard defines a textual and graphical language to represent the runtime architecture of software systems as a component-based model in terms of tasks and their interactions, the hardware platform the system executes on, possibly in a distributed fashion, and the physical environment it interfaces with, such as a plane, car, medical device, robot, satellite, or collections of such systems. This core language includes properties concerning timing, resource consumption in terms of processors, memory, network, deployment alternatives of software on different hardware platforms, and traceability to the application source code. AADL is extensible through user-defined properties and sublanguage annexes. The standard includes a set of annex documents published as SAE AS5506/1 (SAE 2006) that defines the AADL Meta-Model and XMI model interchange format for AADL, as well as the Error Modeling Annex as a standardized extension to support fault modeling and reliability and dependability analysis. Other extensions, such as for security, behavior, and architectures such as ARINC653 exist as draft standards and working documents.

A UML profile of AADL is being standardized jointly with an Object Management Group (OMG) initiative (OMG 2009).

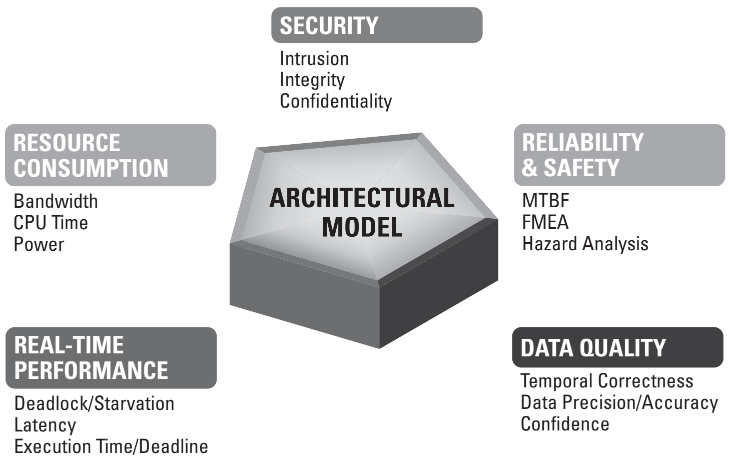
AADL provides several categories of components:

- A generic *abstract* component used for conceptual modeling and for specifying architecture templates or patterns
- Software components such as the following:
  - *Thread* to represent schedulable concurrent tasks
  - *Thread group* to support grouping of threads into groups with a common interface
  - *Process* to represent protected address spaces
  - *Data* to model application data types and static data components
  - *Subprogram* and *subprogram groups* to represent application functions and libraries of functions
- Hardware components such as the following:
  - *Processor* to execute threads
  - *Virtual processor* to represent virtual machines and hierarchical schedulers
  - *Memory* to represent storage hardware
  - *Bus* to represent buses and networks used to support communication between hardware components
  - *Virtual bus* to represent protocols and virtual channels
  - *Device* to represent components of the physical system such as an engine or a camera
- *System* to support hierarchical grouping of both software and hardware components

The AADL standard associates specific semantics to each of the component categories; for example, it defines the execution semantics of threads in terms of a hybrid automaton. AADL imposes a containment relationship on components of different categories. For example, threads and thread groups must be contained in a process. Processes and hardware components and system components can be contained in system components. Interaction relations are expressed through connections, and associations are expressed through reference properties.

The property and annex annotations of the AADL model support the generation of analytical models for different quality attributes from the same architecture model, as shown in Figure C.1 (Lewis and Feiler 2008).

Figure C.1
Multiple dimensions of architecture analysis in AADL



C.2 Documenting a Module Style

Although AADL does not use the term "module," AADL can represent units of implementation and relations among them. An AADL model is organized into packages, each of which defines a namespace. Packages can be placed into a nested naming hierarchy similar to Java packages. A package contains component specifications and may specify use of components from other packages. A package has a public part containing component specifications accessible to other packages and a private part containing component specifications local to the package. The package and its component specifications are maintained in an XMI representation based on the standardized AADL Meta-Model and have both a textual and graphical presentation. The graphical presentation may show subsets of the underlying model according to a specific view point.

Component types can be defined in terms of other component types through an *extends* relation (expressed by the extends keyword in textual AADL). This relation corresponds to the *generalization* concept in UML. This permits an incomplete component type that acts as a template to be refined by completing the specification of features and properties, and to be extended with additional features. These component types effectively represent a family of interfaces for a component.

Multiple component implementations can be associated with a component type through a *realization* relation expressed by naming the type as part of the implementation specification. They represent variants of a component. Implementations themselves can be refinements and extensions of other implementations. These incomplete component types and component implementation can be explicitly parameterized. This allows us to model architecture patterns, reference architectures, and families of system architectures (Feiler et al. 2004, Feiler 2007, Feiler et al. 2009).

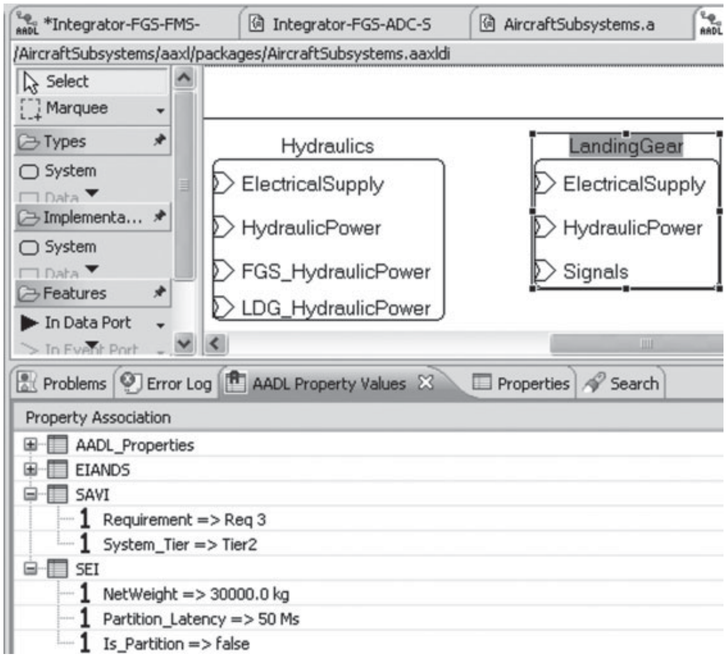
Figure C.2 shows the specification of a landing gear with features indicating that it requires access to an electrical power source, a hydraulic power source, and a signal flow. The component as well as its features can have properties. In our example, the landing gear has a weight property, a property providing traceability to a requirement, and an indication of the intended tier in a multi-tier architecture. The properties of the landing gear features indicate their electrical and hydraulic power requirements.

A graphical view of this component specification is shown in Figure C.3. It shows the landing gear with its features on the right. At the bottom is a property viewer that shows the properties associated with the landing gear specification. You can create a new component specification by selecting the appropriate component category from the palette on the left and by adding features from the palette into the component type.

Users can define data types using the AADL data component type. Such data type specifications can be placed in a separate package, which we call DataDictionary in our example in Figure C.4. This specification may characterize the data type in source code with properties relevant at the architecture level, such as the size of the data type, its source file, its base type representation, and constraints on the data value and its measurement unit. Data component types can have provided subprogram features to reflect methods on classes. The internal details of such data types may have been declared in a programming language or expressed in a data modeling notation such as UML class diagrams, or they can be expressed in AADL.

Figure C.2
Specification sheet of a landing gear

```
system LandingGear
  features
    ElectricalSupply: requires bus access ElectricalPower
        { SEI::PowerBudget => access 6000.0 w;};
    HydraulicPower: requires bus access HydraulicPressure {
      SAVI:: PressureBudget => access 300.0 psi;
      };
    Signals: requires bus access SignalFlow;
  properties
    SAVI::requirement => "Req 3";
    SEI::NetWeight => 30000.0 kg;
  SAVI::SystemTier => tier2;
end Landing Gear
```



Property Association
- AADL_Properties
- EIANDS
- SAVI
  - 1 Requirement => Req 3
  - 1 System_Tier => Tier2
- SEI
  - 1 NetWeight => 30000.0 kg
  - 1 Partition_Latency => 50 Ms
  - 1 Is_Partition => false

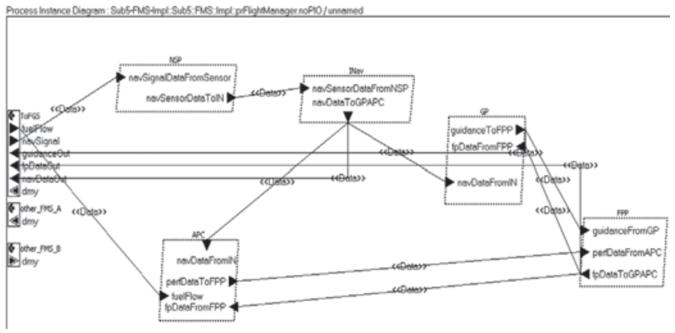
Figure C.3
Graphical view of component specification

```
package DataDictionary
    public
  data NavSignalData
  properties
    Source_Data_Size => 2 Bytes;
    Source_Text => ("DataDictionary.java");
    Data_Model::Base_Type => data BaseTypes::uint16;
    Data_Model::Real_Range => 0.0 .. 255.8;
    Data_Model::Measurement_Unit => "km";
  end NavSignalData;
```

Figure C.4
User-defined data types

A component implementation acts as a blueprint of the realization of a component. Figure C.5 illustrates such a blueprint for the implementation of a flight manager process. It consists of several threads as subcomponents of the process. Connections indicate how these threads communicate with each other and with components outside the process through the features of the process interface, which are shown on the left. In this case the port group graphic is expanded to show the elements of the port group, such that individual ports of the port group can be connected.

Figure C.5
Component blueprint of a flight manager



C.3 Documenting a Component-and-Connector View

Each AADL component category has well-defined semantics, many of which correspond to components in a component-and-connector (C&C) view. For example, AADL *threads* model concurrent tasks or active objects that represent sequential execution of source code. A thread is bound to a virtual processor or processor for execution. AADL threads can be dispatched *periodically* or triggered by events or the arrival of messages. In the latter case, a thread may execute *aperiodically*, that is, in response to the arrival of an event or message. If the thread is already active, newly arriving events or messages are queued. A thread may execute *sporadically*; that is, it will respond to events and messages, but its execution will be limited to a maximum rate. A thread may have a dispatch protocol called *timed*; that is, it will respond to events or messages like an aperiodic thread, but it will time out after a specified period if no event or message arrives. A thread may be declared with a *hybrid* dispatch protocol; that is, it executes periodically and it responds to events and message arrivals. A thread may be dispatched as a *background* thread; that is, it is dispatched once and executes until completion. The semantics of these dispatch protocols and the scheduling states of threads, such as suspended, ready, and running, are defined precisely in the standard using hybrid automata.

An example of a component specification for a process is shown in Figure C.6. An AADL process represents a space partition; that is, it provides runtime address space protection from other processes. It illustrates that at every level of the component hierarchy, we specify the complete interface of a component and its subcomponents to outside components. It also illustrates the use of port groups to indicate a collection of ports through which this process interacts with other software

```
process prFlightManager
  features
    toFGS: port group Integrator::FGS::FMS::ICD::FMS_To_FGS;
    other_FMS_A: port group Integrator::FGS::FMS::ICD::FMS_CrossPlg;
    other_FMS_B: port group Integrator::FGS::FMS::ICD::FMS_CrossSkt;
end prFlightManager
```

Figure C.6
Software process with port groups

components. The interaction with other components will be specified through a single port group connection instead of separate connections for each port.

The details of interaction in terms of ports are specified separately in a port group type declaration that can be placed in a separate AADL package, as shown in Figure C.7. In our example, one port group consists of two incoming ports and four outgoing ports.

Port-based communication may be in the form of messages (AADL *event data port*), in the form of events (AADL *event port*), and in the form of state data (AADL *data port*). Event data ports and event ports have queues associated with them. In addition, arrival of messages or events can trigger the dispatch of a thread according to its dispatch protocol. Data ports and event data ports are typed with user-defined data types, and only ports with compatible data types can be connected.

In AADL the connection concept is used to connect component ports, subprogram features, or access features. Connections can have properties, such as properties that indicate the desired protocol or quality of service provided by a protocol, such as guaranteed delivery. The connection is then bound to a virtual bus or bus that acts as the logical connector in terms of protocols, or a physical connector to perform the communication

```
Package Integrator::FGS::FMS::ICD
    public
  -- with DataDictionary
  port group FMS_to_FGS
    features
      fuelFlow: in data port DataDictionary::FuelFlowData;
      navSignal: in data port DataDictionary:: NavSignalData;
      guidanceOut: out data port DataDictionary:: GuidanceData;
      fpDataOut: out data port DataDictionary:: FPData;
      navDataOut: out data port DataDictionary:: NavData;
      dmy: out event data port;
  end FMS_to_FGS;
```

Figure C.7
Port group specifications

between different hardware components of the sender and receiver.

AADL supports directional flow through ports and connections. The threads may perform periodic sampled processing of signal streams, such as control systems, including communication timing semantics that ensure deterministic sampling. Threads may also perform data-driven message processing, processing of discrete events, and periodic processing of alarms. In addition to port-based communication, AADL supports modeling of access to shared data components with concurrency control—for example, blackboard architectures—and shared access to bus components for communication between hardware components. Finally, AADL supports interaction between threads and with devices through subprogram calls to model service calls.

AADL distinguishes between a set of component specifications and blueprints and an instance of a system model. An instance model is the result of instantiating a top-level system implementation recursively. Typically, such a system consists of the application software, the computing platform, and the physical environment. The AADL standard has defined a separate XMI representation of an instance model that analysis tools can operate on directly, or from which analytical models and runtime executives can be generated.

The instance model represents the complete component containment hierarchy, as illustrated in Figure C.8. Connection instances are between the components that are the leaves of the component hierarchy, for example, between thread instances or between processor instances and bus instances. The AADL standard does not require the full component containment hierarchy to be reflected in the instance model; instance models may be flattened to include only component instances with connection instances.

AADL supports the instantiation of incomplete system models. This allows such models to be analyzed early in the development life cycle and the analyses revisited as the model is refined. For example, only one process has been expanded to the thread level. For such a model we can still perform resource budget analysis by rolling up the data from threads and comparing it against the resource budgets of the processes. Those budgets are compared against the capacity available through the hardware.

AADL supports the analysis of critical flows throughout a system by providing the capability to specify end-to-end flows and annotating them with relevant flow properties, such as latency, precision, and confidentiality. An end-to-end flow is specified

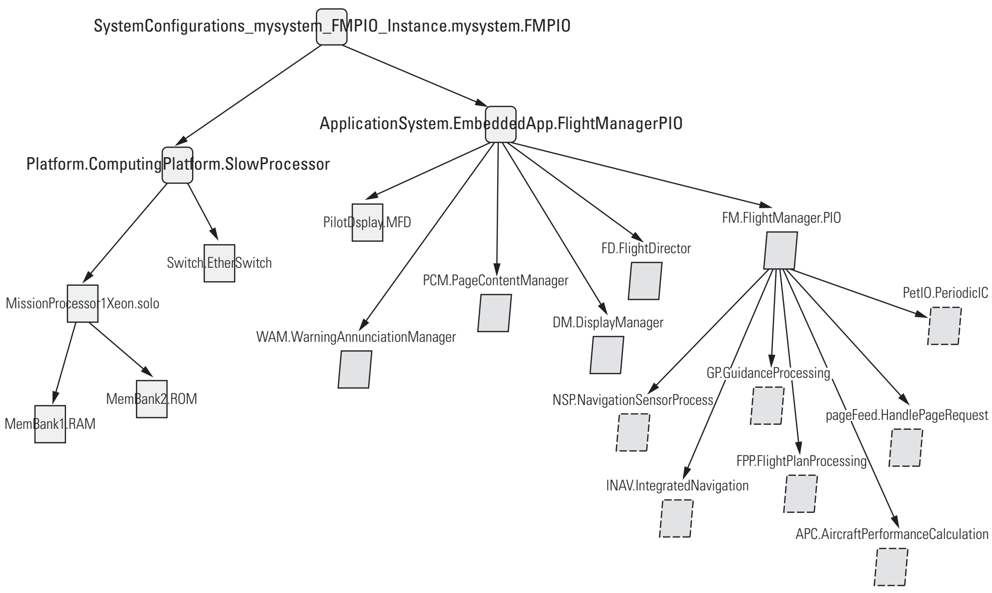

SystemConfigurations_mysystem_FMPIO_Instance.mysystem.FMPIO
Platform.ComputingPlatform.SlowProcessor
ApplicationSystem.EmbeddedApp.FlightManagerPIO
Switch.EtherSwitch
MissionProcessor1Xeon.solo
MemBank2.ROM
MemBank1.RAM
PilotDsplay.MFD
PCM.PageContentManager
WAM.WarningAnnunciationManager
FD.FlightDirector
DM.DisplayManager
FM.FlightManager.PIO
PetIO.PeriodicIC
GP.GuidanceProcessing
NSP.NavigationSensorProcess
pageFeed.HandlePageRequest
FPP.FlightPlanProcessing
INAV.IntegratedNavigation
APC.AircraftPerformanceCalculation

Figure C.8
Component containment hierarchy of system instance model

in terms of a sequence of component flow specifications and connections. A component flow specification specifies a flow from a component input (port) to one of its outputs (ports) without having to expose the component implementation. This allows end-to-end flow analysis, such as latency analysis, to be performed on systems of systems based on specified flow properties, while implementations of individual systems can be separately validated to ensure they meet the specified flow property.

C.4 Documenting a Deployment View

A complete AADL model of an embedded system includes software components, computer hardware components, and components of the physical system. The application software has to be deployed on the computer hardware in order for us to be able to perform analysis of operational quality attributes, such as meeting timing, performance, reliability, safety-criticality, and security requirements.

Figure C.9 shows a graphical representation of the deployment view, as it is often found in architecture documents. It shows the computer hardware components and the software components placed inside them to indicate that they are bound to the respective hardware component. This deployment information

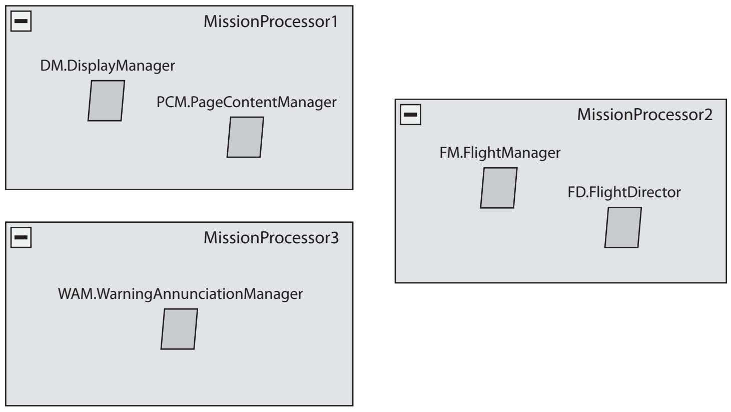


Figure C.9
Graphical representation of a deployment view

is recorded through properties for processor binding, memory binding, and connection binding. This deployment information can be declared as a collection of property values at the top of the model and refer to both the processes and threads to be bound to processors, memory, and buses.

C.5 Documenting Behavior

AADL supports modeling of a variety of system behaviors. The AADL mode and mode transition concept allows users to specify operational modes, different property values for different modes, and different runtime configurations of components and connection for different modes. For example, it can define different sets of threads and port connections during the taxiing mode of an aircraft and a cruise mode.

AADL modes can also be used to define different fault-tolerant configurations. This is illustrated by the architecture redundancy pattern shown in Figure C.10. It shows a replicated component with an observer to determine its health. The replicated component can be a software component or a hardware component. In a hot standby pattern, both the primary and the backup components are active in primary and backup mode; in a passive backup pattern, only one of the components is active at a time. Event-triggered mode transitions the dynamic aspects of switching between these configurations. These architecture patterns can be associated with the architecture as aspects without cluttering the primary functional view.

The AADL property set mechanism allows users to introduce new properties in support of certain analyses. For example, the security behavior of security frameworks, such as the Bell

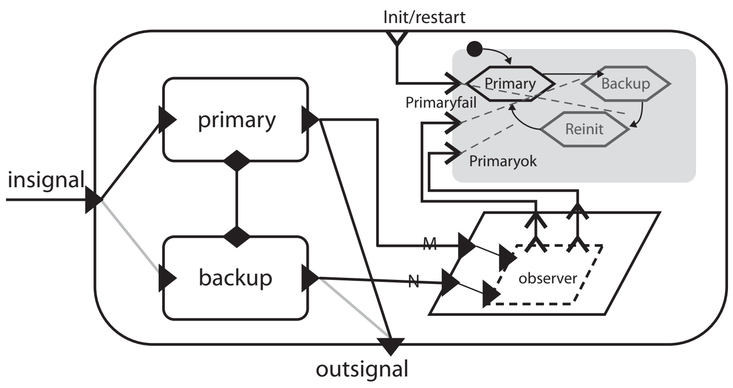


Figure C.10
Dual redundancy pattern

LaPadula and Chinese Wall frameworks, can be expressed as properties on existing AADL model concepts (Hansson and Feiler 2008). Figure C.11 illustrates the definition of security classifications as an enumeration type that is then used to define security properties with values of that type.

AADL also supports the use of a sublanguage in AADL model annotations. Figure C.12 shows the specification of an error state machine using the AADL Error Model Annex sublanguage (Rugina et al. 2008). This error machine specifies fault-free states and error state, intrinsic faults and error propagations with probability of occurrence, and conditions under which error states can change.

An error state machine is associated with a component type or component implementation. As a result, this error state machine is attached to each instance of this component. The error state machines of different components interact by propagating errors based on the logical and physical connectivity as well as the deployment of software to hardware.

AADL also has a draft Behavior Annex standard that was due to be published at the time of this writing. The focus of this

```
property set Security_types is
  -- The levels of security that are applicable to the system.
  -- We require the use of the enumeration type because it
  -- forces an order on the levels, but with the limitation that
  -- the order is a total linear order.
  --
  -- Here we use the standard military/governmental classifications.
Classifications:
  type enumeration (unclassified, confidential, secret, top_secret);
```

Figure C.11
User-defined properties

```
error model basic
features
    Error_Free: initial error state;
    Failed: error state;
    Crashed: error state;
    Fail: error event {occurrence => poisson 10e-3};
    Repair: error event {occurrence => poisson 0.0001};
    KO: in out error propagation {occurrence => fixed p};
    OK: in out error propagation {Occurrence => fixed 0.2-p};
end basic;

error model implementation basic.nominal
transitions
    Error_Free -[Fail]-> Failed;
    Failed -[Repair]-> Error_Free;
    Error_Free -[in KO]-> Failed;
    Failed -[out KO]-> Failed;
    Error_Free-[in KO]-> Failed;
    Failed-[out OK]->Crashed;
end basic.nominal;
```

Figure C.12
An example of the Error Model Annex sublanguage

annex is to support the specification of component interaction behavior and discrete state behavior within components.

C.6 Documenting Interfaces

A component type declares the interface of a component to other components, provides a specification of services, and presents its resource requirements on the hardware platform. The AADL feature concept is used to represent both provided and required features through which the component interacts with other components. AADL supports three types of interactions between components: (1) port-based flow of data, events, and messages from one component to another; (2) communication through shared access to a common resource, such as a shared data component; and (3) calls on subprograms to request services with returning results. AADL also supports the concept of a flow specification to represent the flow through a component without requiring access to its implementation. Flow specifications can have properties such as the expected latency of a flow through the component. They support end-to-end flow analysis of large-scale systems.

C.7 Summary

AADL supports modeling of the static structure and interaction topology, as well as the dynamic nature of system architectures.

The dependencies and the hierarchy reflected in the AADL model are a good basis for analysis of quality attributes that focus on the design of an architecture, such as modifiability. AADL models include the task and communication architecture of application software, the runtime architecture and hardware platform, and the deployment of the former on the latter to support the analysis of operational quality attributes, such as availability.

The AADL standard includes a standard interchange format for models in terms of XMI. This interchange format facilitates integration with existing tools and interchange of AADL models between projects and organizations. There is an open-source tool set for AADL (called OSATE) (SAE AADL 2010) based on Eclipse, as well as commercial tool support. A number of architecture analysis tools as well as automatic generators of runtime executives have been integrated with these AADL tool sets.

This page intentionally left blank

Acronyms

AADL	Architecture Analysis and Design Language
ACSPP	Architecture-centered software project planning
AD	Architecture documentation
ADD	Attribute-Driven Design
ADL	Architecture description language
ADR	Active design review
AOP	Aspect-oriented programming
AOPA	Aircraft Owners and Pilots Association
AOSD	Aspect-oriented software development
API	Application programming interface
ArchE	Architecture Expert
ARID	Active Reviews for Intermediate Designs
ASR	Architecturally significant requirement
ATAM	Architecture Tradeoff Analysis Method
ATIA	U.S. Army Training Information Architecture System
ATM	Asynchronous transfer mode
AV	All-view
BDUF	Big design up front
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Notation
C&C	Component and connector
CCM	CORBA component model
CLR	Common Language Runtime
CM	Complexity

CONOPS	Concepts of Operations
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-shelf
DBMS	Database management system
DCM	Data collection module
DoD	U.S. Department of Defense
DoDAF	Department of Defense Architecture Framework
DSM	Dependency structure matrix
ECS	EOSDIS Core System
ERD	Entity-relationship diagram
FEAF	Federal Enterprise Architecture Framework
FOS	Flight Operations Segment
FTX	Fault-tolerant UNIX
HLA	High-Level Architecture
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDL	Interface Definition Language (CORBA)
IEEE	Institute of Electrical & Electronics Engineers
INCOSE	International Council on Systems Engineering
IP	Internet Protocol
ISO	International Organization for Standardization
Java EE	Java Platform, Enterprise Edition
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JVM	Java Virtual Machine
MVC	Model-view-controller
OMG	Object Management Group
OSATE	Open-source AADL tool environment
OV	Operational view
OWL	Ontology Web language
RAID	Redundant array of independent disks
RMI	Remote Method Invocation
RUP	Rational Unified Process
SaaS	Software as a service
SAE	Society of Automotive Engineers
SARA	Software architecture review and assessment

SCM	Software configuration management
SDL	Specification and Description Language
SDPS	Science Data Processing Segment
SHARK	Sharing and Reusing Architectural Knowledge
SLA	Service-level agreement
SOA	Service-oriented architecture
SV	Systems and services view
SysML	Systems Modeling Language
TCP	Transmission Control Protocol
TDDT	Training and Doctrine Development Tool
TLCD	Top-level context diagram
TOGAF	The Open Group Architecture Framework
TV	Technical standards view
UM	Uncertainty
UML	Unified Modeling Language
UTMC	Unit Training Management Configuration
WBS	Work breakdown structure
WSDL	Web Services Definition Language
XML	Extensible Markup Language

Glossary

Actors the other elements, users, or systems with which an element interacts.

Allocation style a kind of style that describes the mapping of software units to elements of an environment in which the software is developed or executes.

Architectural (architecture) pattern "an architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them" (Buschmann et al. 1996, page 12).

Architecture cartoon the graphical portion of a view's primary presentation, without supporting documentation.

Architecture description language (ADL) a language for representing a software and/or system architecture. ADLs are usually graphical languages that provide semantics that enable analysis and reasoning about architectures, often using associated tools.

Architecture framework "conventions and common practices for architecture description established within a specific domain or stakeholder community" (ISO/IEC 42010:2007). TOGAF and DoDAF are examples of architecture frameworks.

Architecture perspective "a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views" (Rozanski and Woods 2005).

Architecture stakeholder someone who has a vested interest in the architecture.

Architecture style specialization of element and relation types, together with a set of constraints on how they can be used.

Bridging element an element that is common to two views and is used to provide the continuity of understanding from one view to the other. A bridging element appears in both views and has supporting documentation, usually a mapping between views, that makes the correspondence clear, perhaps by showing the combined picture.

Combined view a view that contains elements and relations that come from two or more other views.

Communicating-processes style any C&C style whose components can execute as independent processes.

Component-and-connector (C&C) style a kind of style that introduces a specific set of component and connector types and specifies rules about how elements of those types can be combined. Additionally, given that C&C views capture runtime aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style.

Components the principal computational elements and data stores that execute in a system.

Connector a runtime pathway of interaction between two or more components.

Context diagram a representation that defines the boundary between a system (or part of a system under consideration) and its environment, showing the entities in its environment with which it interacts.

Data integrity a property ensuring consistency and accuracy of the data shared across all applications in a system.

Decomposition refinement a refinement in which a single element is elaborated to reveal its internal structure. Each member of that internal structure may be recursively refined.

Dependency structure matrix (DSM) a table that shows modules as the row and column headers; a cell is nonzero if and only if there is a dependency between the row's module and the column's module.

Descriptive completeness a property of architecture documentation; a document has descriptive completeness if it documents all elements and relations in the system that are in the documentation's scope.

Dynamic architecture an architecture in which architecture variation points are exercised at runtime.

Element an architecture building block native to a style. An element can be a module, a component or connector, or an element in the environment of the system whose architecture we are documenting. The description of an element tells what role it plays in an architecture, lists its important properties, and furnishes guidelines for effective documentation of the element in a view.

Entity in a data model, a particular instance of an entity set or entity type (for example, Earth is an entity of entity set Planet).

Filter a component in the pipe-and-filter style that transforms data read on its input ports to data written on its output ports. Filters typically execute concurrently and incrementally.

Framework a framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. See also *architecture framework.*

Hierarchical element any kind of element that can consist of like-kind elements. A module is a hierarchical element because modules consist of submodules, which are themselves modules. A task or a process is not a hierarchical element.

Hybrid style the combination of two or more existing styles. Hybrid styles, when applied to a particular system, produce views.

Implementation inheritance the definition of a new implementation based on one or more previously defined implementations. The new implementation is usually a modification of the ancestors' behavior.

Implementation refinement a refinement in which some or all of the elements and relations are replaced by other, more implementation-specific, elements and relations.

Interface a boundary across which two elements meet and interact or communicate with each other.

Interface document a specification of what an architect chooses to make publicly known about an element in order for other entities to interact or communicate with it.

Interface inheritance the definition of a new interface based on one or more previously defined interfaces. The new interface is usually a subset of the ancestors' interface(s).

Layer a grouping of modules that together offer a cohesive set of services to other layers. The layers are related to each other by the strictly ordered relation *allowed to use.*

Module an implementation unit of software that provides a coherent set of responsibilities.

Module style a kind of style that introduces a specific set of module types and specifies rules about how elements of those types can be combined.

Overlay a combination of the primary presentations of two or more views followed by supporting documentation for that combined primary presentation.

Pipe a connector in the pipe-and-filter style that conveys streams of data from the output port of one filter to the input port of another filter without changing values or the order of the data.

Port an interface of a component. A port defines a point of interaction of a component with its environment.

Property additional information about elements and relations. When an architect documents a view based on that style, the properties will be given values. Property values are often used to analyze an architecture for its ability to meet quality attribute requirements.

Question set questions that collectively address a narrowly focused purpose for an architecture document review. Besides the questions themselves, a question set contains information to allow a user to ensure the question set is appropriate and to use it effectively. This information includes the name, purpose, stakeholders and concerns, respondents, expected answers, criticality, and advice.

Rationale an explanation of the reasoning that lies behind an architecture decision.

Refinement the process of gradually disclosing information across a series of descriptions.

Relation a definition of how elements cooperate to accomplish the work of the system. The description of a relation names the relations among elements and provides rules on how elements can and cannot be related.

Resource a function, method, data stream, global variable, message end point, event trigger, or any addressable facility within an interface.

Responsibility a general statement about an architecture element and what it is expected to contribute to the architecture. This might include the actions that it performs, the knowledge it maintains, or the role it plays in achieving the system's overall quality attributes or functionality.

Role an interface of a connector. A role defines a point of interaction of a connector and indicates how components may use a connector in interactions.

Software architecture the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

Software product line a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way.

Specialization a style is a specialization of another style if it is consistent with that style—that is, doesn't violate it—and adds more constraints to its element types, relation types, and/or topological restrictions.

Stakeholder see *architecture stakeholder.*

Stereotype a type of modeling element in UML that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes of the metamodel. Stereotypes may extend the semantics but not the structure of preexisting types and classes. Certain stereotypes are predefined in UML; others may be user defined.

Style guide the description of an architecture style that specifies the vocabulary of design (sets of element and relationship types) and the rules (sets of topological and semantic constraints) for how that vocabulary can be used.

Substyle a specialization of another style if it is consistent with that style—that is, doesn't violate it—and adds more constraints to its element types, relation types, and/or topological restrictions.

Subsystem a part of a system that (1) carries out a functionally cohesive subset of the overall system's mission, (2) can be executed independently, and (3) can be developed and deployed incrementally.

System a collection of entities (elements, components, models, and so forth) that are organized for a common purpose.

System of interest ISO 42010 defines "system of interest" as encompassing "individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest."

Tier a mechanism for system partitioning. Usually applied to client-server-based systems, where the various parts (tiers) of the system (user interface, database, business application logic, and so forth) execute on different platforms.

Top-level context diagram a context diagram in which the scope is the entire system.

Topology a definition of constraints on how elements and relations can be associated in a particular style.

Unified Modeling Language (UML) a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software system.

Uses relation a form of dependency that exists between two modules. A uses B if the correctness of A depends on the presence of a correct implementation of B.

Variability the ability quickly to achieve change in preplanned ways.

Variability guide the place in an architecture document that explains what variation points have been designed into the architecture and gives advice about how to exercise them.

Variation mechanism a built-in software mechanism for making a change that, when exercised, results in a new instance of the architecture. The place where a variation mechanism occurs marks a variation point.

Variation point a place in the architecture where a specific kind of flexibility has been built in.

View a representation of a set of system elements and relations among them.

View packet the smallest bundle of view documentation you would show an individual stakeholder, such as a developer assigned to implement a small portion of the system or a customer interested in an overview.

Viewpoint ISO 42010 defines a viewpoint as a work product establishing the conventions for the construction, interpretation, and use of architecture views and associated architecture models.

Virtual machine sometimes called an *abstract machine*, a collection of modules that form an isolated cohesive set of services that can execute programs.

Wiki a collection of Web pages designed to enable anyone with access to contribute or modify content, using a simplified markup language.

References

Abelson, H., and G. Sussman. 1996. *Structure and Interpretation of Computer Programs.* 2nd ed. MIT Press.

Acme 2009. The Acme Project. http://www.cs.cmu.edu/~acme/.

Adventure Builder 2010. Java Adventure Builder Reference Application. https://adventurebuilder.dev.java.net/.

Agile Alliance. 2002a. "Manifesto for Agile Software Development." http://www.agilemanifesto.org.

———. 2002b. Web site. http://www.agilealliance.org.

———. 2002c. "Principles Behind the Agile Manifesto." http://www.agilemanifesto.org/principles.html.

Akerman, A., and J. Tyree. 2005. "Position on Ontology-based Architecture." *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005).* November 6–10, pp. 289–290.

Alexander, Christopher. 1979. *The Timeless Way of Building.* Oxford University Press.

Allen, R. J., and D. Garlan. 1997. "A Formal Basis for Architectural Connection." *ACM Transaction on Software Engineering and Methodology* 6(3): 213–249.

Ambler, Scott W. 2006. "The Object Relational Impedance Mismatch." http://www.agiledata.org/essays/impedanceMismatch.html.

Ambysoft. 2008. "Modeling and Documentation Practices on IT Projects Survey Results: July 2008." http://www.ambysoft.com/surveys/modelingDocumentation2008.html.

Araujo, I., and M. Weiss. 2002. "Linking Patterns and Non-Functional Requirements." *Proceedings of the Pattern Languages* *of Programs Conference (PLoP 2002).* Monticello, Illinois, September 8–12. Available at http://www.hillside.net/plop/plop2002/proceedings.html.

Atlantic. 1956. "Rhythm in My Blood." *The Atlantic* Magazine, February 1956.

Avritzer, A., Y. Cai, and D. Paulish. 2008. "Coordination Implications of Software Architecture in a Global Software Development Project." *Proceedings of WICSA 2008,* pp. 107–116.

Bach, Maurice. 1986. *The Design of the UNIX Operating System.* Prentice Hall.

Bachmann, Felix, and Paulo Merson. 2005. *Experience Using the Web-Based Tool Wiki for Architecture Documentation.* Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2005-TN-041.

Barker, Richard. 1990. *CASE Method: Entity Relationship Modelling.* Addison-Wesley.

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice.* 2nd ed. Addison-Wesley.

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley.

Beck, Kent, and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change.* 2nd ed. Addison-Wesley.

Bianco, Phil, Rick Kotermanski, and Paulo Merson. 2007. *Evaluating a Service-Oriented Architecture.* Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-2007-TR- 015.

Bianco, Phil, Grace Lewis, and Paulo Merson. 2008. *Service Level Agreements in Service-Oriented Architecture Environment.* Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2008-TN-021.

Bloch, Joshua. 2006. "How to Design a Good API and Why It Matters." http://www.infoq.com/presentations/effective-api-design.

Boehm, B., D. Port, A. Egyed, and M. Abi-Antoun. 1999. "The MBASE Life Cycle Architecture Milestone Package: No Architecture Is An Island." 1st Working International Conference on Software Architecture.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 2005. *The Unified Modeling Language User Guide.* 2nd ed. Addison-Wesley.

Bosch, J. 2000. *Design and Use of Software Architecture: Adopting and Evolving a Product Line Approach.* Addison-Wesley.

Brooks, Jr., Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering.* Anniv. ed. Addison-Wesley.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley.

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.* Wiley.

———. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* Wiley.

Cataldo, M., P. Wagstrom, J. D. Herbsleb, and K. Carley. Forthcoming. "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools." In *Proceedings, Computer-Supported Cooperative Work.*

Chen, Peter. 1976. "The Entity-Relationship Model: Toward a Unified View of Data." *ACM Transactions on Database Systems* 1(1): 9–36.

Clements, Paul, and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley.

Clements P., R. Kazman, and M. Klein. 2002. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley.

Clements P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. 2003. *Documenting Software Architectures: Views and Beyond.* 1st ed. Addison-Wesley.

Cockburn, Alistair. 2000. *Writing Effective Use Cases.* Addison-Wesley.

———. 2001. Crystal methodologies Web page. http://alistair.cockburn.us/crystal.

———. 2002. *Agile Software Development.* Addison-Wesley.

———. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley.

Conway, M. E. 1968. "How Do Committees Invent?" *Datamation* 14(4): 28–31.

Date, C. J. 1999. *An Introduction to Database Systems.* 7th ed. Addison-Wesley.

———. 2003. *An Introduction to Database Systems.* 8th ed. Addison-Wesley.

de Boer, R. C., and R. Farenhorst. 2008. "In Search Of 'Architectural Knowledge.' " *3rd Workshop on SHAring and Reusing architectural Knowledge (SHARK).* Leipzig, Germany. May 13, pp. 71–78.

DeRemer, F., and H. J. Kron. 1976. "Programming-in-the-Large versus Programming-in-the-Small." *IEEE Transactions on Software Engineering*, SAE 2(2): 80–86.

Dijkstra, E. W. 1968. "The Structure of the 'T.H.E.' Multiprogramming System." *Communications of the ACM* 18(8): 453–457.

———. 1972. "Notes on Structured Programming: On Program Families." In Ole-Johan Dahl, ed., *Structured Programming*, pp. 39–41. Academic Press.

Dobrica, L., and E. Niemela. 2002. "A Survey on Software Architecture Analysis Methods," *IEEE Transactions on Software Engineering* 28(7): 638–653.

DoDAF. 2007. U.S. Department of Defense Architecture Framework, version 1.5. http://www.defenselink.mil/cio-nii/docs/DoDAF_Volume_I.pdf.

Eeles, P., and P. Cripps. 2009. *The Process of Software Architecting*. Addison-Wesley.

Feiler, Peter H. 2007. *Modeling of System Families*. Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2007-TN-047.

Feiler, Peter H., David Gluch, John Hudak, and Bruce Lewis. 2004. *Embedded Systems Architecture Analysis Using SAE AADL*. Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2004-TN-005.

Feiler, Peter H., David P. Gluch, and John J. Hudak. 2006. *The Architecture Analysis & Design Language (AADL): An Introduction*. Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2006-TN-011.

Feiler P., D. Gluch, K. Weiss, and K. Woodham. 2009. "Model-Based Software Quality Assurance with the Architecture Analysis and Design Language." *Proceedings of AIAA Infotech @Aerospace 2009*, Seattle, Washington, April 6–9.

Flurry, G., and W. Vicknair. 2001. "The IBM Application Framework for e-business." *IBM Systems Journal* 40(1): 8–24.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

———. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley.

Freeman, Eric, Susanne Hupfer, and Ken Arnold. 1999. *JavaSpaces Principles, Patterns, and Practice*. Prentice Hall.

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

Garlan, D., and M. Shaw. 1993. "An Introduction to Software Architecture." In V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering,* vol. 2. World Scientific.

Garlan, D., R. Allen, and J. Ockerbloom. 1995. "Architectural Mismatch or Why It's So Hard to Build Systems out of Existing Parts." *Proceedings of 17th Int. Conf. on Software Engineering,* Seattle, Wash., April 24–28.

Garlan, D., and D. Perry. 1995. "Introduction to the Special Issue on Software Architecture." *IEEE Transactions on Software Engineering* 21(4): 269–274.

Garlan, David, and Bradley Schmerl. 2006. "Architecture-driven Modeling and Analysis." *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06),* Melbourne, Australia.

Garland, Jeff, and Richard Anthony. 2003. *Large-Scale Software Architecture: A Practical Guide Using UML.* Wiley.

Gelernter, D. 1985. "Generative Communication in Linda." *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985), pp. 80–112.

Goethert, W., and J. Siviy. 2004. *Applications of the Indicator Template for Measurement and Analysis.* Carnegie Mellon University, Software Engineering Institute Technical Note CMU/SEI-2004-TN-024.

Gorton, I. 2006. *Essential Software Architecture.* Springer.

Hämäläinen, N., and J. Markkula. 2007. "Quality Evaluation Question Framework for Assessing the Quality of Architecture Documentation." In the CD proceedings of E. Berki, J. Nummenmaa, M. Ross, and G. Staples, eds., *Software Quality Meets Work-Life Quality. International BCS Conference on Software Quality Management—SQM 2007.* Tampere, Finland, August 1–2.

Hansson, Jörgen, and Peter H. Feiler. 2008. "Enforcement of Quality Attributes for Net-centric Systems through Modeling and Validation with Architecture Description Languages." *Proceedings of 4th International Congress on Embedded Real-Time Systems,* January.

Harel, David, and Michael Politi. 1998. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill.

Harvey, Miles. 2000. *The Island of Lost Maps: A True Story of Cartographic Crime.* Random House.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice Hall.

Hoffman, D. M., and D. M. Weiss, eds. 2001. *Software Fundamentals: Collected Papers by David L. Parnas.* Addison-Wesley.

Hofmeister, C., R. Nord, and D. Soni. 2000. *Applied Software Architecture.* Addison-Wesley.

Hohpe, Gregor, and Bobby Wolff. 2003. *Enterprise Integration Patterns.* Addison-Wesley.

Hughart, Barry. 1984. *Bridge of Birds: A Novel of an Ancient China That Never Was.* Random House.

Humphrey, Watts S. 1989. *Managing the Software Process.* Addison-Wesley.

IBM. 2004. Rational Unified Process. http://www-306.ibm.com/software/awdtools/rup/.

IEEE 1471. 2000. *IEEE Product No.: SH94869-TBR: Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Standard No. 1471-2000. Available at http://shop.ieee.org/store/.

IEEE 1516.1. 2000. *IEEE Standard No.: 1516-1-2000: Standard for Modeling and Simulation (M&S)—High Level Architecture (HLA)—Federated Interface Specification.* IEEE Product No.: SS94883-TBR. Available at http://shop.ieee.org/store/.

ISO/IEC 10746-2. 1996. *Information Technology—Open Distributed Processing—Reference Model: Foundations.* Available at http://www.iso.org/iso/catalogue_detail.htm?csnumber=18836.

ISO 15704. 2000. *Industrial Automation Systems—Requirements for Enterprise-Reference Architectures and Methodologies.* Available at http://www.iso.org/iso/catalogue_detail.htm?csnumber=28777.

ISO/IEC 42010. 2007. *Systems and Software Engineering—Recommended Practice for Architectural Description of Software-Intensive Systems.* (Identical to ANSI/IEEE Std 1471-2000). http://www.iso-architecture.org/ieee-1471/.

ISO/IEC CD TR 24748. 2007. *Systems and Software Engineering—Life Cycle Management—Guide for Life Cycle Management.*

ISO/IEC 12207. 2008. *Systems and Software Engineering—Software Life Cycle Processes.* 2008.

ISO/IEC WD2 42010. 2008. *Systems and Software Engineering—Architectural Description.* Working draft dated 1 March 2008. http://www.iso-architecture.org/ieee-1471/docs/ISO-IEC-WD2v1-42010.pdf.

ISO/IEC CD1 42010. 2009. *Recommended Practice for Architectural Description of Software-Intensive Systems*—ANSI/IEEE Std 1471 :: ISO/IEC 42010. http://www.iso-architecture.org/ieee-1471/.

Iverson, K. E. 1987. "A Dictionary of APL." *APL Quote Quad* 18(1): 5.

Jackson, Michael. 1995. *Software Requirements and Specifications: A Lexicon of Practice Principles and Prejudices.* Addison-Wesley.

———. 2000. *Problem Frames: Analysing & Structuring Software Development Problems.* Addison-Wesley.

Jacobson, I. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley.

Jazayeri, Mehdi, Alexander Ran, and Frank van der Linden. 2000. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. 1997. "Aspect-Oriented Programming." *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Published as *Lecture Notes in Computer Science,* Number 1241. Springer Verlag, pp. 220–242.

Kircher, Michael, and Prashant Jain, 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.* Wiley.

Komiya, S. 1994. "A Model for the Recording and Reuse of Software Design Decisions and Decision Rationale." *Third International Conference on Software Reuse: Advances in Software Reusability.* Rio de Janeiro, Brazil, November 1–4, pp. 200–201.

Kruchten, Philippe. 1995. "The 4+1 View Model of Architecture." *IEEE Software* 12(6): 42–50.

———. 2004. "An Ontology of Architectural Design Decisions in Software Intensive Systems." *Proceedings of the 2nd Groningen Workshop on Software Variability,* Groningen, The Netherlands, December 2–3, pp. 54–61.

———. 2009. "Documentation of Software Architecture from a Knowledge Management Perspective—Design Representation." In *Software Architecture Knowledge Management,* ed. M. Ali Babar, T. Dingsøyr, P. Lago, and H. van Vliet, pp. 39–57. Springer Verlag.

Kruchten, P., P. Lago, and H. van Vliet. 2006. "Building Up and Reasoning about Architectural Knowledge." In *QoSA: Quality of Software Architecture.* Published as *Lecture Notes in Computer Science,* Number 4214, ed. C. Hofmeister, pp. 43–58. Springer Verlag.

Kylmäkoski, Roope. 2003. "Efficient Authoring of Software Documentation Using RaPiD7." *Proceedings of the 25th International Conference on Software Engineering,* Portland, Oregon, May 3–10.

Laddad, Ramnivas. 2008. *AspectJ in Action.* Manning.

Lewis, Bruce A., and Peter H. Feiler. 2008. "Multi-Dimensional Model Based Development for Performance Critical Computer Systems Using the AADL." *Proceedings of 4th International Congress on Embedded Real-Time Systems,* January.

Liskov, B. 1987. "Data Abstraction and Hierarchy." *OOPSLA'87: Conference on Object Oriented Programming Systems, Languages and Applications.* Orlando. Also available as *SigPlan Notices* 23(5): 17–34.

Louridas, P., and P. Loucopoulos. 2000. "A Generic Model for Reflective Design." *TOSEM* 9: 199–237.

Martin, James, and Clive Finkelstein. 1981. *Information Engineering.* Technical Report. Savant Institute.

Medvidovic, N., and R. N. Taylor. 1997. "A Framework for Classifying and Comparing Architecture Description Languages." *Proceedings of the 6th European Software Engineering Conference together with FSE4,* pp. 60–76.

Microsoft Developer Network. 2002. *Using .NET to Implement Sun Microsystems' Java Pet Store J2EE BluePrint Application.* http://msdn2.microsoft.com/en-us/library/ms954626.aspx.

Moyers, Bill. 1989. *A World of Ideas,* ed. Betty Sue Flowers. Doubleday.

Nii, H. P. 1986. "Blackboard Systems." *AI Magazine* 7(3): 38–53 and 7(4): 82–107.

Nygaard, K., and O.-J. Dahl. 1981. "The Development of the SIMULA Language." In *History of Programming Languages,* ed. R. Wexelblat, pp. 439–493. Academic Press.

Object Management Group. 2009. *Modeling and Analysis of Real-Time Embedded Systems (MARTE).* http://www.omgmarte.org.

Palmer, Stephen, and John Felsing. 2002. *A Practical Guide to Feature-Driven Development.* Prentice Hall.

Parnas, D. L. 1971. "Information Distribution Aspects of Design Methodology." *Proceedings of the 1971 IFIP Congress.* North Holland Publishing.

———. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15(12): 1053–1058.

———. 1974. "On a 'Buzzword': Hierarchical Structure." *Proceedings of the IFIP Congress '74,* pp. 336–339.

———. 1996. "Why Software Jewels Are Rare." *Computer* 29(2), February, pp. 57–60.

Parnas, D. L., and D. M. Weiss. 1985. "Active Design Reviews: Principles and Practices." *8th International Conference on Software Engineering,* pp. 215–222. Reprinted in *Software Fundamentals: Collected Papers by David L. Parnas,* ed. D. Hoffman and D. Weiss. Addison-Wesley.

Parnas, D. L., and P. C. Clements. 1986. "A Rational Design Process: How and Why to Fake It." *IEEE Transactions in Software Engineering* SE-12(2): 251–257.

Parnas, D., P. Clements, and D. Weiss. 2001. "The Modular Structure of Complex Systems." Reprinted in *Software Fundamentals: Collected Papers by David L. Parnas,* ed. D. Hoffman and D. Weiss. Addison-Wesley.

Parnas, David L., and H. Wuerges. 2001. "Response in Undesired Events in Software Systems." In *Software Fundamentals: Collected Papers by David L. Parnas,* ed. D. Hoffman and D. Weiss. Addison-Wesley.

Paulish, D. J. 2002. *Architecture-Centric Software Project Management: A Practical Guide.* Addison-Wesley.

Perry, D. E., and A. L. Wolf. 1992. "Foundations for the Study of Software Architecture." *Software Engineering Notes* 17(2): 40–52.

Ponniah, Paulraj. 2007. *Data Modeling Fundamentals.* Wiley.

Prieto-Diaz, R., and J. M. Neighbors. 1986. "Module Interconnection Languages." *The Journal of Systems and Software* 6(4): 307–334.

Rozanski, N., and E. Woods. 2005. *Software Systems Architecture.* Addison-Wesley.

Rugina, Ana-Elena, Karama Kanoun, Mohamed Kaaniche, and Peter Feiler. 2008. "Software Dependability Modeling using an Industry-Standard Architecture Description Language." *Proceedings of 4th International Congress on Embedded Real-Time Systems,* January 2008.

SAE. 2004/2009. SAE International, Avionics Systems Division AS-2C Subcommittee: Standard Document SAE AS-5506A. Nov. 2004, rev. Jan 2009. *Avionics Architecture Description Language Standard.*

SAE. 2006. SAE International, Avionics Systems Division AS-2C Subcommittee. Annex Document AS-5506/1. June 2006. *SAE Architecture Analysis & Design Language (AADL) Annex Volume 1: Graphical AADL Notation, AADL Meta-Model and Interchange Formats, Language Compliance and Application Program Interface.*

SAE. 2010. AADL Standard Web site. http://www.aadl.info.

SARA. 2002. *Final Report of the Software Architecture Review and Assessment (SARA) Group*, version 1.0. http://philippe.kruchten.com/architecture/SARAv1.pdf.

Scaffidi, Christopher, and Mary Shaw. 2007. "Developing Confidence in Software through Credentials and Low-Ceremony Evidence." *International Workshop on Living with Uncertainties at the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007),* Atlanta, Georgia, November 2007.

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* Wiley.

Schwaber, Ken. 2001. *Agile Software Development with Scrum.* Prentice Hall.

SEI. 2010."Defining Software Architecture." http://www.sei.cmu.edu/architecture/start/definitions.cfm.

Shaw, Mary. 1990. "Elements of a Design Language for Software Architecture." Position Paper for IEEE Design Automation Workshop. January 1990. Unpaginated.

———. 1991. "Heterogeneous Design Idioms for Software Architecture." *Proceedings of the 6th International Workshop on Software Specification and Design,* Como, Italy, October 25–26, 1991, pp. 158–165.

———. 1995. "Making Choices: A Comparison of Styles for Software Architecture." *IEEE Software, Special Issue on Software Architecture* 12(6): 27–41.

———. 1996a. "Truth vs. Knowledge: The Difference Between What a Component Does and What We Know it Does." *Proceedings of the 8th International Workshop on Software Specification and Design,* pp. 181–185.

———. 1996b. “Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First Class Status.” In *Studies of Software Design, Proceedings of a 1993 Workshop,* ed. D. A. Lamb; published as *Lecture Notes in Computer Science* No. 1978, pp. 17–32. Springer Verlag.

Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

Smith, C., and L. Williams. 2002. *Performance Solutions: A Practical Guide for Creating Responsive, Scalable Software.* Addison-Wesley.

Smith, John Miles, and Diane C. P. Smith. 1977. “Database Abstractions: Aggregation and Generalization.” *ACM Transactions on Database Systems* 2 (2).

Snyder, A. 1986. “Encapsulaton and Inheritance in Object-Oriented Programming Languages.” In *Proceedings of the Conferences on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’86),* ed. Norman K. Meyerowitz, pp. 38–45. Available as SIGPLAN Notices 21(11), November.

Soni, D., R. L. Nord, and C. Hofmeister. 1995. “Software Architecture in Industrial Applications.” *Proceedings of the 17th International Conferences on Software Engineering,* pp. 196–207.

Spivey, J. M. 1988. *The Z Notation: A Reference Manual.* 2nd ed. Available at http://spivey.oriel.ox.ac.uk/mike/zrm/.

Stafford, J. A., and A. L. Wolf. 2001. “Software Architecture.” In *Component-Based Software Engineering: Putting the Pieces Together,* ed. G. T. Heineman and W. T. Council. Addison-Wesley.

Steward, Donald. 1981. “Design Structure System: A Method for Managing the Design of Complex Systems.” *IEEE Transactions on Engineering Management* 28(33): 71–74.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

Taylor, R. N., N. Medvidovic, and E. M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice.* Wiley.

TOGAF. 2010. The Open Group Architecture Framework, version 9. http://www.opengroup.org/architecture/togaf9-doc/arch/.

Trachtenberg, Marvin, and Isabelle Hyman. 1986. *Architecture: From Prehistory to Post-Modernism/The Western Tradition.* Prentice Hall.

Urdangarin, R., P. Fernandes, A. Avritzer, and D. Paulish. 2008. "Experiences with Agile Practices in the Global Studio Project." *IEEE International Conference on Global Software Engineering (ICGSE)*, Bangalore, India, August 17–20, pp. 77–86.

Weeks, Edward, and Emily Flint, eds. 1957. "Emily Dickinson's Letters." In *Jubilee: One Hundred Years of the Atlantic.* Little, Brown and Company.

Wikipedia. 2010a. "Architectural style." http://en.wikipedia.org/wiki/Architectural_style.

———. 2010b. "Representational State Transfer." http://en.wikipedia.org/wiki/REST.

———. 2010c. "Wiki." http://en.wikipedia.org/wiki/Wiki.

Wright, Tim. 2003. "Flying Like the Birds." *AOPA Pilot.* June 2003: 81–89. Also available at http://roman-hartmann.de/html/flying_like_the_birds.html.

Yahoo!. 2010. Pipes Web site. http://pipes.yahoo.com/pipes/.

Zachman, J. A. 1987. "A Framework for Information Systems Architecture." *IBM Systems Journal* 26(3).

About the Authors

Paul Clements is a Senior Member of the Technical Staff at the Carnegie Mellon Software Engineering Institute (SEI), where he has worked since 1994 leading or coleading projects in software product-line engineering and software architecture documentation and analysis. Besides this one, Clements is the coauthor of two other practitioner-oriented books about software architecture: *Software Architecture in Practice* (Addison-Wesley, 1998; Second Edition 2003) and *Evaluating Software Architectures: Methods and Case Studies* (Addison-Wesley, 2001). He also cowrote *Software Product Lines: Practices and Patterns* (Addison-Wesley, 2001) and was coauthor and editor of *Constructing Superior Software* (Sams, 1999). In addition, Clements has authored dozens of papers in software engineering, reflecting his longstanding interest in the design and specification of challenging software systems. In 2005 and 2006 he spent a year as a visiting faculty member at the Indian Institute of Technology in Mumbai. He received a Ph.D. in computer sciences from the University of Texas at Austin in 1994. He is a founding member of the IFIP Working Group on Software Architecture (WG2.10).

Felix Bachmann is a Senior Member of the Technical Staff at the SEI, working in the Architecture Centric Engineering Initiative. He is coauthor of the Attribute-Driven Design Method, a contributor to and instructor for the ATAM Evaluator Training course, and a contributor to the book *Software Architecture in Practice, Second Edition.* Before joining the SEI, he was a software engineer at Robert Bosch GmbH in corporate research, where he worked with software development departments to address the issues of software engineering in small and large embedded systems.

Len Bass is a Senior Member of the Technical Staff at the SEI. He has coauthored two award-winning books in software architecture as well as several other books and numerous papers in a wide variety of areas of computer science and software engineering. He has been a keynote speaker or a distinguished lecturer on six continents. He is currently working on applying the concepts of ultra-large-scale systems to the smart grid. He has been involved in the development of numerous different production or research software systems, ranging from operating systems to database management systems to automotive systems. He is a member of the IFIP Working Group on Software Architecture (WG2.10).

David Garlan is a Professor of Computer Science and Director of Software Engineering Professional Programs in the School of Computer Science at Carnegie Mellon University (CMU). He received his Ph.D. from CMU in 1987 and worked as a software architect in industry between 1987 and 1990. His interests include software architecture, self-adaptive systems, formal methods, and cyber-physical systems. He is considered to be one of the founders of the field of software architecture and, in particular, formal representation and analysis of architectural designs. In 2005 he received a Stevens Award Citation for fundamental contributions to the development and understanding of software architecture as a discipline in software engineering.

James Ivers is a Senior Member of the Technical Staff at the SEI, where he works in the areas of software architecture and program analysis. He received a Master of Software Engineering from CMU in 1996 and has worked for and with a variety of development organizations, from start-up to multinational corporations. He has written numerous papers, contributed to the development of an international standard for distributed simulations, and has recently been working in a public-private collaboration to draft security recommendations for the smart grid.

Reed Little is a Senior Member of the Technical Staff at the SEI. He applies more than 35 years of experience in computer simulation, software architecture, software product lines, man-machine interface, artificial intelligence, and programming language design to various aspects of applied research and hands-on customer assistance for large (more than three million lines of code) software systems.

Paulo Merson has more than 20 years of software development experience. He works for the SEI in the areas of software architecture, service-oriented architecture, and aspect-oriented software development. He is also a practicing software architect in industry. One of his assignments at the SEI is to teach a two-day course in "Documenting Software Architectures" for industry and government practitioners. His speaking experience also includes tutorials at various conferences, such as SD Best Practices, Dr. Dobb's Architecture & Design World, and JavaOne. Prior to joining the SEI, he was a Java EE consultant. Paulo holds a B.Sc. in Computer Science from University of Brasilia, and a Master of Software Engineering from CMU.

Robert Nord is a Senior Member of the Technical Staff in the Research, Technology, and System Solutions Program at the SEI, where he works to develop and communicate effective methods and practices for software architecture. He is coauthor of the practitioner-oriented book *Applied Software Architecture* (Addison-Wesley, 2000) and lectures on architecture-centric approaches. He is a member of the IFIP Working Group on Software Architecture (WG2.10).

Judith Stafford is a Senior Lecturer at Tufts University and a Visiting Scientist at the SEI. Before joining the faculty at Tufts University, she was a Senior Member of the Technical Staff at the SEI in the Product Lines Systems Program, working in the Software Architecture Technologies Initiative. She has authored several book chapters on the topic of software architecture analysis, software architecture support for software component composition, and software architecture documentation. Stafford has been an organizer and program committee member for several conferences and workshops, and a guest editor on several leading software engineering journal special issues. She received her Ph.D. and M.S. degrees in Computer Science from the University of Colorado at Boulder. She is a member of the IEEE Computer Society, ACM SIGSOFT and SIGPLAN, and the IFIP Working Group on Software Architecture (WG2.10).

This page intentionally left blank

About the Contributors

Art Akerman is a Director, Enterprise Architecture, at a Fortune 500 financial services company. He has more than 15 years of experience developing and architecting complex, mission-critical systems for government, insurance, and financial services industries. Art is currently leading an effort to consolidate, virtualize, and standardize the company's technology portfolio and to apply service-oriented architecture principles to IT infrastructure.

Peter Eeles, Executive IT Architect at IBM Rational Software, has spent much of his career architecting and implementing large-scale, distributed systems. His current role is focused on helping organizations improve their software development capability. He coauthored *Building J2EE Applications with the Rational Unified Process* (Addison-Wesley, 2003), *Building Business Objects* (Wiley, 1998), and *The Process of Software Architecting* (Addison-Wesley, 2009).

David Emery is Chief Software Architect at DSCI, a systems/software engineering company, working on the Army's Future Combat Systems program. He has spent the last 18 years working on defining and improving the practice of architecture as a distinct discipline within software and systems engineering. David is head of the U.S. Technical Working Group for ISO/IEC JTC1/SC7 WG42, revising the ISO/IEC 42010:2007 standard for architecture description, and he was a major contributor to the predecessor IEEE Std 1471-2000.

George Fairbanks has been teaching software architecture and object-oriented design for ten years. In the spring of 2008 he was the co-instructor for the graduate software architecture course at Carnegie Mellon University (CMU). He holds a Ph.D. in Software Engineering from CMU. His dissertation introduced design fragments, a new way to specify and assure the correct use of frameworks through static analysis. He has written production code for telephone switches, plug-ins for the Eclipse IDE, and everything from soup to nuts for his dot-com start-up.

Rik Farenhorst has been a researcher at the Information Management and Software Engineering department of the VU University Amsterdam for four years. He conducts research on architectural knowledge management, which focuses on the effective application of knowledge management practices in the software architecture domain. His research results have been published in over a dozen refereed articles.

Peter Feiler has been with the CMU Software Engineering Institute (SEI) for 23 years. He is the technical lead and author of the Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) standard. His research interests include dependable real-time systems, architecture languages for embedded systems, and predictable system engineering.

James Herbsleb is a Professor in the School of Computer Science at CMU. For the last 18 years, his research has focused on coordination in software engineering projects. His practical experience comes from working in industry, leading the Bell Labs Collaboratory project, as well as consulting and collaborating with many industry partners, including IBM, Accenture, Bosch, and Siemens. His current work focuses on developing organizational and architectural tactics for improving coordination, and identifying and cataloguing socio-technical patterns that meld organizational and architectural solutions.

Rich Hilliard is a software systems architect consulting to industry, government, and academia. He is editor of ISO/IEC 42010, *Systems and Software Engineering—Architecture description* (the standard formerly known as ANSI/IEEE Std 1471). He has been writing about architecture since 1990.

John Klein is a Senior Member of the Technical Staff at SEI. Prior to joining SEI in 2008, he was a chief architect for communication application products at Avaya, Inc. John has more than 25 years' experience developing systems architectures, in domains ranging from sensors and weapons to videoconferencing and collaboration systems to telephone call centers.

Philippe Kruchten was a system and software architect with Alcatel and Rational Software for about 20 years on a variety of large software systems, in telecommunication, defense, and transportation. While at Rational, he developed the concept of multiple architecture views, a representation technique that he later included in the Rational Unified Process. He now teaches software engineering at the University of British Columbia in Vancouver.

John D. McGregor is an associate professor of computer science at Clemson University, a Visiting Scientist at the Software Engineering Institute, and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines, model-driven development, and component-based software engineering. Dr. McGregor advises systems engineers on software engineering decisions, including software architecture. He has defined a tool chain consisting of SysML, AADL, and UML as a means of providing continuity of information from system definition to code generation and test. He is a coauthor of *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley, 2001).

Don O'Connell is a Technical Fellow in Software/Systems Architecture and works for The Boeing Company. For the past nine years he has worked in Boeing Phantom Works, and he is leading an effort to increase Boeing's architecture competence through the introduction of key practices such as architecture evaluation, architecture development, architecture analysis, and architect certification.

T. V. Prabhakar is currently a Professor in Computer Science and Engineering at the Indian Institute of Technology, in Kanpur, where he has been since 1986. He has taught many courses on architecture and design for the industry; his forte is teaching architecture in a classroom. His current interests are software architecture and knowledge processing.

Nick Rozanski has worked in IT since the early 1980s. During his career he has worked as developer, designer, requirements analyst, and, more recently, architect, on a wide range of projects in finance, retail, manufacturing, and government. He currently leads the Enterprise Architecture group at Barclays Global Investors in London. He and his team are charged with delivering the vision and roadmaps for IT; for providing guidance and oversight for projects and programs; for supporting the IT Group's planning and investment processes; and fostering innovative solutions to challenging business problems.

Darpan Saini is a research fellow at the Master of Software Engineering program at Carnegie Mellon University. His primary research interests include programming language design and software architecture. He has prior experience developing tools that generate code from UML models.

Jeff Tyree is a Senior Director, IT Architecture, at a Fortune 500 financial services company. His interests include large-scale system design, system evolution processes, refactoring, and performance engineering. Jeff has more than 20 years of experience developing software for financial and defense industries. He received his bachelor's degree in Mathematics from Berea College in Berea, Kentucky, and a master's degree in Mathematics from the University of Tennessee, Knoxville.

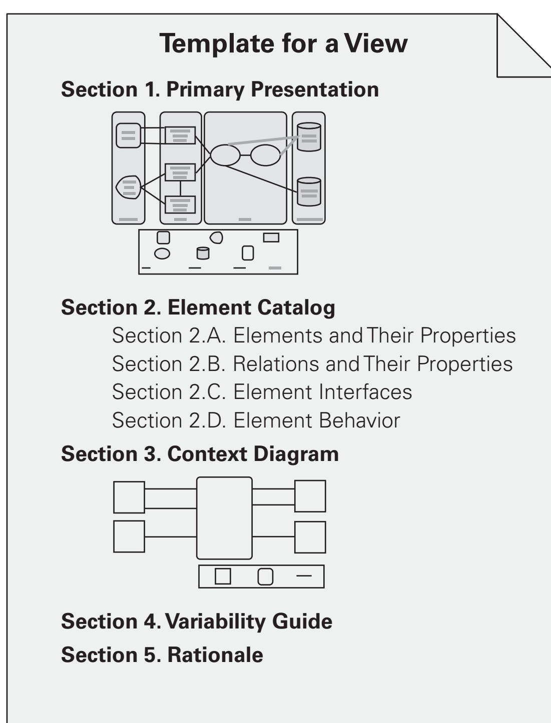
David M. Weiss received a B.S. in Mathematics in 1964 from Union College and a Ph.D. in Computer Science in 1981 from the University of Maryland. He is currently the Lanh and Oahn Nguyen professor of software engineering at Iowa State University. Dr. Weiss's best-known work is the goal-question-metric approach to software measurement, his explorations of the modular structure of software systems, and his work in software product-line engineering as a coinventor of the FAST process. He is coauthor and coeditor of two books: *Software Product Line Engineering* (Springer, 2005) and *Software Fundamentals: Collected Papers by David L. Parnas* (Addison-Wesley, 2001).

Eoin Woods is a software architect with Barclays Global Investors, responsible for the architecture of the firm's next-generation portfolio management system. Eoin has worked in software engineering since the early 1990s and has worked primarily as a software architect for the last ten years. He is coauthor, with Nick Rozanski, of the widely used book *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* (Addison-Wesley, 2005).

Index

Note: *Italicized* page locators indicate figures/tables.

This page intentionally left blank



Template for a View

Section 1. Primary Presentation

Section 2. Element Catalog

Section 2.A. Elements and Their Properties

Section 2.B. Relations and Their Properties

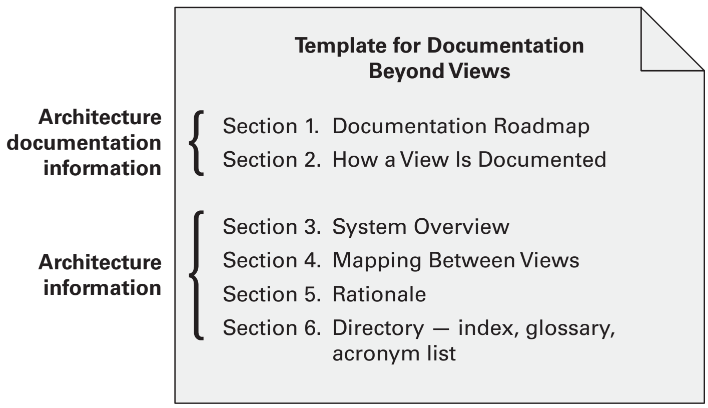
Section 2.C. Element Interfaces

Section 2.D. Element Behavior

Section 3. Context Diagram

Section 4. Variability Guide

Section 5. Rationale



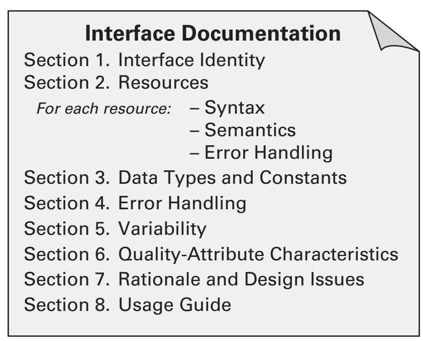
Template for Documentation Beyond Views

Architecture documentation information

- Section 1. Documentation Roadmap
- Section 2. How a View Is Documented

Architecture information

- Section 3. System Overview
- Section 4. Mapping Between Views
- Section 5. Rationale
- Section 6. Directory — index, glossary, acronym list



Interface Documentation

- Section 1. Interface Identity
- Section 2. Resources
  - *For each resource:* – Syntax
  - – Semantics
  - – Error Handling
- Section 3. Data Types and Constants
- Section 4. Error Handling
- Section 5. Variability
- Section 6. Quality-Attribute Characteristics
- Section 7. Rationale and Design Issues
- Section 8. Usage Guide