

Software Architecture in Practice

Third Edition

Uwe Gelsev | Paul C. Lohmeyer | Rolf Küller



The SEI Series in Software Engineering

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

CMM, CMMI, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ATAM; Architecture Tradeoff Analysis Method; CMM Integration; COTS Usage-Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Appraiser; SCAMPI Lead Assessor; SCE; SEI; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

Special permission to reproduce portions of works copyright by Carnegie Mellon University, as listed on page [588](#), is granted by the Software Engineering Institute.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informati.com/aw

Library of Congress Cataloging-in-Publication Data

Bass, Len.

Software architecture in practice / Len Bass, Paul Clements, Rick Kazman.—3rd ed.
p. cm.—(SEI series in software engineering)
Includes bibliographical references and index.
ISBN 978-0-321-81573-6 (hardcover : alk. paper) 1. Software architecture. 2. System
design. I. Clements, Paul, 1955– II. Kazman, Rick. III. Title.
QA76.754.B37 2012
005.1—dc23

2012023744

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson

Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-81573-6
ISBN-10: 0-321-81573-4

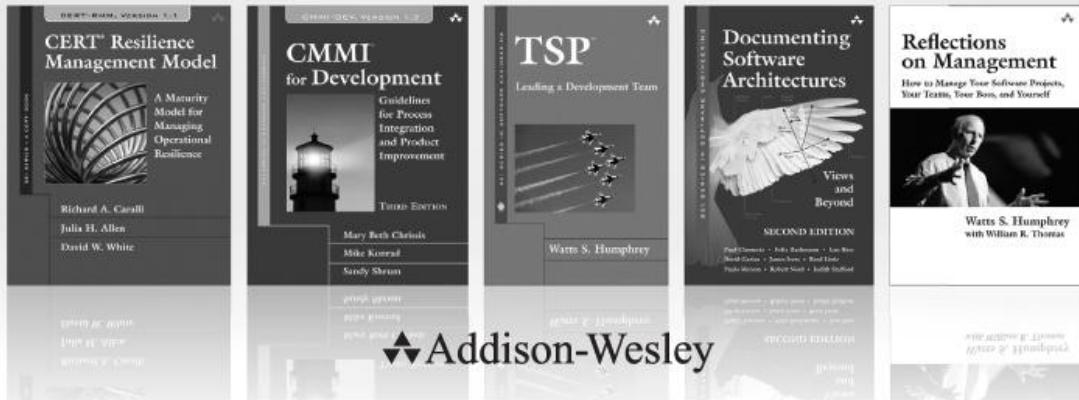
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, September 2012

The SEI Series in Software Engineering



Software Engineering Institute

Carnegie Mellon



▲ Addison-Wesley

Visit informit.com/sei for a complete list of available products.

The **SEI Series in Software Engineering** represents a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Books in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all books in the series address critical problems in software engineering for which practical solutions are available.

PEARSON

▲ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM
Press..

QUE'

PRENTICE HALL

SAMS

Safari

Contents

[Preface](#)

[Reader's Guide](#)

[Acknowledgments](#)

Part One Introduction

Chapter 1 What Is Software Architecture?

- [1.1 What Software Architecture Is and What It Isn't](#)
- [1.2 Architectural Structures and Views](#)
- [1.3 Architectural Patterns](#)
- [1.4 What Makes a "Good" Architecture?](#)
- [1.5 Summary](#)
- [1.6 For Further Reading](#)
- [1.7 Discussion Questions](#)

Chapter 2 Why Is Software Architecture Important?

- [2.1 Inhibiting or Enabling a System's Quality Attributes](#)
- [2.2 Reasoning About and Managing Change](#)
- [2.3 Predicting System Qualities](#)
- [2.4 Enhancing Communication among Stakeholders](#)
- [2.5 Carrying Early Design Decisions](#)
- [2.6 Defining Constraints on an Implementation](#)
- [2.7 Influencing the Organizational Structure](#)
- [2.8 Enabling Evolutionary Prototyping](#)
- [2.9 Improving Cost and Schedule Estimates](#)
- [2.10 Supplying a Transferable, Reusable Model](#)
- [2.11 Allowing Incorporation of Independently Developed Components](#)
- [2.12 Restricting the Vocabulary of Design Alternatives](#)
- [2.13 Providing a Basis for Training](#)
- [2.14 Summary](#)
- [2.15 For Further Reading](#)
- [2.16 Discussion Questions](#)

Chapter 3 The Many Contexts of Software Architecture

- [3.1 Architecture in a Technical Context](#)
- [3.2 Architecture in a Project Life-Cycle Context](#)
- [3.3 Architecture in a Business Context](#)
- [3.4 Architecture in a Professional Context](#)
- [3.5 Stakeholders](#)
- [3.6 How Is Architecture Influenced?](#)
- [3.7 What Do Architectures Influence?](#)
- [3.8 Summary](#)
- [3.9 For Further Reading](#)
- [3.10 Discussion Questions](#)

Part Two Quality Attributes

Chapter 4 Understanding Quality Attributes

- [4.1 Architecture and Requirements](#)
- [4.2 Functionality](#)
- [4.3 Quality Attribute Considerations](#)
- [4.4 Specifying Quality Attribute Requirements](#)
- [4.5 Achieving Quality Attributes through Tactics](#)
- [4.6 Guiding Quality Design Decisions](#)
- [4.7 Summary](#)
- [4.8 For Further Reading](#)
- [4.9 Discussion Questions](#)

Chapter 5 Availability

- [5.1 Availability General Scenario](#)
- [5.2 Tactics for Availability](#)
- [5.3 A Design Checklist for Availability](#)
- [5.4 Summary](#)
- [5.5 For Further Reading](#)
- [5.6 Discussion Questions](#)

Chapter 6 Interoperability

- [6.1 Interoperability General Scenario](#)
- [6.2 Tactics for Interoperability](#)
- [6.3 A Design Checklist for Interoperability](#)
- [6.4 Summary](#)
- [6.5 For Further Reading](#)
- [6.6 Discussion Questions](#)

Chapter 7 Modifiability

- [7.1 Modifiability General Scenario](#)
- [7.2 Tactics for Modifiability](#)
- [7.3 A Design Checklist for Modifiability](#)
- [7.4 Summary](#)
- [7.5 For Further Reading](#)
- [7.6 Discussion Questions](#)

Chapter 8 Performance

- [8.1 Performance General Scenario](#)
- [8.2 Tactics for Performance](#)
- [8.3 A Design Checklist for Performance](#)
- [8.4 Summary](#)
- [8.5 For Further Reading](#)
- [8.6 Discussion Questions](#)

Chapter 9 Security

- [9.1 Security General Scenario](#)
- [9.2 Tactics for Security](#)
- [9.3 A Design Checklist for Security](#)

[9.4 Summary](#)

[9.5 For Further Reading](#)

[9.6 Discussion Questions](#)

Chapter 10 Testability

[10.1 Testability General Scenario](#)

[10.2 Tactics for Testability](#)

[10.3 A Design Checklist for Testability](#)

[10.4 Summary](#)

[10.5 For Further Reading](#)

[10.6 Discussion Questions](#)

Chapter 11 Usability

[11.1 Usability General Scenario](#)

[11.2 Tactics for Usability](#)

[11.3 A Design Checklist for Usability](#)

[11.4 Summary](#)

[11.5 For Further Reading](#)

[11.6 Discussion Questions](#)

Chapter 12 Other Quality Attributes

[12.1 Other Important Quality Attributes](#)

[12.2 Other Categories of Quality Attributes](#)

[12.3 Software Quality Attributes and System Quality Attributes](#)

[12.4 Using Standard Lists of Quality Attributes—or Not](#)

[12.5 Dealing with “X-ability”: Bringing a New Quality Attribute into the Fold](#)

[12.6 For Further Reading](#)

[12.7 Discussion Questions](#)

Chapter 13 Architectural Tactics and Patterns

[13.1 Architectural Patterns](#)

[13.2 Overview of the Patterns Catalog](#)

[13.3 Relationships between Tactics and Patterns](#)

[13.4 Using Tactics Together](#)

[13.5 Summary](#)

[13.6 For Further Reading](#)

[13.7 Discussion Questions](#)

Chapter 14 Quality Attribute Modeling and Analysis

[14.1 Modeling Architectures to Enable Quality Attribute Analysis](#)

[14.2 Quality Attribute Checklists](#)

[14.3 Thought Experiments and Back-of-the-Envelope Analysis](#)

[14.4 Experiments, Simulations, and Prototypes](#)

[14.5 Analysis at Different Stages of the Life Cycle](#)

[14.6 Summary](#)

[14.7 For Further Reading](#)

[14.8 Discussion Questions](#)

Part Three Architecture in the Life Cycle

Chapter 15 Architecture in Agile Projects

- [15.1 How Much Architecture?](#)
- [15.2 Agility and Architecture Methods](#)
- [15.3 A Brief Example of Agile Architecting](#)
- [15.4 Guidelines for the Agile Architect](#)
- [15.5 Summary](#)
- [15.6 For Further Reading](#)
- [15.7 Discussion Questions](#)

Chapter 16 Architecture and Requirements

- [16.1 Gathering ASRs from Requirements Documents](#)
- [16.2 Gathering ASRs by Interviewing Stakeholders](#)
- [16.3 Gathering ASRs by Understanding the Business Goals](#)
- [16.4 Capturing ASRs in a Utility Tree](#)
- [16.5 Tying the Methods Together](#)
- [16.6 Summary](#)
- [16.7 For Further Reading](#)
- [16.8 Discussion Questions](#)

Chapter 17 Designing an Architecture

- [17.1 Design Strategy](#)
- [17.2 The Attribute-Driven Design Method](#)
- [17.3 The Steps of ADD](#)
- [17.4 Summary](#)
- [17.5 For Further Reading](#)
- [17.6 Discussion Questions](#)

Chapter 18 Documenting Software Architectures

- [18.1 Uses and Audiences for Architecture Documentation](#)
- [18.2 Notations for Architecture Documentation](#)
- [18.3 Views](#)
- [18.4 Choosing the Views](#)
- [18.5 Combining Views](#)
- [18.6 Building the Documentation Package](#)
- [18.7 Documenting Behavior](#)
- [18.8 Architecture Documentation and Quality Attributes](#)
- [18.9 Documenting Architectures That Change Faster Than You Can Document Them](#)
- [18.10 Documenting Architecture in an Agile Development Project](#)
- [18.11 Summary](#)
- [18.12 For Further Reading](#)
- [18.13 Discussion Questions](#)

Chapter 19 Architecture, Implementation, and Testing

- [19.1 Architecture and Implementation](#)
- [19.2 Architecture and Testing](#)
- [19.3 Summary](#)

[19.4 For Further Reading](#)
[19.5 Discussion Questions](#)

Chapter 20 Architecture Reconstruction and Conformance

[20.1 Architecture Reconstruction Process](#)
[20.2 Raw View Extraction](#)
[20.3 Database Construction](#)
[20.4 View Fusion](#)
[20.5 Architecture Analysis: Finding Violations](#)
[20.6 Guidelines](#)
[20.7 Summary](#)
[20.8 For Further Reading](#)
[20.9 Discussion Questions](#)

Chapter 21 Architecture Evaluation

[21.1 Evaluation Factors](#)
[21.2 The Architecture Tradeoff Analysis Method](#)
[21.3 Lightweight Architecture Evaluation](#)
[21.4 Summary](#)
[21.5 For Further Reading](#)
[21.6 Discussion Questions](#)

Chapter 22 Management and Governance

[22.1 Planning](#)
[22.2 Organizing](#)
[22.3 Implementing](#)
[22.4 Measuring](#)
[22.5 Governance](#)
[22.6 Summary](#)
[22.7 For Further Reading](#)
[22.8 Discussion Questions](#)

Part Four Architecture and Business

Chapter 23 Economic Analysis of Architectures

[23.1 Decision-Making Context](#)
[23.2 The Basis for the Economic Analyses](#)
[23.3 Putting Theory into Practice: The CBAM](#)
[23.4 Case Study: The NASA ECS Project](#)
[23.5 Summary](#)
[23.6 For Further Reading](#)
[23.7 Discussion Questions](#)

Chapter 24 Architecture Competence

[24.1 Competence of Individuals: Duties, Skills, and Knowledge of Architects](#)
[24.2 Competence of a Software Architecture Organization](#)
[24.3 Summary](#)
[24.4 For Further Reading](#)

[24.5 Discussion Questions](#)

[Chapter 25 Architecture and Software Product Lines](#)

- [25.1 An Example of Product Line Variability](#)
- [25.2 What Makes a Software Product Line Work?](#)
- [25.3 Product Line Scope](#)
- [25.4 The Quality Attribute of Variability](#)
- [25.5 The Role of a Product Line Architecture](#)
- [25.6 Variation Mechanisms](#)
- [25.7 Evaluating a Product Line Architecture](#)
- [25.8 Key Software Product Line Issues](#)
- [25.9 Summary](#)
- [25.10 For Further Reading](#)
- [25.11 Discussion Questions](#)

[Part Five The Brave New World](#)

[Chapter 26 Architecture in the Cloud](#)

- [26.1 Basic Cloud Definitions](#)
- [26.2 Service Models and Deployment Options](#)
- [26.3 Economic Justification](#)
- [26.4 Base Mechanisms](#)
- [26.5 Sample Technologies](#)
- [26.6 Architecting in a Cloud Environment](#)
- [26.7 Summary](#)
- [26.8 For Further Reading](#)
- [26.9 Discussion Questions](#)

[Chapter 27 Architectures for the Edge](#)

- [27.1 The Ecosystem of Edge-Dominant Systems](#)
- [27.2 Changes to the Software Development Life Cycle](#)
- [27.3 Implications for Architecture](#)
- [27.4 Implications of the Metropolis Model](#)
- [27.5 Summary](#)
- [27.6 For Further Reading](#)
- [27.7 Discussion Questions](#)

[Chapter 28 Epilogue](#)

- [References](#)
- [About the Authors](#)
- [Index](#)

Preface

I should have no objection to go over the same life from its beginning to the end: requesting only the advantage authors have, of correcting in a [third] edition the faults of the first [two].

—Benjamin Franklin

It has been a decade since the publication of the second edition of this book. During that time, the field of software architecture has broadened its focus from being primarily internally oriented—How does one design, evaluate, and document software?—to including external impacts as well—a deeper understanding of the influences on architectures and a deeper understanding of the impact architectures have on the life cycle, organizations, and management.

The past ten years have also seen dramatic changes in the types of systems being constructed. Large data, social media, and the cloud are all areas that, at most, were embryonic ten years ago and now are not only mature but extremely influential.

We listened to some of the criticisms of the previous editions and have included much more material on patterns, reorganized the material on quality attributes, and made interoperability a quality attribute worthy of its own chapter. We also provide guidance about how you can generate scenarios and tactics for your own favorite quality attributes.

To accommodate this plethora of new material, we had to make difficult choices. In particular, this edition of the book does not include extended case studies as the prior editions did. This decision also reflects the maturing of the field, in the sense that case studies about the choices made in software architectures are more prevalent than they were ten years ago, and they are less necessary to convince readers of the importance of software architecture. The case studies from the first two editions are available, however, on the book’s website, at www.informit.com/title/9780321815736. In addition, on the same website, we have slides that will assist instructors in presenting this material.

We have thoroughly reworked many of the topics covered in this edition. In particular, we realize that the methods we present—for architecture design, analysis, and documentation—are one version of how to achieve a particular goal, but there are others. This led us to separate the methods that we present in detail from their underlying theory. We now present the theory first with specific methods given as illustrations of possible realizations of the theories. The new topics in this edition include architecture-centric project management; architecture competence; requirements modeling and analysis; Agile methods; implementation and testing; the cloud; and the edge.

As with the prior editions, we firmly believe that the topics are best discussed in either reading groups or in classroom settings, and to that end we have included a collection of discussion questions at the end of each chapter. Most of these questions are open-ended, with no absolute right or wrong answers, so you, as a reader, should emphasize how you justify your answer rather than just answer the question itself.

Reader's Guide

We have structured this book into five distinct portions. [Part One](#) introduces architecture and the various contextual lenses through which it could be viewed. These are the following:

- *Technical*. What technical role does the software architecture play in the system or systems of which it's a part?
- *Project*. How does a software architecture relate to the other phases of a software development life cycle?
- *Business*. How does the presence of a software architecture affect an organization's business environment?
- *Professional*. What is the role of a software architect in an organization or a development project?

[Part Two](#) is focused on technical background. [Part Two](#) describes how decisions are made. Decisions are based on the desired quality attributes for a system, and [Chapters 5–11](#) describe seven different quality attributes and the techniques used to achieve them. The seven are availability, interoperability, maintainability, performance, security, testability, and usability. [Chapter 12](#) tells you how to add other quality attributes to our seven, [Chapter 13](#) discusses patterns and tactics, and [Chapter 14](#) discusses the various types of modeling and analysis that are possible.

[Part Three](#) is devoted to how a software architecture is related to the other portions of the life cycle. Of special note is how architecture can be used in Agile projects. We discuss individually other aspects of the life cycle: requirements, design, implementation and testing, recovery and conformance, and evaluation.

[Part Four](#) deals with the business of architecting from an economic perspective, from an organizational perspective, and from the perspective of constructing a series of similar systems.

[Part Five](#) discusses several important emerging technologies and how architecture relates to these technologies.

Acknowledgments

We had a fantastic collection of reviewers for this edition, and their assistance helped make this a better book. Our reviewers were Muhammad Ali Babar, Felix Bachmann, Joe Batman, Phil Bianco, Jeromy Carriere, Roger Champagne, Steve Chenoweth, Viktor Clerc, Andres Diaz Pace, George Fairbanks, Rik Farenhorst, Ian Gorton, Greg Hartman, Rich Hilliard, James Ivers, John Klein, Philippe Kruchten, Phil Laplante, George Leih, Grace Lewis, John McGregor, Tommi Mikkonen, Linda Northrop, Ipek Ozkaya, Eltjo Poort, Eelco Rommes, Nick Rozanski, Jungwoo Ryoo, James Scott, Antony Tang, Arjen Uittenbogaard, Hans van Vliet, Hiroshi Wada, Rob Wojcik, Eoin Woods, and Liming Zhu.

In addition, we had significant contributions from Liming Zhu, Hong-Mei Chen, Jungwoo Ryoo, Phil Laplante, James Scott, Grace Lewis, and Nick Rozanski that helped give the book a richer flavor than one written by just the three of us.

The issue of build efficiency in [Chapter 12](#) came from Rolf Siegers and John McDonald of Raytheon. John Klein and Eltjo Poort contributed the “abstract system clock” and “sandbox mode” tactics, respectively, for testability. The list of stakeholders in [Chapter 3](#) is from *Documenting Software Architectures: Views and Beyond, Second Edition*. Some of the material in [Chapter 28](#) was inspired by a talk given by Anthony Lattanze called “Organizational Design Thinking” in 2011.

Joe Batman was instrumental in the creation of the seven categories of design decisions we describe in [Chapter 4](#). In addition, the descriptions of the security view, communications view, and exception view in [Chapter 18](#) are based on material that Joe wrote while planning the documentation for a real system’s architecture. Much of the new material on modifiability tactics was based on the work of Felix Bachmann and Rod Nord. James Ivers helped us with the security tactics.

Both Paul Clements and Len Bass have taken new positions since the last edition was published, and we thank their new respective managements (BigLever Software for Paul and NICTA for Len) for their willingness to support our work on this edition. We would also like to thank our (former) colleagues at the Software Engineering Institute for multiple contributions to the evolution of the ideas expressed in this edition.

Finally, as always, we thank our editor at Addison-Wesley, Peter Gordon, for providing guidance and support during the writing and production processes.

Part One. Introduction

What is a software architecture? What is it good for? How does it come to be? What effect does its existence have? These are the questions we answer in Part I.

[Chapter 1](#) deals with a technical perspective on software architecture. We define it and relate it to system and enterprise architectures. We discuss how the architecture can be represented in different views to emphasize different perspectives on the architecture. We define patterns and discuss what makes a “good” architecture.

In [Chapter 2](#), we discuss the uses of an architecture. You may be surprised that we can find so many—ranging from a vehicle for communication among stakeholders to a blueprint for implementation, to the carrier of the system’s quality attributes. We also discuss how the architecture provides a reasoned basis for schedules and how it provides the foundation for training new members on a team.

Finally, in [Chapter 3](#), we discuss the various contexts in which a software architecture exists. It exists in a technical context, in a project life-cycle context, in a business context, and in a professional context. Each of these contexts defines a role for the software architecture to play, or an influence on it. These impacts and influences define the Architecture Influence Cycle.

1. What Is Software Architecture?

Good judgment is usually the result of experience. And experience is frequently the result of bad judgment. But to learn from the experience of others requires those who have the experience to share the knowledge with those who follow.

—Barry LePatner

Writing (on our part) and reading (on your part) a book about software architecture, which distills the experience of many people, presupposes that

1. having a software architecture is important to the successful development of a software system and
2. there is a sufficient, and sufficiently generalizable, body of knowledge about software architecture to fill up a book.

One purpose of this book is to convince you that both of these assumptions are true, and once you are convinced, give you a basic knowledge so that you can apply it yourself.

Software systems are constructed to satisfy organizations' business goals. The architecture is a bridge between those (often abstract) business goals and the final (concrete) resulting system. While the path from abstract goals to concrete systems can be complex, the good news is that software architectures can be designed, analyzed, documented, and implemented using known techniques that will support the achievement of these business and mission goals. The complexity can be tamed, made tractable.

These, then, are the topics for this book: the design, analysis, documentation, and implementation of architectures. We will also examine the influences, principally in the form of business goals and quality attributes, which inform these activities.

In this chapter we will focus on architecture strictly from a software engineering point of view. That is, we will explore the value that a software architecture brings to a development project. (Later chapters will take a business and organizational perspective.)

1.1. What Software Architecture Is and What It Isn't

There are many definitions of software architecture, easily discoverable with a web search, but the one we like is this one:

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

This definition stands in contrast to other definitions that talk about the system's "early" or "major" design decisions. While it is true that many architectural decisions are made early, not all are—especially in Agile or spiral-development projects. It's also true that very many decisions are made early that are not architectural. Also, it's hard to look at a decision and tell whether or not it's "major." Sometimes only time will tell. And since writing down an architecture is one of the architect's most important obligations, we need to know now which decisions an architecture comprises.

Structures, on the other hand, are fairly easy to identify in software, and they form a powerful tool for system design.

Let us look at some of the implications of our definition.

Architecture Is a Set of Software Structures

This is the first and most obvious implication of our definition. A structure is simply a set of elements held together by a relation. Software systems are composed of many structures, and no single structure holds claim to being *the* architecture. There are three categories of architectural structures, which will play an important role in the design, documentation, and analysis of architectures:

1. First, some structures partition systems into implementation units, which in this book we call *modules*. Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams (Team A works on the database, Team B works on the business rules, Team C works on the user interface, etc.). In large projects, these elements (modules) are subdivided for assignment to subteams. For example, the database for a large enterprise resource planning (ERP) implementation might be so complex that its implementation is split into many parts. The structure that captures that decomposition is a kind of module structure, the *module decomposition structure* in fact. Another kind of module structure emerges as an output of object-oriented analysis and design—class diagrams. If you aggregate your modules into layers, you've created another (and very useful) module structure. Module structures are static structures, in that they focus on the way the system's functionality is divided up and assigned to implementation teams.
2. Other structures are dynamic, meaning that they focus on the way the elements interact with each other at runtime to carry out the system's functions. Suppose the system is to be built as a set of services. The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system. These services are made up of (compiled from) the programs in the various implementation units that we just described. In this book we will call runtime structures *component-and-connector* (C&C) structures. The term *component* is overloaded in software engineering. In our use, a component is always a runtime entity.
3. A third kind of structure describes the mapping from software structures to the system's organizational, developmental, installation, and execution environments. For example, modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing. Components are deployed onto hardware in order to execute. These mappings are called *allocation* structures.

Although software comprises an endless supply of structures, not all of them are architectural. For example, the set of lines of source code that contain the letter "z," ordered by increasing length from shortest to longest, is a software structure. But it's not a very interesting one, nor is it architectural. A structure is architectural if it supports reasoning about the system and the system's properties. The reasoning should be about an attribute of the system that is important to some stakeholder. These include functionality achieved by the system, the availability of the

system in the face of faults, the difficulty of making specific changes to the system, the responsiveness of the system to user requests, and many others. We will spend a great deal of time in this book on the relationship between architecture and quality attributes like these.

Thus, the set of architectural structures is not fixed or limited. What is architectural is what is useful in your context for your system.

Architecture Is an Abstraction

Because architecture consists of structures and structures consist of elements¹ and relations, it follows that an architecture comprises software elements and how the elements relate to each other. This means that architecture specifically omits certain information about elements that is not useful for reasoning about the system—in particular, it omits information that has no ramifications outside of a single element. Thus, an architecture is foremost an *abstraction* of a system that selects certain details and suppresses others. In all modern systems, elements interact with each other by means of interfaces that partition details about an element into public and private parts. Architecture is concerned with the public side of this division; private details of elements—details having to do solely with internal implementation—are not architectural. Beyond just interfaces, though, the architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth. This abstraction is essential to taming the complexity of a system—we simply cannot, and do not want to, deal with all of the complexity all of the time.

¹. In this book we use the term “element” when we mean either a module or a component, and don’t want to distinguish.

Every Software System Has a Software Architecture

Every system can be shown to comprise elements and relations among them to support some type of reasoning. In the most trivial case, a system is itself a single element—an uninteresting and probably non-useful architecture, but an architecture nevertheless.

Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone. Perhaps all of the people who designed the system are long gone, the documentation has vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code. This reveals the difference between the architecture of a system and the *representation* of that architecture. Because an architecture can exist independently of its description or specification, this raises the importance of *architecture documentation*, which is described in [Chapter 18](#), and *architecture reconstruction*, discussed in [Chapter 20](#).

Architecture Includes Behavior

The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system. This behavior embodies how elements interact with each other, which is clearly part of our definition of architecture.

This tells us that box-and-line drawings that are passed off as architectures are in fact not architectures at all. When looking at the names of the boxes (database, graphical user interface, executive, etc.), a reader may well imagine the functionality and behavior of the corresponding elements. This mental image approaches an architecture, but it springs from the imagination of the observer’s mind and relies on information that is not present. This does not mean that the exact behavior and performance of every element must be documented in all circumstances—some aspects of behavior are fine-grained and below the architect’s level of concern. But to the extent that an element’s behavior influences another element or influences the acceptability of the system as a whole, this behavior must be considered, and should be documented, as part of the software architecture.

Not All Architectures Are Good Architectures

The definition is indifferent as to whether the architecture for a system is a good one or a bad one. An architecture may permit or preclude a system’s achievement of its behavioral, quality attribute, and life-cycle requirements. Assuming that we do not accept trial and error as the best way to choose an architecture for a system—that is, picking an architecture at random, building the

system from it, and then hacking away and hoping for the best—this raises the importance of *architecture design*, which is treated in [Chapter 17](#), and *architecture evaluation*, which we deal with in [Chapter 21](#).

System and Enterprise Architectures

Two disciplines related to software architecture are system architecture and enterprise architecture. Both of these disciplines have broader concerns than software and affect software architecture through the establishment of constraints within which a software system must live. In both cases, the software architect for a system should be on the team that provides input into the decisions made about the system or the enterprise.

System architecture

A system’s architecture is a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components. That is, system architecture is concerned with a total system, including hardware, software, and humans.

A system architecture will determine, for example, the functionality that is assigned to different processors and the type of network that connects those processors. The software architecture on each of those processors will determine how this functionality is implemented and how the various processors interact through the exchange of messages on the network.

A description of the software architecture, as it is mapped to hardware and networking components, allows reasoning about qualities such as performance and reliability. A description of the system architecture will allow reasoning about additional qualities such as power consumption, weight, and physical footprint.

When a particular system is designed, there is frequently negotiation between the system architect and the software architect as to the distribution of functionality and, consequently, the constraints placed on the software architecture.

Enterprise architecture

Enterprise architecture is a description of the structure and behavior of an organization’s processes, information flow, personnel, and organizational subunits, aligned with the organization’s core goals and strategic direction. An enterprise architecture need not include information systems—clearly organizations had architectures that fit the preceding definition prior to the advent of computers—but these days, enterprise architectures for all but the smallest businesses are unthinkable without information system support. Thus, a modern enterprise architecture is concerned with how an enterprise’s software systems support the business processes and goals of the enterprise. Typically included in this set of concerns is a process for deciding which systems with which functionality should be supported by an enterprise.

An enterprise architecture will specify the data model that various systems use to interact, for example. It will specify rules for how the enterprise’s systems interact with external systems.

Software is only one concern of enterprise architecture. Two other common concerns addressed by enterprise architecture are how the software is used by humans to perform business processes, and the standards that determine the computational environment.

Sometimes the software infrastructure that supports communication among systems and with the external world is considered a portion of the enterprise architecture; other times, this infrastructure is considered one of the systems within an enterprise. (In either case, the architecture of that infrastructure is a *software architecture!*) These two views will result in different management structures and spheres of influence for the individuals concerned with the infrastructure.

The system and the enterprise provide environments for, and constraints on, the software architecture. The software architecture must live within the system and enterprise, and increasingly it is the focus for achieving the organization’s business goals. But all three forms of architecture share important commonalities: They are concerned with major

elements taken as abstractions, the relationships among the elements, and how the elements together meet the behavioral and quality goals of the thing being built.

Are these in scope for this book? Yes! (Well, no.)

System and enterprise architectures share a great deal with software architectures. All can be designed, evaluated, and documented; all answer to requirements; all are intended to satisfy stakeholders; all consist of structures, which in turn consist of elements and relationships; all have a repertoire of patterns and styles at their respective architects' disposal; and the list goes on. So to the extent that these architectures share commonalities with software architecture, they are in the scope of this book. But like all technical disciplines, each has its own specialized vocabulary and techniques, and we won't cover those. Copious other sources do.

1.2. Architectural Structures and Views

The neurologist, the orthopedist, the hematologist, and the dermatologist all have different views of the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on specific subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these views are pictured differently and have very different properties, all are inherently related, interconnected: together they describe the architecture of the human body. [Figure 1.1](#) shows several different views of the human body: the skeletal, the vascular, and the X-ray.

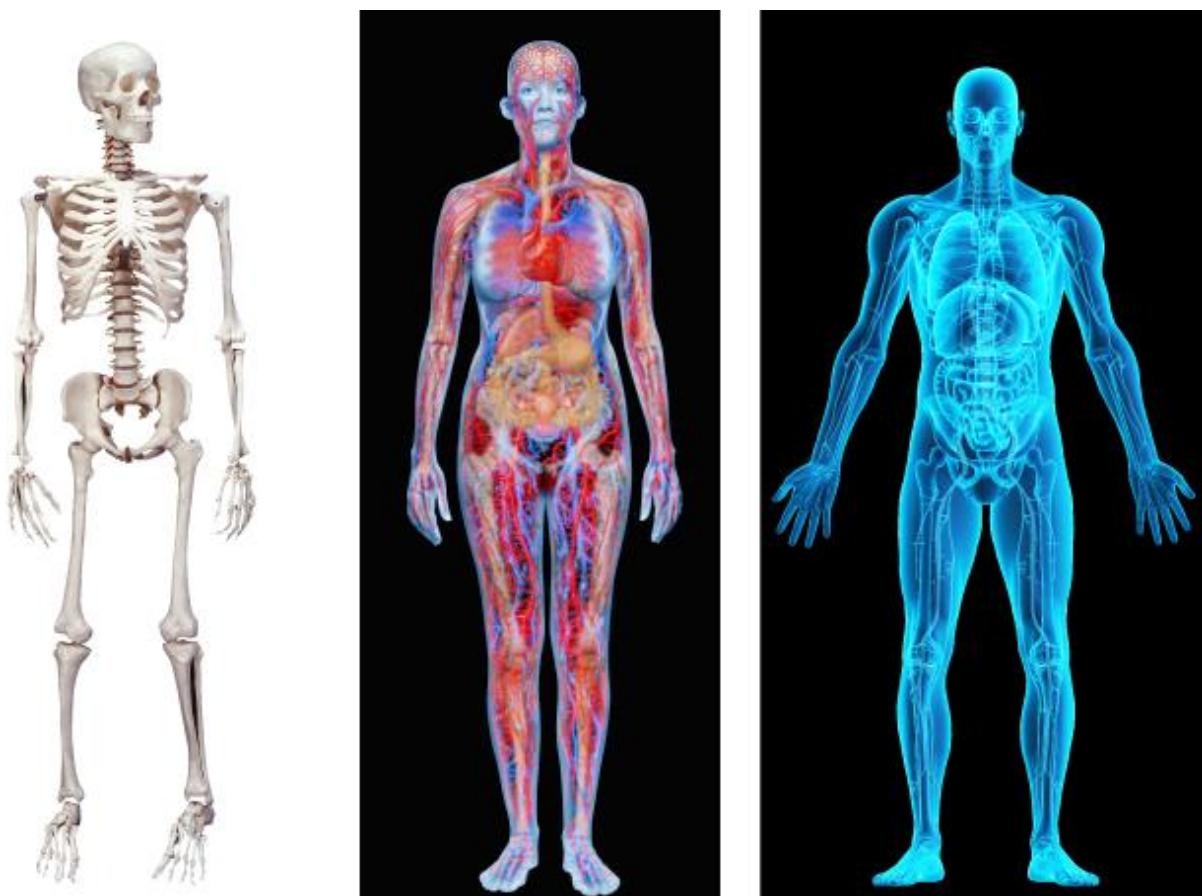


Figure 1.1. Physiological structures (Getty images: Brand X Pictures [skeleton], Don Farrall [woman], Mads Abildgaard [man])

So it is with software. Modern systems are frequently too complex to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which

structure or structures we are discussing at the moment—which *view* we are taking of the architecture.

Structures and Views

We will be using the related terms *structure* and *view* when discussing architecture representation.

- A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.
- A structure is the set of elements itself, as they exist in software or hardware.

In short, a view is a representation of a structure. For example, a module *structure* is the set of the system's modules and their organization. A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.

So: Architects design structures. They document views of those structures.

Three Kinds of Structures

As we saw in the previous section, architectural structures can be divided into three major categories, depending on the broad nature of the elements they show. These correspond to the three broad kinds of decisions that architectural design involves:

1. *Module structures* embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured. In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation). Modules represent a static way of considering the system. Modules are assigned areas of functional responsibility; there is less emphasis in these structures on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as these:

- What is the primary functional responsibility assigned to each module?
- What other software elements is a module allowed to use?
- What other software does it actually use and depend on?
- What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Module structures convey this information directly, but they can also be used by extension to ask questions about the impact on the system when the responsibilities assigned to each module change. In other words, examining a system's module structures—that is, looking at its module views—is an excellent way to reason about a system's modifiability.

2. *Component-and-connector structures* embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors). In these structures, the elements are runtime components (which are the principal units of computation and could be services, peers, clients, servers, filters, or many other types of runtime elements) and connectors (which are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others). Component-and-connector views help us answer questions such as these:

- What are the major executing components and how do they interact at runtime?
- What are the major shared data stores?
- Which parts of the system are replicated?
- How does data progress through the system?
- What parts of the system can run in parallel?
- Can the system's structure change as it executes and, if so, how?

By extension, component-and-connector views are crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.

3. Allocation structures embody decisions as to how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.). These structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation views help us answer questions such as these:

- What processor does each software element execute on?
- In what directories or files is each element stored during development, testing, and system building?
- What is the assignment of each software element to development teams?

Structures Provide Insight

Structures play such an important role in our perspective on software architecture because of the analytical and engineering power they hold. Each structure provides a perspective for reasoning about some of the relevant quality attributes. For example:

- The module “uses” structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.
- The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.
- The deployment structure is strongly tied to the achievement of performance, availability, and security goals.

And so forth. Each structure provides the architect with a different insight into the design (that is, each structure can be analyzed for its ability to deliver a quality attribute). But perhaps more important, each structure presents the architect with an engineering leverage point: By designing the structures appropriately, the desired quality attributes emerge.

Scenarios, described in [Chapter 4](#), are useful for exercising a given structure as well as its connections to other structures. For example, a software engineer wanting to make a change to the concurrency structure of a system would need to consult the concurrency and deployment views, because the affected mechanisms typically involve processes and threads, and physical distribution might involve different control mechanisms than would be used if the processes were co-located on a single machine. If control mechanisms need to be changed, the module decomposition would need to be consulted to determine the extent of the changes.

Some Useful Module Structures

Useful module structures include the following:

- **Decomposition structure.** The units are modules that are related to each other by the *is-a-submodule-of* relation, showing how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent (more detailed) design and eventual implementation. Modules often have products (such as interface specifications, code, test plans, etc.) associated with them. The decomposition structure determines, to a large degree, the system’s modifiability, by assuring that likely changes are localized. That is, changes fall within the purview of at most a few (preferably small) modules. This structure is often used as the basis for the development project’s organization, including the structure of the documentation, and the project’s integration and test plans. The units in this structure tend to have names that are organization-specific such as “segment” or “subsystem.”
- **Uses structure.** In this important but overlooked structure, the units here are also modules, perhaps classes. The units are related by the *uses* relation, a specialized form of dependency. A unit of software uses another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted. The ability to easily create a subset of a system allows for incremental development.

- **Layer structure.** The modules in this structure are called layers. A layer is an abstract “virtual machine” that provides a cohesive set of services through a managed interface. Layers are allowed to use other layers in a strictly managed fashion; in strictly layered systems, a layer is only allowed to use the layer immediately below. This structure is used to imbue a system with portability, the ability to change the underlying computing platform.
- **Class (or generalization) structure.** The module units in this structure are called classes. The relation is *inherits from* or *is an instance of*. This view supports reasoning about collections of similar behavior or capability (e.g., the classes that other classes inherit from) and parameterized differences. The class structure allows one to reason about reuse and the incremental addition of functionality. If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.
- **Data model.** The data model describes the static information structure in terms of data entities and their relationships. For example, in a banking system, entities will 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.

Some Useful C&C Structures

Component-and-connector structures show a runtime view of the system. In these structures the modules described above have all been compiled into executable forms. All component-and-connector structures are thus orthogonal to the module-based structures and deal with the dynamic aspects of a running system. The relation in all component-and-connector structures is *attachment*, showing how the components and the connectors are hooked together. (The connectors themselves can be familiar constructs such as “invokes.”) Useful C&C structures include the following:

- **Service structure.** The units here are services that interoperate with each other by service coordination mechanisms such as SOAP (see [Chapter 6](#)). The service structure is an important structure to help engineer a system composed of components that may have been developed anonymously and independently of each other.
- **Concurrency structure.** This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are their communication mechanisms. The components are arranged into *logical threads*; a logical thread is a sequence of computations that could be allocated to a separate physical thread later in the design process. The concurrency structure is used early in the design process to identify the requirements to manage the issues associated with concurrent execution.

Some Useful Allocation Structures

Allocation structures define how the elements from C&C or module structures map onto things that are not software: typically hardware, teams, and file systems. Useful allocation structures include these:

- **Deployment structure.** The deployment structure shows how software is assigned to hardware processing and communication elements. The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways. Relations are *allocated-to*, showing on which physical units the software elements reside, and *migrates-to* if the allocation is dynamic. This structure can be used to reason about performance, data integrity, security, and availability. It is of particular interest in distributed and parallel systems.
- **Implementation structure.** This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system’s development, integration, or configuration control environments. This is critical for the management of development activities and build processes. (In practice, a screenshot of your development environment tool, which manages the implementation environment, often makes a very useful and sufficient diagram of your implementation view.)

- **Work assignment structure.** This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out. Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications. The architect will know the expertise required on each team. Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning those to a single team, rather than having them implemented by everyone who needs them. This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.

[Table 1.1](#) summarizes these structures. The table lists the meaning of the elements and relations in each structure and tells what each might be used for.

Table 1.1. Useful Architectural Structures

	Software Structure	Element Types	Relations	Useful For	Quality Attributes Affected
Module Structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Engineering subsets, engineering extensions	"Subsetability," extensibility
	Layers	Layer	Requires the correct presence of, uses the services of, provides abstraction to	Incremental development, implementing systems on top of "virtual machines"	Portability
	Class	Class, object	Is an instance of, shares access methods of	In object-oriented design systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}, generalizes, specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C Structures	Service	Service, ESB, registry, others	Runs concurrently with, may run concurrently with, excludes, precedes, etc.	Scheduling analysis, performance analysis	Interoperability, modifiability
	Concurrency	Processes, threads	Can run in parallel	Identifying locations where resource contention exists, or where threads may fork, join, be created, or be killed	Performance, availability
Allocation Structures	Deployment	Components, hardware elements	Allocated to, migrates to	Performance, availability, security analysis	Performance, security, availability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

Relating Structures to Each Other

Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the structures give different system perspectives, they are not independent. Elements of one structure will be related to elements of other structures, and we need to reason about these relations. For example, a module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures, reflecting its runtime alter ego. In general, mappings between structures are many to many.

[Figure 1.2](#) shows a very simple example of how two structures might relate to each other. The figure on the left shows a module decomposition view of a tiny client-server system. In this system, two modules must be implemented: The client software and the server software. The figure on the right shows a component-and-connector view of the same system. At runtime there are ten clients running and accessing the server. Thus, this little system has two modules and eleven components (and ten connectors).

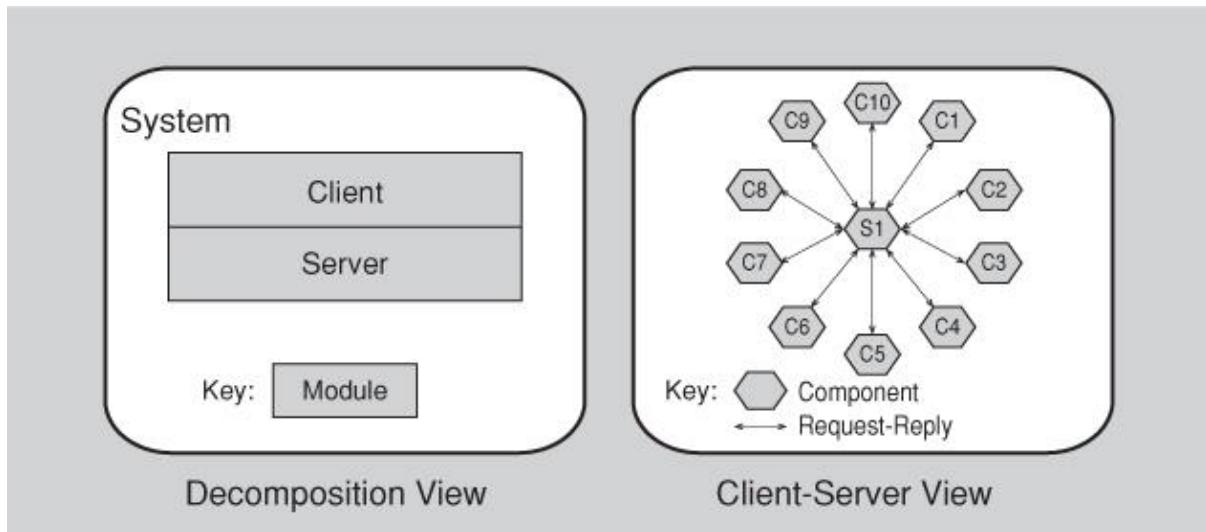


Figure 1.2. Two views of a client-server system

Whereas the correspondence between the elements in the decomposition structure and the client-server structure is obvious, these two views are used for very different things. For example, the view on the right could be used for performance analysis, bottleneck prediction, and network traffic management, which would be extremely difficult or impossible to do with the view on the left.

(In [Chapter 13](#) we'll learn about the map-reduce pattern, in which copies of simple, identical functionality are distributed across hundreds or thousands of processing nodes—one module for the whole system, but one component per node.)

Individual projects sometimes consider one structure dominant and cast other structures, when possible, in terms of the dominant structure. Often the dominant structure is the module decomposition structure. This is for a good reason: it tends to spawn the project structure, because it mirrors the team structure of development. In other projects, the dominant structure might be a C&C structure that shows how the system's functionality and/or critical quality attributes are achieved.

Fewer Is Better

Not all systems warrant consideration of many architectural structures. The larger the system, the more dramatic the difference between these structures tends to be; but for small systems we can often get by with fewer. Instead of working with each of several component-and-connector structures, usually a single one will do. If there is only one process, then the process structure collapses to a single node and need not be explicitly represented in the design. If there is to be no distribution (that is, if the system is implemented on a single processor), then the deployment structure is trivial and need not be considered further. In general, design and document a structure only if doing so brings a positive return on the investment, usually in terms of decreased development or maintenance costs.

Which Structures to Choose?

We have briefly described a number of useful architectural structures, and there are many more. Which ones shall an architect choose to work on? Which ones shall the architect choose to document? Surely not all of them. [Chapter 18](#) will treat this topic in more depth, but for now a good answer is that you should think about how the various structures available to you provide insight and leverage into the system's most important quality attributes, and then choose the ones that will play the best role in delivering those attributes.

Ask Cal

More than a decade ago I went to a customer site to do an architecture evaluation—one of the first instances of the Architecture Tradeoff Analysis Method (ATAM) that I had ever performed (you can read about the ATAM, and other architecture evaluation topics, in [Chapter 21](#)). In those early days, we were still figuring out how to make architecture evaluations repeatable and predictable, and how to guarantee useful outcomes from them. One of the ways that we ensured useful outcomes was to enforce certain preconditions on the evaluation. A precondition that we figured out rather quickly was this: if the architecture has not been documented, we will not proceed with the evaluation. The reason for this precondition was simple: we could not evaluate the architecture by reading the code—we didn't have the time for that—and we couldn't just ask the architect to sketch the architecture in real time, since that would produce vague and very likely erroneous representations.

Okay, it's not completely true to say that they had *no* architecture documentation. They did produce a single-page diagram, with a few boxes and lines. Some of those boxes were, however, clouds. Yes, they actually used a cloud as one of their icons. When I pressed them on the meaning of this icon—Was it a process? A class? A thread?—they waffled. This was not, in fact, architecture documentation. It was, at best, “marketecture.”

But in those early days we had no preconditions and so we didn't stop the evaluation there. We just blithely waded in to whatever swamp we found, and we enforced nothing. As I began this evaluation, I interviewed some of the key project stakeholders: the project manager and several of the architects (this was a large project with one lead architect and several subordinates). As it happens, the lead architect was away, and so I spent my time with the subordinate architects. Every time I asked the subordinates a tough question—“How do you ensure that you will meet your latency goal along this critical execution path?” or “What are your rules for layering?”—they would answer: “Ask Cal. Cal knows that.” Cal was the lead architect. Immediately I noted a risk for this system: What if Cal gets hit by a bus? What then?

In the end, because of my pestering, the architecture team did in fact produce respectable architecture documentation. About halfway through the evaluation, the project manager came up to me and shook my hand and thanked me for the great job I had done. I was dumbstruck. In my mind I hadn't done anything, at that point; the evaluation was only partially complete and I hadn't produced a single report or finding. I said that to the manager and he said: “You got those guys to document the architecture. I've never been able to get them to do that. So . . . thanks!”

If Cal had been hit by a bus or just left the company, they would have had a serious problem on their hands: all of that architectural knowledge located in one guy's head and he is no longer with the organization. It can happen. It *does* happen.

The moral of this story? An architecture that is not documented, and not communicated, may still be a good architecture, but the risks surrounding it are enormous.

—RK

1.3. Architectural Patterns

In some cases, architectural elements are composed in ways that solve particular problems. The compositions have been found useful over time, and over many different domains, and so they have been documented and disseminated. These compositions of architectural elements, called *architectural patterns*, provide packaged strategies for solving some of the problems facing a system.

An architectural pattern delineates the element types and their forms of interaction used in solving the problem. Patterns can be characterized according to the type of architectural elements they use. For example, a common module type pattern is this:

- *Layered pattern.* When the *uses* relation among software elements is strictly unidirectional, a system of layers emerges. A layer is a coherent set of related functionality. In a *strictly* layered structure, a layer can only use the services of the layer

immediately below it. Many variations of this pattern, lessening the structural restriction, occur in practice. Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.

Common component-and-connector type patterns are these:

- *Shared-data (or repository) pattern*. This pattern comprises components and connectors that create, store, and access persistent data. The repository usually takes the form of a (commercial) database. The connectors are protocols for managing the data, such as SQL.
- *Client-server pattern*. The components are the clients and the servers, and the connectors are protocols and messages they share among each other to carry out the system's work.

Common allocation patterns include the following:

- *Multi-tier pattern*, which describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium. This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- *Competence center and platform*, which are patterns that specialize a software system's work assignment structure. *Incompetence center*, 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. In *platform*, one site is tasked with developing reusable core assets of a software product line (see [Chapter 25](#)), and other sites develop applications that use the core assets.

Architectural patterns will be investigated much further in [Chapter 13](#).

1.4. What Makes a “Good” Architecture?

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose. A three-tier layered service-oriented architecture may be just the ticket for a large enterprise's web-based B2B system but completely wrong for an avionics application. An architecture carefully crafted to achieve high modifiability does not make sense for a throwaway prototype (and vice versa!). One of the messages of this book is that architectures can in fact be *evaluated*—one of the great benefits of paying attention to them—but only in the context of specific stated goals.

Nevertheless, there are rules of thumb that should be followed when designing most architectures. Failure to apply any of these does not automatically mean that the architecture will be fatally flawed, but it should at least serve as a warning sign that should be investigated.

We divide our observations into two clusters: process recommendations and product (or structural) recommendations. Our process recommendations are the following:

1. The architecture should be the product of a single architect or a small group of architects with an identified technical leader. This approach gives the architecture its conceptual integrity and technical consistency. This recommendation holds for Agile and open source projects as well as “traditional” ones. There should be a strong connection between the architect(s) and the development team, to avoid ivory tower designs that are impractical.
2. The architect (or architecture team) should, on an ongoing basis, base the architecture on a prioritized list of well-specified quality attribute requirements. These will inform the tradeoffs that always occur. Functionality matters less.
3. The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline. This might mean minimal documentation at first, elaborated later. Concerns usually are related to construction, analysis, and maintenance of the system, as well as education of new stakeholders about the system.
4. The architecture should be evaluated for its ability to deliver the system's important quality attributes. This should occur early in the life cycle, when it returns the most benefit, and repeated as appropriate, to ensure that changes to the architecture (or the environment for which it is intended) have not rendered the design obsolete.

5. The architecture should lend itself to incremental implementation, to avoid having to integrate everything at once (which almost never works) as well as to discover problems early. One way to do this is to create a “skeletal” system in which the communication paths are exercised but which at first has minimal functionality. This skeletal system can be used to “grow” the system incrementally, refactoring as necessary.

Our structural rules of thumb are as follows:

1. The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns. The information-hiding modules should encapsulate things likely to change, thus insulating the software from the effects of those changes. Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software that uses its facilities. These interfaces should allow their respective development teams to work largely independently of each other.
2. Unless your requirements are unprecedented—possible, but unlikely—your quality attributes should be achieved using well-known architectural patterns and tactics (described in [Chapter 13](#)) specific to each attribute.
3. The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.
4. Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are frequently confined to either the production or the consumption side of data. If new data is added, both sides will have to change, but the separation allows for a staged (incremental) upgrade.
5. Don’t expect a one-to-one correspondence between modules and components. For example, in systems with concurrency, there may be multiple instances of a component running in parallel, where each component is built from the same module. For systems with multiple threads of concurrency, each thread may use services from several components, each of which was built from a different module.
6. Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
7. The architecture should feature a small number of ways for components to interact. That is, the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.
8. The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained. For example, if network utilization is an area of concern, the architect should produce (and enforce) for each development team guidelines that will result in a minimum of network traffic. If performance is a concern, the architect should produce (and enforce) time budgets for the major threads.

1.5. Summary

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

A structure is a set of elements and the relations among them.

A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A view is a representation of one or more structures.

There are three categories of structures:

- Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.
- Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
- Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).

Structures represent the primary engineering leverage points of an architecture. Each structure brings with it the power to manipulate one or more quality attributes. They represent a powerful approach for creating the architecture (and later, for analyzing it and explaining it to its stakeholders). And as we will see in [Chapter 18](#), the structures that the architect has chosen as engineering leverage points are also the primary candidates to choose as the basis for architecture documentation.

Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.

1.6. For Further Reading

The early work of David Parnas laid much of the conceptual foundation for what became the study of software architecture. A quintessential Parnas reader would include his foundational article on information hiding [\[Parnas 72\]](#) as well as his works on program families [\[Parnas 76\]](#), the structures inherent in software systems [\[Parnas 74\]](#), and introduction of the uses structure to build subsets and supersets of systems [\[Parnas 79\]](#). All of these papers can be found in the more easily accessible collection of his important papers [\[Hoffman 00\]](#).

An early paper by Perry and Wolf [\[Perry 92\]](#) drew an analogy between software architecture views and structures and the structures one finds in a house (plumbing, electrical, and so forth).

Software architectural patterns have been extensively catalogued in the series *Pattern-Oriented Software Architecture* [\[Buschmann 96\]](#) and others. [Chapter 13](#) of this book also deals with architectural patterns.

Early papers on architectural views as used in industrial development projects are [\[Soni 95\]](#) and [\[Kruhnen 95\]](#). The former grew into a book [\[Hofmeister 00\]](#) that presents a comprehensive picture of using views in development and analysis. The latter grew into the Rational Unified Process, about which there is no shortage of references, both paper and online. A good one is [\[Kruhnen 03\]](#).

Cristina Gacek and her colleagues discuss the process issues surrounding software architecture in [\[Gacek 95\]](#).

Garlan and Shaw's seminal work on software architecture [\[Garlan 93\]](#) provides many excellent examples of architectural styles (a concept similar to patterns).

In [\[Clements 10a\]](#) you can find an extended discussion on the difference between an architectural pattern and an architectural style. (It argues that a pattern is a context-problem-solution triple; a style is simply a condensation that focuses most heavily on the solution part.)

See [\[Taylor 09\]](#) for a definition of software architecture based on decisions rather than on structure.

1.7. Discussion Questions

1. Software architecture is often compared to the architecture of buildings as a conceptual analogy. What are the strong points of that analogy? What is the correspondence in buildings to software architecture structures and views? To patterns? What are the weaknesses of the analogy? When does it break down?
2. Do the architectures you've been exposed to document different structures and relations like those described in this chapter? If so, which ones? If not, why not?
3. Is there a different definition of software architecture that you are familiar with? If so, compare and contrast it with the definition given in this chapter. Many definitions include considerations like "rationale" (stating the reasons why the architecture is what it is) or how the architecture will evolve over time. Do you agree or disagree that these considerations should be part of the definition of software architecture?
4. Discuss how an architecture serves as a basis for analysis. What about decision-making? What kinds of decision-making does an architecture empower?
5. What is architecture's role in project risk reduction?

- 6.** Find a commonly accepted definition of *system architecture* and discuss what it has in common with software architecture. Do the same for *enterprise architecture*.
- 7.** Find a published example of an architecture. What structure or structures are shown? Given its purpose, what structure or structures *should* have been shown? What analysis does the architecture support? Critique it: What questions do you have that the representation does not answer?
- 8.** Sailing ships have architectures, which means they have “structures” that lend themselves to reasoning about the ship’s performance and other quality attributes. Look up the technical definitions for *barque*, *brig*, *cutter*, *frigate*, *ketch*, *schooner*, and *sloop*. Propose a useful set of “structures” for distinguishing and reasoning about ship architectures.

2. Why Is Software Architecture Important?

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

—Eoin Woods

If architecture is the answer, what was the question?

While [Chapter 3](#) will cover the business importance of architecture to an enterprise, this chapter focuses on why architecture matters from a technical perspective. We will examine a baker's dozen of the most important reasons.

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Even if you already believe us that architecture is important and don't need the point hammered thirteen more times, think of these thirteen points (which form the outline for this chapter) as thirteen useful ways to use architecture in a project.

2.1. Inhibiting or Enabling a System's Quality Attributes

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.

This is such an important message that we've devoted all of [Part 2](#) of this book to expounding that message in detail. Until then, keep these examples in mind as a starting point:

- If your system requires high performance, then you need to pay attention to managing the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
- If modifiability is important, then you need to pay careful attention to assigning responsibilities to elements so that the majority of changes to the system will affect a small number of those elements. (Ideally each change will affect just a single element.)
- If your system must be highly secure, then you need to manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism) into the architecture.
- If you believe that scalability will be important to the success of your system, then you need to carefully localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hard-coding in resource assumptions or limits.
- If your projects need the ability to deliver incremental subsets of the system, then you must carefully manage intercomponent usage.
- If you want the elements from your system to be reusable in other systems, then you need to restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment to be useful.

The strategies for these and other quality attributes are supremely architectural. But an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an adequate architectural design. As we like to say (*mostly* in jest): The architecture giveth and the implementation taketh away. Decisions at all stages of the life cycle—from architectural design to coding and implementation—affect system quality. Therefore, quality is not completely a function of an architectural design.

A good architecture is necessary, but not sufficient, to ensure quality. Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct.

For example, modifiability is determined by how functionality is divided and coupled (architectural) and by coding techniques within a module (nonarchitectural). Thus, a system is typically modifiable if changes involve the fewest possible number of distinct elements. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure, tangled code.

2.2. Reasoning About and Managing Change

This point is a corollary to the previous point.

Modifiability—the ease with which changes can be made to a system—is a quality attribute (and hence covered by the arguments in the previous section), but it is such an important quality that we have awarded it its own spot in the List of Thirteen. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's total cost occurs *after* initial deployment. A corollary of this statistic is that most systems that people work on are in this phase. Many programmers

and software designers *never* get to work on new development; they work under the constraints of the existing architecture and the existing body of code. Virtually all software systems change over their lifetime, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But these changes are often fraught with difficulty.

Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

- A local change can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.
- A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact. For example, adding a new business rule to a pricing logic module, then adding new fields to the database that this new business rule requires, and then revealing the results of the rule in the user interface.
- An architectural change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system. For example, changing a system from client-server to peer-to-peer.

Obviously, local changes are the most desirable, and so an *effective* architecture is one in which the most common changes are local, and hence easy to make.

Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, performance, and behaviors of system software elements. These activities are in the job description for an architect. Reasoning about the architecture and analyzing the architecture can provide the insight necessary to make decisions about anticipated changes.

2.3. Predicting System Qualities

This point follows from the previous two. Architecture not only imbues systems with qualities, but it does so in a predictable way.

Were it not possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed, then choosing an architecture would be a hopeless task—randomly making architecture selections would perform as well as any other method. Fortunately, it *is* possible to make quality predictions about a system based solely on an evaluation of its architecture. If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, then we can make those decisions and rightly expect to be rewarded with the associated quality attributes. After the fact, when we examine an architecture, we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.

This is no different from any mature engineering discipline, where design analysis is a standard part of the development process. The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Even if you don't do the quantitative analytic modeling sometimes necessary to ensure that an architecture will deliver its prescribed benefits, this principle of evaluating

decisions based on their quality attribute implications is invaluable for at least spotting potential trouble spots early.

The architecture modeling and analysis techniques described in [Chapter 14](#), as well as the architecture evaluation techniques covered in [Chapter 21](#), allow early insight into the software product qualities made possible by software architectures.

2.4. Enhancing Communication among Stakeholders

Software architecture represents a common abstraction of a system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other. The architecture—or at least parts of it—is sufficiently abstract that most nontechnical people can understand it adequately, particularly with some coaching from the architect, and yet that abstraction can be refined into sufficiently rich technical specifications to guide implementation, integration, test, and deployment.

Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different characteristics of the system that are affected by its architecture. For example:

- The user is concerned that the system is fast, reliable, and available when needed.
- The customer is concerned that the architecture can be implemented on schedule and according to budget.
- The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
- The architect is worried about strategies to achieve all of those goals.

Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness. Architectural analysis, as we will see in [Chapter 21](#), both depends on this level of communication and enhances it.

[Section 3.5](#) covers stakeholders and their concerns in greater depth.

“What Happens When I Push This Button?” Architecture as a Vehicle for Stakeholder Communication

The project review droned on and on. The government-sponsored development was behind schedule and over budget and was large enough that these lapses were attracting congressional attention. And now the government was making up for past neglect by holding a marathon come-one-come-all review session. The contractor had recently undergone a buyout, which hadn't helped matters. It was the afternoon of the second day, and the agenda called for the software architecture to be presented. The young architect—an apprentice to the chief architect for the system—was bravely explaining how the software architecture for the massive system would enable it to meet its very demanding real-time,

distributed, high-reliability requirements. He had a solid presentation and a solid architecture to present. It was sound and sensible. But the audience—about 30 government representatives who had varying roles in the management and oversight of this sticky project—was tired. Some of them were even thinking that perhaps they should have gone into real estate instead of enduring another one of these marathon let's-finally-get-it-right-this-time reviews.

The viewgraph showed, in semiformal box-and-line notation, what the major software elements were in a runtime view of the system. The names were all acronyms, suggesting no semantic meaning without explanation, which the young architect gave. The lines showed data flow, message passing, and process synchronization. The elements were internally redundant, the architect was explaining. “In the event of a failure,” he began, using a laser pointer to denote one of the lines, “a restart mechanism triggers along this path when—”

“What happens when the mode select button is pushed?” interrupted one of the audience members. He was a government attendee representing the user community for this system.

“Beg your pardon?” asked the architect.

“The mode select button,” he said. “What happens when you push it?”

“Um, that triggers an event in the device driver, up here,” began the architect, laser-pointing. “It then reads the register and interprets the event code. If it’s mode select, well, then, it signals the blackboard, which in turns signals the objects that have subscribed to that event. . . .”

“No, I mean what does the system *do*,” interrupted the questioner. “Does it reset the displays? And what happens if this occurs during a system reconfiguration?”

The architect looked a little surprised and flicked off the laser pointer. This was not an architectural question, but since he was an architect and therefore fluent in the requirements, he knew the answer. “If the command line is in setup mode, the displays will reset,” he said. “Otherwise an error message will be put on the control console, but the signal will be ignored.” He put the laser pointer back on. “Now, the restart mechanism that I was talking about—”

“Well, I was just wondering,” said the users’ delegate. “Because I see from your chart that the display console is sending signal traffic to the target location module.”

“What *should* happen?” asked another member of the audience, addressing the first questioner. “Do you really want the user to get mode data during its reconfiguring?” And for the next 45 minutes, the architect watched as the audience consumed his time slot by debating what the correct behavior of the system was supposed to be in various esoteric states.

The debate was not architectural, but the architecture (and the graphical rendition of it) had sparked debate. It is natural to think of architecture as the basis for communication among some of the stakeholders besides the architects and developers: Managers, for example, use the architecture to create teams and allocate resources among them. But users? The architecture is invisible to users, after all; why should they latch on to it as a tool for understanding the system?

The fact is that they do. In this case, the questioner had sat through two days of viewgraphs all about function, operation, user interface, and testing. But it was the first slide on architecture that—even though he was tired and wanted to go

home—made him realize he didn’t understand something. Attendance at many architecture reviews has convinced me that seeing the system in a new way prods the mind and brings new questions to the surface. For users, architecture often serves as that new way, and the questions that a user poses will be behavioral in nature. In a memorable architecture evaluation exercise a few years ago, the user representatives were much more interested in what the system was going to do than in how it was going to do it, and naturally so. Up until that point, their only contact with the vendor had been through its marketers. The architect was the first legitimate expert on the system to whom they had access, and they didn’t hesitate to seize the moment.

Of course, careful and thorough requirements specifications would ameliorate this situation, but for a variety of reasons they are not always created or available. In their absence, a specification of the architecture often serves to trigger questions and improve clarity. It is probably more prudent to recognize this reality than to resist it.

Sometimes such an exercise will reveal unreasonable requirements, whose utility can then be revisited. A review of this type that emphasizes synergy between requirements and architecture would have let the young architect in our story off the hook by giving him a place in the overall review session to address that kind of information. And the user representative wouldn’t have felt like a fish out of water, asking his question at a clearly inappropriate moment.

—PCC

2.5. Carrying Early Design Decisions

Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system’s remaining development, its deployment, and its maintenance life. It is also the earliest point at which these important design decisions affecting the system can be scrutinized.

Any design, in any discipline, can be viewed as a set of decisions. When painting a picture, an artist decides on the material for the canvas, on the media for recording—oil paint, watercolor, crayon—even before the picture is begun. Once the picture is begun, other decisions are immediately made: Where is the first line? What is its thickness? What is its shape? All of these early design decisions have a strong influence on the final appearance of the picture. Each decision constrains the many decisions that follow. Each decision, in isolation, might appear innocent enough, but the early ones in particular have disproportionate weight simply because they influence and constrain so much of what follows.

So it is with architecture design. An architecture design can also be viewed as a set of decisions. The early design decisions constrain the decisions that follow, and changing these decisions has enormous ramifications. Changing these early decisions will cause a ripple effect, in terms of the additional decisions that must now be changed. Yes, sometimes the architecture must be refactored or redesigned, but this is not a task we undertake lightly (because the “ripple” might turn into a tsunami).

What are these early design decisions embodied by software architecture? Consider:

- Will the system run on one processor or be distributed across multiple processors?
- Will the software be layered? If so, how many layers will there be? What will each one do?
- Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?
- Will the system depend on specific features of the operating system or hardware?
- Will the information that flows through the system be encrypted or not?
- What operating system will we use?
- What communication protocol will we choose?

Imagine the nightmare of having to change any of these or a myriad other related decisions. Decisions like these begin to flesh out some of the structures of the architecture and their interactions. In [Chapter 4](#), we

describe seven categories of these early design decisions. In [Chapters 5–11](#) we show the implications of these design decision categories on achieving quality attributes.

2.6. Defining Constraints on an Implementation

An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture. This means that the implementation must be implemented as the set of prescribed elements, these elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the other elements as dictated by the architecture. Each of these prescriptions is a constraint on the implementer.

Element builders must be fluent in the specifications of their individual elements, but they may not be aware of the architectural tradeoffs—the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs. A classic example of this phenomenon is when an architect assigns performance budget to the pieces of software involved in some larger piece of functionality. If each software unit stays within its budget, the overall transaction will meet its performance requirement. Implementers of each of the constituent pieces may not know the overall budget, only their own.

Conversely, the architects need not be experts in all aspects of algorithm design or the intricacies of the programming language—although they should certainly know enough not to design something that is difficult to build—but they are the ones responsible for establishing, analyzing, and enforcing the architectural tradeoffs.

2.7. Influencing the Organizational Structure

Not only does architecture prescribe the structure of the system being developed, but that structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization). The normal method for dividing up the labor in a large project is to assign different groups different portions of the system to construct. This is called the work-breakdown structure of a system. Because the architecture includes the broadest decomposition of the system, it is typically used as the basis for the work-breakdown structure. The work-breakdown structure in turn dictates units of planning, scheduling, and budget; interteam communication channels; configuration control and file-system organization; integration and test plans and procedures; and even project minutiae such as how the project intranet is organized and who sits with whom at the company picnic. Teams communicate with each other in terms of the interface specifications for the major elements. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural elements from the architecture: the database, the business rules, the user interface, the device drivers, and so forth.

A side effect of establishing the work-breakdown structure is to freeze some aspects of the software architecture. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Thus, once the architecture has been agreed on, it becomes very costly—for managerial and business reasons—to significantly modify it. This is one argument (among many) for carrying out extensive analysis before settling on the software architecture for a large system—because so much depends on it.

2.8. Enabling Evolutionary Prototyping

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. A skeletal system is one in which at least some of the infrastructure—how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth—is built before much of the system’s functionality has been created. (The two can go hand in hand: build a little infrastructure to support a little end-to-end functionality; repeat until done.)

For example, systems built as plug-in architectures are skeletal systems: the plug-ins provide the actual functionality. This approach aids the development process because the system is executable early in the product’s life cycle. The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software. In some cases the prototype parts can be low-fidelity versions of the final functionality, or they can be *surrogates* that consume and produce data at the appropriate rates but do little else. Among other things, this approach allows potential performance problems to be identified early in the product’s life cycle.

These benefits reduce the potential risk in the project. Furthermore, if the architecture is part of a family of related systems, the cost of creating a framework for prototyping can be distributed over the development of many systems.

2.9. Improving Cost and Schedule Estimates

Cost and schedule estimates are important tools for the project manager both to acquire the necessary resources and to monitor progress on the project, to know if and when a project is in trouble. One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle. Although top-down estimates are useful for setting goals and apportioning budgets, cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.

As we have said, the organizational and work-breakdown structure of a project is almost always based on its architecture. Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true. But the best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers). The discussion and negotiation that results from this process creates a far more accurate estimate than either approach by itself.

It helps if the requirements for a system have been reviewed and validated. The more up-front knowledge you have about the scope, the more accurate the cost and schedule estimates will be.

[Chapter 22](#) delves into the use of architecture in project management.

2.10. Supplying a Transferable, Reusable Model

The earlier in the life cycle that reuse is applied, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse of architectures provides tremendous leverage for systems with similar requirements. Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we just described are also transferred.

A software product line or family is a set of software systems that are all built using the same set of reusable assets. Chief among these assets is the architecture that was designed to handle the needs of the entire family. Product-line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line. The architecture defines what is fixed for all members of the product line and what is variable. Software product lines represent a powerful approach to multi-system development that is showing order-of-magnitude payoffs in time to market, cost, productivity, and product quality. The power of architecture lies at the heart of the paradigm. Similar to other capital investments, the architecture for a product line becomes a developing organization's core asset. Software product lines are explained in [Chapter 25](#).

2.11. Allowing Incorporation of Independently Developed Components

Whereas earlier software paradigms have focused on *programming* as the prime activity, with progress measured in lines of code, architecture-based development often focuses on *composing* or *assembling elements* that are likely to have been developed separately, even independently, from each other. This composition is possible because the architecture defines the elements that can be incorporated into the system. The architecture constrains possible replacements (or additions) according to how they interact with their environment, how they receive and relinquish control, what data they consume and produce, how they access data, and what protocols they use for communication and resource sharing.

In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, signaled the dawn of the industrial age. In the days before physical measurements were reliable, manufacturing interchangeable parts was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting and just as significant.

Commercial off-the-shelf components, open source software, publicly available apps, and networked services are all modern-day software instantiations of Whitney's basic idea. Whitney's musket parts had "interfaces" (having to do with fit and durability) and so do today's interchangeable software components.

For software, the payoff can be

- Decreased time to market (it should be easier to use someone else's ready solution than build your own)
- Increased reliability (widely used software should have its bugs ironed out already)
- Lower cost (the software supplier can amortize development cost across their customer base)

- Flexibility (if the component you want to buy is not terribly special-purpose, it's likely to be available from several sources, thus increasing your buying leverage)

2.12. Restricting the Vocabulary of Design Alternatives

As useful architectural patterns are collected, it becomes clear that although software elements can be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions. By doing so we minimize the design complexity of the system we are building.

A software engineer is not an *artiste*, whose creativity and freedom are paramount. Engineering is about discipline, and discipline comes in part by *restricting* the vocabulary of alternatives to proven solutions.

Advantages of this approach include enhanced reuse, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability.

Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number.

Properties of software design follow from the choice of an architectural pattern. Those patterns that are more desirable for a particular problem should improve the implementation of the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insight into poorly understood design contexts, or by helping to surface inconsistencies in requirements. We will discuss architectural patterns in more detail in [Chapter 13](#).

2.13. Providing a Basis for Training

The architecture, including a description of how the elements interact with each other to carry out the required behavior, can serve as the first introduction to the system for new project members. This reinforces our point that one of the important uses of software architecture is to support and encourage communication among the various stakeholders. The architecture is a common reference point.

Module views are excellent for showing someone the structure of a project: Who does what, which teams are assigned to which parts of the system, and so forth. Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

We will discuss these views in more detail in [Chapter 18](#).

2.14. Summary

Software architecture is important for a wide variety of technical and nontechnical reasons. Our list includes the following:

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. An architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training of a new team member.

2.15. For Further Reading

Rebecca Grinter has observed architects from a sociological standpoint. In [\[Grinter 99\]](#) she argues eloquently that the architect's primary role is to facilitate stakeholder communication. The way she puts it is that architects enable communication among parties who would otherwise not be able to talk to each other.

The granddaddy of papers about architecture and organization is [\[Conway 68\]](#). Conway's law states that "organizations which design systems. . . are constrained to produce designs which are copies of the communication structures of these organizations."

There is much about software development through composition that remains unresolved. When the components that are candidates for importation and reuse are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications can increase the effort required to integrate their functions. David Garlan and his colleagues coined the term *architectural mismatch* to describe this situation, and their paper on it is worth reading [\[Garlan 95\]](#).

Paulish [\[Paulish 02\]](#) discusses architecture-based project management, and in particular the ways in which an architecture can help in the estimation of project cost and schedule.

2.16. Discussion Questions

1. For each of the thirteen reasons articulated in this chapter why architecture is important, take the contrarian position: Propose a set of circumstances under which architecture is not necessary to achieve the result indicated. Justify your position. (Try to come up with different circumstances for each of the thirteen.)
2. This chapter argues that architecture brings a number of tangible benefits. How would you measure the benefits, on a particular project, of each of the thirteen points?
3. Suppose you want to introduce architecture-centric practices to your organization. Your management is open to the idea, but wants to know the ROI for doing so. How would you respond?
4. Prioritize the list of thirteen points in this chapter according to some criteria meaningful to you. Justify your answer. Or, if you could choose only two or three of the reasons to promote the use of architecture in a project, which would you choose and why?

3. The Many Contexts of Software Architecture

People in London think of London as the center of the world, whereas New Yorkers think the world ends three miles outside of Manhattan.

—Toby Young

In 1976, a *New Yorker* magazine cover featured a cartoon by Saul Steinberg showing a New Yorker's view of the world. You've probably seen it; if not, you can easily find it online. Looking to the west from 9th Avenue in Manhattan, the illustration shows 10th Avenue, then the wide Hudson River, then a thin strip of completely nondescript land called "Jersey," followed by a somewhat thicker strip of land representing the entire rest of the United States. The mostly empty United States has a cartoon mountain or two here and there and a few cities haphazardly placed "out there," and is flanked by featureless "Canada" on the right and "Mexico" on the left. Beyond is the Pacific Ocean, only slightly wider than the Hudson, and beyond that lie tiny amorphous shapes for Japan and China and Russia, and that's pretty much the world from a New Yorker's perspective.

In a book about architecture, it is tempting to view architecture in the same way, as the most important piece of the software universe. And in some chapters, we unapologetically will do exactly that. But in this chapter we put software architecture in its place, showing how it supports and is informed by other critical forces and activities in the various contexts in which it plays a role.

These contexts, around which we structured this book, are as follows:

- *Technical*. What technical role does the software architecture play in the system or systems of which it's a part?
- *Project life cycle*. How does a software architecture relate to the other phases of a software development life cycle?
- *Business*. How does the presence of a software architecture affect an organization's business environment?
- *Professional*. What is the role of a software architect in an organization or a development project?

These contexts all play out throughout the book, but this chapter introduces each one. Although the contexts are unchanging, the specifics for your system may change over time. One challenge for the architect is to envision what in their context might change and to adopt mechanisms to protect the system and its development if the envisioned changes come to pass.

3.1. Architecture in a Technical Context

Architectures inhibit or enable the achievement of quality attributes, and one use of an architecture is to support reasoning about the consequences of change in the particular quality attributes important for a system at its inception.

Architectures Inhibit or Enable the Achievement of Quality Attributes

[Chapter 2](#) listed thirteen reasons why software architecture is important and merits study. Several of those reasons deal with exigencies that go beyond the bounds of a particular development project (such as communication among stakeholders, many of whom may reside outside the project's organization). Others deal with nontechnical aspects of a project (such as the architecture's influence on a project's team structure, or its contribution to accurate budget and schedule estimation). The first three reasons in that List of Thirteen deal specifically with an architecture's technical impact on every system that uses it:

1. An architecture will inhibit or enable the achievement of a system's quality attributes.
2. You can predict many aspects of a system's qualities by studying its architecture.
3. An architecture makes it easier for you to reason about and manage change.

These are all about the architecture's effect on a system's quality attributes, although the first one states it the most explicitly. While all of the reasons enumerated in [Chapter 2](#) are valid statements of the contribution of architecture, probably the most important reason that it warrants attention is its critical effect on quality attributes.

This is such a critical point that, with your indulgence, we'll add a few more points to the bullet list that we gave in [Section 2.1](#). Remember? The one that started like this:

- If your system requires high performance, then you need to pay attention to managing the time-based behavior of elements, their use of shared resources, and the frequency and volume of interelement communication.

To that list, we'll add the following:

- If you care about a system's availability, you *have* to be concerned with how components take over for each other in the event of a failure, and how the system responds to a fault.
- If you care about usability, you *have* to be concerned about isolating the details of the user interface and those elements responsible for the user experience from the rest of the system, so that those things can be tailored and improved over time.
- If you care about the testability of your system, you *have* to be concerned about the testability of individual elements, which means making their state observable and controllable, plus understanding the emergent behavior of the elements working together.
- If you care about the safety of your system, you *have* to be concerned about the behavioral envelope of the elements and the emergent behavior of the elements working in concert.
- If you care about interoperability between your system and another, you *have* to be concerned about which elements are responsible for external interactions so that you can control those interactions.

These and other representations are all saying the same thing in different ways: If you care about *this* quality attribute, you have to be concerned with *these* decisions, all of which are thoroughly architectural in nature. An architecture inhibits or enables a system's quality attributes. And conversely, nothing else influences an architecture more than the quality attribute requirements it must satisfy.

If you care about architecture for no other reason, you should care about it for this one. We feel so strongly about architecture's importance with respect to achieving system quality attributes that all of [Part II](#) of this book is devoted to the topic.

Why is functionality missing from the preceding list? It is missing because the architecture mainly provides containers into which the architect places functionality. Functionality is not so much a driver for the architecture as it is a consequence of it. We return to this point in more detail in [Part II](#).

Architectures and the Technical Environment

The technical environment that is current when an architecture is designed will influence that architecture. It might include standard industry practices or software engineering techniques prevalent in the architect's professional community. It is a brave architect who, in today's environment, does not at least consider a web-based, object-oriented, service-oriented, mobility-aware, cloud-based, social-networking-friendly design for an information system. It wasn't always so, and it won't be so ten years from now when another crop of technological trends has come to the fore.

The Swedish Ship Vasa

In the 1620s, Sweden and Poland were at war. The king of Sweden, Gustavus Adolphus, was determined to put a swift and favorable end to it and commissioned a new warship the likes of which had never been seen before. The *Vasa*, shown in [Figure 3.1](#), was to be the world's most formidable instrument of war: 70 meters long, able to carry 300 soldiers, and with an astonishing 64 heavy guns mounted on two gun decks. Seeking to add overwhelming firepower to his navy to strike a decisive blow, the king insisted on stretching the *Vasa*'s armaments to the limits. Her architect, Henrik Hybertsson, was a seasoned Dutch shipbuilder with an impeccable reputation, but the *Vasa* was beyond even his broad experience. Two-gun-deck ships were rare, and none had been built of the *Vasa*'s size and armament.

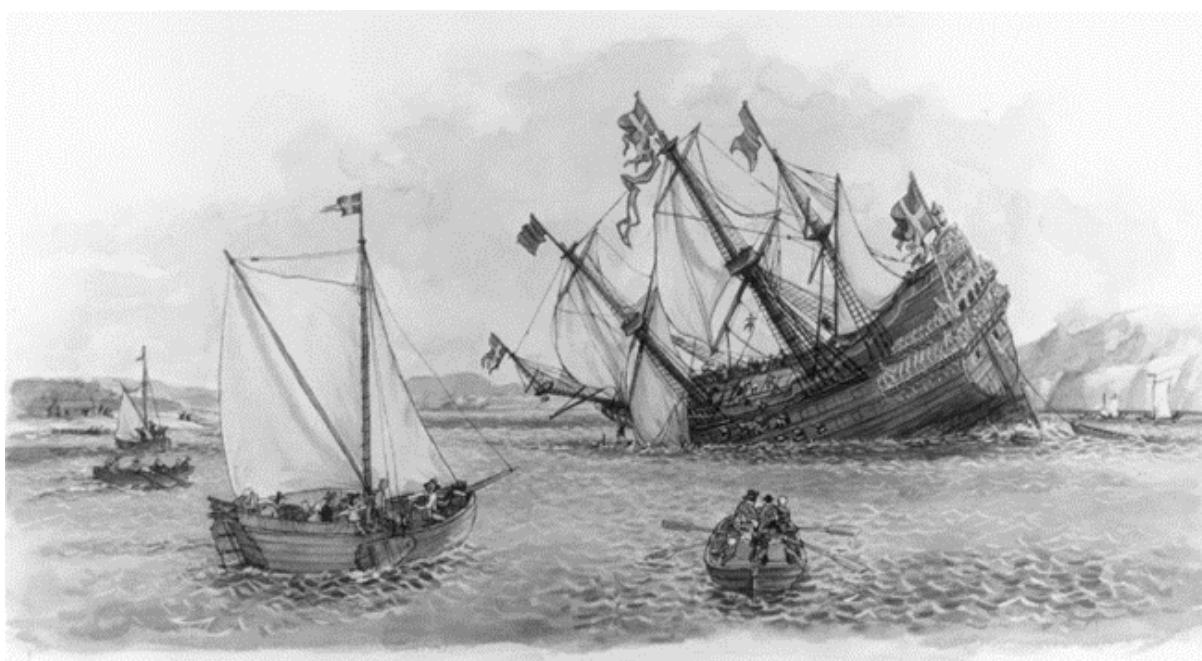


Figure 3.1. The warship. Used with permission of The Vasa Museum, Stockholm, Sweden.

Like all architects of systems that push the envelope of experience, Hybertsson had to balance many concerns. Swift time to deployment was critical, but so were performance, functionality, safety, reliability, and cost. He was also responsible to a variety of stakeholders. In this case, the primary customer was the king, but Hybertsson also was responsible to the crew that would sail his creation. Also like all architects, Hybertsson brought his experience with him to the task. In this case, his experience told him to design the *Vasa* as though it were a single-gun-deck ship and then extrapolate, which was in accordance with the technical environment of the day. Faced with an impossible task, Hybertsson had the good sense to die about a year before the ship was finished.

The project was completed to his specifications, however, and on Sunday morning, August 10, 1628, the mighty ship was ready. She set her sails, waddled out into Stockholm's deep-water harbor, fired her guns in salute, and promptly rolled over. Water

poured in through the open gun ports, and the *Vasa* plummeted. A few minutes later her first and only voyage ended 30 meters beneath the surface. Dozens among her 150-man crew drowned.

Inquiries followed, which concluded that the ship was well built but “badly proportioned.” In other words, its architecture was flawed. Today we know that Hybertsson did a poor job of balancing all of the conflicting constraints levied on him. In particular, he did a poor job of risk management and a poor job of customer management (not that anyone could have fared better). He simply acquiesced in the face of impossible requirements.

The story of the *Vasa*, although more than 375 years old, well illustrates the Architecture Influence Cycle: organization goals beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect’s experience and the technical environment of the day. Hybertsson suffered from the fact that neither of those were up to the task before him.

In this book, we provide three things that Hybertsson could have used:

1. Examples of successful architectural practices that work under demanding requirements, so as to help set the technical playing field of the day.
2. Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs.
3. Techniques for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

Our goal is to give architects another way out of their design dilemmas than the one that befell the ill-fated Dutch ship designer. Death before deployment is not nearly so admired these days.

—PCC

3.2. Architecture in a Project Life-Cycle Context

Software development processes are standard approaches for developing software systems. They impose a discipline on software engineers and, more important, teams of software engineers. They tell the members of the team what to do next. There are four dominant software development processes, which we describe in roughly the order in which they came to prominence:

1. *Waterfall*. For many years the Waterfall model dominated the field of software development. The Waterfall model organized the life cycle into a series of connected sequential activities, each with entry and exit conditions and a formalized relationship with its upstream and downstream neighbors. The process began with requirements specification, followed by design, then implementation, then integration, then testing, then installation, all followed by maintenance. Feedback paths from later to earlier steps allowed for the revision of artifacts (requirements documents, design documents, etc.) on an as-needed basis, based on the knowledge acquired in the later stage. For example, designers might push back against overly stringent requirements, which would then be reworked and flow back down. Testing that uncovered defects would trigger reimplementation (and maybe even redesign). And then the cycle continued.
2. *Iterative*. Over time the feedback paths of the Waterfall model became so pronounced that it became clear that it was better to think of software development as a series of short cycles through the steps—*some* requirements lead to *some* design, which can be implemented and tested while the next cycle’s worth of requirements are being captured and designed. These cycles are called iterations, in the sense of iterating toward the ultimate software solution for the given problem. Each iteration should deliver something working and useful. The trick here is to uncover early those requirements that have the most far-reaching effect on the design; the corresponding danger is to overlook requirements that, when discovered later, will capsize the design decisions made so far. An especially well-known iterative process is called the Unified Process (originally named the Rational Unified Process, after Rational Software, which originated it). It defines four phases of each iteration: inception, elaboration,

construction, and transition. A set of chosen use cases defines the goals for each iteration, and the iterations are ordered to address the greatest risks first.

3. *Agile*. The term “Agile software development” refers to a group of software development methodologies, the best known of which include Scrum, Extreme Programming, and Crystal Clear. These methodologies are all incremental and iterative. As such, one can consider some iterative methodologies as Agile. What distinguishes Agile practices is early and frequent delivery of working software, close collaboration between developers and customers, self-organizing teams, and a focus on adaptation to changing circumstances (such as late-arriving requirements). All Agile methodologies focus on teamwork, adaptability, and close collaboration (both within the team and between team members and customers/end users). These methodologies typically eschew substantial up-front work, on the assumption that requirements *always* change, and they continue to change throughout the project’s life cycle. As such, it might seem that Agile methodologies and architecture cannot happily coexist. As we will show in [Chapter 15](#), this is not so.
4. *Model-driven development*. Model-driven development is based on the idea that humans should not be writing code in programming languages, but they should be creating models of the domain, from which code is automatically generated. Humans create a platform-independent model (PIM), which is combined with a platform-definition model (PDM) to generate running code. In this way the PIM is a pure realization of the functional requirements while the PDM addresses platform specifics and quality attributes.

All of these processes include design among their obligations, and because architecture is a special kind of design, architecture finds a home in each one. Changing from one development process to another in the middle of a project requires the architect to save useful information from the old process and determine how to integrate it into the new process.

No matter what software development process or life-cycle model you’re using, there are a number of activities that are involved in creating a software architecture, using that architecture to realize a complete design, and then implementing or managing the evolution of a target system or application. The process you use will determine how often and when you revisit and elaborate each of these activities. These activities include:

1. Making a business case for the system
2. Understanding the architecturally significant requirements
3. Creating or selecting the architecture
4. Documenting and communicating the architecture
5. Analyzing or evaluating the architecture
6. Implementing and testing the system based on the architecture
7. Ensuring that the implementation conforms to the architecture

Each of these activities is covered in a chapter in [Part III](#) of this book, and described briefly below.

Making a Business Case for the System

A business case is, briefly, a justification of an organizational investment. It is a tool that helps you make business decisions by predicting how they will affect your organization. Initially, the decision will be a go/no-go for pursuing a new business opportunity or approach. After initiation, the business case is reviewed to assess the accuracy of initial estimates and then updated to examine new or alternative angles on the opportunity. By documenting the expected costs, benefits, and risks, the business case serves as a repository of the business and marketing data. In this role, management uses the business case to determine possible courses of action.

Knowing the business goals for the system—[Chapter 16](#) will show you how to elicit and capture them in a systematic way—is also critical in the creation of a business case for a system.

Creating a business case is broader than simply assessing the market need for a system. It is an important step in shaping and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market and lifetime? Will it need to interface with other systems? Are there system limitations that it must work within?

These are all questions about which the system's architects have specialized knowledge; they must contribute to the answers. These questions cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, the organization may be unable to achieve its business goals. Typically, a business case is created prior to the initiation of a project, but it also may be revisited during the course of the project for the organization to determine whether to continue making investments in the project. If the circumstances assumed in the initial version of the business case change, the architect may be called upon to establish how the system will change to reflect the new set of circumstances.

Understanding the Architecturally Significant Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, object-oriented analysis uses use cases and scenarios to embody requirements. Safety-critical systems sometimes use more rigorous approaches, such as finite-state-machine models or formal specification languages. In [Part II](#) of this book, which covers quality attributes, we introduce a collection of quality attribute scenarios that aid in the brainstorming, discussion, and capture of quality attribute requirements for a system.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Because it is a rare system these days that is not similar to other systems, requirements elicitation techniques involve understanding these prior systems' characteristics. We discuss the architectural implications of software product lines in [Chapter 25](#).

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model and explore desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly build support for the project and catalyze decisions on the system's design and the design of its user interface.

Creating or Selecting the Architecture

In the landmark book *The Mythical Man-Month*, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture. We firmly believe this as well. Good architecture almost never results as an emergent phenomenon.

[Chapters 5–12](#) and [17](#) will provide practical techniques that will aid you in creating an architecture to achieve its behavioral and quality requirements.

Documenting and Communicating the Architecture

For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments that the architecture requires of them, testers must understand the task structure that the architecture imposes on them, management must understand the scheduling implications it contains, and so forth.

Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds. Architectural documentation should also be *minimal* and aimed at the stakeholders who will use it; we are no fans of documentation for documentation's sake. We discuss the documentation of architectures and provide examples of good documentation practices in [Chapter 18](#). We will also discuss keeping the architecture up to date when there is a change in something on which the architecture documentation depends.

Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these competing designs in a rational way is one of the architect's greatest challenges.

Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Analysis techniques to

evaluate the quality attributes that an architecture imparts to a system have become much more widespread in the past decade. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture. The most mature methodological approach is found in the Architecture Tradeoff Analysis Method (ATAM) of [Chapter 21](#), while the economic implications of architectural decisions are explored in [Chapter 23](#).

Implementing and Testing the System Based on the Architecture

If the architect designs and analyzes a beautiful, conceptually sound architecture which the implementers then ignore, what was the point? If architecture is important enough to devote the time and effort of your best minds to, then it is just as important to keep the developers faithful to the structures and interaction protocols constrained by the architecture. Having an explicit and well-communicated architecture is the first step toward ensuring *architectural conformance*. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture (as opposed to just the code) is better.

There are many reasons why developers might not be faithful to the architecture: It might not have been properly documented and disseminated. It might be too confusing. It might be that the architect has not built ground-level support for the architecture (particularly if it presents a different way of “doing business” than the developers are used to), and so the developers resist it. Or the developers may sincerely want to implement the architecture but, being human, they occasionally slip up. This is not to say that the architecture should not change, but it should not change purely on the basis of the whims of the developers, because they may not have the overall picture.

Ensuring That the Implementation Conforms to the Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase. And when they do get significantly out of sync, effort must be expended to either fix the implementation or update the architectural documentation.

Although work in this area is still relatively immature, it has been an area of intense activity in recent years. [Chapter 20](#) will present the current state of recovering an architecture from an existing system and ensuring that it conforms to the specified architecture.

3.3. Architecture in a Business Context

Architectures and systems are not constructed frivolously. They serve some business purposes, although as mentioned before, these purposes may change over time.

Architectures and Business Goals

Systems are created to satisfy the business goals of one or more organizations. Development organizations want to make a profit, or capture market, or stay in business, or help their customers do their jobs better, or keep their staff gainfully employed, or make their stockholders happy, or a little bit of each. Customers have their own goals for acquiring a system, usually involving some aspect of making their lives easier or more productive. Other organizations involved in a project’s life cycle, such as subcontractors or government regulatory agencies, have their own goals dealing with the system.

Architects need to understand who the vested organizations are and what their goals are. Many of these goals will have a profound influence on the architecture.

Many business goals will be manifested as quality attribute requirements. In fact, every quality attribute—such as a user-visible response time or platform flexibility or ironclad security or any of a dozen other needs—should originate from some higher purpose that can be described in terms of added value. If we ask, for example, “*Why* do you want this system to have a really fast response time?” we might hear that this will differentiate the product from its competition and let the developing organization capture market share.

Some business goals, however, will not show up in the form of requirements. We know of one software architect who was informed by his manager that the architecture should include a database. The architect was perplexed, because the requirements for the system really didn't warrant a database and the architect's design had nicely avoided putting one in, thereby simplifying the design and lowering the cost of the product. The architect was perplexed, that is, until the manager reminded the architect that the company's database department was currently overstaffed and underworked. They needed something to do! The architect put in the database, and all was well. That kind of business goal—keeping staff gainfully employed—is not likely to show up in any requirements document, but if the architect had failed to meet it, the manager would have considered the architecture as unacceptable, just as the customer would have if it failed to provide a key piece of functionality.

Still other business goals have no effect on the architecture whatsoever. A business goal to lower costs might be realized by asking employees to work from home, or turn the office thermostats down in the winter, or using less paper in the printers. [Chapter 16](#) will deal with uncovering business goals and the requirements they lead to.

[Figure 3.2](#) illustrates the major points from the preceding discussion. In the figure, the arrows mean "leads to." The solid arrows highlight the relationships of most interest to us.

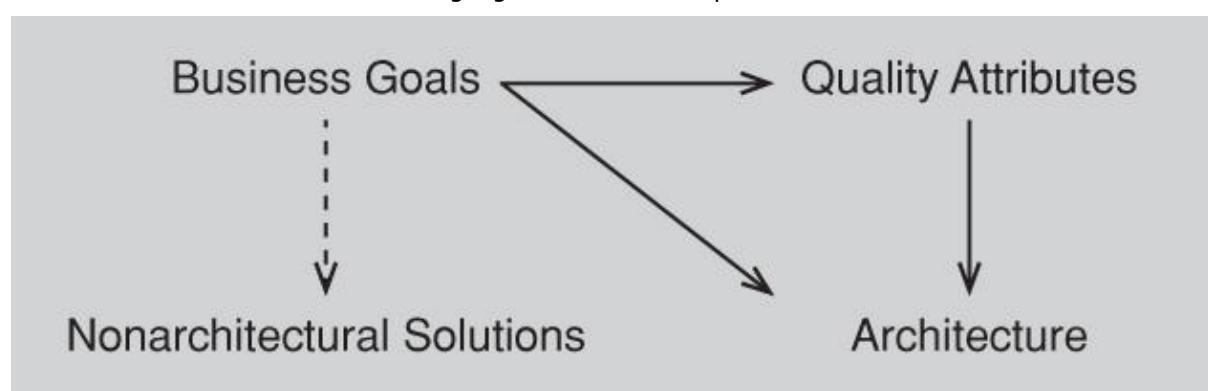


Figure 3.2. Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

Architectures and the Development Organization

A development organization contributes many of the business goals that influence an architecture. For example, if the organization has an abundance of experienced and idle programmers skilled in peer-to-peer communications, then a peer-to-peer architecture might be the approach supported by management. If not, it may well be rejected. This would support the business goal, perhaps left implicit, of not wanting to hire new staff or lay off existing staff, or not wanting to invest significantly in the retraining of existing staff.

More generally, an organization often has an investment in assets, such as existing architectures and the products based on them. The foundation of a development project may be that the proposed system is the next in a sequence of similar systems, and the cost estimates assume a high degree of asset reuse and a high degree of skill and productivity from the programmers.

Additionally, an organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may view the proposed system as one means of financing and extending that infrastructure. For example, an organization may decide that it wants to develop a reputation for supporting solutions based on cloud computing or service-oriented architecture or high-performance real-time computing. This long-term goal would be supported, in part, by infrastructural investments that will affect the developing organization: a cloud-computing group needs to be hired or grown, infrastructure needs to be purchased, or perhaps training needs to be planned.

Finally, the organizational structure can shape the software architecture, and vice versa. Organizations are often organized around technology and application concepts: a database group, a networking group, a business rules team, a user-interface group. So the explicit identification of

a distinct subsystem in the architecture will frequently lead to the creation of a group with the name of the subsystem. Furthermore, if the user-interface team frequently needs to communicate with the business rules team, these teams will need to either be co-located or they will need some regular means of communicating and coordinating.

3.4. Architecture in a Professional Context

What do architects do? How do you become an architect? In this section we talk about the many facets of being an architect that go beyond what you learned in a programming or software engineering course.

You probably know by now that architects need more than just technical skills. Architects need to explain to one stakeholder or another the chosen priorities of different properties, and why particular stakeholders are not having all of their expectations fulfilled. To be an effective architect, then, you will need diplomatic, negotiation, and communication skills.

You will perform many activities beyond directly producing an architecture. These activities, which we call *duties*, form the backbone of individual architecture competence. We surveyed the broad body of information aimed at architects (such as websites, courses, books, and position descriptions for architects), as well as practicing architects, and duties are but one aspect. Writers about architects also speak of skills and knowledge. For example, architects need the ability to communicate ideas clearly and need to have up-to-date knowledge about (for example) patterns, or database platforms, or web services standards.

Duties, skills, and knowledge form a triad on which architecture competence rests. You will need to be involved in supporting management and dealing with customers. You will need to manage a diverse workload and be able to switch contexts frequently. You will need to know business considerations. You will need to be a leader in the eyes of developers and management. In [Chapter 24](#) we examine at length the architectural competence of organizations and people.

Architects' Background and Experience

We are all products of our experiences, architects included. If you have had good results using a particular architectural approach, such as three-tier client-server or publish-subscribe, chances are that you will try that same approach on a new development effort. Conversely, if your experience with an approach was disastrous, you may be reluctant to try it again.

Architectural choices may also come from your education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well. You may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a training course.

Why do we mention this? Because you (and your organization) must be aware of this influence, so that you can manage it to the best of your abilities. This may mean that you will critically examine proposed architectural solutions, to ensure that they are not simply the path of least resistance. It may mean that you will take training courses in interesting new technologies. It may mean that you will invest in exploratory projects, to "test the water" of a new technology. Each of these steps is a way to proactively manage your background and experience.

3.5. Stakeholders

Many people and organizations are interested in a software system. We call these entities *stakeholders*. A stakeholder is anyone who has a stake in the success of the system: the customer, the end users, the developers, the project manager, the maintainers, and even those who market the system, for example. But stakeholders, despite all having a shared stake in the success of the system, typically have different specific concerns that they wish the system to guarantee or optimize. These concerns are as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions. [Figure 3.3](#) shows the architect receiving a few helpful stakeholder "suggestions."

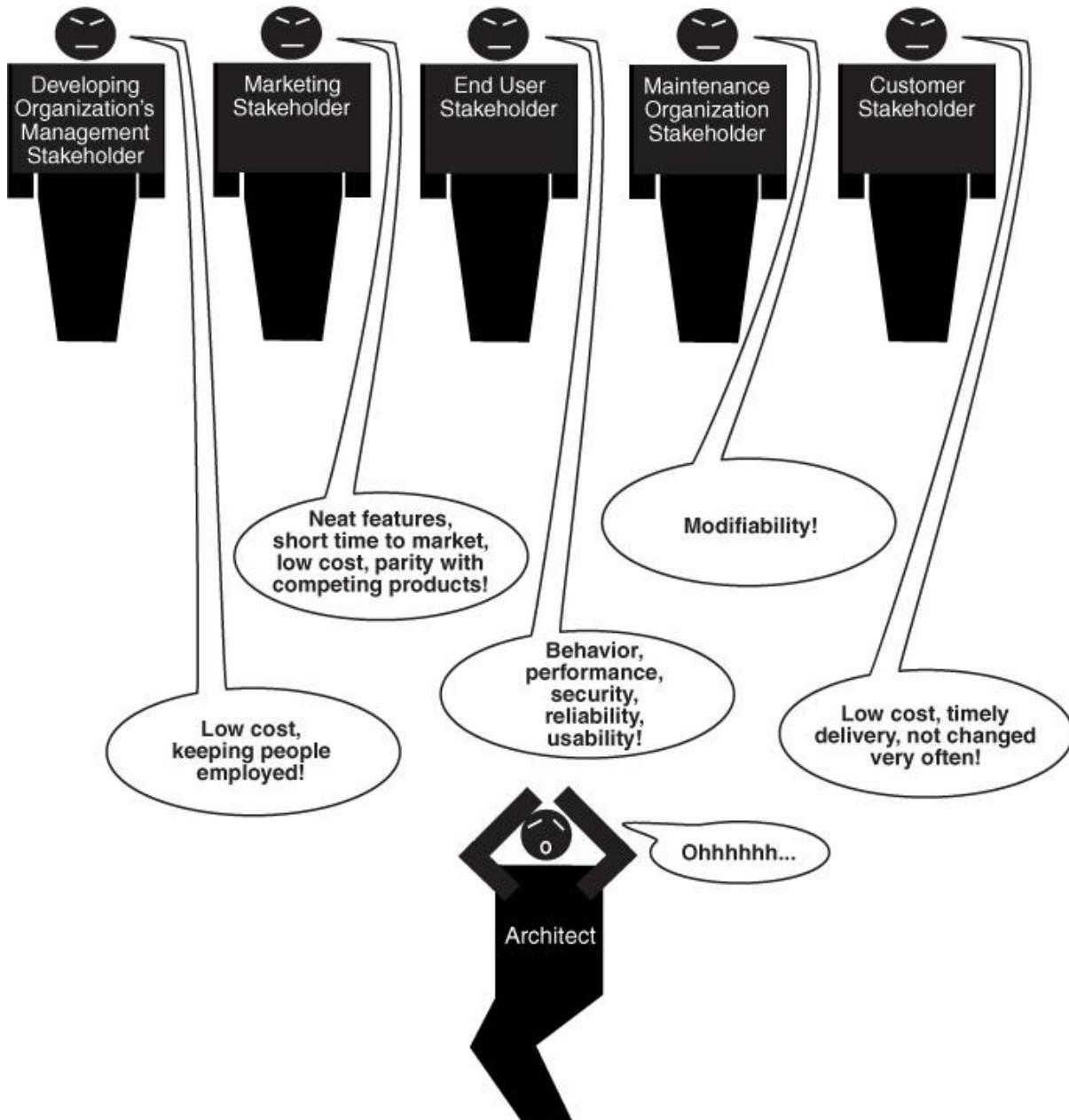


Figure 3.3. Influence of stakeholders on the architect

You will need to know and understand the nature, source, and priority of constraints on the project as early as possible. Therefore, you must identify and actively engage the stakeholders to solicit their needs and expectations. Early engagement of stakeholders allows you to understand the constraints of the task, manage expectations, negotiate priorities, and make tradeoffs. Architecture evaluation (covered in [Part III](#) of this book) and iterative prototyping are two means for you to achieve stakeholder engagement.

Having an acceptable system involves appropriate performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior. All of these qualities, and others, affect how the delivered system is viewed by its eventual recipients, and so such quality attributes will be demanded by one or more of the system's stakeholders.

The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory. It is a rare requirements document that does a good job of capturing all of a system's quality requirements in testable detail (a property is testable if it is falsifiable; "make the system easy to use" is not falsifiable but "deliver audio packets with no more

than 10 ms. jitter” is falsifiable). The architect often has to fill in the blanks—the quality attribute requirements that have not been explicitly stated—and mediate the conflicts that frequently emerge.

Therefore, one of the best pieces of advice we can give to architects is this: Know your stakeholders. Talk to them, engage them, listen to them, and put yourself in their shoes. [Table 3.1](#) enumerates a set of stakeholders. Notice the remarkable variety and length of this set, but remember that not every stakeholder named in this list may play a role in every system, and one person may play many roles.

Table 3.1. Stakeholders for a System and Their Interests

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.
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, 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, 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 distinguished kinds of users, such as administrators, superusers, etc.	Users, in the role of reviewers, might use architecture documentation to check whether desired functionality is being delivered. Users might also use the documentation to understand what the major system elements are, which can aid them in emergency field maintenance.

3.6. How Is Architecture Influenced?

For decades, software designers have been taught to build systems based on the software’s technical requirements. In the older Waterfall model, the requirements document is “tossed over the wall” into the designer’s cubicle, and the designer must come forth with a satisfactory design. Requirements beget design, which begets system. In an iterative or Agile approach to development, an increment of requirements begets an increment of design, and so forth.

This vision of software development is short-sighted. In any development effort, the requirements make explicit some—but only some—of the desired properties of the final system. Not all requirements are focused directly on desired system properties; some requirements might mandate a development process or the use of a particular tool. Furthermore, the requirements specification only begins to tell the story. Failure to satisfy other constraints may render the system just as problematic as if it functioned poorly.

What do you suppose would happen if two different architects, working in two different organizations, were given the same requirements specification for a system? Do you think they would produce the same architecture or different ones?

The answer is that they would very likely produce different ones, which immediately belies the notion that requirements *determine* architecture. Other factors are at work.

A software architecture is a result of business and social influences, as well as technical ones. The existence of an architecture in turn affects the technical, business, and social environments that subsequently influence future architectures. In particular, each of the contexts for architecture that we just covered—technical, project, business, and professional—plays a role in influencing an architect and the architecture, as shown in [Figure 3.4](#).

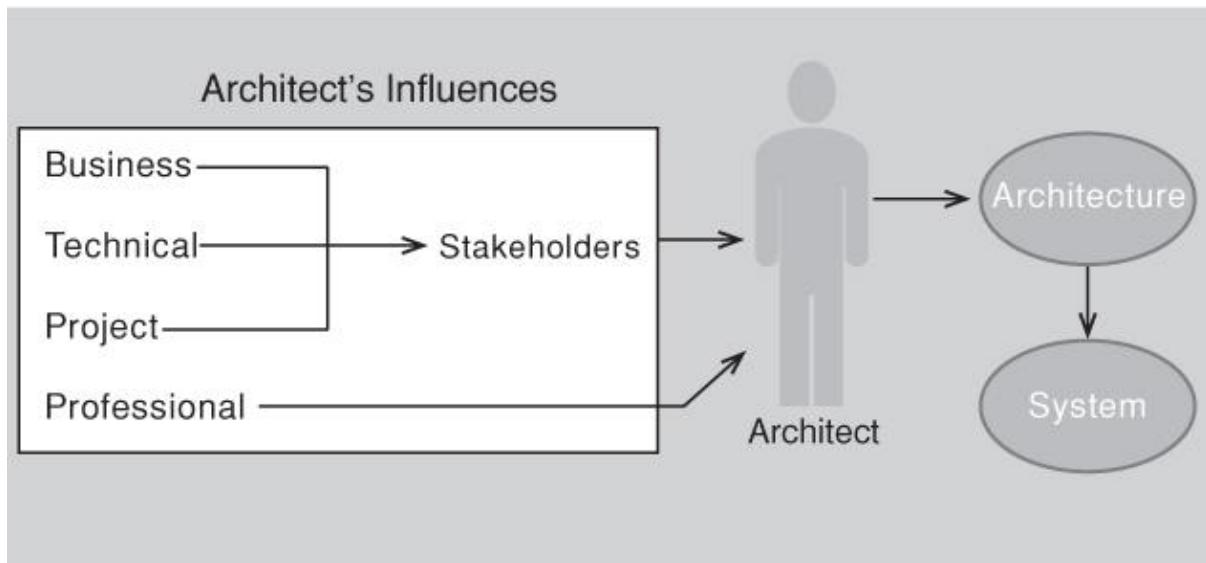


Figure 3.4. Influences on the architect

An architect designing a system for which the real-time deadlines are tight will make one set of design choices; the same architect, designing a similar system in which the deadlines can be easily satisfied, will make different choices. And the same architect, designing a non-real-time system, is likely to make quite different choices still. Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.

3.7. What Do Architectures Influence?

The story about contexts influencing architectures has a flip side. It turns out that architectures have an influence on the very factors that influence them. Specifically, the existence of an architecture affects the technical, project, business, and professional contexts that subsequently influence future architectures.

Here is how the cycle works:

- *Technical context.* The architecture can affect stakeholder requirements for the next system by giving the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if the subsequent system were to be built from scratch, and typically with fewer defects. A customer may in fact be willing to relax some of their requirements to gain these economies. Shrink-wrapped software has clearly affected people's requirements by providing solutions that are not tailored to any individual's precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality. Software product lines have the same effect on customers who cannot be so flexible with their requirements.
- *Project context.* The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly prescribes the units of software that must be implemented (or otherwise obtained) and integrated to form the system. These units are the basis for the development project's structure. Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization. In any design undertaken by the organization at large, these groups have a strong voice in the system's decomposition, pressuring for the continued existence of the portions they control.
- *Business context.* The architecture can affect the business goals of the developing organization. A successful system built from an architecture can enable a company to establish a foothold in a particular market segment—think of the iPhone or Android app.

platforms as examples. The architecture can provide opportunities for the efficient production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.

- *Professional context.* The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base. A system that was successfully built around a particular technical approach will make the architect more inclined to build systems using the same approach in the future. On the other hand, architectures that fail are less likely to be chosen for future projects.

These and other feedback mechanisms form what we call the Architecture Influence Cycle, or AIC, illustrated in [Figure 3.5](#), which depicts the influences of the culture and business of the development organization on the software architecture. That architecture is, in turn, a primary determinant of the properties of the developed system or systems. But the AIC is also based on a recognition that shrewd organizations can take advantage of the organizational and experiential effects of developing an architecture and can use those effects to position their business strategically for future projects.

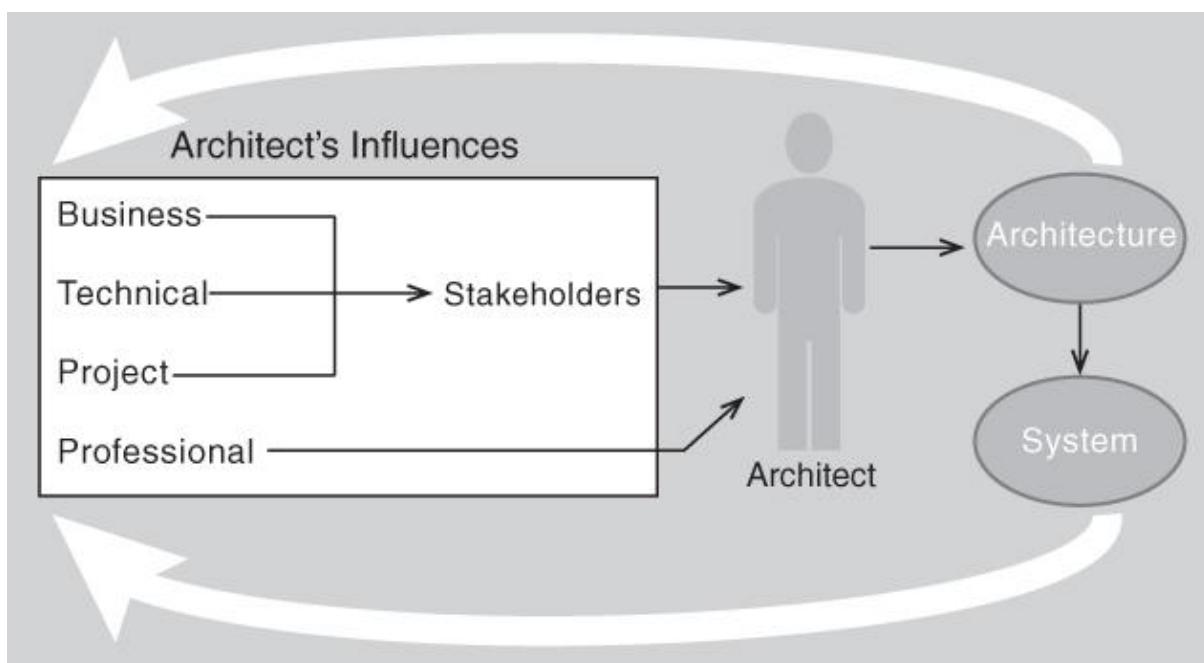


Figure 3.5. Architecture Influence Cycle

3.8. Summary

Architectures exist in four different contexts.

1. *Technical.* The technical context includes the achievement of quality attribute requirements. We spend [Part II](#) discussing how to do this. The technical context also includes the current technology. The cloud (discussed in [Chapter 26](#)) and mobile computing (discussed in [Chapter 27](#)) are important current technologies.
2. *Project life cycle.* Regardless of the software development methodology you use, you must make a business case for the system, understand the architecturally significant requirements, create or select the architecture, document and communicate the architecture, analyze or evaluate the architecture, implement and test the system based on the architecture, and ensure that the implementation conforms to the architecture.
3. *Business.* The system created from the architecture must satisfy the business goals of a wide variety of stakeholders, each of whom has different expectations for the system. The architecture is also influenced by and influences the structure of the development organization.

- 4. Professional.** You must have certain skills and knowledge to be an architect, and there are certain duties that you must perform as an architect. These are influenced not only by coursework and reading but also by your experiences.

An architecture has some influences that lead to its creation, and its existence has an impact on the architect, the organization, and, potentially, the industry. We call this cycle the Architecture Influence Cycle.

3.9. For Further Reading

The product line framework produced by the Software Engineering Institute includes a discussion of business cases from which we drew [[SEI 12](#)].

The SEI has also published a case study of Celsius Tech that includes an example of how organizations and customers change over time [[Brownword 96](#)].

Several other SEI reports discuss how to find business goals and the business goals that have been articulated by certain organizations [[Kazman 05](#), [Clements 10b](#)].

Ruth Malan and Dana Bredemeyer provide a description of how an architect can build buy-in within an organization [[Malan 00](#)].

3.10. Discussion Questions

- 1.** Enumerate six different software systems used by your organization. For each of these systems:
 - a.** What are the contextual influences?
 - b.** Who are the stakeholders?
 - c.** How do these systems reflect or impact the organizational structure?
- 2.** What kinds of business goals have driven the construction of the following:
 - a.** The World Wide Web
 - b.** Amazon's EC2 cloud infrastructure
 - c.** Google's Android platform
- 3.** What mechanisms are available to improve your skills and knowledge? What skills are you lacking?
- 4.** Describe a system you are familiar with and place it into the AIC. Specifically, identify the forward and reverse influences on contextual factors.

Part Two. Quality Attributes

In Part II, we provide the technical foundations for you to design or analyze an architecture to achieve particular quality attributes. We do not discuss design or analysis processes here; we cover those topics in [Part III](#). It is impossible, however, to understand how to improve the performance of a design, for example, without understanding something about performance.

In [Chapter 4](#) we describe how to specify a quality attribute requirement and motivate design techniques called tactics to enable you to achieve a particular quality attribute requirement. We also enumerate seven categories of design decisions. These are categories of decisions that are universally important, and so we provide material to help an architect focus on these decisions. In [Chapter 4](#), we describe these categories, and in each of the following chapters devoted to a particular quality attribute—[Chapters 5–11](#)—we use those categories to develop a checklist that tells you how to focus your attention on the important aspects associated with that quality attribute. Many of the items in our checklists may seem obvious, but the purpose of a checklist is to help ensure the completeness of your design and analysis process.

In addition to providing a treatment of seven specific quality attributes (availability, interoperability, modifiability, performance, security, testability, and usability), we also describe how you can generate the material provided in [Chapters 5–11](#) for other quality attributes that we have not covered.

Architectural patterns provide known solutions to a number of common problems in design. In [Chapter 13](#), we present some of the most important patterns and discuss the relationship between patterns and tactics.

Being able to analyze a design for a particular quality attribute is a key skill that you as an architect will need to acquire. In [Chapter 14](#), we discuss modeling techniques for some of the quality attributes.

4. Understanding Quality Attributes

Between stimulus and response, there is a space. In that space is our power to choose our response. In our response lies our growth and our freedom.

—Viktor E. Frankl, *Man's Search for Meaning*

As we have seen in the Architecture Influence Cycle (in [Chapter 3](#)), many factors determine the qualities that must be provided for in a system’s architecture. These qualities go beyond functionality, which is the basic statement of the system’s capabilities, services, and behavior. Although functionality and other qualities are closely related, as you will see, functionality often takes the front seat in the development scheme. This preference is shortsighted, however. Systems are frequently redesigned not because they are functionally deficient—the replacements are often functionally identical—but because they are difficult to maintain, port, or scale; or they are too slow; or they have been compromised by hackers. In [Chapter 2](#), we said that architecture was the first place in software creation in which quality requirements could be addressed. It is the mapping of a system’s functionality onto software structures that determines the architecture’s support for qualities. In [Chapters 5–11](#) we discuss how various qualities are supported by architectural design decisions. In [Chapter 17](#) we show how to integrate all of the quality attribute decisions into a single design.

We have been using the term “quality attribute” loosely, but now it is time to define it more carefully. A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the “goodness” of a product along some dimension of interest to a stakeholder.

In this chapter our focus is on understanding the following:

- How to express the qualities we want our architecture to provide to the system or systems we are building from it
- How to achieve those qualities

- How to determine the design decisions we might make with respect to those qualities

This chapter provides the context for the discussion of specific quality attributes in [Chapters 5–11](#).

4.1. Architecture and Requirements

Requirements for a system come in a variety of forms: textual requirements, mockups, existing systems, use cases, user stories, and more. [Chapter 16](#) discusses the concept of an *architecturally significant requirement*, the role such requirements play in architecture, and how to identify them. No matter the source, all requirements encompass the following categories:

- 1. Functional requirements.** These requirements state what the system must do, and how it must behave or react to runtime stimuli.
- 2. Quality attribute requirements.** These requirements are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input. A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.
- 3. Constraints.** A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that's already been made. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a management fiat to make your system service oriented. These choices are arguably in the purview of the architect, but external factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

What is the “response” of architecture to each of these kinds of requirements?

- Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.
- Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures. [Chapter 17](#) will show this approach in more detail.
- Constraints are satisfied by accepting the design decision and reconciling it with other affected design decisions.

4.2. Functionality

Functionality is the ability of the system to do the work for which it was intended. Of all of the requirements, functionality has the strangest relationship to architecture.

First of all, functionality does not determine architecture. That is, given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality. At the very least, you could divide up the functionality in any number of ways and assign the subpieces to different architectural elements.

In fact, if functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine. Instead, we design our systems as structured sets of cooperating architectural elements—modules, layers, classes, services, databases, apps, threads, peers, tiers, and on and on—to make them understandable and to support a variety of other purposes. Those “other purposes” are the other quality attributes that we'll turn our attention to in the remaining sections of this chapter, and the remaining chapters of [Part II](#).

But although functionality is independent of any particular structure, functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

Although responsibilities can be allocated arbitrarily to any modules, software architecture constrains this allocation when other quality attributes are important. For example, systems are frequently divided so that several people can cooperatively build them. The architect's interest in functionality is in how it interacts with and constrains other qualities.

4.3. Quality Attribute Considerations

Just as a system's functions do not stand on their own without due consideration of other quality attributes, neither do quality attributes stand on their own; they pertain to the functions of the system. If a functional requirement is "When the user presses the green button, the Options dialog appears," a performance QA annotation might describe how quickly the dialog will appear; an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Functional Requirements

After more than 15 years of writing and discussing the distinction between functional requirements and quality requirements, the definition of functional requirements still eludes me. Quality attribute requirements are well defined: performance has to do with the timing behavior of the system, modifiability has to do with the ability of the system to support changes in its behavior or other qualities after initial deployment, availability has to do with the ability of the system to survive failures, and so forth.

Function, however, is much more slippery. An international standard (ISO 25010) defines functional suitability as "the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions." That is, functionality is the ability to provide functions. One interpretation of this definition is that functionality describes what the system does and quality describes how well the system does its function. That is, qualities are attributes of the system and function is the purpose of the system.

This distinction breaks down, however, when you consider the nature of some of the "function." If the function of the software is to control engine behavior, how can the function be correctly implemented without considering timing behavior? Is the ability to control access through requiring a user name/password combination not a function even though it is not the purpose of any system?

I like much better the use of the word "responsibility" to describe computations that a system must perform. Questions such as "What are the timing constraints on that set of responsibilities?", "What modifications are anticipated with respect to that set of responsibilities?", and "What class of users is allowed to execute that set of responsibilities?" make sense and are actionable.

The achievement of qualities induces responsibility; think of the user name/password example just mentioned. Further, one can identify responsibilities as being associated with a particular set of requirements.

So does this mean that the term "functional requirement" shouldn't be used? People have an understanding of the term, but when precision is desired, we should talk about sets of specific responsibilities instead.

Paul Clements has long ranted against the careless use of the term "nonfunctional," and now it's my turn to rant against the careless use of the term "functional"—probably equally ineffectually.

—LB

Quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an architect's perspective, there are three problems with previous discussions of system quality attributes:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be "modifiable." Every system may be modifiable with respect to one set of changes and not modifiable with respect to another. The other quality attributes are similar in this regard: a system may be robust with respect to some faults and brittle with respect to others. And so forth.

2. Discussion often focuses on which quality a particular concern belongs to. Is a system failure due to a denial-of-service attack an aspect of availability, an aspect of performance, an aspect of security, or an aspect of usability? All four attribute communities would claim ownership of a system failure due to a denial-of-service attack. All are, to some extent, correct. But this doesn't help us, as architects, understand and create architectural solutions to manage the attributes of concern.
3. Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but they are described using different terms.

A solution to the first two of these problems (untestable definitions and overlapping concerns) is to use *quality attribute scenarios* as a means of characterizing quality attributes (see the next section). A solution to the third problem is to provide a discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.

There are two categories of quality attributes on which we focus. The first is those that describe some property of the system at runtime, such as availability, performance, or usability. The second is those that describe some property of the development of the system, such as modifiability or testability.

Within complex systems, quality attributes can never be achieved in isolation. The achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others. For example, almost every quality attribute negatively affects performance. Take portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this hurts performance. Determining the design that satisfies all of the quality attribute requirements is partially a matter of making the appropriate tradeoffs; we discuss design in [Chapter 17](#). Our purpose here is to provide the context for discussing each quality attribute. In particular, we focus on how quality attributes can be specified, what architectural decisions will enable the achievement of particular quality attributes, and what questions about quality attributes will enable the architect to make the correct design decisions.

4.4. Specifying Quality Attribute Requirements

A quality attribute requirement should be unambiguous and testable. We use a common form to specify all quality attribute requirements. This has the advantage of emphasizing the commonalities among all quality attributes. It has the disadvantage of occasionally being a force-fit for some aspects of quality attributes.

Our common form for quality attribute expression has these parts:

- *Stimulus*. We use the term "stimulus" to describe an event arriving at the system. The stimulus can be an event to the performance community, a user operation to the usability community, or an attack to the security community. We use the same term to describe a motivating action for developmental qualities. Thus, a stimulus for modifiability is a request for a modification; a stimulus for testability is the completion of a phase of development.
- *Stimulus source*. A stimulus must have a source—it must come from somewhere. The source of the stimulus may affect how it is treated by the system. A request from a trusted user will not undergo the same scrutiny as a request by an untrusted user.
- *Response*. How the system should respond to the stimulus must also be specified. The response consists of the responsibilities that the system (for runtime qualities) or the developers (for development-time qualities) should perform in response to the stimulus. For example, in a performance scenario, an event arrives (the stimulus) and the system should process that event and generate a response. In a modifiability scenario, a request for a modification arrives (the stimulus) and the developers should implement the modification—without side effects—and then test and deploy the modification.
- *Response measure*. Determining whether a response is satisfactory—whether the requirement is satisfied—is enabled by providing a response measure. For performance

this could be a measure of latency or throughput; for modifiability it could be the labor or wall clock time required to make, test, and deploy the modification.

These four characteristics of a scenario are the heart of our quality attribute specifications. But there are two more characteristics that are important: environment and artifact.

- *Environment*. The environment of a requirement is the set of circumstances in which the scenario takes place. The environment acts as a qualifier on the stimulus. For example, a request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze. A failure that is the fifth successive failure of a component may be treated differently than the first failure of that component.
- *Artifact*. Finally, the artifact is the portion of the system to which the requirement applies. Frequently this is the entire system, but occasionally specific portions of the system may be called out. A failure in a data store may be treated differently than a failure in the metadata store. Modifications to the user interface may have faster response times than modifications to the middleware.

To summarize how we specify quality attribute requirements, we capture them formally as six-part scenarios. While it is common to omit one or more of these six parts, particularly in the early stages of thinking about quality attributes, knowing that all parts are there forces the architect to consider whether each part is relevant.

In summary, here are the six parts:

1. *Source of stimulus*. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. *Stimulus*. The stimulus is a condition that requires a response when it arrives at a system.
3. *Environment*. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, “normal” operation can refer to one of a number of modes. For these kinds of systems, the environment should specify in which mode the system is executing.
4. *Artifact*. Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. *Response*. The response is the activity undertaken as the result of the arrival of the stimulus.
6. *Response measure*. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

We distinguish *general* quality attribute scenarios (which we call “general scenarios” for short)—those that are system independent and can, potentially, pertain to any system—from *concrete* quality attribute scenarios (concrete scenarios)—those that are specific to the particular system under consideration.

We can characterize quality attributes as a collection of general scenarios. Of course, to translate these generic attribute characterizations into requirements for a particular system, the general scenarios need to be made system specific. Detailed examples of these scenarios will be given in [Chapters 5–11](#). [Figure 4.1](#) shows the parts of a quality attribute scenario that we have just discussed. [Figure 4.2](#) shows an example of a general scenario, in this case for availability.

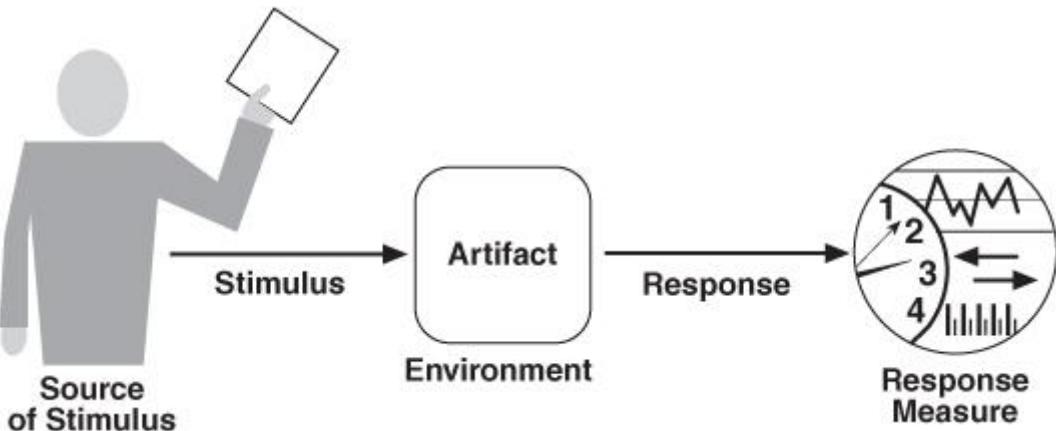


Figure 4.1. The parts of a quality attribute scenario

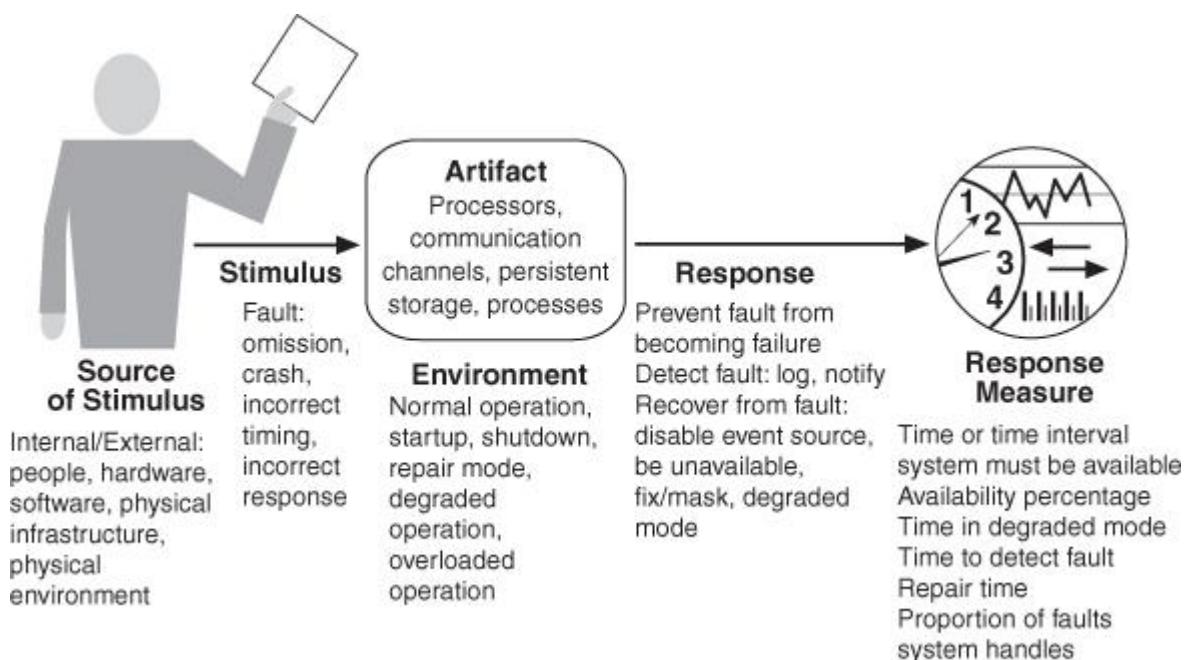


Figure 4.2. A general scenario for availability

4.5. Achieving Quality Attributes through Tactics

The quality attribute requirements specify the responses of the system that, with a bit of luck and a dose of good planning, realize the goals of the business. We now turn to the techniques an architect can use to achieve the required quality attributes. We call these techniques *architectural tactics*. A tactic is a design decision that influences the achievement of a quality attribute response—tactics directly affect the system's response to some stimulus. Tactics impart portability to one design, high performance to another, and integrability to a third.

Not My Problem

One time I was doing an architecture analysis on a complex system created by and for Lawrence Livermore National Laboratory. If you visit their website (www.llnl.gov) and try to figure out what Livermore Labs does, you will see the word “security” mentioned over and over. The lab focuses on nuclear security, international and domestic security, and environmental and energy security. Serious stuff. . .

Keeping this emphasis in mind, I asked them to describe the quality attributes of concern for the system that I was analyzing. I'm sure you can imagine my surprise when security wasn't mentioned once! The system stakeholders mentioned performance, modifiability, evolvability, interoperability, configurability, and portability, and one or two more, but the word security never passed their lips.

Being a good analyst, I questioned this seemingly shocking and obvious omission. Their answer was simple and, in retrospect, straightforward: "We don't care about it. Our systems are not connected to any external network and we have barbed-wire fences and guards with machine guns." Of course, *someone* at Livermore Labs was very interested in security. But it was clearly not the software architects.

—RK

The focus of a tactic is on a single quality attribute response. Within a tactic, there is no consideration of tradeoffs. Tradeoffs must be explicitly considered and controlled by the designer. In this respect, tactics differ from architectural patterns, where tradeoffs are built into the pattern. (We visit the relation between tactics and patterns in [Chapter 14](#). [Chapter 13](#) explains how sets of tactics for a quality attribute can be constructed, which are the steps we used to produce the set in this book.)

A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality. We represent the relationship between stimulus, tactics, and response in [Figure 4.3](#). The tactics, like design patterns, are design techniques that architects have been using for years. Our contribution is to isolate, catalog, and describe them. We are not inventing tactics here, we are just capturing what architects do in practice.

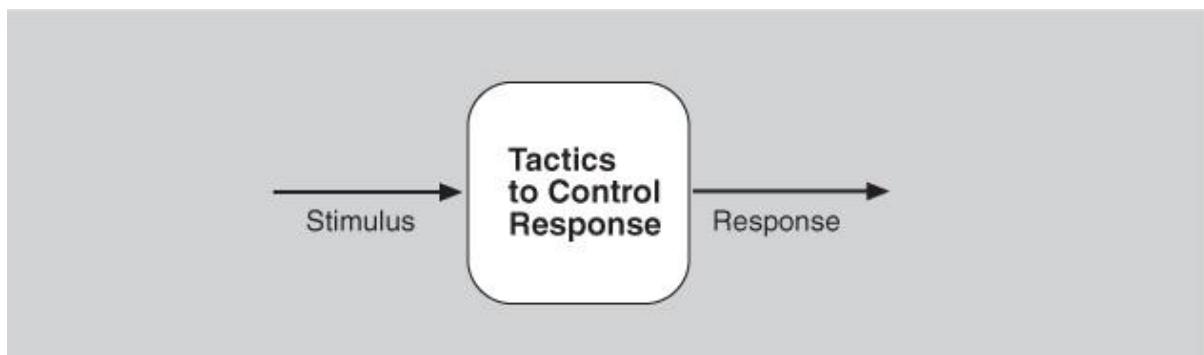


Figure 4.3. Tactics are intended to control responses to stimuli.

Why do we do this? There are three reasons:

1. Design patterns are complex; they typically consist of a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them. By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.
2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles." Tactics give the architect insight into the properties of the resulting design fragment.
3. By cataloging tactics, we provide a way of making design more systematic within some limitations. Our list of tactics does not provide a taxonomy. We only provide a categorization. The tactics will overlap, and you frequently will have a choice among multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement. These considerations transcend the discussion of tactics for particular quality attributes. [Chapter 17](#) provides some techniques for choosing among competing tactics.

The tactics that we present can and should be refined. Consider performance: *Schedule resources* is a common performance tactic. But this tactic needs to be refined into a specific scheduling strategy, such as shortest-job-first, round-robin, and so forth, for specific purposes. *Use an intermediary* is a modifiability tactic. But there are multiple types of intermediaries (layers, brokers, and proxies, to name just a few). Thus there are refinements that a designer will employ to make each tactic concrete.

In addition, the application of a tactic depends on the context. Again considering performance: *Manage sampling rate* is relevant in some real-time systems but not in all real-time systems and certainly not in database systems.

4.6. Guiding Quality Design Decisions

Recall that one can view an architecture as the result of applying a collection of design decisions. What we present here is a systematic categorization of these decisions so that an architect can focus attention on those design dimensions likely to be most troublesome.

The seven categories of design decisions are

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

These categories are not the only way to classify architectural design decisions, but they do provide a rational division of concerns. These categories might overlap, but it's all right if a particular decision exists in two different categories, because the concern of the architect is to ensure that every important decision is considered. Our categorization of decisions is partially based on our definition of software architecture in that many of our categories relate to the definition of structures and the relations among them.

Allocation of Responsibilities

Decisions involving allocation of responsibilities include the following:

- Identifying the important responsibilities, including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
- Determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

Strategies for making these decisions include functional decomposition, modeling real-world objects, grouping based on the major modes of system operation, or grouping based on similar quality requirements: processing frame rate, security level, or expected changes.

In [Chapters 5–11](#), where we apply these design decision categories to a number of important quality attributes, the checklists we provide for the allocation of responsibilities category is derived systematically from understanding the stimuli and responses listed in the general scenario for that QA.

Coordination Model

Software works by having elements interact with each other through designed mechanisms. These mechanisms are collectively referred to as a coordination model. Decisions about the coordination model include these:

- Identifying the elements of the system that must coordinate, or are prohibited from coordinating.
- Determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency.

- Choosing the communication mechanisms (between systems, between our system and external entities, between elements of our system) that realize those properties. Important properties of the communication mechanisms include stateful versus stateless, synchronous versus asynchronous, guaranteed versus nonguaranteed delivery, and performance-related properties such as throughput and latency.

Data Model

Every system must represent artifacts of system-wide interest—data—in some internal fashion. The collection of those representations and how to interpret them is referred to as the data model. Decisions about the data model include the following:

- Choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
- Compiling metadata needed for consistent interpretation of the data.
- Organizing the data. This includes determining whether the data is going to be kept in a relational database, a collection of objects, or both. If both, then the mapping between the two different locations of the data must be determined.

Management of Resources

An architect may need to arbitrate the use of shared resources in the architecture. These include hard resources (e.g., CPU, memory, battery, hardware buffers, system clock, I/O ports) and soft resources (e.g., system locks, software buffers, thread pools, and non-thread-safe code).

Decisions for management of resources include the following:

- Identifying the resources that must be managed and determining the limits for each.
- Determining which system element(s) manage each resource.
- Determining how resources are shared and the arbitration strategies employed when there is contention.
- Determining the impact of saturation on different resources. For example, as a CPU becomes more heavily loaded, performance usually just degrades fairly steadily. On the other hand, when you start to run out of memory, at some point you start paging/swapping intensively and your performance suddenly crashes to a halt.

Mapping among Architectural Elements

An architecture must provide two types of mappings. First, there is mapping between elements in different types of architecture structures—for example, mapping from units of development (modules) to units of execution (threads or processes). Next, there is mapping between software elements and environment elements—for example, mapping from processes to the specific CPUs where these processes will execute.

Useful mappings include these:

- The mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element.
- The assignment of runtime elements to processors.
- The assignment of items in the data model to data stores.
- The mapping of modules and runtime elements to units of delivery.

Binding Time Decisions

Binding time decisions introduce allowable ranges of variation. This variation can be bound at different times in the software life cycle by different entities—from design time by a developer to runtime by an end user. A binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation.

The decisions in the other six categories have an associated binding time decision. Examples of such binding time decisions include the following:

- For allocation of responsibilities, you can have build-time selection of modules via a parameterized makefile.
- For choice of coordination model, you can design runtime negotiation of protocols.
- For resource management, you can design a system to accept new peripheral devices plugged in at runtime, after which the system recognizes them and downloads and installs the right drivers automatically.
- For choice of technology, you can build an app store for a smartphone that automatically downloads the version of the app appropriate for the phone of the customer buying the app.

When making binding time decisions, you should consider the costs to implement the decision and the costs to make a modification after you have implemented the decision. For example, if you are considering changing platforms at some time after code time, you can insulate yourself from the effects caused by porting your system to another platform at some cost. Making this decision depends on the costs incurred by having to modify an early binding compared to the costs incurred by implementing the mechanisms involved in the late binding.

Choice of Technology

Every architecture decision must eventually be realized using a specific technology. Sometimes the technology selection is made by others, before the intentional architecture design process begins. In this case, the chosen technology becomes a constraint on decisions in each of our seven categories. In other cases, the architect must choose a suitable technology to realize a decision in every one of the categories.

Choice of technology decisions involve the following:

- Deciding which technologies are available to realize the decisions made in the other categories.
- Determining whether the available tools to support this technology choice (IDEs, simulators, testing tools, etc.) are adequate for development to proceed.
- Determining the extent of internal familiarity as well as the degree of external support available for the technology (such as courses, tutorials, examples, and availability of contractors who can provide expertise in a crunch) and deciding whether this is adequate to proceed.
- Determining the side effects of choosing a technology, such as a required coordination model or constrained resource management opportunities.
- Determining whether a new technology is compatible with the existing technology stack. For example, can the new technology run on top of or alongside the existing technology stack? Can it communicate with the existing technology stack? Can the new technology be monitored and managed?

4.7. Summary

Requirements for a system come in three categories:

- 1. Functional.** These requirements are satisfied by including an appropriate set of responsibilities within the design.
- 2. Quality attribute.** These requirements are satisfied by the structures and behaviors of the architecture.
- 3. Constraints.** A constraint is a design decision that's already been made.

To express a quality attribute requirement, we use a quality attribute scenario. The parts of the scenario are these:

- 1. Source of stimulus**
- 2. Stimulus**
- 3. Environment**
- 4. Artifact**
- 5. Response**

6. Response measure

An architectural tactic is a design decision that affects a quality attribute response. The focus of a tactic is on a single quality attribute response. Architectural patterns can be seen as "packages" of tactics.

The seven categories of architectural design decisions are these:

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

4.8. For Further Reading

Philippe Kruchten [\[Kruchten 04\]](#) provides another categorization of design decisions.

Pena [\[Pena 87\]](#) uses categories of Function/Form/Economy/Time as a way of categorizing design decisions.

Binding time and mechanisms to achieve different types of binding times are discussed in [\[Bachmann 05\]](#).

Taxonomies of quality attributes can be found in [\[Boehm 78\]](#), [\[McCall 77\]](#), and [\[ISO 11\]](#).

Arguments for viewing architecture as essentially independent from function can be found in [\[Shaw 95\]](#).

4.9. Discussion Questions

1. What is the relationship between a use case and a quality attribute scenario? If you wanted to add quality attribute information to a use case, how would you do it?
2. Do you suppose that the set of tactics for a quality attribute is finite or infinite? Why?
3. Discuss the choice of programming language (an example of choice of technology) and its relation to architecture in general, and the design decisions in the other six categories? For instance, how can certain programming languages enable or inhibit the choice of particular coordination models?
4. We will be using the automatic teller machine as an example throughout the chapters on quality attributes. Enumerate the set of responsibilities that an automatic teller machine should support and propose an initial design to accommodate that set of responsibilities. Justify your proposal.
5. Think about the screens that your favorite automatic teller machine uses. What do those screens tell you about binding time decisions reflected in the architecture?
6. Consider the choice between synchronous and asynchronous communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?
7. Consider the choice between stateful and stateless communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?
8. Most peer-to-peer architecture employs late binding of the topology. What quality attributes does this promote or inhibit?

5. Availability

With James Scott

Ninety percent of life is just showing up.

—Woody Allen

Availability refers to a property of software that it is there and ready to carry out its task when you need it to be. This is a broad perspective and encompasses what is normally called reliability (although it may encompass additional considerations such as downtime due to periodic maintenance). In fact, availability builds upon the concept of reliability by adding the notion of recovery—that is, when the system breaks, it repairs itself. Repair may be accomplished by various means, which we'll see in this chapter. More precisely, Avižienis and his colleagues have defined dependability:

Dependability is the ability to avoid failures that are more frequent and more severe than is acceptable.

Our definition of availability as an aspect of dependability is this: "Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval." These definitions make the concept of failure subject to the judgment of an external agent, possibly a human. They also subsume concepts of reliability, confidentiality, integrity, and any other quality attribute that involves a concept of unacceptable failure.

Availability is closely related to security. A denial-of-service attack is explicitly designed to make a system fail—that is, to make it unavailable. Availability is also closely related to performance, because it may be difficult to tell when a system has failed and when it is simply being outrageously slow to respond. Finally, availability is closely allied with safety, which is concerned with keeping the system from entering a hazardous state and recovering or limiting the damage when it does.

Fundamentally, availability is about minimizing service outage time by mitigating faults. Failure implies visibility to a system or human observer in the environment. That is, a failure is the deviation of the system from its specification, where the deviation is externally visible. One of the most demanding tasks in building a high-availability, fault-tolerant system is to understand the nature of the failures that can arise during operation (see the sidebar "[Planning for Failure](#)"). Once those are understood, mitigation strategies can be designed into the software.

A failure's cause is called a fault. A fault can be either internal or external to the system under consideration. Intermediate states between the occurrence of a fault and the occurrence of a failure are called errors. Faults can be prevented, tolerated, removed, or forecast. In this way a system becomes "resilient" to faults.

Among the areas with which we are concerned are how system faults are detected, how frequently system faults may occur, what happens when a fault occurs, how long a system is allowed to be out of operation, when faults or failures may occur safely, how faults or failures can be prevented, and what kinds of notifications are required when a failure occurs.

Because a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be a brief delay in the response time or it may be the time it takes someone to fly to a remote location in the Andes to repair a piece of mining machinery (as was recounted to us by a person responsible for repairing the software in a mining machine engine). The notion of "observability" can be a tricky one: the Stuxnet virus, as an example, went unobserved for a very long time even though it was doing damage. In addition, we are often concerned with the level of capability that remains when a failure has occurred—a degraded operating mode.

The distinction between faults and failures allows discussion of automatic repair strategies. That is, if code containing a fault is executed but the system is able to recover from the fault without any deviation from specified behavior being observable, there is no failure.

The availability of a system can be calculated as the probability that it will provide the specified services within required bounds over a specified time interval. When referring to hardware, there is a well-known expression used to derive steady-state availability:

$$\frac{MTBF}{(MTBF + MTTR)}$$

where *MTBF* refers to the mean time between failures and *MTTR* refers to the mean time to repair. In the software world, this formula should be interpreted to mean that when thinking about availability, you should think about what will make your system fail, how likely that is to occur, and that there will be some time required to repair it.

From this formula it is possible to calculate probabilities and make claims like “99.999 percent availability,” or a 0.001 percent probability that the system will not be operational when needed. Scheduled downtimes (when the system is intentionally taken out of service) may not be considered when calculating availability, because the system is deemed “not needed” then; of course, this depends on the specific requirements for the system, often encoded in service-level agreements (SLAs). This arrangement may lead to seemingly odd situations where the system is down and users are waiting for it, but the downtime is scheduled and so is not counted against any availability requirements.

In operational systems, faults are detected and correlated prior to being reported and repaired. Fault correlation logic will categorize a fault according to its severity (critical, major, or minor) and service impact (service-affecting or non-service-affecting) in order to provide the system operator with timely and accurate system status and allow for the appropriate repair strategy to be employed. The repair strategy may be automated or may require manual intervention.

The availability provided by a computer system or hosting service is frequently expressed as a service-level agreement. This SLA specifies the availability level that is guaranteed and, usually, the penalties that the computer system or hosting service will suffer if the SLA is violated. The SLA that Amazon provides for its EC2 cloud service is

AWS will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage [defined elsewhere] of at least 99.95% during the Service Year. In the event Amazon EC2 does not meet the Annual Uptime Percentage commitment, you will be eligible to receive a Service Credit as described below.

[Table 5.1](#) provides examples of system availability requirements and associated threshold values for acceptable system downtime, measured over observation periods of 90 days and one year. The term *high availability* typically refers to designs targeting availability of 99.999 percent (“5 nines”) or greater. By definition or convention, only unscheduled outages contribute to system downtime.

Table 5.1. System Availability Requirements

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

Planning for Failure

When designing a high-availability or safety-critical system, it’s tempting to say that failure is not an option. It’s a catchy phrase, but it’s a lousy design philosophy. In fact, failure is not only an option, it’s almost inevitable. What will make your system safe and available is planning for the occurrence of failure or (more likely) failures, and handling them with aplomb. The first step is to understand what kinds of failures your system is prone to, and what the consequences of each will be. Here are three well-known techniques for getting a handle on this.

Hazard analysis

Hazard analysis is a technique that attempts to catalog the hazards that can occur during the operation of a system. It categorizes each hazard according to its severity. For example, the DO-178B standard used in the aeronautics industry defines these failure condition levels in terms of their effects on the aircraft, crew, and passengers:

- *Catastrophic*. This kind of failure may cause a crash. This failure represents the loss of critical function required to safely fly and land aircraft.
- *Hazardous*. This kind of failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
- *Major*. This kind of failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries) or significantly increases crew workload to the point where safety is affected.
- *Minor*. This kind of failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine flight plan change).
- *No effect*. This kind of failure has no impact on safety, aircraft operation, or crew workload.

Other domains have their own categories and definitions. Hazard analysis also assesses the probability of each hazard occurring. Hazards for which the product of cost and probability exceed some threshold are then made the subject of mitigation activities.

Fault tree analysis

Fault tree analysis is an analytical technique that specifies a state of the system that negatively impacts safety or reliability, and then analyzes the system's context and operation to find all the ways that the undesired state could occur. The technique uses a graphic construct (the fault tree) that helps identify all sequential and parallel sequences of contributing faults that will result in the occurrence of the undesired state, which is listed at the top of the tree (the "top event"). The contributing faults might be hardware failures, human errors, software errors, or any other pertinent events that can lead to the undesired state.

[Figure 5.1](#), taken from a NASA handbook on fault tree analysis, shows a very simple fault tree for which the top event is failure of component D. It shows that component D can fail if A fails *and* either B or C fails.

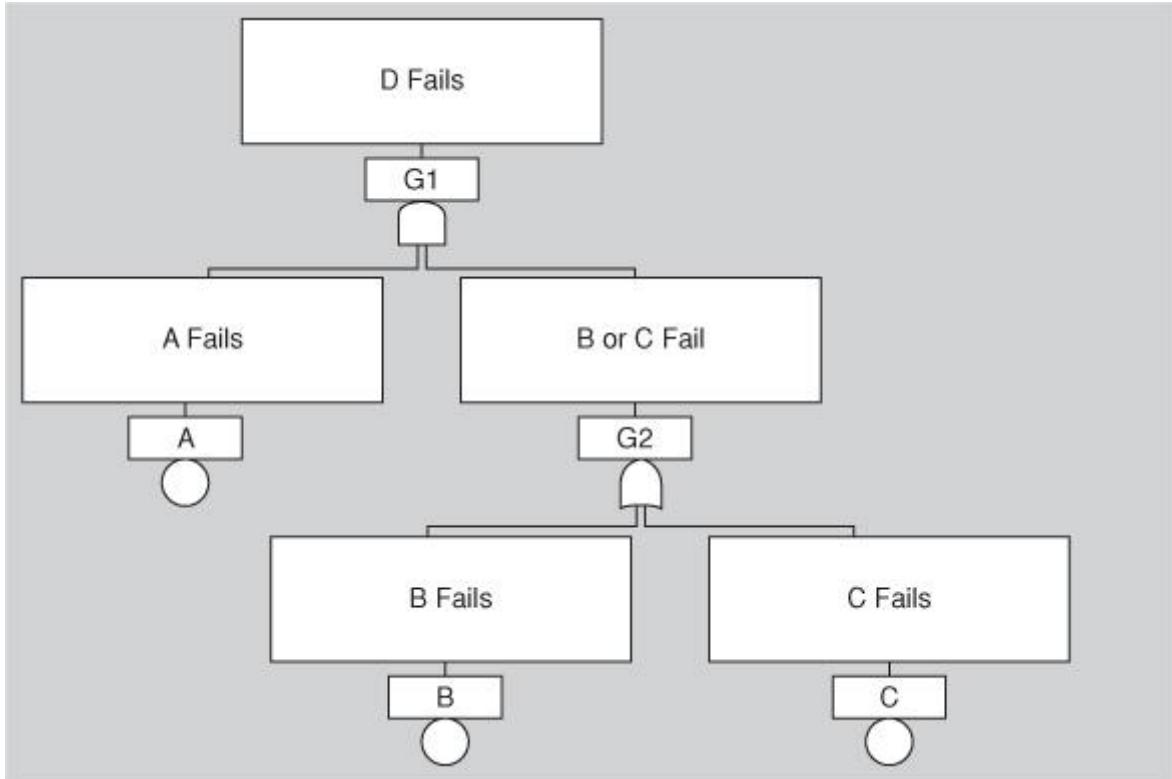


Figure 5.1. A simple fault tree. D fails if A fails and either B or C fails.

The symbols that connect the events in a fault tree are called gate symbols, and are taken from Boolean logic diagrams. [Figure 5.2](#) illustrates the notation.

GATE SYMBOLS



AND Output fault occurs if all of the input faults occur



OR Output fault occurs if at least one of the input faults occurs



COMBINATION Output fault occurs if n of the input faults occur



EXCLUSIVE OR Output fault occurs if exactly one of the input faults occurs



PRIORITY AND Output fault occurs if all of the input faults occur in a specific sequence (the sequence is represented by a CONDITIONING EVENT drawn to the right of the gate)



INHIBIT Output fault occurs if the (single) input fault occurs in the presence of an enabling condition (the enabling condition is represented by a CONDITIONING EVENT drawn to the right of the gate)

Figure 5.2. Fault tree gate symbols

A fault tree lends itself to static analysis in various ways. For example, a “minimal cut set” is the smallest combination of events along the bottom of the tree that together can cause the top event. The set of minimal cut sets shows all the ways the bottom events can combine to cause the overarching failure. Any singleton minimal cut set reveals a single point of failure, which should be carefully scrutinized. Also, the probabilities of various contributing failures can be combined to come up with a probability of the top event occurring. Dynamic analysis occurs when the order of contributing failures matters. In this case, techniques such as Markov analysis can be used to calculate probability of failure over different failure sequences.

Fault trees aid in system design, but they can also be used to diagnose failures at runtime. If the top event has occurred, then (assuming the fault tree model is complete) one or more of the contributing failures has occurred, and the fault tree can be used to track it down and initiate repairs.

Failure Mode, Effects, and Criticality Analysis (FMECA) catalogs the kinds of failures that systems of a given type are prone to, along with how severe the effects of each one can be. FMECA relies on the history of failure of similar systems in the past. [Table 5.2](#), also taken from the NASA handbook, shows the data for a system of redundant amplifiers. Historical data shows that amplifiers fail most often when there is a short circuit or the circuit is left open, but there are several other failure modes as well (lumped together as “Other”).

Table 5.2. Failure Probabilities and Effects

Component	Failure Probability	Failure Mode	% Failures by Mode	Effects	
				Critical	Noncritical
A	1×10^{-3}	Open	90		X
		Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	
B	1×10^{-3}	Open	90		X
		Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	

Adding up the critical column gives us the probability of a critical system failure: $5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} = 2 \times 10^{-4}$.

These techniques, and others, are only as good as the knowledge and experience of the people who populate their respective data structures. One of the worst mistakes you can make, according to the NASA handbook, is to let form take priority over substance. That is, don't let safety engineering become a matter of just filling out the tables. Instead, keep pressing to find out what else can go wrong, and then plan for it.

5.1. Availability General Scenario

From these considerations we can now describe the individual portions of an availability general scenario. These are summarized in [Table 5.3](#):

- *Source of stimulus.* We differentiate between internal and external origins of faults or failure because the desired system response may be different.
- *Stimulus.* A fault of one of the following classes occurs:
 - *Omission.* A component fails to respond to an input.
 - *Crash.* The component repeatedly suffers omission faults.
 - *Timing.* A component responds but the response is early or late.
 - *Response.* A component responds with an incorrect value.
- *Artifact.* This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
- *Environment.* The state of the system when the fault or failure occurs may also affect the desired system response. For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally. However, if this is the first fault observed, some degradation of response time or function may be preferred.
- *Response.* There are a number of possible reactions to a system fault. First, the fault must be detected and isolated (correlated) before any other response is possible. (One exception to this is when the fault is prevented before it occurs.) After the fault is detected, the system must recover from it. Actions associated with these possibilities include logging the failure, notifying selected users or other systems, taking actions to limit the damage caused by the fault, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair.
- *Response measure.* The response measure can specify an availability percentage, or it can specify a time to detect the fault, time to repair the fault, times or time intervals

during which the system must be available, or the duration for which the system must be available.

Table 5.3. Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	<p>Prevent the fault from becoming a failure</p> <p>Detect the fault:</p> <ul style="list-style-type: none"> ▪ Log the fault ▪ Notify appropriate entities (people or systems) <p>Recover from the fault:</p> <ul style="list-style-type: none"> ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	<p>Time or time interval when the system must be available</p> <p>Availability percentage (e.g., 99.999%)</p> <p>Time to detect the fault</p> <p>Time to repair the fault</p> <p>Time or time interval in which system can be in degraded mode</p> <p>Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing</p>

[Figure 5.3](#) shows a concrete scenario generated from the general scenario: The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.

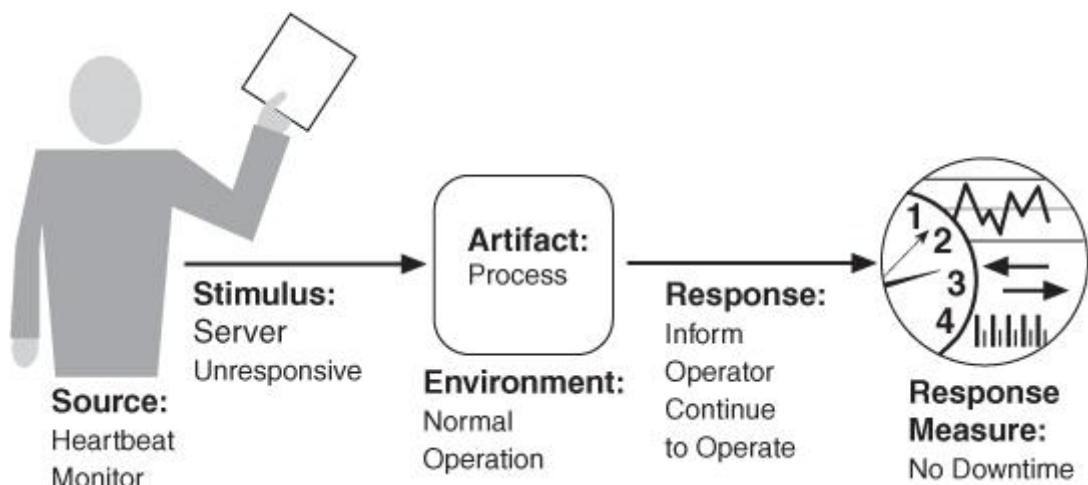


Figure 5.3. Sample concrete availability scenario

5.2. Tactics for Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, therefore, are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specification. The tactics we discuss in this section will keep faults from becoming failures or at least bound the effects of the fault and make repair possible. We illustrate this approach in [Figure 5.4](#).

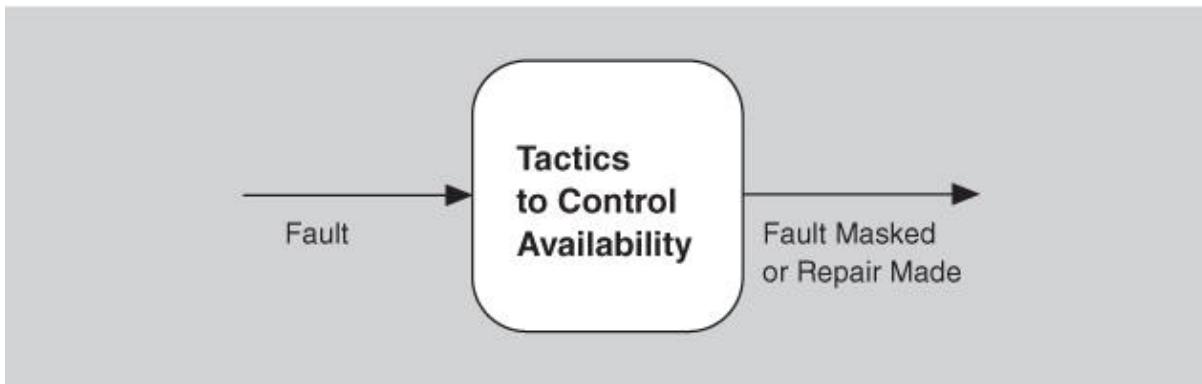


Figure 5.4. Goal of availability tactics

Availability tactics may be categorized as addressing one of three categories: fault detection, fault recovery, and fault prevention. The tactics categorization for availability is shown in [Figure 5.5](#) (on the next page). Note that it is often the case that these tactics will be provided for you by a software infrastructure, such as a middleware package, so your job as an architect is often one of choosing and assessing (rather than implementing) the right availability tactics and the right combination of tactics.

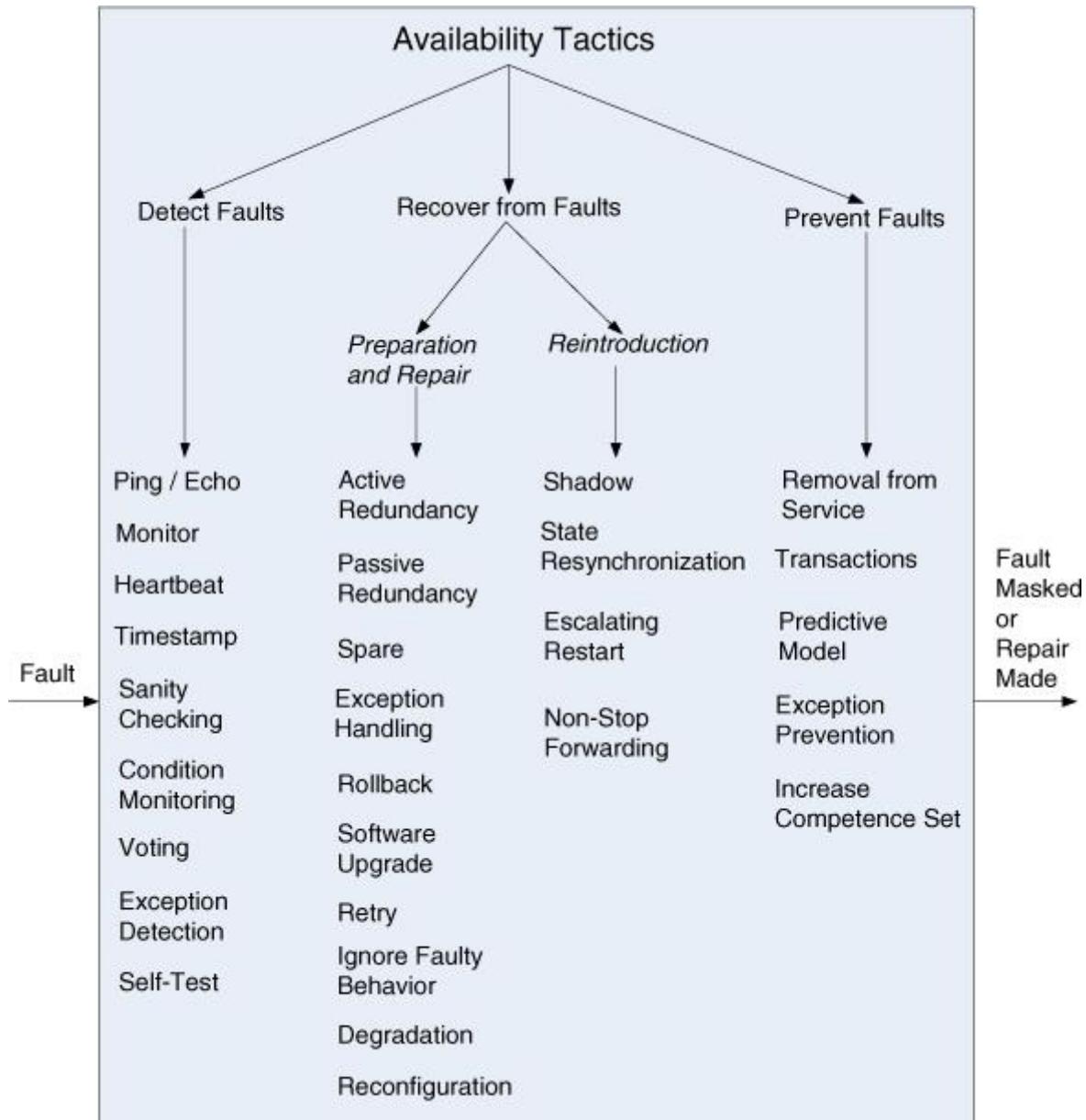


Figure 5.5. Availability tactics

Detect Faults

Before any system can take action regarding a fault, the presence of the fault must be detected or anticipated. Tactics in this category include the following:

- *Ping/echo* refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed ("timed out"). Standard implementations of ping/echo are available for nodes interconnected via IP.
- *Monitor*. A monitor is a component that is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so on. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. It orchestrates software using other tactics in this category to detect malfunctioning components. For example, the system monitor can

initiate self-tests, or be the component that detects faulty time stamps or missed heartbeats.¹

1. When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of system monitor is referred to as a "watchdog." During nominal operation, the process being monitored will periodically reset the watchdog counter/timer as part of its signal that it's working correctly; this is sometimes referred to as "petting the watchdog."
 - *Heartbeat* is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check—the monitor or the component itself.
 - *Time stamp*. This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A time stamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Simple sequence numbers can also be used for this purpose, if time information is not important.
 - *Sanity checking* checks the validity or reasonableness of specific operations or outputs of a component. This tactic is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny. It is most often employed at interfaces, to examine a specific information flow.
 - *Condition monitoring* involves checking conditions in a process or device, or validating assumptions made during the design. By monitoring conditions, this tactic prevents a system from producing faulty behavior. The computation of checksums is a common example of this tactic. However, the monitor must itself be simple (and, ideally, provable) to ensure that it does not introduce new software errors.
 - *Voting*. The most common realization of this tactic is referred to as triple modular redundancy (TMR), which employs three components that do the same thing, each of which receives identical inputs, and forwards their output to voting logic, used to detect any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide what output to use. It can let the majority rule, or choose some computed average of the disparate outputs. This tactic depends critically on the voting logic, which is usually realized as a simple, rigorously reviewed and tested singleton so that the probability of error is low.
 - *Replication* is the simplest form of voting; here, the components are exact clones of each other. Having multiple copies of identical components can be effective in protecting against random failures of hardware, but this cannot protect against design or implementation errors, in hardware or software, because there is no form of diversity embedded in this tactic.
 - *Functional redundancy* is a form of voting intended to address the issue of common-mode failures (design or implementation faults) in hardware or software components. Here, the components must always give the same output given the same input, but they are diversely designed and diversely implemented.
 - *Analytic redundancy* permits not only diversity among components' private sides, but also diversity among the components' inputs and outputs. This tactic is intended to tolerate specification errors by using separate requirement specifications. In embedded systems, analytic redundancy also helps when some input sources are likely to be unavailable at times. For example, avionics programs have multiple ways to compute aircraft altitude, such as using barometric pressure, the radar altimeter, and geometrically using the straight-line distance and look-down angle of a point ahead on the ground. The voter mechanism used with analytic redundancy needs to be more sophisticated than just letting majority rule or computing a simple average. It may have to understand which sensors are currently reliable or not, and it may be

asked to produce a higher-fidelity value than any individual component can, by blending and smoothing individual values over time.

- *Exception detection* refers to the detection of a system condition that alters the normal flow of execution. The exception detection tactic can be further refined:
 - *System exceptions* will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.
 - The *parameter fence* tactic incorporates an *a priori* data pattern (such as 0xDEADBEEF) placed immediately after any variable-length parameters of an object. This allows for runtime detection of overwriting the memory allocated for the object's variable-length parameters.
 - *Parameter typing* employs a base class that defines functions that add, find, and iterate over type-length-value (TLV) formatted message parameters. Derived classes use the base class functions to implement functions that provide parameter typing according to each parameter's structure. Use of strong typing to build and parse messages results in higher availability than implementations that simply treat messages as byte buckets. Of course, all design involves tradeoffs. When you employ strong typing, you typically trade higher availability against ease of evolution.
 - *Timeout* is a tactic that raises an exception when a component detects that it or another component has failed to meet its timing constraints. For example, a component awaiting a response from another component can raise an exception if the wait time exceeds a certain value.
- *Self-test*. Components (or, more likely, whole subsystems) can run procedures to test themselves for correct operation. Self-test procedures can be initiated by the component itself, or invoked from time to time by a system monitor. These may involve employing some of the techniques found in condition monitoring, such as checksums.

Recover from Faults

Recover-from-faults tactics are refined into *preparation-and-repair* tactics and *reintroduction* tactics. The latter are concerned with reintroducing a failed (but rehabilitated) component back into normal operation.

Preparation-and-repair tactics are based on a variety of combinations of retrying a computation or introducing redundancy. They include the following:

- *Active redundancy (hot spare)*. This refers to a configuration where all of the nodes (active or redundant spare) in a protection group² receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, it can take over from a failed component in a matter of milliseconds. The simple case of one active node and one redundant spare node is commonly referred to as 1+1 ("one plus one") redundancy. Active redundancy can also be used for facilities protection, where active and standby network links are used to ensure highly available network connectivity.

2. A protection group is a group of processing nodes where one or more nodes are "active," with the remaining nodes in the protection group serving as redundant spares.

- *Passive redundancy (warm spare)*. This refers to a configuration where only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates. Because the state maintained by the redundant spares is only loosely coupled with that of the active node(s) in the protection group (with the looseness of the coupling being a function of the checkpointing mechanism employed between active and redundant nodes), the redundant nodes are referred to as warm spares. Depending on a system's availability requirements, passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy tactic and the less available but significantly less complex cold spare tactic (which is also significantly cheaper). (For an example of implementing passive redundancy, see the section on code templates in [Chapter 19](#).)

- *Spare (cold spare)*. Cold sparing refers to a configuration where the redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, cold sparing is better suited for systems having only high-reliability (MTBF) requirements as opposed to those also having high-availability requirements.
- *Exception handling*. Once an exception has been detected, the system must handle it in some fashion. The easiest thing it can do is simply to crash, but of course that's a terrible idea from the point of availability, usability, testability, and plain good sense. There are much more productive possibilities. The mechanism employed for exception handling depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception classes that contain information helpful in fault correlation, such as the name of the exception thrown, the origin of the exception, and the cause of the exception thrown. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.
- *Rollback*. This tactic permits the system to revert to a previous known good state, referred to as the "rollback line"—rolling back time—upon the detection of a failure. Once the good state is reached, then execution can continue. This tactic is often combined with active or passive redundancy tactics so that after a rollback has occurred, a standby version of the failed component is promoted to active status. Rollback depends on a copy of a previous good state (a checkpoint) being available to the components that are rolling back. Checkpoints can be stored in a fixed location and updated at regular intervals, or at convenient or significant times in the processing, such as at the completion of a complex operation.
- *Software upgrade* is another preparation-and-repair tactic whose goal is to achieve in-service upgrades to executable code images in a non-service-affecting manner. This may be realized as a function patch, a class patch, or a hitless in-service software upgrade (ISSU). A function patch is used in procedural programming and employs an incremental linker/loader to store an updated software function into a pre-allocated segment of target memory. The new version of the software function will employ the entry and exit points of the deprecated function. Also, upon loading the new software function, the symbol table must be updated and the instruction cache invalidated. The class patch tactic is applicable for targets executing object-oriented code, where the class definitions include a back-door mechanism that enables the runtime addition of member data and functions. Hitless in-service software upgrade leverages the active redundancy or passive redundancy tactics to achieve non-service-affecting upgrades to software and associated schema. In practice, the function patch and class patch are used to deliver bug fixes, while the hitless in-service software upgrade is used to deliver new features and capabilities.
- *Retry*. The retry tactic assumes that the fault that caused a failure is transient and retrying the operation may lead to success. This tactic is used in networks and in server farms where failures are expected and common. There should be a limit on the number of retries that are attempted before a permanent failure is declared.
- *Ignore faulty behavior*. This tactic calls for ignoring messages sent from a particular source when we determine that those messages are spurious. For example, we would like to ignore the messages of an external component launching a denial-of-service attack by establishing Access Control List filters, for example.
- The *degradation* tactic maintains the most critical system functions in the presence of component failures, dropping less critical functions. This is done in circumstances where individual component failures gracefully reduce system functionality rather than causing a complete system failure.
- *Reconfiguration* attempts to recover from component failures by reassigning responsibilities to the (potentially restricted) resources left functioning, while maintaining as much functionality as possible.

Reintroduction is where a failed component is reintroduced after it has been corrected. Reintroduction tactics include the following:

- The *shadow* tactic refers to operating a previously failed or in-service upgraded component in a "shadow mode" for a predefined duration of time prior to reverting the

component back to an active role. During this duration its behavior can be monitored for correctness and it can repopulate its state incrementally.

- *State resynchronization* is a reintroduction partner to the active redundancy and passive redundancy preparation-and-repair tactics. When used alongside the active redundancy tactic, the state resynchronization occurs organically, because the active and standby components each receive and process identical inputs in parallel. In practice, the states of the active and standby components are periodically compared to ensure synchronization. This comparison may be based on a cyclic redundancy check calculation (checksum) or, for systems providing safety-critical services, a message digest calculation (a one-way hash function). When used alongside the passive redundancy (warm spare) tactic, state resynchronization is based solely on periodic state information transmitted from the active component(s) to the standby component(s), typically via checkpointing. A special case of this tactic is found in stateless services, whereby any resource can handle a request from another (failed) resource.
- *Escalating restart* is a reintroduction tactic that allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected. For example, consider a system that supports four levels of restart, as follows. The lowest level of restart (call it Level 0), and hence having the least impact on services, employs passive redundancy (warm spare), where all child threads of the faulty component are killed and recreated. In this way, only data associated with the child threads is freed and reinitialized. The next level of restart (Level 1) frees and reinitializes all unprotected memory (protected memory would remain untouched). The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. And the final level of restart (Level 3) would involve completely reloading and reinitializing the executable image and associated data segments. Support for the escalating restart tactic is particularly useful for the concept of graceful degradation, where a system is able to degrade the services it provides while maintaining support for mission-critical or safety-critical applications.
- *Non-stop forwarding* (NSF) is a concept that originated in router design. In this design functionality is split into two parts: supervisory, or control plane (which manages connectivity and routing information), and data plane (which does the actual work of routing packets from sender to receiver). If a router experiences the failure of an active supervisor, it can continue forwarding packets along known routes—with neighboring routers—while the routing protocol information is recovered and validated. When the control plane is restarted, it implements what is sometimes called “graceful restart,” incrementally rebuilding its routing protocol database even as the data plane continues to operate.

Prevent Faults

Instead of detecting faults and then trying to recover from them, what if your system could prevent them from occurring in the first place? Although this sounds like some measure of clairvoyance might be required, it turns out that in many cases it is possible to do just that.³

3. These tactics deal with runtime means to prevent faults from occurring. Of course, an excellent way to prevent faults—at least in the system you’re building, if not in systems that your system must interact with—is to produce high-quality code. This can be done by means of code inspections, pair programming, solid requirements reviews, and a host of other good engineering practices.

- *Removal from service*. This tactic refers to temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures. One example involves taking a component of a system out of service and resetting the component in order to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults affects service (resulting in system failure). Another term for this tactic is *software rejuvenation*.
- *Transactions*. Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*. These four properties are called the “ACID properties.” The most common realization of the transactions tactic is “two-

phase commit” (a.k.a. 2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item.

- *Predictive model.* A predictive model, when combined with a monitor, is employed to monitor the state of health of a system process to ensure that the system is operating within its nominal operating parameters, and to take corrective action when conditions are detected that are predictive of likely future faults. The operational performance metrics monitored are used to predict the onset of faults; examples include session establishment rate (in an HTTP server), threshold crossing (monitoring high and low water marks for some constrained, shared resource), or maintaining statistics for process state (in service, out of service, under maintenance, idle), message queue length statistics, and so on.
- *Exception prevention.* This tactic refers to techniques employed for the purpose of preventing system exceptions from occurring. The use of exception classes, which allows a system to transparently recover from system exceptions, was discussed previously. Other examples of exception prevention include abstract data types, such as smart pointers, and the use of wrappers to prevent faults, such as dangling pointers and semaphore access violations from occurring. Smart pointers prevent exceptions by doing bounds checking on pointers, and by ensuring that resources are automatically deallocated when no data refers to it. In this way resource leaks are avoided.
- *Increase competence set.* A program’s competence set is the set of states in which it is “competent” to operate. For example, the state when the denominator is zero is outside the competence set of most divide programs. When a component raises an exception, it is signaling that it has discovered itself to be outside its competence set; in essence, it doesn’t know what to do and is throwing in the towel. Increasing a component’s competence set means designing it to handle more cases—faults—as part of its normal operation. For example, a component that assumes it has access to a shared resource might throw an exception if it discovers that access is blocked. Another component might simply wait for access, or return immediately with an indication that it will complete its operation on its own the next time it does have access. In this example, the second component has a larger competence set than the first.

5.3. A Design Checklist for Availability

[Table 5.4](#) is a checklist to support the design and analysis process for availability.

Table 5.4. Checklist to Support the Design and Analysis Process for Availability

Category	Checklist
Allocation of Responsibilities	Determine the system responsibilities that need to be highly available. Within those responsibilities, ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response. Additionally, ensure that there are responsibilities to do the following: <ul style="list-style-type: none">▪ Log the fault▪ Notify appropriate entities (people or systems)▪ Disable the source of events causing the fault▪ Be temporarily unavailable▪ Fix or mask the fault/failure▪ Operate in a degraded mode
Coordination Model	Determine the system responsibilities that need to be highly available. With respect to those responsibilities, do the following: <ul style="list-style-type: none">▪ Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under conditions of degraded communication?▪ Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode.▪ Ensure that the coordination model supports the replacement of the artifacts used (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate?

- Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. For example, how much lost information can the coordination model withstand and with what consequences?
- Data Model**
- Determine which portions of the system need to be highly available. Within those portions, determine which data abstractions, along with their operations or their properties, could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response.
- For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault.
- For example, ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.
- Mapping among Architectural Elements**
- Determine which artifacts (processors, communication channels, persistent storage, or processes) may produce a fault: omission, crash, incorrect timing, or incorrect response.
- Ensure that the mapping (or remapping) of architectural elements is flexible enough to permit the recovery from the fault. This may involve a consideration of the following:
- Which processes on failed processors need to be reassigned at runtime
 - Which processors, data stores, or communication channels can be activated or reassigned at runtime
 - How data on failed processors or storage can be served by replacement units

- How quickly the system can be reinstalled based on the units of delivery provided
- How to (re)assign runtime elements to processors, communication channels, and data stores
- When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant components is important. For example, it is possible to write one module that contains code appropriate for both the active component and backup components in a protection group.

Resource Management

Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable the source of events causing the fault; be temporarily unavailable; fix or mask the fault/failure; operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation.

Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals.

For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.

Binding Time	<p>Determine how and when architectural elements are bound. If late binding is used to alternate between components that can themselves be sources of faults (e.g., processes, processors, communication channels), ensure the chosen availability strategy is sufficient to cover faults introduced by all sources. For example:</p> <ul style="list-style-type: none"> ▪ If late binding is used to switch between artifacts such as processors that will receive or be the subject of faults, will the chosen fault detection and recovery mechanisms work for all possible bindings? ▪ If late binding is used to change the definition or tolerance of what constitutes a fault (e.g., how long a process can go without responding before a fault is assumed), is the recovery strategy chosen sufficient to handle all cases? For example, if a fault is flagged after 0.1 milliseconds, but the recovery mechanism takes 1.5 seconds to work, that might be an unacceptable mismatch. ▪ What are the availability characteristics of the late binding mechanism itself? Can it fail?
Choice of Technology	<p>Determine the available technologies that can (help) detect faults, recover from faults, or reintroduce failed components.</p> <p>Determine what technologies are available that help the response to a fault (e.g., event loggers).</p> <p>Determine the availability characteristics of chosen technologies themselves: What faults can they recover from? What faults might they introduce into the system?</p>

5.4. Summary

Availability refers to the ability of the system to be available for use, especially after a fault occurs. The fault must be recognized (or prevented) and then the system must respond in some fashion. The response desired will depend on the criticality of the application and the type of fault and can range from "ignore it" to "keep on going as if it didn't occur."

Tactics for availability are categorized into detect faults, recover from faults and prevent faults. Detection tactics depend, essentially, on detecting signs of life from various components. Recovery tactics are some combination of retrying an operation or maintaining redundant data or computations. Prevention tactics depend either on removing elements from service or utilizing mechanisms to limit the scope of faults.

All of the availability tactics involve the coordination model because the coordination model must be aware of faults that occur to generate an appropriate response.

5.5. For Further Reading

Patterns for availability:

- You can find patterns for fault tolerance in [\[Hanmer 07\]](#).

Tactics for availability, overall:

- A more detailed discussion of some of the availability tactics in this chapter is given in [\[Scott 09\]](#). This is the source of much of the material in this chapter.
- The Internet Engineering Task Force has promulgated a number of standards supporting availability tactics. These standards include non-stop forwarding [\[IETF 04\]](#), ping/echo ICMPv6 [\[IETF 06b\]](#), echo request/response), and MPLS (LSP Ping) networks[\[IETF 06a\]](#).

Tactics for availability, fault detection:

- The parameter fence tactic was first used (to our knowledge) in the Control Data Series computers of the late 1960s.
- Triple modular redundancy (TMR), part of the voting tactic, was developed in the early 1960s by Lyons [\[Lyons 62\]](#).
- The fault detection tactic of voting is based on the fundamental contributions to automata theory by Von Neumann, who demonstrated how systems having a prescribed reliability could be built from unreliable components [\[Von Neumann 56\]](#).

Tactics for availability, fault recovery:

- Standards-based realizations of active redundancy exist for protecting network links (i.e., facilities) at both the physical layer [\[Bellcore 99\]](#), [\[Telcordia 00\]](#) and the network/link layer [\[IETF 05\]](#).
- Exception handling has been written about by [\[Powel Douglass 99\]](#). Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.
- [\[Morelos-Zaragoza 06\]](#) and [\[Schneier 96\]](#) have written about the comparison of state during resynchronization.
- Some examples of how a system can degrade through use (degradation) are given in [\[Nygard 07\]](#).
- [\[Utas 05\]](#) has written about escalating restart.
- Mountains of papers have been written about parameter typing, but [\[Utas 05\]](#) writes about it in the context of availability (as opposed to bug prevention, its usual context).
- Hardware engineers often use preparation-and-repair tactics. Examples include error detection and correction (EDAC) coding, forward error correction (FEC), and temporal redundancy. EDAC coding is typically used to protect control memory structures in high-availability distributed real-time embedded systems [\[Hamming 80\]](#). Conversely, FEC coding is typically employed to recover from physical-layer errors occurring on external network links [\[Morelos-Zaragoza 06\]](#). Temporal redundancy involves sampling spatially redundant clock or data lines at time intervals that exceed the pulse width of any transient pulse to be tolerated, and then voting out any defects detected [\[Mavis 02\]](#).

Tactics for availability, fault prevention:

- Parnas and Madey have written about increasing an element's competence set [\[Parnas 95\]](#).
- The ACID properties, important in the transactions tactic, were introduced by Gray in the 1970s and discussed in depth in [\[Gray 93\]](#).

Analysis:

- Fault tree analysis dates from the early 1960s, but the granddaddy of resources for it is the U.S. Nuclear Regulatory Commission's "Fault Tree Handbook," published in 1981 [\[Vesely 81\]](#). NASA's 2002 "Fault Tree Handbook with Aerospace Applications" [\[Vesely 02\]](#) is an updated comprehensive primer of the NRC handbook, and the source for the notation used in this chapter. Both are available online as downloadable PDF files.

5.6. Discussion Questions

1. Write a set of concrete scenarios for availability using each of the possible responses in the general scenario.
2. Write a concrete availability scenario for the software for a (hypothetical) pilotless passenger aircraft.
3. Write a concrete availability scenario for a program like Microsoft Word.
4. Redundancy is often cited as a key strategy for achieving high availability. Look at the tactics presented in this chapter and decide how many of them exploit some form of redundancy and how many do not.
5. How does availability trade off against modifiability? How would you make a change to a system that is required to have "24/7" availability (no scheduled or unscheduled downtime, ever)?
6. Create a fault tree for an automatic teller machine. Include faults dealing with hardware component failure, communications failure, software failure, running out of supplies, user errors, and security attacks. How would you modify your automatic teller machine design to accommodate these faults?
7. Consider the fault detection tactics (ping/echo, heartbeat, system monitor, voting, and exception detection). What are the performance implications of using these tactics?

6. Interoperability

With Liming Zhu

The early bird (A) arrives and catches worm (B), pulling string (C) and shooting off pistol (D). Bullet (E) bursts balloon (F), dropping brick (G) on bulb (H) of atomizer (I) and shooting perfume (J) on sponge (K). As sponge gains in weight, it lowers itself and pulls string (L), raising end of board (M). Cannon ball (N) drops on nose of sleeping gentleman. String tied to cannon ball releases cork (O) of vacuum bottle (P) and ice water falls on sleeper's face to assist the cannon ball in its good work.

—Rube Goldberg, instructions for “a simple alarm clock”

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context. The definition includes not only having the ability to exchange data (syntactic interoperability) but also having the ability to correctly interpret the data being exchanged (semantic interoperability). A system cannot be interoperable in isolation. Any discussion of a system’s interoperability needs to identify with whom, with what, and under what circumstances—hence, the need to include the context.

Interoperability is affected by the systems expected to interoperate. If we already know the interfaces of external systems with which our system will interoperate, then we can design that knowledge into the system. Or we can design our system to interoperate in a more generic fashion, so that the identity and the services that another system provides can be bound later in the life cycle, at build time or runtime.

Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning. There are several characterizing frameworks for interoperability, all of which seem to define five levels of interoperability “maturity” (see the “[For Further Reading](#)” section at the end of this chapter for a pointer). The lowest level signifies systems that do not share data at all, or do not do so with any success. The highest level signifies systems that work together seamlessly, never make any mistakes interpreting each other’s communications, and share the same underlying semantic model of the world in which they work.

“Exchanging Information via Interfaces”

Interoperability, as we said, is about two or more systems exchanging information via interfaces.

At this point, we need to clarify two critical concepts central to this discussion and emphasize that we are taking a broad view of each.

The first is what it means to “exchange information.” This can mean something as simple as program A calling program B with some parameters. However, two systems (or parts of a system) can exchange information even if they never communicate directly with each other. Did you ever have a conversation like the following in junior high school? “Charlene said that Kim told her that Trevor heard that Heather wants to come to your party.” Of course, junior high school protocol would preclude the possibility of responding directly to Heather. Instead, your response (if you like Heather) might be, “Cool,” which would make its way back through Charlene, Kim, and Trevor. You and Heather exchanged information, but never talked to each other. (We hope you got to talk to each other at the party.)

Entities can exchange information in even less direct ways. If I have an idea of a program’s behavior, and I design my program to work assuming that behavior, the two programs have also exchanged information—just not at runtime.

One of the more infamous software disasters in history occurred when an antimissile system failed to intercept an incoming ballistic rocket in Operation Desert Storm in 1991, resulting in 28 fatalities. One of the missile’s software components “expected” to be shut down and restarted periodically, so it could recalibrate its orientation framework from a known initial point. The software had been running for some 100 hours when the missile was launched, and calculation errors had accumulated to the point where the software component’s idea of its orientation had wandered hopelessly away from truth.

Systems (or components within systems) often have or embody expectations about the behaviors of its “information exchange” partners. The assumption of everything interacting with the errant component in the preceding example was that its accuracy did *not* degrade over time. The result was a system of parts that did not work together correctly to solve the problem they were supposed to.

The second concept we need to stress is what we mean by “interface.” Once again, we mean something beyond the simple case—a syntactic description of a component’s programs and the type and number of their parameters, most commonly realized as an API. That’s necessary for interoperability—heck, it’s necessary if you want your software to compile successfully—but it’s not sufficient. To illustrate this concept, we’ll use another “conversation” analogy. Has your partner or spouse ever come home, slammed the door, and when you ask what’s wrong, replied “Nothing!”? If so, then you should be able to appreciate the keen difference between syntax and semantics and the role of expectations in understanding how an entity behaves. Because we want interoperable systems and components, and not simply ones that compile together nicely, we require a higher bar for interfaces than just a statement of syntax. By “interface,” we mean the set of assumptions that you can safely make about an entity. For example, it’s a safe assumption that whatever’s wrong with your spouse/partner, it’s not “Nothing,” and you know that because that “interface” extends waybeyond just the words they say. And it’s also a safe assumption that nothing about our missile component’s accuracy degradation over time was in its API, and yet that was a critical part of its interface.

—PCC

Here are some of the reasons you might want systems to interoperate:

- Your system provides a service to be used by a collection of unknown systems. These systems need to interoperate with your system even though you may know nothing about them. An example is a service such as Google Maps.
- You are constructing capabilities from existing systems. For example, one of the existing systems is responsible for sensing its environment, another one is responsible for processing the raw data, a third is responsible for interpreting the data, and a final one is responsible for producing and distributing a representation of what was sensed. An example is a traffic sensing system where the input comes from individual vehicles, the raw data is processed into common units of measurement, is interpreted and fused, and traffic congestion information is broadcast.

These examples highlight two important aspects of interoperability:

1. *Discovery.* The consumer of a service must discover (possibly at runtime, possibly prior to runtime) the location, identity, and the interface of the service.
2. *Handling of the response.* There are three distinct possibilities:
 - The service reports back to the requester with the response.
 - The service sends its response on to another system.
 - The service broadcasts its response to any interested parties.

These elements, discovery and disposition of response, along with management of interfaces, govern our discussion of scenarios and tactics for interoperability.

Systems of Systems

If you have a group of systems that are interoperating to achieve a joint purpose, you have what is called a *system of systems* (SoS). An SoS is an arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities. [Table 6.1](#) shows a categorization of SoSs.

Table 6.1. Taxonomy of Systems of Systems*

Directed	SoS objectives, centralized management, funding, and authority for the overall SoS are in place. Systems are subordinated to the SoS.
Acknowledged	SoS objectives, centralized management, funding, and authority in place. However, systems retain their own management, funding, and authority in parallel with the SoS.
Collaborative	There are no overall objectives, centralized management, authority, responsibility, or funding at the SoS level. Systems voluntarily work together to address shared or common interests.
Virtual	Like collaborative, but systems don't know about each other.

* The taxonomy shown is an extension of work done by Mark Maier in 1998.

In directed and acknowledged SoSs, there is a deliberate attempt to create an SoS. The key difference is that in the former, there is SoS-level management that exercises control over the constituent systems, while in the latter, the constituent systems retain a high degree of autonomy in their own evolution. Collaborative and virtual systems of systems are more ad hoc, absent an overarching authority or source of funding and, in the case of a virtual SoS, even absent the knowledge about the scope and membership of the SoS.

The collaborative case is quite common. Consider the Google Maps example from the introduction. Google is the manager and funding authority for the map service. Each use of the maps in an application (an SoS) has its own management and funding authority, and there is no overall management of all of the applications that use Google Maps. The various organizations involved in the applications collaborate (either explicitly or implicitly) to enable the applications to work correctly.

A virtual SoS involves large systems and is much more ad hoc. For example, there are over 3,000 electric companies in the U.S. electric grid, each state has a public utility commission that oversees the utility companies operating in its state, and the federal Department of Energy provides some level of policy guidance. Many of the systems within the electric grid must interoperate, but there is no management authority for the overall system.

6.1. Interoperability General Scenario

The following are the portions of an interoperability general scenario:

- *Source of stimulus.* A system initiates a request to interoperate with another system.
- *Stimulus.* A request to exchange information among system(s).
- *Artifacts.* The systems that wish to interoperate.
- *Environment.* The systems that wish to interoperate are discovered at runtime or are known prior to runtime.
- *Response.* The request to interoperate results in the exchange of information. The information is understood by the receiving party both syntactically and semantically. Alternatively, the request is rejected and appropriate entities are notified. In either case, the request may be logged.

- **Response measure.** The percentage of information exchanges correctly processed or the percentage of information exchanges correctly rejected.

[Figure 6.1](#) gives an example: Our vehicle information system sends our current location to the traffic monitoring system. The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and broadcasts it. Our location information is correctly included with a probability of 99.9%.

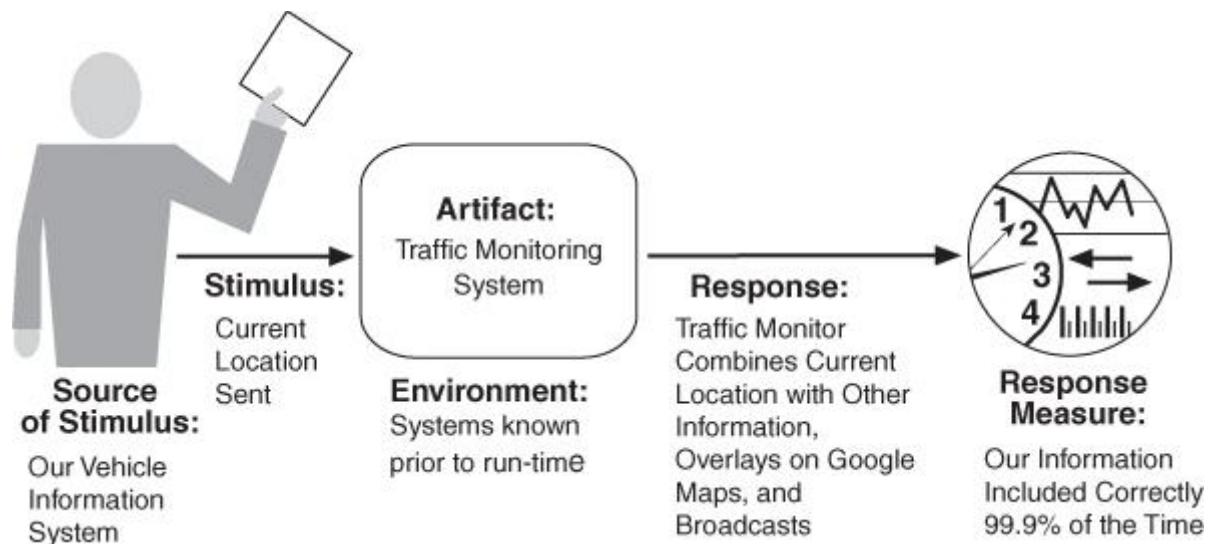


Figure 6.1. Sample concrete interoperability scenario

[Table 6.2](#) presents the possible values for each portion of an interoperability scenario.

Table 6.2. General Interoperability Scenario

Portion of Scenario	Possible Values
Source	A system initiates a request to interoperate with another system.
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate.
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime.
Response	One or more of the following: <ul style="list-style-type: none"> ▪ The request is (appropriately) rejected and appropriate entities (people or systems) are notified. ▪ The request is (appropriately) accepted and information is exchanged successfully. ▪ The request is logged by one or more of the involved systems.
Response Measure	One or more of the following: <ul style="list-style-type: none"> ▪ Percentage of information exchanges correctly processed ▪ Percentage of information exchanges correctly rejected

SOAP vs. REST

If you want to allow web-based applications to interoperate, you have two major off-the-shelf technology options today: (1) WS* and SOAP (which once stood for "Simple Object Access Protocol," but that acronym is no longer blessed) and (2) REST (which stands for "Representation State Transfer," and therefore is sometimes spelled ReST). How can we compare these technologies? What is each good for? What are the road hazards you need to be aware of? This is a bit of an apples-and-oranges comparison, but I will try to sketch the landscape.

SOAP is a protocol specification for XML-based information that distributed applications can use to exchange information and hence interoperate. It is most often accompanied by a set of SOA middleware interoperability standards and compliant implementations, referred to (collectively) as WS*. SOAP and WS* together define many standards, including the following:

- *An infrastructure for service composition.* SOAP can employ the Business Process Execution Language (BPEL) as a way to let developers express business processes that are implemented as WS* services.
- *Transactions.* There are several web-service standards for ensuring that transactions are properly managed: WS-AT, WS-BA, WS-CAF, and WS-Transaction.
- *Service discovery.* The Universal Description, Discovery and Integration (UDDI) language enables businesses to publish service listings and discover each other.
- *Reliability.* SOAP, by itself, does not ensure reliable message delivery. Applications that require such guarantees must use services compliant with SOAP's reliability standard: WS-Reliability.

SOAP is quite general and has its roots in a remote procedure call (RPC) model of interacting applications, although other models are certainly possible. SOAP has a simple type system, comparable to that found in the major programming languages. SOAP relies on HTTP and RPC for message transmission, but it could, in theory, be implemented on top of any communication protocol. SOAP does not mandate a service's method names, addressing model, or procedural conventions. Thus, choosing SOAP buys little actual interoperability between applications—it is just an information exchange standard. The interacting applications need to agree on *how to interpret* the payload, which is where you get semantic interoperability.

REST, on the other hand, is a client-server-based architectural style that is structured around a small set of create, read, update, delete (CRUD) operations (called POST, GET, PUT, DELETE respectively in the REST world) and a single addressing scheme (based on a URI, or uniform resource identifier). REST imposes few constraints on an architecture: SOAP offers completeness; REST offers simplicity.

REST is about state and state transfer and views the web (and the services that service-oriented systems can string together) as a huge network of information that is accessible by a single URI-based addressing scheme. There is no notion of type and hence no type checking in REST—it is up to the applications to get the semantics of interaction right.

Because REST interfaces are so simple and general, any HTTP client can talk to any HTTP server, using the REST operations (POST, GET, PUT, DELETE) with no further configuration. That buys you syntactic interoperability, but of course there must be organization-level agreement about what these programs actually do and what information they exchange. That is, semantic interoperability is not guaranteed between services just because both have REST interfaces.

REST, on top of HTTP, is meant to be self-descriptive and in the best case is a stateless protocol. Consider the following example, in REST, of a phone book service that allows someone to look up a person, given some unique identifier for that person:

<http://www.XYZdirectory.com/phonebook/UserInfo/99999>

The same simple lookup, implemented in SOAP, would be specified as something like the following:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/
  12/soap-envelope
  soap:encodingStyle="http://www.w3.org/2001/12/">
```

```

    soap-encoding">
<soap:Body pb="http://www.XYZdirectory.com/
    phonebook">
    <pb:GetUserInfo>
        <pb:UserIdentifier>99999</pb:UserIdentifier>
    </pb:GetUserInfo>
</soap:Body>
</soap:Envelope>

```

One aspect of the choice between SOAP and REST is whether you want to accept the complexity and restrictions of SOAP+WSDL (the Web Services Description Language) to get more standardized interoperability or if you want to avoid the overhead by using REST, but perhaps benefit from less standardization. What are the other considerations?

A message exchange in REST has somewhat fewer characters than a message exchange in SOAP. So one of the tradeoffs in the choice between REST and SOAP is the size of the individual messages. For systems exchanging a large number of messages, another tradeoff is between performance (favoring REST) and structured messages (favoring SOAP).

The decision to implement WS* or REST will depend on aspects such as the quality of service (QoS) required—WS* implementation has greater support for security, availability, and so on—and type of functionality. A RESTful implementation, because of its simplicity, is more appropriate for read-only functionality, typical of mashups, where there are minimal QoS requirements and concerns.

OK, so if you are building a service-based system, how do you choose? The truth is, you don't have to make a single choice, once and for all time; each technology is reasonably easy to use, at least for simple applications. And each has its strengths and weaknesses. Like everything else in architecture, it's all about the tradeoffs; your decision will likely hinge on the way those tradeoffs affect your system in your context.

—RK

6.2. Tactics for Interoperability

[Figure 6.2](#) shows the goal of the set of interoperability tactics.

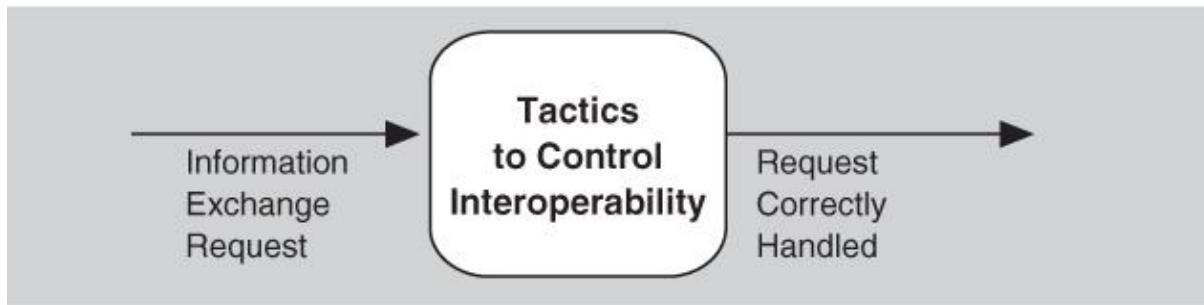


Figure 6.2. Goal of interoperability tactics

We identify two categories of interoperability tactics: locate and manage interfaces.

Locate

There is only one tactic in this category: *discover service*. It is used when the systems that interoperate must be discovered at runtime.

- *Discover service*. Locate a service through searching a known directory service. (By “service,” we simply mean a set of capabilities that is accessible via some kind of interface.) There may be multiple levels of indirection in this location process—that is, a

known location points to another location that in turn can be searched for the service. The service can be located by type of service, by name, by location, or by some other attribute.

Manage Interfaces

Managing interfaces consists of two tactics: *orchestrate* and *tailor interface*.

- *Orchestrate*. Orchestrate is a tactic that uses a control mechanism to coordinate and manage and sequence the invocation of particular services (which could be ignorant of each other). Orchestration is used when the interoperating systems must interact in a complex fashion to accomplish a complex task; orchestration “scripts” the interaction. Workflow engines are an example of the use of the orchestrate tactic. The mediator design pattern can serve this function for simple orchestration. Complex orchestration can be specified in a language such as BPEL.
- *Tailor interface*. Tailor interface is a tactic that adds or removes capabilities to an interface. Capabilities such as translation, adding buffering, or smoothing data can be added. Capabilities may be removed as well. An example of removing capabilities is to hide particular functions from untrusted users. The decorator pattern is an example of the tailor interface tactic.

The enterprise service bus that underlies many service-oriented architectures combines both of the manage interface tactics.

[Figure 6.3](#) shows a summary of the tactics to achieve interoperability.

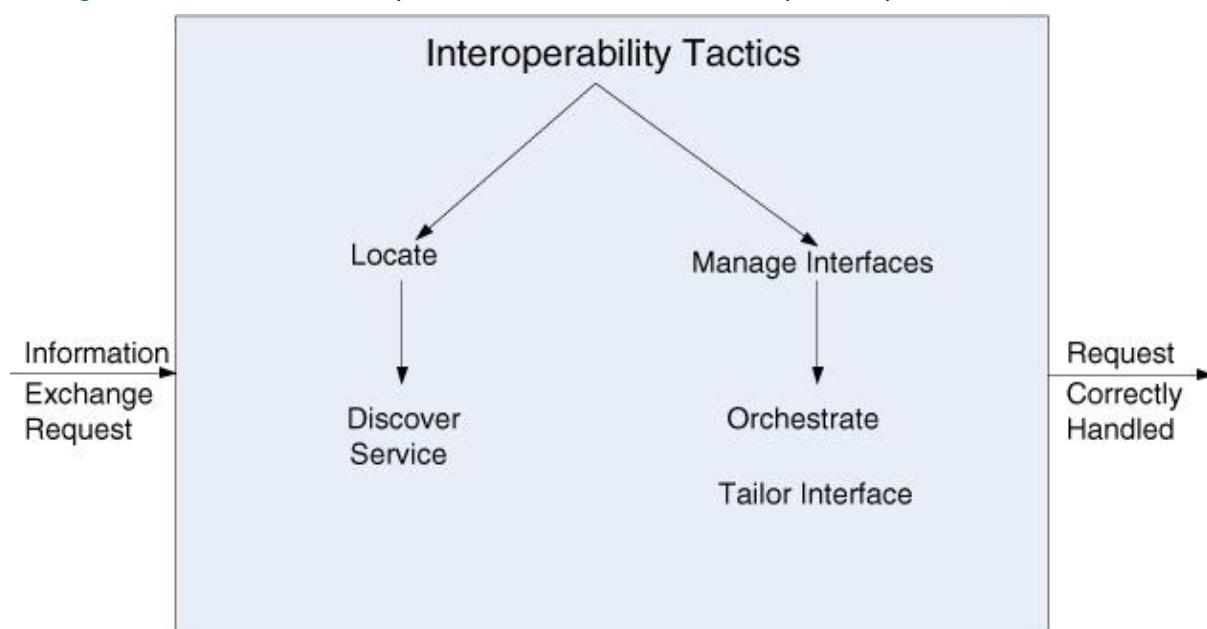


Figure 6.3. Summary of interoperability tactics

Why Standards Are Not Enough to Guarantee Interoperability *By Grace Lewis*

Developer of System A needs to exchange product data with System B. Developer A finds that there is an existing WS* web service interface for sending product data that among other fields contains price expressed in XML Schema as a decimal with two fraction digits. Developer A writes code to interact with the web service and the system works perfectly. However, after two weeks of operation, there is a huge discrepancy between the totals reported by System A and the totals reported by System B. After conversations between the two developers, they discover that System B expected to receive a price that included tax and System A was sending it without tax.

This is a simple example of why standards are not enough. The systems exchanged data perfectly because they both agreed that the price was a decimal with two fractions digits expressed in XML Schema and the message was sent via SOAP over HTTP (syntax)—standards used in the implementation of WS* web services—but they did not agree on whether the price included tax or not (semantics).

Of course, the only realistic approach to getting diverse applications to share information is by reaching agreements on the structure and function of the information to be shared. These agreements are often reflected in standards that provide a common interface that multiple vendors and application builders support. Standards have indeed been instrumental in achieving a significant level of interoperability that we rely on in almost every domain. However, while standards are useful and in many ways indispensable, expectations of what can be achieved through standards are unrealistic. Here are some of the challenges that organizations face related to standards and interoperability:

1. Ideally, every implementation of a standard should be identical and thus completely interoperable with any other implementation. However, this is far from reality. Standards, when incorporated into products, tools, and services, undergo customizations and extensions because every vendor wants to create a unique selling point as a competitive advantage.
2. Standards are often deliberately open-ended and provide extension points. The actual implementation of these extension points is left to the discretion of implementers, leading to proprietary implementations.
3. Standards, like any technology, have a life cycle of their own and evolve over time in compatible and noncompatible ways. Deciding when to adopt a new or revised standard is a critical decision for organizations. Committing to a new standard that is not ready or eventually not adopted by the community is a big risk for organizations. On the other hand, waiting too long may also become a problem, which can lead to unsupported products, incompatibilities, and workarounds, because everyone else is using the standard.
4. Within the software community, there are as many bad standards as there are engineers with opinions. Bad standards include underspecified, overspecified, inconsistently specified, unstable, or irrelevant standards.
5. It is quite common for standards to be championed by competing organizations, resulting in conflicting standards due to overlap or mutual exclusion.
6. For new and rapidly emerging domains, the argument often made is that standardization will be destructive because it will hinder flexibility: premature standardization will force the use of an inadequate approach and lead to abandoning other presumably better approaches. So what do organizations do in the meantime?

What these challenges illustrate is that because of the way in which standards are usually created and evolved, we cannot let standards drive our architectures. We need to architect systems first and then decide which standards can support desired system requirements and qualities. This approach allows standards to change and evolve without affecting the overall architecture of the system.

I once heard someone in a keynote address say that “The nice thing about standards is that there are so many to choose from.”

6.3. A Design Checklist for Interoperability

Table 6.3 is a checklist to support the design and analysis process for inter-operability.

Table 6.3. Checklist to Support the Design and Analysis Process for Interoperability

Category	Checklist
Allocation of Responsibilities	<p>Determine which of your system responsibilities will need to interoperate with other systems.</p>
	<p>Ensure that responsibilities have been allocated to detect a request to interoperate with known or unknown external systems.</p>
	<p>Ensure that responsibilities have been allocated to carry out the following tasks:</p>
	<ul style="list-style-type: none"> ▪ Accept the request ▪ Exchange information ▪ Reject the request ▪ Notify appropriate entities (people or systems) ▪ Log the request (for interoperability in an untrusted environment, logging for nonrepudiation is essential)
Coordination Model	<p>Ensure that the coordination mechanisms can meet the critical quality attribute requirements. Considerations for performance include the following:</p>
	<ul style="list-style-type: none"> ▪ Volume of traffic on the network both created by the systems under your control and generated by systems not under your control ▪ Timeliness of the messages being sent by your systems ▪ Currency of the messages being sent by your systems ▪ Jitter of the messages' arrival times ▪ Ensure that all of the systems under your control make assumptions about protocols and underlying networks that are consistent with the systems not under your control.

Data Model	<p>Determine the syntax and semantics of the major data abstractions that may be exchanged among interoperating systems.</p> <p>Ensure that these major data abstractions are consistent with data from the interoperating systems. (If your system's data model is confidential and must not be made public, you may have to apply transformations to and from the data abstractions of systems with which yours interoperates.)</p>
Mapping among Architectural Elements	<p>For interoperability, the critical mapping is that of components to processors. Beyond the necessity of making sure that components that communicate externally are hosted on processors that can reach the network, the primary considerations deal with meeting the security, availability, and performance requirements for the communication. These will be dealt with in their respective chapters.</p>
Resource Management	<p>Ensure that interoperation with another system (accepting a request and/or rejecting a request) can never exhaust critical system resources (e.g., can a flood of such requests cause service to be denied to legitimate users?).</p> <p>Ensure that the resource load imposed by the communication requirements of interoperation is acceptable.</p> <p>Ensure that if interoperation requires that resources be shared among the participating systems, an adequate arbitration policy is in place.</p>
Binding Time	<p>Determine the systems that may interoperate, and when they become known to each other. For each system over which you have control:</p> <ul style="list-style-type: none"> ▪ Ensure that it has a policy for dealing with binding to both known and unknown external systems. ▪ Ensure that it has mechanisms in place to reject unacceptable bindings and to log such requests. ▪ In the case of late binding, ensure that mechanisms will support the discovery of relevant new services or protocols, or the sending of information using chosen protocols.
Choice of Technology	<p>For any of your chosen technologies, are they “visible” at the interface boundary of a system? If so, what interoperability effects do they have? Do they support, undercut, or have no effect on the interoperability scenarios that apply to your system? Ensure the effects they have are acceptable.</p> <p>Consider technologies that are designed to support interoperability, such as web services. Can they be used to satisfy the interoperability requirements for the systems under your control?</p>

6.4. Summary

Interoperability refers to the ability of systems to usefully exchange information. These systems may have been constructed with the intention of exchanging information, they may be existing systems that are desired to exchange information, or they may provide general services without knowing the details of the systems that wish to utilize those services.

The general scenario for interoperability provides the details of these different cases. In any interoperability case, the goal is to intentionally exchange information or reject the request to exchange information.

Achieving interoperability involves the relevant systems locating each other and then managing the interfaces so that they can exchange information.

6.5. For Further Reading

An SEI report gives a good overview of interoperability, and it highlights some of the “maturity frameworks” for interoperability [[Brownword 04](#)].

The various WS* services are being developed under the auspices of the World Wide Web Consortium (W3C) and can be found at www.w3.org/2002/ws.

Systems of systems are of particular interest to the U.S. Department of Defense. An engineering guide can be found at [[ODUSD 08](#)].

6.6. Discussion Questions

1. Find a web service mashup. Write several concrete interoperability scenarios for this system.
2. What is the relationship between interoperability and the other quality attributes highlighted in this book? For example, if two systems fail to exchange information properly, could a security flaw result? What other quality attributes seem strongly related (at least potentially) to interoperability?
3. Is a service-oriented system a system of systems? If so, describe a service-oriented system that is directed, one that is acknowledged, one that is collaborative, and one that is virtual.
4. Universal Description, Discovery, and Integration (UDDI) was touted as a discovery service, but commercial support for UDDI is being withdrawn. Why do you suppose this is? Does it have anything to do with the quality attributes delivered or not delivered by UDDI solutions?
5. Why has the importance of orchestration grown in recent years?
6. If you are a technology producer, what are the advantages and disadvantages of adhering to interoperability standards? Why would a producer not adhere to a standard?
7. With what other systems will an automatic teller machine need to interoperate? How would you change your automatic teller system design to accommodate these other systems?

7. Modifiability

Adapt or perish, now as ever, is nature's inexorable imperative.

—H.G. Wells

Change happens.

Study after study shows that most of the cost of the typical software system occurs after it has been initially released. If change is the only constant in the universe, then software change is not only constant but ubiquitous. Changes happen to add new features, to change or even retire old ones. Changes happen to fix defects, tighten security, or improve performance. Changes happen to enhance the user's experience. Changes happen to embrace new technology, new platforms, new protocols, new standards. Changes happen to make systems work together, even if they were never designed to do so.

Modifiability is about change, and our interest in it centers on the cost and risk of making changes. To plan for modifiability, an architect has to consider four questions:

- *What can change?* A change can occur to any aspect of a system: the functions that the system computes, the platform (the hardware, operating system, middleware), the environment in which the system operates (the systems with which it must interoperate, the protocols it uses to communicate with the rest of the world), the qualities the system exhibits (its performance, its reliability, and even its future modifications), and its capacity (number of users supported, number of simultaneous operations).
- *What is the likelihood of the change?* One cannot plan a system for all potential changes—the system would never be done, or if it was done it would be far too expensive and would likely suffer quality attribute problems in other dimensions. Although anything *might* change, the architect has to make the tough decisions about which changes are likely, and hence which changes are to be supported, and which are not.
- *When is the change made and who makes it?* Most commonly in the past, a change was made to source code. That is, a developer had to make the change, which was tested and then deployed in a new release. Now, however, the question of when a change is made is intertwined with the question of who makes it. An end user changing the screen saver is clearly making a change to one of the aspects of the system. Equally clear, it is not in the same category as changing the system so that it can be used over the web rather than on a single machine. Changes can be made to the implementation (by modifying the source code), during compile (using compile-time switches), during build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting), or during execution (by parameter settings, plugins, etc.). A change can also be made by a developer, an end user, or a system administrator.
- *What is the cost of the change?* Making a system more modifiable involves two types of cost:
 - The cost of introducing the mechanism(s) to make the system more modifiable
 - The cost of making the modification using the mechanism(s)

For example, the simplest mechanism for making a change is to wait for a change request to come in, then change the source code to accommodate the request. The cost of introducing the mechanism is zero; the cost of exercising it is the cost of changing the source code and revalidating the system. At the other end of the spectrum is an application generator, such as a user interface builder. The builder takes as input a description of the designer user interface produced through direct manipulation techniques and produces (usually) source code. The cost of introducing the mechanism is the cost of constructing the UI builder, which can be substantial. The cost of using the mechanism is the cost of producing the input to feed the builder (cost can be substantial or negligible), the cost of running the builder (approximately zero), and then the cost of whatever testing is performed on the result (usually much less than usual).

For N similar modifications, a simplified justification for a change mechanism is that

$$N \times \text{Cost of making the change without the mechanism} \leq \text{Cost of installing the mechanism} + (N \times \text{Cost of making the change using the mechanism}).$$

N is the anticipated number of modifications that will use the modifiability mechanism, but N is a prediction. If fewer changes than expected come in, then an expensive modification mechanism may not be warranted. In addition, the cost of creating the modifiability mechanism could be applied elsewhere—in adding functionality, in improving the performance, or even in nonsoftware investments such as buying tech stocks. Also, the equation does not take time into account. It might be cheaper in the long run to build a sophisticated change-handling mechanism, but you might not be able to wait for that.

7.1. Modifiability General Scenario

From these considerations, we can see the portions of the modifiability general scenario:

- *Source of stimulus.* This portion specifies who makes the change: the developer, a system administrator, or an end user.
- *Stimulus.* This portion specifies the change to be made. A change can be the addition of a function, the modification of an existing function, or the deletion of a function. (For this categorization, we regard fixing a defect as changing a function, which presumably wasn't working correctly as a result of the defect.) A change can also be made to the qualities of the system: making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Accommodating an increasing number of simultaneous users is a frequent requirement. Finally, changes may happen to accommodate new technology of some sort, the most common of which is porting the system to a different type of computer or communication network.
- *Artifact.* This portion specifies what is to be changed: specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates.
- *Environment.* This portion specifies when the change can be made: design time, compile time, build time, initiation time, or runtime.
- *Response.* Make the change, test it, and deploy it.
- *Response measure.* All of the possible responses take time and cost money; time and money are the most common response measures. Although both sound simple to measure, they aren't. You can measure calendar time or staff time. But do you measure the time it takes for the change to wind its way through configuration control boards and approval authorities (some of whom may be outside your organization), or merely the time it takes your engineers to make the change? Cost usually means direct outlay, but it might also include opportunity cost of having your staff work on changes instead of other tasks. Other measures include the extent of the change (number of modules or other artifacts affected) or the number of new defects introduced by the change, or the effect on other quality attributes. If the change is being made by a user, you may wish to measure the efficacy of the change mechanisms provided, which somewhat overlaps with measures of usability (see [Chapter 11](#)).

[Figure 7.1](#) illustrates a concrete modifiability scenario: The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.

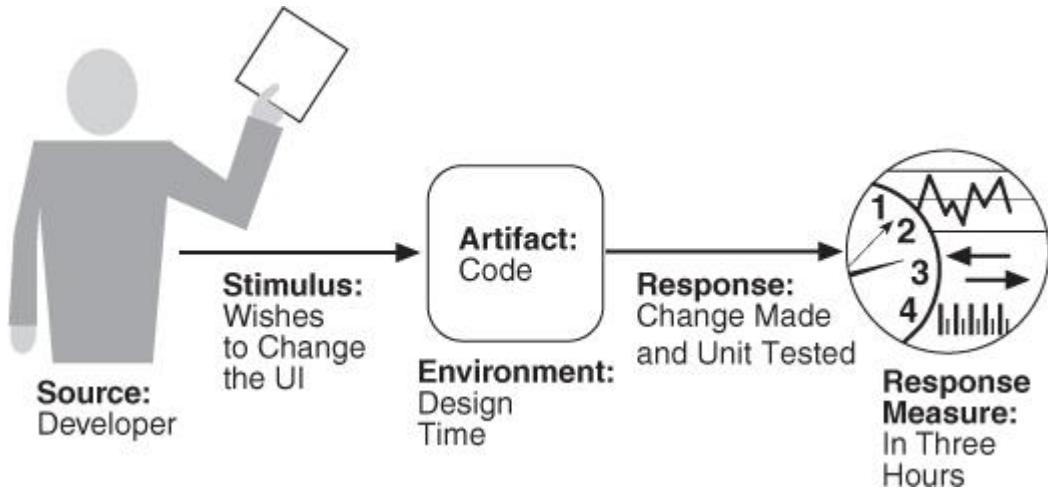


Figure 7.1. Sample concrete modifiability scenario

Table 7.1 enumerates the elements of the general scenario that characterize modifiability.

Table 7.1. Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, ...
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: <ul style="list-style-type: none"> ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: <ul style="list-style-type: none"> ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

7.2. Tactics for Modifiability

Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes. [Figure 7.2](#) shows this relationship.

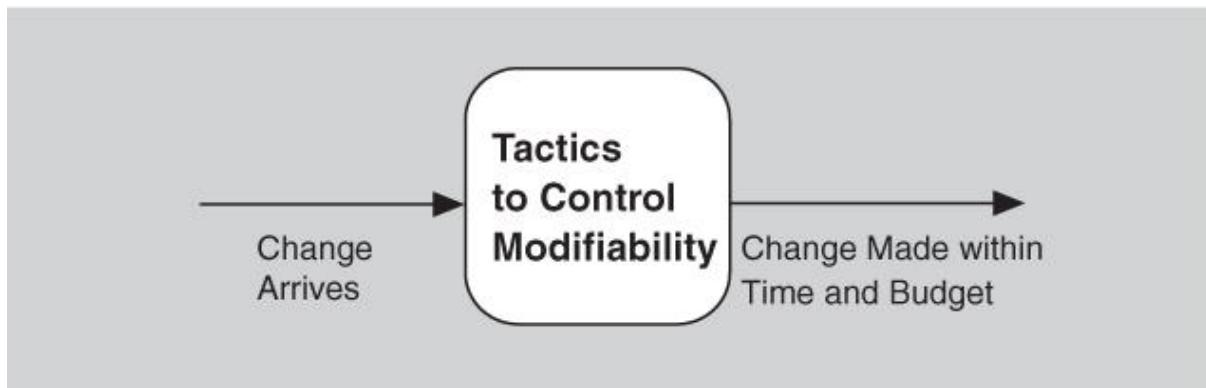


Figure 7.2. The goal of modifiability tactics

To understand modifiability, we begin with coupling and cohesion.

Modules have responsibilities. When a change causes a module to be modified, its responsibilities are changed in some way. Generally, a change that affects one module is easier and less expensive than if it changes more than one module. However, if two modules' responsibilities overlap in some way, then a single change may well affect them both. We can measure this overlap by measuring the probability that a modification to one module will propagate to the other. This is called *coupling*, and high coupling is an enemy of modifiability.

Cohesion measures how strongly the responsibilities of a module are related. Informally, it measures the module's "unity of purpose." Unity of purpose can be measured by the change scenarios that affect a module. The cohesion of a module is the probability that a change scenario that affects a responsibility will also affect other (different) responsibilities. The higher the cohesion, the lower the probability that a given change will affect multiple responsibilities. High cohesion is good; low cohesion is bad. The definition allows for two modules with similar purposes each to be cohesive.

Given this framework, we can now identify the parameters that we will use to motivate modifiability tactics:

- *Size of a module.* Tactics that split modules will reduce the cost of making a modification to the module that is being split as long as the split is chosen to reflect the type of change that is likely to be made.
- *Coupling.* Reducing the strength of the coupling between two modules A and B will decrease the expected cost of any modification that affects A. Tactics that reduce coupling are those that place intermediaries of various sorts between modules A and B.
- *Cohesion.* If module A has a low cohesion, then cohesion can be improved by removing responsibilities unaffected by anticipated changes.

Finally we need to be concerned with when in the software development life cycle a change occurs. If we ignore the cost of preparing the architecture for the modification, we prefer that a change is bound as late as possible. Changes can only be successfully made (that is, quickly and at lowest cost) late in the life cycle if the architecture is suitably prepared to accommodate them. Thus the fourth and final parameter in a model of modifiability is this:

- *Binding time of modification.* An architecture that is suitably equipped to accommodate modifications late in the life cycle will, on average, cost less than an architecture that forces the same modification to be made earlier. The preparedness of the system means that some costs will be zero, or very low, for late life-cycle modifications. This, however, neglects the cost of preparing the architecture for the late binding.

Now we may understand tactics and their consequences as affecting one or more of the previous parameters: reducing the size of a module, increasing cohesion, reducing coupling, and deferring binding time. These tactics are shown in [Figure 7.3](#).

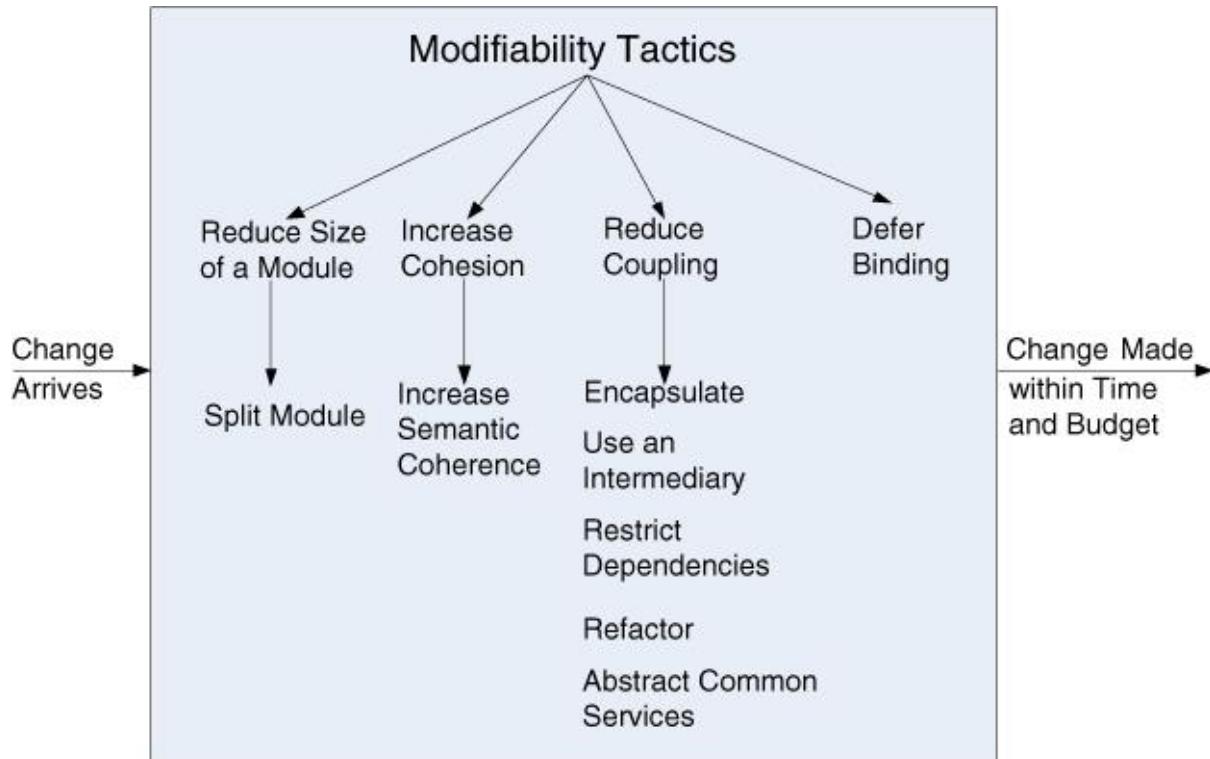


Figure 7.3. Modifiability tactics

Reduce the Size of a Module

- *Split module.* If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

Several tactics involve moving responsibilities from one module to another. The purpose of moving a responsibility from one module to another is to reduce the likelihood of side effects affecting other responsibilities in the original module.

- *Increase semantic coherence.* If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module. One method for identifying responsibilities to be moved is to hypothesize likely changes that affect a module. If some responsibilities are not affected by these changes, then those responsibilities should probably be removed.

Reduce Coupling

We now turn to tactics that reduce the coupling between modules.

- *Encapsulate.* Encapsulation introduces an explicit interface to a module. This interface includes an application programming interface (API) and its associated responsibilities, such as "perform a syntactic transformation on an input parameter to an internal representation." Perhaps the most common modifiability tactic, encapsulation reduces the probability that a change to one module propagates to other modules. The strengths of coupling that previously went to the module now go to the interface for the module. These strengths are, however, reduced because the interface limits the ways in which external responsibilities can interact with the module (perhaps through a wrapper). The external responsibilities can now only directly interact with the module through the exposed interface (indirect interactions, however, such as dependence on quality of

service, will likely remain unchanged). Interfaces designed to increase modifiability should be abstract with respect to the details of the module that are likely to change—that is, they should hide those details.

- *Use an intermediary* breaks a dependency. Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary. The type of intermediary depends on the type of dependency. For example, a publish-subscribe intermediary will remove the data producer's knowledge of its consumers. So will a shared data repository, which separates readers of a piece of data from writers of that data. In a service-oriented architecture in which services discover each other by dynamic lookup, the directory service is an intermediary.
- *Restrict dependencies* is a tactic that restricts the modules that a given module interacts with or depends on. In practice this tactic is achieved by restricting a module's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (restricting access to only authorized modules). This tactic is seen in layered architectures, in which a layer is only allowed to use lower layers (sometimes only the next lower layer) and in the use of wrappers, where external entities can only see (and hence depend on) the wrapper and not the internal functionality that it wraps.
- *Refactor* is a tactic undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other. Code refactoring is a mainstay practice of Agile development projects, as a cleanup step to make sure that teams have not produced duplicative or overly complex code; however, the concept applies to architectural elements as well. Common responsibilities (and the code that implements them) are “factored out” of the modules where they exist and assigned an appropriate home of their own. By co-locating common responsibilities—that is, making them submodules of the same parent module—the architect can reduce coupling.
- *Abstract common services*. In the case where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form. Any modification to the (common) service would then need to occur just in one place, reducing modification costs. A common way to introduce an abstraction is by parameterizing the description (and implementation) of a module's activities. The parameters can be as simple as values for key variables or as complex as statements in a specialized language that are subsequently interpreted.

Defer Binding

Because the work of people is almost always more expensive than the work of computers, letting computers handle a change as much as possible will almost always reduce the cost of making that change. If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.

Parameters are perhaps the best-known mechanism for introducing flexibility, and that is reminiscent of the abstract common services tactic. A parameterized function $f(a, b)$ is more general than the similar function $f(a)$ that assumes $b = 0$. When we bind the value of some parameters at a different phase in the life cycle than the one in which we defined the parameters, we are applying the defer binding tactic.

In general, the later in the life cycle we can bind values, the better. However, putting the mechanisms in place to facilitate that late binding tends to be more expensive—yet another tradeoff. And so the equation on page 118 comes into play. We want to bind as late as possible, as long as the mechanism that allows it is cost-effective.

Tactics to bind values at compile time or build time include these:

- Component replacement (for example, in a build script or makefile)
- Compile-time parameterization
- Aspects

Tactics to bind values at deployment time include this:

- Configuration-time binding

Tactics to bind values at startup or initialization time include this:

- Resource files

Tactics to bind values at runtime include these:

- Runtime registration
- Dynamic lookup (e.g., for services)
- Interpret parameters
- Startup time binding
- Name servers
- Plug-ins
- Publish-subscribe
- Shared repositories
- Polymorphism

Separating building a mechanism for modifiability from using the mechanism to make a modification admits the possibility of different stakeholders being involved—one stakeholder (usually a developer) to provide the mechanism and another stakeholder (an installer, for example, or a user) to exercise it later, possibly in a completely different life-cycle phase. Installing a mechanism so that someone else can make a change to the system without having to change any code is sometimes called *externalizing* the change.

7.3. A Design Checklist for Modifiability

[Table 7.2](#) is a checklist to support the design and analysis process for modifiability.

Table 7.2. Checklist to Support the Design and Analysis Process for Modifiability

Category	Checklist
Allocation of Responsibilities	<p>Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes:</p> <ul style="list-style-type: none"> ▪ Determine the responsibilities that would need to be added, modified, or deleted to make the change. ▪ Determine what responsibilities are impacted by the change. ▪ Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.
Coordination Model	<p>Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number set of modules.</p> <p>Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules.</p> <p>For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish-subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.</p>
Data Model	<p>Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction.</p>

For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties.

For each potential change or category of change:

- Determine which data abstractions would need to be added, modified, or deleted to make the change.
- Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions.
- Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, their creation, initialization, persistence, manipulation, translation, or destruction.
- Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes.

Design your data model so that items allocated to each element of the data model are likely to change together.

Mapping among Architectural Elements

Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time.

Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of the following, for example:

- Execution dependencies
- Assignment of data to databases
- Assignment of runtime elements to processes, threads, or processors

	<p>Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.</p>
Resource Management	<p>Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example:</p> <ul style="list-style-type: none"> ▪ Determining what changes might introduce new resources or remove old ones or affect existing resource usage ▪ Determining what resource limits will change and how <p>Ensure that the resources after the modification are sufficient to meet the system requirements.</p> <p>Encapsulate all resource managers and ensure that the policies implemented by those resource managers are themselves encapsulated and bindings are deferred to the extent possible.</p>
Binding Time	<p>For each change or category of change:</p> <ul style="list-style-type: none"> ▪ Determine the latest time at which the change will need to be made. ▪ Choose a defer-binding mechanism (see Section 7.2) that delivers the appropriate capability at the time chosen. ▪ Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. Use the equation on page 118 to assess your choice of mechanism. ▪ Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	<p>Determine what modifications are made easier or harder by your technology choices.</p> <ul style="list-style-type: none"> ▪ Will your technology choices help to make, test, and deploy modifications? ▪ How easy is it to modify your choice of technologies (in case some of these technologies change or become obsolete)? <p>Choose your technologies to support the most likely modifications. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.</p>

7.4. Summary

Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.

Changes can be made by developers, installers, or end users, and these changes need to be prepared for. There is a cost of preparing for change as well as a cost of making a change. The modifiability tactics are designed to prepare for subsequent changes.

Tactics to reduce the cost of making a change include making modules smaller, increasing cohesion, and reducing coupling. Deferring binding will also reduce the cost of making a change.

Reducing coupling is a standard category of tactics that includes encapsulating, using an intermediary, restricting dependencies, co-locating related responsibilities, refactoring, and abstracting common services.

Increasing cohesion is another standard tactic that involves separating responsibilities that do not serve the same purpose.

Defer binding is a category of tactics that affect build time, load time, initialization time, or runtime.

7.5. For Further Reading

Serious students of software engineering should read two early papers about designing for modifiability. The first is Edsger Dijkstra's 1968 paper about the T.H.E. operating system [\[Dijkstra 68\]](#), which is the first paper that talks about designing systems to be layered, and the modifiability benefits it brings. The second is David Parnas's 1972 paper that introduced the concept of information hiding [\[Parnas 72\]](#). Parnas prescribed defining modules not by their functionality but by their ability to internalize the effects of changes.

The tactics that we have presented in this chapter are a variant on those introduced by [\[Bachmann 07\]](#).

Additional tactics for modifiability within the avionics domain can be found in [\[EOSAN 07\]](#), published by the European Organization for the Safety of Air Navigation.

7.6. Discussion Questions

1. Modifiability comes in many flavors and is known by many names. Find one of the IEEE or ISO standards dealing with quality attributes and compile a list of quality attributes that refer to some form of modifiability. Discuss the differences.
2. For each quality attribute that you discovered as a result of the previous question, write a modifiability scenario that expresses it.
3. In a certain metropolitan subway system, the ticket machines accept cash but do not give change. There is a separate machine that dispenses change but does not sell tickets. In an average station there are six or eight ticket machines for every change machine. What modifiability tactics do you see at work in this arrangement? What can you say about availability?
4. For the subway system in the previous question, describe the specific form of modifiability (using a modifiability scenario) that seems to be the aim of arranging the ticket and change machines as described.
5. A *wrapper* is a common aid to modifiability. A wrapper for a component is the only element allowed to use that component; every other piece of software uses the component's services by going through the wrapper. The wrapper transforms the data or control information for the component it wraps. For example, a component may expect input using English measures but find itself in a system in which all of the other components produce metric measures. A wrapper could be employed to translate. What modifiability tactics does a wrapper embody?
6. Once an intermediary has been introduced into an architecture, some modules may attempt to circumvent it, either inadvertently (because they are not aware of the intermediary) or intentionally (for performance, for convenience, or out of habit). Discuss some architectural means to prevent inadvertent circumvention of an intermediary.
7. In some projects, *deployability* is an important quality attribute that measures how easy it is to get a new version of the system into the hands of its users. This might mean a trip to your auto dealer or transmitting updates over the Internet. It also includes the time it takes to install the update once it arrives. In projects that measure deployability separately, should the cost of a modification stop when the new version is ready to ship? Justify your answer.

- 8.** The abstract common services tactic is intended to reduce coupling, but it also might reduce cohesion. Discuss.
- 9.** Identify particular change scenarios for an automatic teller machine. What modifications would you make to your automatic teller machine design to accommodate these changes?

8. Performance

An ounce of performance is worth pounds of promises.

—Mae West

It's about time.

Performance, that is: It's about time and the software system's ability to meet timing requirements. When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence of discussing performance.

Web-based system events come in the form of requests from users (numbering in the tens or tens of millions) via their clients such as web browsers. In a control system for an internal combustion engine, events come from the operator's controls and the passage of time; the system must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and efficiency and minimize pollution.

For a web-based system, the desired response might be expressed as number of transactions that can be processed in a minute. For the engine control system, the response might be the allowable variation in the firing time. In each case, the pattern of events arriving and the pattern of responses can be characterized, and this characterization forms the language with which to construct performance scenarios.

For much of the history of software engineering, performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. As the price/performance ratio of hardware continues to plummet and the cost of developing software continues to rise, other qualities have emerged as important competitors to performance.

Nevertheless, all systems have performance requirements, even if they are not expressed. For example, a word processing tool may not have any explicit performance requirement, but no doubt everyone would agree that waiting an hour (or a minute, or a second) before seeing a typed character appear on the screen is unacceptable. Performance continues to be a fundamentally important quality attribute for all software.

Performance is often linked to scalability—that is, increasing your system's capacity for work, while still performing well. Technically, scalability is making your system easy to change in a particular way, and so is a kind of modifiability. In addition, we address scalability explicitly in [Chapter 12](#).

8.1. Performance General Scenario

A performance scenario begins with an event arriving at the system. Responding correctly to the event requires resources (including time) to be consumed. While this is happening, the system may be simultaneously servicing other events.

Concurrency

Concurrency is one of the more important concepts that an architect must understand and one of the least-taught in computer science courses. Concurrency refers to operations occurring in parallel. For example, suppose there is a thread that executes the statements

```
x := 1;  
x++;
```

and another thread that executes the same statements. What is the value of x after both threads have executed those statements? It could be either 2 or 3. I leave it to you to figure out how the value 3 could occur—or should I say I interleave it to you?

Concurrency occurs any time your system creates a new thread, because threads, by definition, are independent sequences of control. Multi-tasking on your system is supported by independent threads. Multiple users are simultaneously supported on your system through the use of threads. Concurrency also occurs any time your system is executing on more than one processor, whether the processors are packaged separately or as multi-core processors. In addition, you must consider concurrency when parallel algorithms, parallelizing infrastructures such as map-reduce, or NoSQL databases are used by your system, or you utilize one of a variety of concurrent scheduling algorithms. In other words, concurrency is a tool available to you in many ways.

Concurrency, when you have multiple CPUs or wait states that can exploit it, is a good thing. Allowing operations to occur in parallel improves performance, because delays introduced in one thread allow the processor to progress on another thread. But because of the interleaving phenomenon just described (referred to as a *race condition*), concurrency must also be carefully managed by the architect.

As the example shows, race conditions can occur when there are two threads of control and there is shared state. The management of concurrency frequently comes down to managing how state is shared. One technique for preventing race conditions is to use locks to enforce sequential access to state. Another technique is to partition the state based on the thread executing a portion of code. That is, if there are two instances of x in our example, x is not shared by the two threads and there will not be a race condition.

Race conditions are one of the hardest types of bugs to discover; the occurrence of the bug is sporadic and depends on (possibly minute) differences in timing. I once had a race condition in an operating system that I could not track down. I put a test in the code so that the next time the race condition occurred, a debugging process was triggered. It took over a year for the bug to recur so that the cause could be determined.

Do not let the difficulties associated with concurrency dissuade you from utilizing this very important technique. Just use it with the knowledge that you must carefully identify critical sections in your code and ensure that race conditions will not occur in those sections.

—LB

Events can arrive in predictable patterns or mathematical distributions, or be unpredictable. An arrival pattern for events is characterized as *periodic*, *stochastic*, or *sporadic*:

- Periodic events arrive predictably at regular time intervals. For instance, an event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems.
- Stochastic arrival means that events arrive according to some probabilistic distribution.
- Sporadic events arrive according to a pattern that is neither periodic nor stochastic. Even these can be characterized, however, in certain circumstances. For example, we might know that at most 600 events will occur in a minute, or that there will be at least 200 milliseconds between the arrival of any two events. (This might describe a system in which events correspond to keyboard strokes from a human user.) These are helpful characterizations, even though we don't know when any single event will arrive.

The response of the system to a stimulus can be measured by the following:

- *Latency*. The time between the arrival of the stimulus and the system's response to it.
- *Deadlines in processing*. In the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline.
- The *throughput* of the system, usually given as the number of transactions the system can process in a unit of time.
- The *jitter* of the response—the allowable variation in latency.
- The *number of events not processed* because the system was too busy to respond.

From these considerations we can now describe the individual portions of a general scenario for performance:

- *Source of stimulus.* The stimuli arrive either from external (possibly multiple) or internal sources.
- *Stimulus.* The stimuli are the event arrivals. The arrival pattern can be periodic, stochastic, or sporadic, characterized by numeric parameters.
- *Artifact.* The artifact is the system or one or more of its components.
- *Environment.* The system can be in various operational modes, such as normal, emergency, peak load, or overload.
- *Response.* The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode).
- *Response measure.* The response measures are the time it takes to process the arriving events (latency or a deadline), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate).

The general scenario for performance is summarized in [Table 8.1](#).

Table 8.1. Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

[Figure 8.1](#) gives an example concrete performance scenario: Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

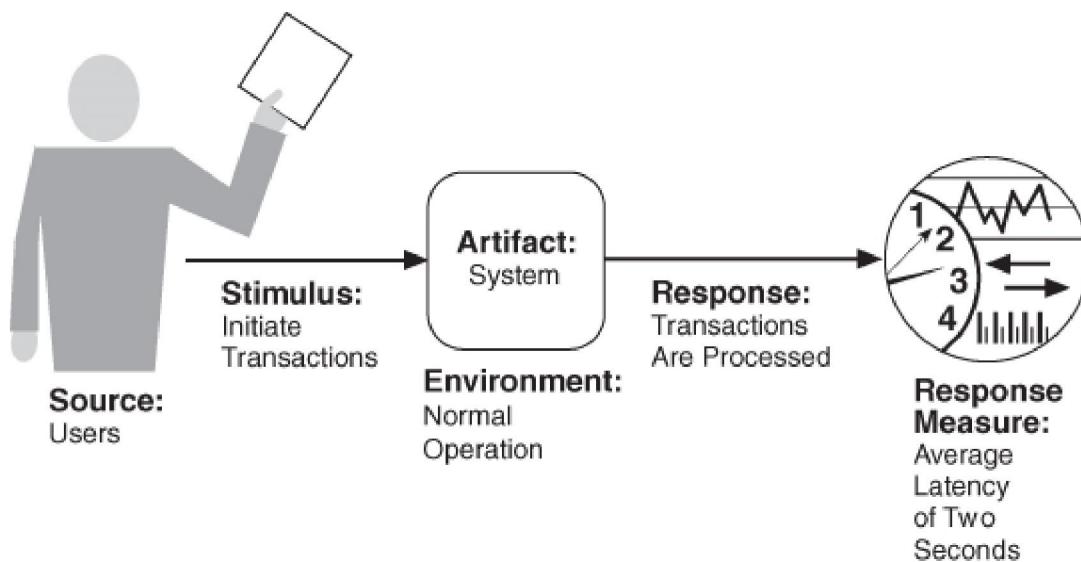


Figure 8.1. Sample concrete performance scenario

8.2. Tactics for Performance

The goal of performance tactics is to generate a response to an event arriving at the system within some time-based constraint. The event can be single or a stream and is the trigger to perform computation. Performance tactics control the time within which a response is generated, as illustrated in [Figure 8.2](#).

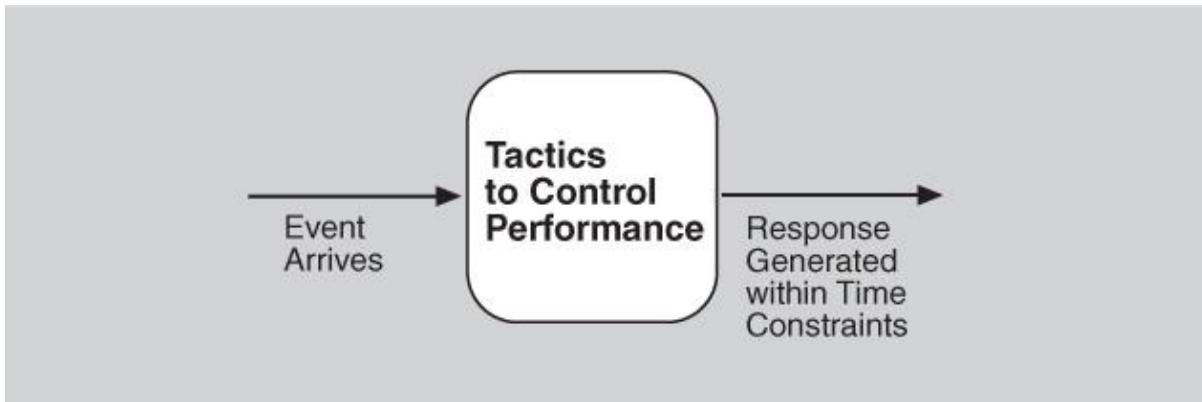


Figure 8.2. The goal of performance tactics

At any instant during the period after an event arrives but before the system's response to it is complete, either the system is working to respond to that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: processing time (when the system is working to respond) and blocked time (when the system is unable to respond).

- *Processing time.* Processing consumes resources, which takes time. Events are handled by the execution of one or more components, whose time expended is a resource. Hardware resources include CPU, data stores, network communication bandwidth, and memory. Software resources include entities defined by the system under design. For example, buffers must be managed and access to critical sections¹ must be made sequential.

1. A critical section is a section of code in a multi-threaded system in which at most one thread may be active at any time.

For example, suppose a message is generated by one component. It might be placed on the network, after which it arrives at another component. It is then placed in a buffer; transformed in some fashion; processed according to some algorithm; transformed for output; placed in an output buffer; and sent onward to another component, another system, or some actor. Each of these steps consumes resources and time and contributes to the overall latency of the processing of that event.

Different resources behave differently as their utilization approaches their capacity—that is, as they become saturated. For example, as a CPU becomes more heavily loaded, performance usually degrades fairly steadily. On the other hand, when you start to run out of memory, at some point the page swapping becomes overwhelming and performance crashes suddenly.

- *Blocked time.* A computation can be blocked because of contention for some needed resource, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available:

- *Contention for resources.* Many resources can only be used by a single client at a time. This means that other clients must wait for access to those resources. [Figure 8.2](#) shows events arriving at the system. These events may be in a single stream or in multiple streams. Multiple streams vying for the same resource or different events in the same stream vying for the same resource contribute to latency. The more contention for a resource, the more likelihood of latency being introduced.
- *Availability of resources.* Even in the absence of contention, computation cannot proceed if a resource is unavailable. Unavailability may be caused by the resource being offline or by failure of the component or for some other reason. In any case, you must identify places where resource unavailability might cause a significant

contribution to overall latency. Some of our tactics are intended to deal with this situation.

- *Dependency on other computation.* A computation may have to wait because it must synchronize with the results of another computation or because it is waiting for the results of a computation that it initiated. If a component calls another component and must wait for that component to respond, the time can be significant if the called component is at the other end of a network (as opposed to co-located on the same processor).

With this background, we turn to our tactic categories. We can either reduce demand for resources or make the resources we have handle the demand more effectively:

- *Control resource demand.* This tactic operates on the demand side to produce smaller demand on the resources that will have to service the events.
- *Manage resources.* This tactic operates on the response side to make the resources at hand work more effectively in handling the demands put to them.

Control Resource Demand

One way to increase performance is to carefully manage the demand for resources. This can be done by reducing the number of events processed by enforcing a sampling rate, or by limiting the rate at which the system responds to events. In addition, there are a number of techniques for ensuring that the resources that you do have are applied judiciously:

- *Manage sampling rate.* If it is possible to reduce the sampling frequency at which a stream of environmental data is captured, then demand can be reduced, typically with some attendant loss of fidelity. This is common in signal processing systems where, for example, different codecs can be chosen with different sampling rates and data formats. This design choice is made to maintain predictable levels of latency; you must decide whether having a lower fidelity but consistent stream of data is preferable to losing packets of data.
- *Limit event response.* When discrete events arrive at the system (or element) too rapidly to be processed, then the events must be queued until they can be processed. Because these events are discrete, it is typically not desirable to "downsample" them. In such a case, you may choose to process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed. This tactic could be triggered by a queue size or processor utilization measure exceeding some warning level. If you adopt this tactic and it is unacceptable to lose any events, then you must ensure that your queues are large enough to handle the worst case. If, on the other hand, you choose to drop events, then you need to choose a policy for handling this situation: Do you log the dropped events, or simply ignore them? Do you notify other systems, users, or administrators?
- *Prioritize events.* If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them. If there are not enough resources available to service them when they arise, low-priority events might be ignored. Ignoring events consumes minimal resources (including time), and thus increases performance compared to a system that services all events all the time. For example, a building management system may raise a variety of alarms. Life-threatening alarms such as a fire alarm should be given higher priority than informational alarms such as a room is too cold.
- *Reduce overhead.* The use of intermediaries (so important for modifiability, as we saw in [Chapter 7](#)) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff. Separation of concerns, another linchpin of modifiability, can also increase the processing overhead necessary to service an event if it leads to an event being serviced by a chain of components rather than a single component. The context switching and intercomponent communication costs add up, especially when the components are on different nodes on a network. A strategy for reducing computational overhead is to co-locate resources. Co-location may mean hosting cooperating components on the same processor to avoid the time delay of network communication; it may mean putting the resources in the same runtime software component to avoid even the expense of a subroutine call. A special case of reducing computational overhead is to perform a periodic cleanup of resources

that have become inefficient. For example, hash tables and virtual memory maps may require recalculation and reinitialization. Another common strategy is to execute single-threaded servers (for simplicity and avoiding contention) and split workload across them.

- *Bound execution times.* Place a limit on how much execution time is used to respond to an event. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times. The cost is usually a less accurate computation. If you adopt this tactic, you will need to assess its effect on accuracy and see if the result is “good enough.” This resource management tactic is frequently paired with the manage sampling rate tactic.
- *Increase resource efficiency.* Improving the algorithms used in critical areas will decrease latency.

Manage Resources

Even if the demand for resources is not controllable, the management of these resources can be. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a cache or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk. Here are some resource management tactics:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency. Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency and in many cases is the cheapest way to get immediate improvement.
- *Introduce concurrency.* If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, scheduling policies can be used to achieve the goals you find desirable. Different scheduling policies may maximize fairness (all requests get equal time), throughput (shortest time to finish first), or other goals. (See the sidebar.)
- *Maintain multiple copies of computations.* Multiple servers in a client-server pattern are replicas of computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server. A *load balancer* is a piece of software that assigns new work to one of the available duplicate servers; criteria for assignment vary but can be as simple as round-robin or assigning the next request to the least busy server.
- *Maintain multiple copies of data.* *Caching* is a tactic that involves keeping copies of data (possibly one a subset of the other) on storage with different access speeds. The different access speeds may be inherent (memory versus secondary storage) or may be due to the necessity for network communication. *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses. Because the data being cached or replicated is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume. Another responsibility is to choose the data to be cached. Some caches operate by merely keeping copies of whatever was recently requested, but it is also possible to predict users’ future requests based on patterns of behavior, and begin the calculations or prefetches necessary to comply with those requests before the user has made them.
- *Bound queue sizes.* This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to adopt a policy for what happens when the queues overflow and decide if not responding to lost events is acceptable. This tactic is frequently paired with the limit event response tactic.
- *Schedule resources.* Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your goal is to understand the characteristics of each resource’s use and choose the scheduling strategy that is compatible with it. (See the sidebar.)

The tactics for performance are summarized in [Figure 8.3](#).

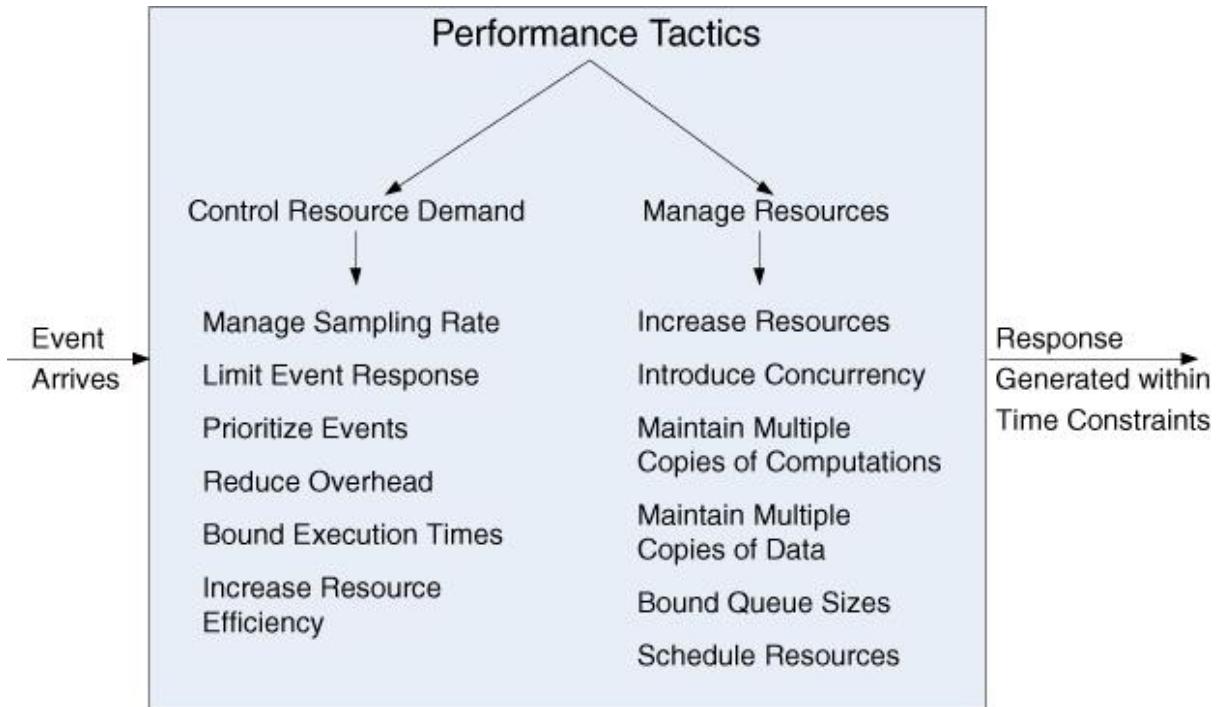


Figure 8.3. Performance tactics

Scheduling Policies

A *scheduling policy* conceptually has two parts: a priority assignment and dispatching. All scheduling policies assign priorities. In some cases the assignment is as simple as first-in/first-out (or FIFO). In other cases, it can be tied to the deadline of the request or its semantic importance. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, preventing starvation to ensure fairness, and so forth. You need to be aware of these possibly conflicting criteria and the effect that the chosen tactic has on meeting them.

A high-priority event stream can be dispatched only if the resource to which it is being assigned is available. Sometimes this depends on preempting the current user of the resource. Possible preemption options are as follows: can occur anytime, can occur only at specific preemption points, and executing processes cannot be preempted. Some common scheduling policies are these:

- *First-in/first-out*. FIFO queues treat all requests for resources as equals and satisfy them in turn. One possibility with a FIFO queue is that one request will be stuck behind another one that takes a long time to generate a response. As long as all of the requests are truly equal, this is not a problem, but if some requests are of higher priority than others, it is problematic.
- *Fixed-priority scheduling*. Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. This strategy ensures better service for higher priority requests. But it admits the possibility of a lower priority, but important, request taking an arbitrarily long time to be serviced, because it is stuck behind a series of higher priority requests. Three common prioritization strategies are these:
 - *Semantic importance*. Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
 - *Deadline monotonic*. Deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines. This

scheduling policy is used when streams of different priorities with real-time deadlines are to be scheduled.

- *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods. This scheduling policy is a special case of deadline monotonic but is better known and more likely to be supported by the operating system.
- *Dynamic priority scheduling.* Strategies include these:
 - *Round-robin.* Round-robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order. A special form of round-robin is a cyclic executive, where assignment possibilities are at fixed time intervals.
 - *Earliest-deadline-first.* Earliest-deadline-first. Earliest-deadline-first assigns priorities based on the pending requests with the earliest deadline.
 - *Least-slack-first.* This strategy assigns the highest priority to the job having the least “slack time,” which is the difference between the execution time remaining and the time to the job’s deadline.

For a single processor and processes that are preemptible (that is, it is possible to suspend processing of one task in order to service a task whose deadline is drawing near), both the earliest-deadline and least-slack scheduling strategies are optimal. That is, if the set of processes can be scheduled so that all deadlines are met, then these strategies will be able to schedule that set successfully.

- *Static scheduling.* A cyclic executive schedule is a scheduling strategy where the preemption points and the sequence of assignment to the resource are determined offline. The runtime overhead of a scheduler is thereby obviated.
-
-

Performance Tactics on the Road

Tactics are generic design principles. To exercise this point, think about the design of the systems of roads and highways where you live. Traffic engineers employ a bunch of design “tricks” to optimize the performance of these complex systems, where performance has a number of measures, such as throughput (how many cars per hour get from the suburbs to the football stadium), average-case latency (how long it takes, on average, to get from your house to downtown), and worst-case latency (how long does it take an emergency vehicle to get you to the hospital). What are these tricks? None other than our good old buddies, tactics.

Let’s consider some examples:

- *Manage event rate.* Lights on highway entrance ramps let cars onto the highway only at set intervals, and cars must wait (queue) on the ramp for their turn.
- *Prioritize events.* Ambulances and police, with their lights and sirens going, have higher priority than ordinary citizens; some highways have high-occupancy vehicle (HOV) lanes, giving priority to vehicles with two or more occupants.
- *Maintain multiple copies.* Add traffic lanes to existing roads, or build parallel routes.

In addition, there are some tricks that users of the system can employ:

- *Increase resources.* Buy a Ferrari, for example. All other things being equal, the fastest car with a competent driver on an open road will get you to your destination more quickly.
- *Increase efficiency.* Find a new route that is quicker and/or shorter than your current route.
- *Reduce computational overhead.* You can drive closer to the car in front of you, or you can load more people into the same vehicle (that is, carpooling).

What is the point of this discussion? To paraphrase Gertrude Stein: performance is performance is performance. Engineers have been analyzing and optimizing systems for centuries, trying to improve their performance, and they have been employing the same design strategies to do so. So you should feel some comfort in knowing that when you try to

improve the performance of your computer-based system, you are applying tactics that have been thoroughly “road tested.”

—RK

8.3. A Design Checklist for Performance

[Table 8.2](#) is a checklist to support the design and analysis process for performance.

Table 8.2. Checklist to Support the Design and Analysis Process for Performance

Category	Checklist
Allocation of Responsibilities	<p>Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur.</p> <p>For those responsibilities, identify the processing requirements of each responsibility, and determine whether they may cause bottlenecks.</p> <p>Also, identify additional responsibilities to recognize and process requests appropriately, including</p> <ul style="list-style-type: none"> ▪ Responsibilities that result from a thread of control crossing process or processor boundaries ▪ Responsibilities to manage the threads of control—allocation and deallocation of threads, maintaining thread pools, and so forth ▪ Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches <p>For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).</p>
Coordination Model	<p>Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that do the following:</p> <ul style="list-style-type: none"> ▪ Support any introduced concurrency (for example, is it thread safe?), event prioritization, or scheduling strategy ▪ Ensure that the required performance response can be delivered ▪ Can capture periodic, stochastic, or sporadic event arrivals, as needed ▪ Have the appropriate properties of the communication mechanisms; for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency

Data Model	<p>Determine those portions of the data model that will be heavily loaded, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur.</p> <p>For those data abstractions, determine the following:</p> <ul style="list-style-type: none"> ▪ Whether maintaining multiple copies of key data would benefit performance ▪ Whether partitioning data would benefit performance ▪ Whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible ▪ Whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible
Mapping among Architectural Elements	<p>Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency.</p> <p>Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity.</p> <p>Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance.</p> <p>Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.</p>
Resource Management	<p>Determine which resources in your system are critical for performance. For these resources, ensure that they will be monitored and managed under normal and overloaded system operation. For example:</p>

	<ul style="list-style-type: none"> ▪ System elements that need to be aware of, and manage, time and other performance-critical resources ▪ Process/thread models ▪ Prioritization of resources and access to resources ▪ Scheduling and locking strategies ▪ Deploying additional resources on demand to meet increased loads
Binding Time	<p>For each element that will be bound after compile time, determine the following:</p> <ul style="list-style-type: none"> ▪ Time necessary to complete the binding ▪ Additional overhead introduced by using the late binding mechanism <p>Ensure that these values do not pose unacceptable performance penalties on the system.</p>
Choice of Technology	<p>Will your choice of technology let you set and meet hard, real-time deadlines? Do you know its characteristics under load and its limits?</p> <p>Does your choice of technology give you the ability to set the following:</p> <ul style="list-style-type: none"> ▪ Scheduling policy ▪ Priorities ▪ Policies for reducing demand ▪ Allocation of portions of the technology to processors ▪ Other performance-related parameters <p>Does your choice of technology introduce excessive overhead for heavily used operations?</p> <hr/>

8.4. Summary

Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior. Performance can be measured in terms of throughput and latency for both interactive and embedded real-time systems, although throughput is usually more important in interactive systems, and latency is more important in embedded systems.

Performance can be improved by reducing demand or by managing resources more appropriately. Reducing demand will have the side effect of reducing fidelity or refusing to service some requests. Managing resources more appropriately can be done through scheduling, replication, or just increasing the resources available.

8.5. For Further Reading

Performance has a rich body of literature. Here are some books we recommend:

- *Software Performance and Scalability: A Quantitative Approach* [[Liu 09](#)]. This book covers performance geared toward enterprise applications, with an emphasis on queuing theory and measurement.

- *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* [[Smith 01](#)]. This book covers designing with performance in mind, with emphasis on building (and populating with real data) practical predictive performance models.
- *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* [[Douglass 99](#)].
- *Real-Time Systems* [[Liu 00](#)].
- *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management* [[Kircher 03](#)].

8.6. Discussion Questions

- 1.** “Every system has real-time performance constraints.” Discuss. Or provide a counterexample.
- 2.** Write a performance scenario that describes the average on-time flight arrival performance for an airline.
- 3.** Write several performance scenarios for an automatic teller machine. Think about whether your major concern is worst-case latency, average-case latency, throughput, or some other response measure. How would you modify your automatic teller machine design to accommodate these scenarios?
- 4.** Web-based systems often use *proxy servers*, which are the first element of the system to receive a request from a client (such as your browser). Proxy servers are able to serve up often-requested web pages, such as a company’s home page, without bothering the real application servers that carry out transactions. There may be many proxy servers, and they are often located geographically close to large user communities, to decrease response time for routine requests. What performance tactics do you see at work here?
- 5.** A fundamental difference between coordination mechanisms is whether interaction is synchronous or asynchronous. Discuss the advantages and disadvantages of each with respect to each of the performance responses: latency, deadline, throughput, jitter, miss rate, data loss, or any other required performance-related response you may be used to.
- 6.** Find real-world (that is, nonsoftware) examples of applying each of the manage-resources tactics. For example, suppose you were managing a brick-and-mortar big-box retail store. How would you get people through the checkout lines faster using these tactics?
- 7.** User interface frameworks typically are single-threaded. Why is this so and what are the performance implications of this single-threading?

9. Security

With Jungwoo Ryoo and Phil Laplante

Your personal identity isn't worth quite as much as it used to be—at least to thieves willing to swipe it. According to experts who monitor such markets, the value of stolen credit card data may range from \$3 to as little as 40 cents. That's down tenfold from a decade ago—even though the cost to an individual who has a credit card stolen can soar into the hundreds of dollars.

—Forbes.com (Taylor Buley. "Hackonomics," Forbes.com, October 27, 2008,www.forbes.com/2008/10/25/credit-card-theft-tech-security-cz_tb1024theft.html)

Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized. An action taken against a computer system with the intention of doing harm is called an attack and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

The simplest approach to characterizing security has three characteristics: confidentiality, integrity, and availability (CIA):

1. *Confidentiality* is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
2. *Integrity* is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
3. *Availability* is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering book from an online bookstore.

Other characteristics that are used to support CIA are these:

4. *Authentication* verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an email purporting to come from a bank, authentication guarantees that it actually comes from the bank.
5. *Nonrepudiation* guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.
6. *Authorization* grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

We will use these characteristics in our general scenarios for security. Approaches to achieving security can be characterized as those that detect attacks, those that resist attacks, those that react to attacks, and those that recover from successful attacks. The objects that are being protected from attacks are data at rest, data in transit, and computational processes.

9.1. Security General Scenario

One technique that is used in the security domain is threat modeling. An “attack tree,” similar to a fault tree discussed in [Chapter 5](#), is used by security engineers to determine possible threats. The root is a successful attack and the nodes are possible direct causes of that successful attack. Children nodes decompose the direct causes, and so forth. An attack is an attempt to break CIA, and the leaves of attack trees would be the stimulus in the scenario. The response to the attack is to preserve CIA or deter attackers through monitoring of their activities. From these considerations we can now describe the individual portions of a security general scenario. These are summarized in [Table 9.1](#), and an example security scenario is given in [Figure 9.1](#).

Table 9.1. Security General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	<p>Transactions are carried out in a fashion such that</p> <ul style="list-style-type: none">▪ Data or services are protected from unauthorized access.▪ Data or services are not being manipulated without authorization.▪ Parties to a transaction are identified with assurance.▪ The parties to the transaction cannot repudiate their involvements.▪ The data, resources, and system services will be available for legitimate use. <p>The system tracks activities within it by</p> <ul style="list-style-type: none">▪ Recording access or modification▪ Recording attempts to access data, resources, or services▪ Notifying appropriate entities (people or systems) when an apparent attack is occurring
Response Measure	<p>One or more of the following:</p> <ul style="list-style-type: none">▪ How much of a system is compromised when a particular component or data value is compromised▪ How much time passed before an attack was detected▪ How many attacks were resisted▪ How long does it take to recover from a successful attack▪ How much data is vulnerable to a particular attack

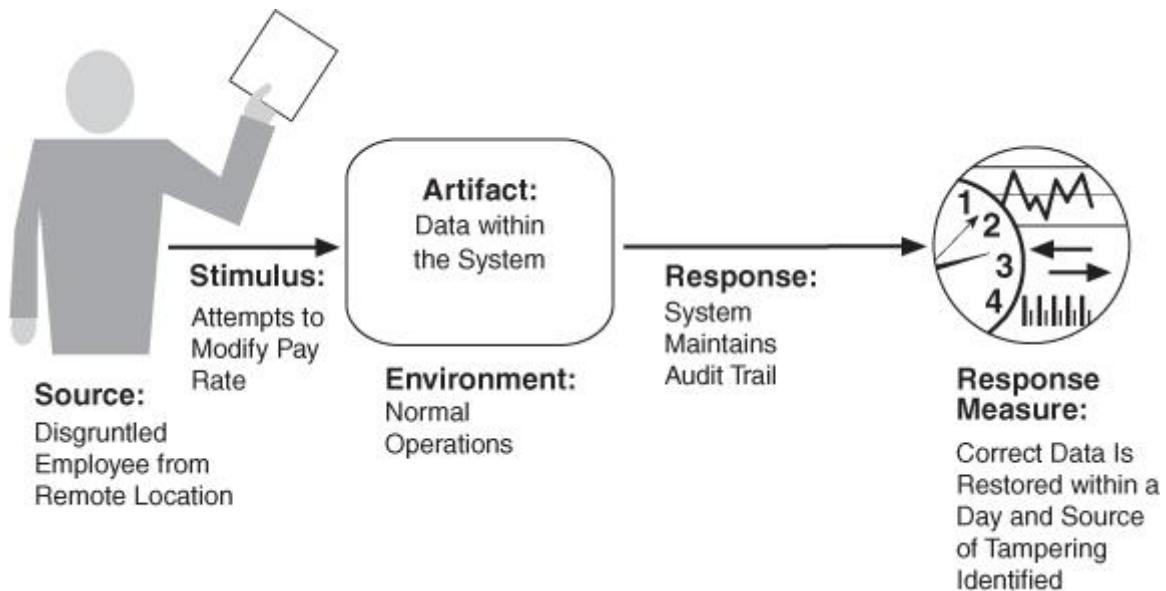


Figure 9.1. Sample concrete security scenario

- *Source of stimulus.* The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
- *Stimulus.* The stimulus is an attack. We characterize this as an unauthorized attempt to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
- *Artifact.* The target of the attack can be either the services of the system, the data within it, or the data produced or consumed by the system. Some attacks are made on particular components of the system known to be vulnerable.
- *Environment.* The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to a network, fully operational, partially operational, or not operational.
- *Response.* The system should ensure that transactions are carried out in a fashion such that data or services are protected from unauthorized access; data or services are not being manipulated without authorization; parties to a transaction are identified with assurance; the parties to the transaction cannot repudiate their involvements; and the data, resources, and system services will be available for legitimate use.

The system should also track activities within it by recording access or modification; attempts to access data, resources, or services; and notifying appropriate entities (people or systems) when an apparent attack is occurring.

- *Response measure.* Measures of a system's response include how much of a system is compromised when a particular component or data value is compromised, how much time passed before an attack was detected, how many attacks were resisted, how long it took to recover from a successful attack, and how much data was vulnerable to a particular attack.

[Table 9.1](#) enumerates the elements of the general scenario, which characterize security, and [Figure 9.1](#) shows a sample concrete scenario: A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The system maintains an audit trail, and the correct data is restored within a day.

9.2. Tactics for Security

One method for thinking about how to achieve security in a system is to think about physical security. Secure installations have limited access (e.g., by using security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms such as armed guards, have reaction mechanisms such as automatic locking of doors, and have recovery mechanisms such as off-site backup. These lead to our four categories of tactics: detect, resist, react, and recover. [Figure 9.2](#) shows these categories as the goal of security tactics.

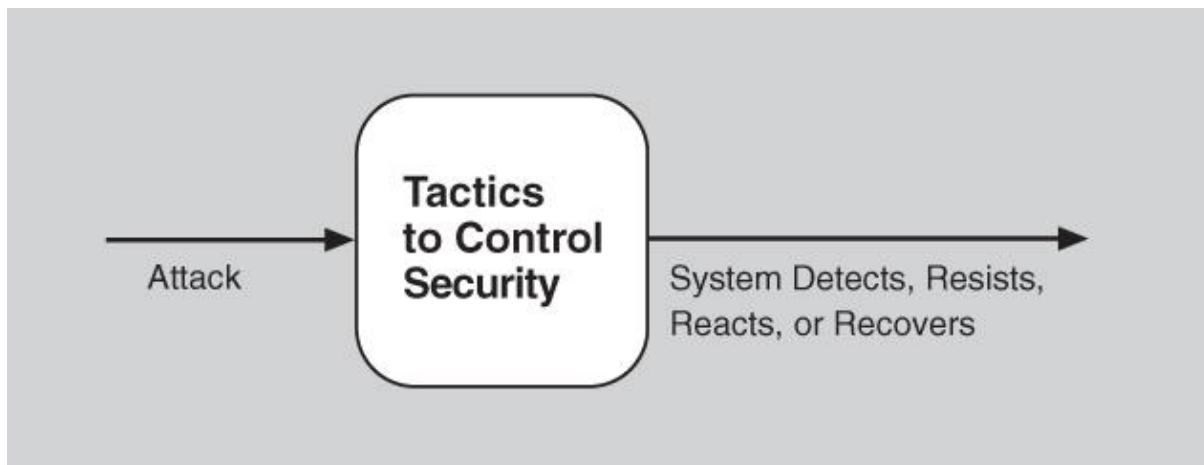


Figure 9.2. The goal of security tactics

Detect Attacks

The detect attacks category consists of four tactics: detect intrusion, detect service denial, verify message integrity, and detect message delay.

- *Detect intrusion* is the comparison of network traffic or service request patterns *within* a system to a set of signatures or known patterns of malicious behavior stored in a database. The signatures can be based on protocol, TCP flags, payload sizes, applications, source or destination address, or port number.
- *Detect service denial* is the comparison of the pattern or signature of network traffic *coming into* a system to historic profiles of known denial-of-service attacks.
- *Verify message integrity*. This tactic employs techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files. A checksum is a validation mechanism wherein the system maintains redundant information for configuration files and messages, and uses this redundant information to verify the configuration file or message when it is used. A hash value is a unique string generated by a hashing function whose input could be configuration files or messages. Even a slight change in the original files or messages results in a significant change in the hash value.
- *Detect message delay* is intended to detect potential man-in-the-middle attacks, where a malicious party is intercepting (and possibly modifying) messages. By checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior, where the time it takes to deliver a message is highly variable.

Resist Attacks

There are a number of well-known means of resisting an attack:

- *Identify actors*. Identifying “actors” is really about identifying the source of any external input to the system. Users are typically identified through user IDs. Other systems may be “identified” through access codes, IP addresses, protocols, ports, and so on.

- *Authenticate actors.* Authentication means ensuring that an actor (a user or a remote computer) is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, and biometric identification provide a means for authentication.
- *Authorize actors.* Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. This mechanism is usually enabled by providing some access control mechanisms within a system. Access control can be by an actor or by an actor class. Classes of actors can be defined by actor groups, by actor roles, or by lists of individuals.
- *Limit access.* Limiting access to computing resources involves limiting access to resources such as memory, network connections, or access points. This may be achieved by using memory protection, blocking a host, closing a port, or rejecting a protocol. For example, a demilitarized zone (DMZ) is used when an organization wants to let external users access certain services and not access other services. It sits between the Internet and a firewall in front of the internal intranet. The firewall is a single point of access to the intranet (limit exposure). It also restricts access using a variety of techniques to authorize users (authorize actors).
- *Limit exposure.* The limit exposure tactic minimizes the attack surface of a system. This tactic focuses on reducing the probability of and minimizing the effects of damage caused by a hostile action. It is a passive defense because it does not proactively prevent attackers from doing harm. Limit exposure is typically realized by having the least possible number of access points for resources, data, or services and by reducing the number of connectors that may provide unanticipated exposure.
- *Encrypt data.* Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication. Encryption provides extra protection to persistently maintained data beyond that available from authorization. Communication links, on the other hand, may not have authorization controls. In such cases, encryption is the only protection for passing data over publicly accessible communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for a web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).
- *Separate entities.* Separating different entities within the system can be done through physical separation on different servers that are attached to different networks; the use of virtual machines (see [Chapter 26](#) for a discussion of virtual machines); or an “air gap,” that is, by having no connection between different portions of a system. Finally, sensitive data is frequently separated from nonsensitive data to reduce the attack possibilities from those who have access to nonsensitive data.
- *Change default settings.* Many systems have default settings assigned when the system is delivered. Forcing the user to change those settings will prevent attackers from gaining access to the system through settings that are, generally, publicly available.

React to Attacks

Several tactics are intended to respond to a potential attack:

- *Revoke access.* If the system or a system administrator believes that an attack is underway, then access can be severely limited to sensitive resources, even for normally legitimate users and uses. For example, if your desktop has been compromised by a virus, your access to certain resources may be limited until the virus is removed from your system.
- *Lock computer.* Repeated failed login attempts may indicate a potential attack. Many systems limit access from a particular computer if there are repeated failed attempts to access an account from that computer. Legitimate users may make mistakes in attempting to log in. Therefore, the limited access may only be for a certain time period.
- *Inform actors.* Ongoing attacks may require action by operators, other personnel, or cooperating systems. Such personnel or systems—the set of relevant actors—must be notified when the system has detected an attack.

Recover from Attacks

Once a system has detected and attempted to resist an attack, it needs to recover. Part of recovery is restoration of services. For example, additional servers or network connections may be kept in reserve for such a purpose. Since a successful attack can be considered a kind of failure, the set of availability tactics (from [Chapter 5](#)) that deal with recovering from a failure can be brought to bear for this aspect of security as well.

In addition to the availability tactics that permit restoration of services, we need to maintain an audit trail. We audit—that is, keep a record of user and system actions and their effects—to help trace the actions of, and to identify, an attacker. We may analyze audit trails to attempt to prosecute attackers, or to create better defenses in the future.

The set of security tactics is shown in [Figure 9.3](#).

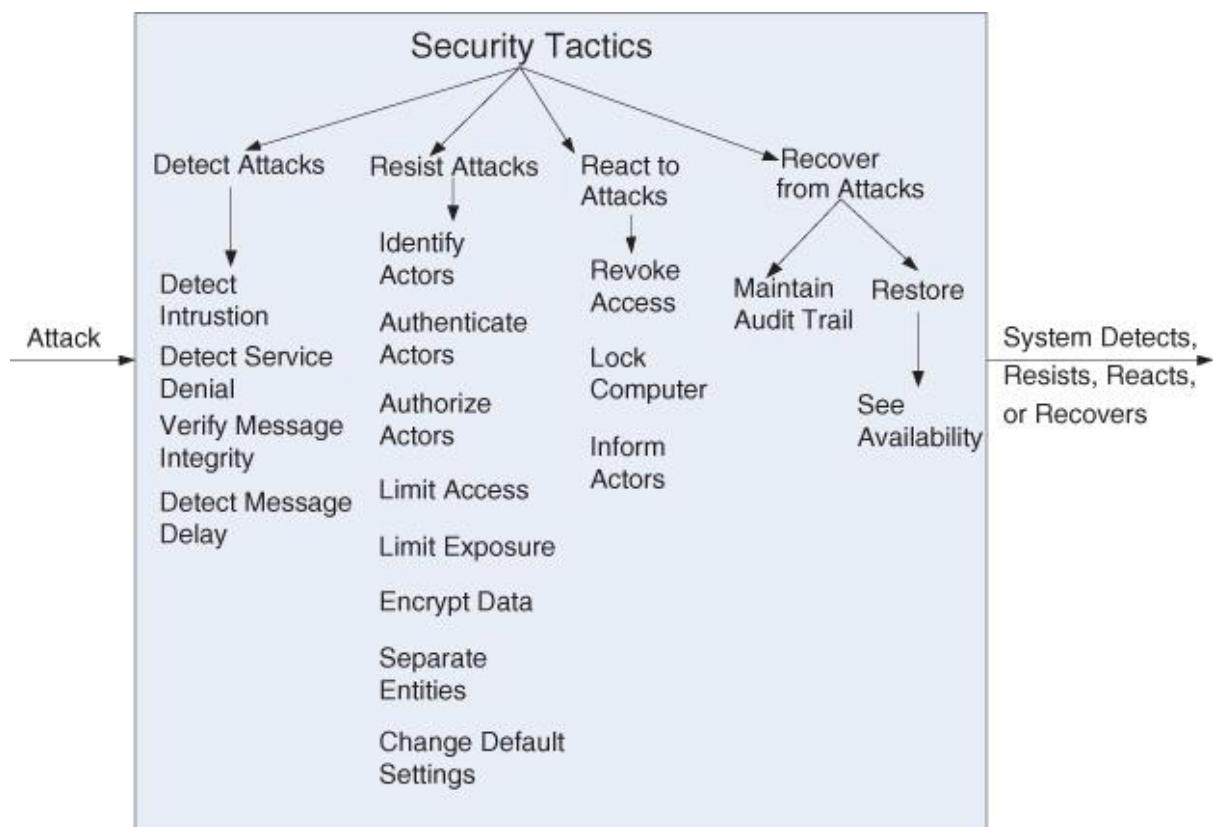


Figure 9.3. Security tactics

9.3. A Design Checklist for Security

[Table 9.2](#) is a checklist to support the design and analysis process for security.

Table 9.2. Checklist to Support the Design and Analysis Process for Security

Category	Checklist
Allocation of Responsibilities	<p>Determine which system responsibilities need to be secure. For each of these responsibilities, ensure that additional responsibilities have been allocated to do the following:</p> <ul style="list-style-type: none">▪ Identify the actor▪ Authenticate the actor▪ Authorize actors▪ Grant or deny access to data or services▪ Record attempts to access or modify data or services▪ Encrypt data▪ Recognize reduced availability for resources or services and inform appropriate personnel and restrict access▪ Recover from an attack▪ Verify checksums and hash values
Coordination Model	<p>Determine mechanisms required to communicate and coordinate with other systems or individuals. For these communications, ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection, are in place. Ensure also that mechanisms exist for monitoring and recognizing unexpectedly high demands for resources or services as well as mechanisms for restricting or terminating the connection.</p>
Data Model	<p>Determine the sensitivity of different data fields. For each data abstraction:</p> <ul style="list-style-type: none">▪ Ensure that data of different sensitivity is separated.▪ Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access.▪ Ensure that access to sensitive data is logged and that the log file is suitably protected.▪ Ensure that data is suitably encrypted and that keys are separated from the encrypted data.▪ Ensure that data can be restored if it is inappropriately modified.

Mapping among Architectural Elements	<p>Determine how alternative mappings of architectural elements that are under consideration may change how an individual or system may read, write, or modify data; access system services or resources; or reduce availability to system services or resources. Determine how alternative mappings may affect the recording of access to data, services or resources and the recognition of unexpectedly high demands for resources.</p> <p>For each such mapping, ensure that there are responsibilities to do the following:</p> <ul style="list-style-type: none"> ▪ Identify an actor ▪ Authenticate an actor ▪ Authorize actors ▪ Grant or deny access to data or services ▪ Record attempts to access or modify data or services ▪ Encrypt data ▪ Recognize reduced availability for resources or services, inform appropriate personnel, and restrict access ▪ Recover from an attack
Resource Management	<p>Determine the system resources required to identify and monitor a system or an individual who is internal or external, authorized or not authorized, with access to specific resources or all resources. Determine the resources required to authenticate the actor, grant or deny access to data or resources, notify appropriate entities (people or systems), record attempts to access data or resources, encrypt data, recognize inexplicably high demand for resources, inform users or systems, and restrict access.</p>
Binding Time	<p>For these resources consider whether an external entity can access a critical resource or exhaust a critical resource; how to monitor the resource; how to manage resource utilization; how to log resource utilization; and ensure that there are sufficient resources to perform the necessary security operations.</p> <p>Ensure that a contaminated element can be prevented from contaminating other elements.</p> <p>Ensure that shared resources are not used for passing sensitive data from an actor with access rights to that data to an actor without access rights to that data.</p>
Choice of Technology	<p>Determine cases where an instance of a late-bound component may be untrusted. For such cases ensure that late-bound components can be qualified; that is, if ownership certificates for late-bound components are required, there are appropriate mechanisms to manage and validate them; that access to late-bound data and services can be managed; that access by late-bound components to data and services can be blocked; that mechanisms to record the access, modification, and attempts to access data or services by late-bound components are in place; and that system data is encrypted where the keys are intentionally withheld for late-bound components</p> <p>Determine what technologies are available to help user authentication, data access rights, resource protection, and data encryption.</p> <p>Ensure that your chosen technologies support the tactics relevant for your security needs.</p>

9.4. Summary

Attacks against a system can be characterized as attacks against the confidentiality, integrity, or availability of a system or its data. Confidentiality means keeping data away from those who should not have access while granting access to those who should. Integrity means that there are no unauthorized modifications to or deletion of data, and availability means that the system is accessible to those who are entitled to use it.

The emphasis of distinguishing various classes of actors in the characterization leads to many of the tactics used to achieve security. Identifying, authenticating, and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.

An assumption is made that no security tactic is foolproof and that systems will be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack, and to react and recover from an attack.

Recovering from an attack involves many of the same tactics as availability and, in general, involves returning the system to a consistent state prior to any attack.

9.5. For Further Reading

The architectural tactics that we have described in this chapter are only one aspect of making a system secure. Other aspects are these:

- *Coding.* *Secure Coding in C and C++* [\[Seacord 05\]](#) describes how to code securely. The Common Weakness Enumeration [\[CWE 12\]](#) is a list of the most common vulnerabilities discovered in systems.
- *Organizational processes.* Organizations must have processes that provide for responsibility for various aspects of security, including ensuring that systems are patched to put into place the latest protections. The National Institute of Standards and Technology (NIST) provides an enumeration of organizational processes [\[NIST 09\]](#). [\[Cappelli 12\]](#) discusses insider threats.
- *Technical processes.* Microsoft has a life-cycle development process (The Secure Development Life Cycle) that includes modeling of threats. Four training classes are publicly available. www.microsoft.com/download/en/details.aspx?id=16420

NIST has several volumes that give definitions of security terms [\[NIST 04\]](#), categories of security controls [\[NIST 06\]](#), and an enumeration of security controls that an organization could employ [\[NIST 09\]](#). A security control could be a tactic, but it could also be organizational, coding-related, or a technical process.

The attack surface of a system is the code that can be run by unauthorized users. A discussion of how to minimize the attack surface for a system can be found at [\[Howard 04\]](#).

Encryption and certificates of various types and strengths are commonly used to resist certain types of attacks. Encryption algorithms are particularly difficult to code correctly. A document produced by NIST [\[NIST 02\]](#) gives requirements for these algorithms.

Good books on engineering systems for security have been written by Ross Anderson [\[Anderson 08\]](#) and Bruce Schneier [\[Schneier 08\]](#).

Different domains have different specific sets of practices. The Payment Card Industry (PCI) has a set of standards intended for those involved in credit card processing (www.pcisecuritystandards.org). There is also a set of recommendations for securing various portions of the electric grid (www.smartgridipedia.org/index.php/ASAP-SG).

Data on the various sources of data breaches can be found in the Verizon 2012 Data Breach Investigations Report [\[Verizon 12\]](#).

John Viega has written several books about secure software development in various environments. See, for example, [\[Viega 01\]](#).

9.6. Discussion Questions

- 1.** Write a set of concrete scenarios for security for an automatic teller machine. How would you modify your design for the automatic teller machine to satisfy these scenarios?
- 2.** One of the most sophisticated attacks on record was carried out by a virus known as Stuxnet. Stuxnet first appeared in 2009 but became widely known in 2011 when it was revealed that it had apparently severely damaged or incapacitated the high-speed centrifuges involved in Iran's uranium enrichment program. Read about Stuxnet and see if you can devise a defense strategy against it based on the tactics in this chapter.
- 3.** Some say that inserting security awareness into the software development life cycle is at least as important as designing software with security countermeasures. What are some examples of software development processes that can lead to more-secure systems?
- 4.** Security and usability are often seen to be at odds with each other. Security often imposes procedures and processes that seem like needless overhead to the casual user. But some say that security and usability go (or should go) hand in hand and argue that making the system easy to use securely is the best way to promote security to the user. Discuss.
- 5.** List some examples of critical resources for security that might become exhausted.
- 6.** List an example of a mapping of architectural elements that has strong security implications. Hint: think of where data is stored.
- 7.** Which of the tactics in our list will protect against an insider threat? Can you think of any that should be added?
- 8.** In the United States, Facebook can account for more than 5 percent of all Internet traffic in a given week. How would you recognize a denial-of-service attack on Facebook.com?
- 9.** The public disclosure of vulnerabilities in production systems is a matter of controversy. Discuss why this is so and the pros and cons of public disclosure of vulnerabilities.

10. Testability

Testing leads to failure, and failure leads to understanding

—Burt Rutan

Industry estimates indicate that between 30 and 50 percent (or in some cases, even more) of the cost of developing well-engineered systems is taken up by testing. If the software architect can reduce this cost, the payoff is large.

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution. Intuitively, a system is testable if it “gives up” its faults easily. If a fault is present in a system, then we want it to fail during testing as quickly as possible. Of course, calculating this probability is not easy and, as you will see when we discuss response measures for testability, other measures will be used.

[Figure 10.1](#) shows a model of testing in which a program processes input and produces output. An oracle is an agent (human or mechanical) that decides whether the output is correct or not by comparing the output to the program’s specification. Output is not just the functionally produced value, but it also can include derived measures of quality attributes such as how long it took to produce the output. [Figure 10.1](#) also shows that the program’s internal state can also be shown to the oracle, and an oracle can decide whether that is correct or not—that is, it can detect whether the program has entered an erroneous state and render a judgment as to the correctness of the program.

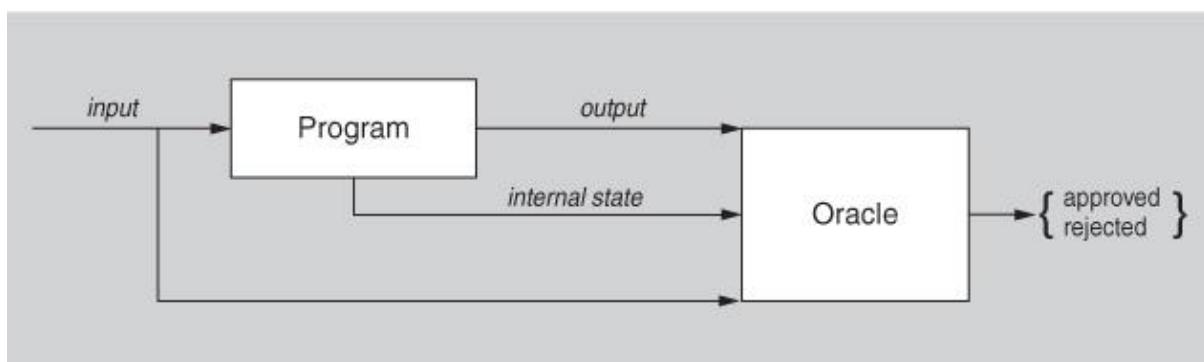


Figure 10.1. A model of testing

Setting and examining a program’s internal state is an aspect of testing that will figure prominently in our tactics for testability.

For a system to be properly testable, it must be possible to control each component’s inputs (and possibly manipulate its internal state) and then to observe its outputs (and possibly its internal state, either after or on the way to computing the outputs). Frequently this control and observation is done through the use of a test harness, which is specialized software (or in some cases, hardware) designed to exercise the software under test. Test harnesses come in various forms, such as a record-and-playback capability for data sent across various interfaces, or a simulator for an external environment in which a piece of embedded software is tested, or even during production (see sidebar). The test harness can provide assistance in executing the test procedures and recording the output. A test harness can be a substantial piece of software in its own right, with its own architecture, stakeholders, and quality attribute requirements.

Testing is carried out by various developers, users, or quality assurance personnel. Portions of the system or the entire system may be tested. The response measures for testability deal with how effective the tests are in discovering faults and how long it takes to perform the tests to some desired level of coverage. Test cases can be written by the developers, the testing group, or the customer. The test cases can be a portion of acceptance testing or can drive the development as they do in certain types of Agile methodologies.

Netflix's Simian Army

Netflix distributes movies and television shows both via DVD and via streaming video. Their streaming video service has been extremely successful. In May 2011 Netflix streaming video accounted for 24 percent of the Internet traffic in North America. Naturally, high availability is important to Netflix.

Netflix hosts their computer services in the Amazon EC2 cloud, and they utilize what they call a "Simian Army" as a portion of their testing process. They began with a Chaos Monkey, which randomly kills processes in the running system. This allows the monitoring of the effect of failed processes and gives the ability to ensure that the system does not fail or suffer serious degradation as a result of a process failure.

Recently, the Chaos Monkey got some friends to assist in the testing. Currently, the Netflix Simian Army includes these:

- The Latency Monkey induces artificial delays in the client-server communication layer to simulate service degradation and measures if upstream services respond appropriately.
- The Conformity Monkey finds instances that don't adhere to best practices and shuts them down. For example, if an instance does not belong to an auto-scaling group, it will not appropriately scale when demand goes up.
- The Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g., CPU load) to detect unhealthy instances.
- The Janitor Monkey ensures that the Netflix cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.
- The Security Monkey is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured security groups, and terminates the offending instances. It also ensures that all the SSL and digital rights management (DRM) certificates are valid and are not coming up for renewal.
- The 10-18 Monkey (localization-internationalization) detects configuration and runtime problems in instances serving customers in multiple geographic regions, using different languages and character sets. The name 10-18 comes from *L10n-i18n*, a sort of shorthand for the words *localization* and *internationalization*.

Some of the members of the Simian Army use fault injection to place faults into the running system in a controlled and monitored fashion. Other members monitor various specialized aspects of the system and its environment. Both of these techniques have broader applicability than just Netflix.

Not all faults are equal in terms of severity. More emphasis should be placed on finding the most severe faults than on finding other faults. The Simian Army reflects a determination by Netflix that the faults they look for are the most serious in terms of their impact.

This strategy illustrates that some systems are too complex and adaptive to be tested fully, because some of their behaviors are emergent. An aspect of testing in that arena is logging of operational data produced by the system, so that when failures occur, the logged data can be analyzed in the lab to try to reproduce the faults. Architecturally this can require mechanisms to access and log certain system state. The Simian Army is one way to discover and log behavior in systems of this ilk.

—LB

Testing of code is a special case of validation, which is making sure that an engineered artifact meets the needs of its stakeholders or is suitable for use. In [Chapter 21](#) we will discuss architectural design reviews. This is another kind of validation, where the artifact being tested is the architecture. In this chapter we are concerned only with the testability of a running system and of its source code.

10.1. Testability General Scenario

We can now describe the general scenario for testability.

- *Source of stimulus.* The testing is performed by unit testers, integration testers, or system testers (on the developing organization side), or acceptance testers and end users (on the customer side). The source could be human or an automated tester.
- *Stimulus.* A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
- *Artifact.* A unit of code (corresponding to a module in the architecture), a subsystem, or the whole system is the artifact being tested.
- *Environment.* The test can happen at development time, at compile time, at deployment time, or while the system is running (perhaps in routine use). The environment can also include the test harness or test environments in use.
- *Response.* The system can be controlled to perform the desired tests and the results from the test can be observed.
- *Response measure.* Response measures are aimed at representing how easily a system under test "gives up" its faults. Measures might include the effort involved in finding a fault or a particular class of faults, the effort required to test a given percentage of statements, the length of the longest test chain (a measure of the difficulty of performing the tests), measures of effort to perform the tests, measures of effort to actually find faults, estimates of the probability of finding additional faults, and the length of time or amount of effort to prepare the test environment.

Maybe one measure is the ease at which the system can be brought into a specific state. In addition, measures of the reduction in risk of the remaining errors in the system can be used. Not all faults are equal in terms of their possible impact. Measures of risk reduction attempt to rate the severity of faults found (or to be found).

[Figure 10.2](#) shows a concrete scenario for testability. The unit tester completes a code unit during development and performs a test sequence whose results are captured and that gives 85 percent path coverage within three hours of testing.

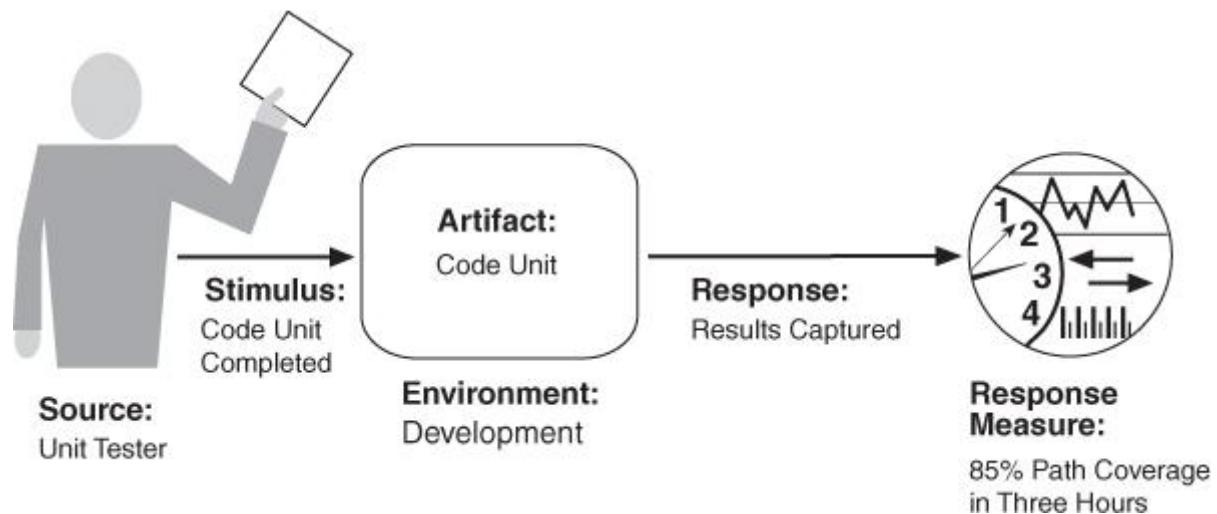


Figure 10.2. Sample concrete testability scenario

[Table 10.1](#) enumerates the elements of the general scenario that characterize testability.

Table 10.1. Testability General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results, capture activity that resulted in the fault, control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure ($\text{size}(\text{loss}) \times \text{prob}(\text{loss})$)

10.2. Tactics for Testability

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. [Figure 10.3](#) displays the use of tactics for testability. Architectural techniques for enhancing the software testability have not received as much attention as more mature quality attribute disciplines such as modifiability, performance, and availability, but as we stated before, anything the architect can do to reduce the high cost of testing will yield a significant benefit.

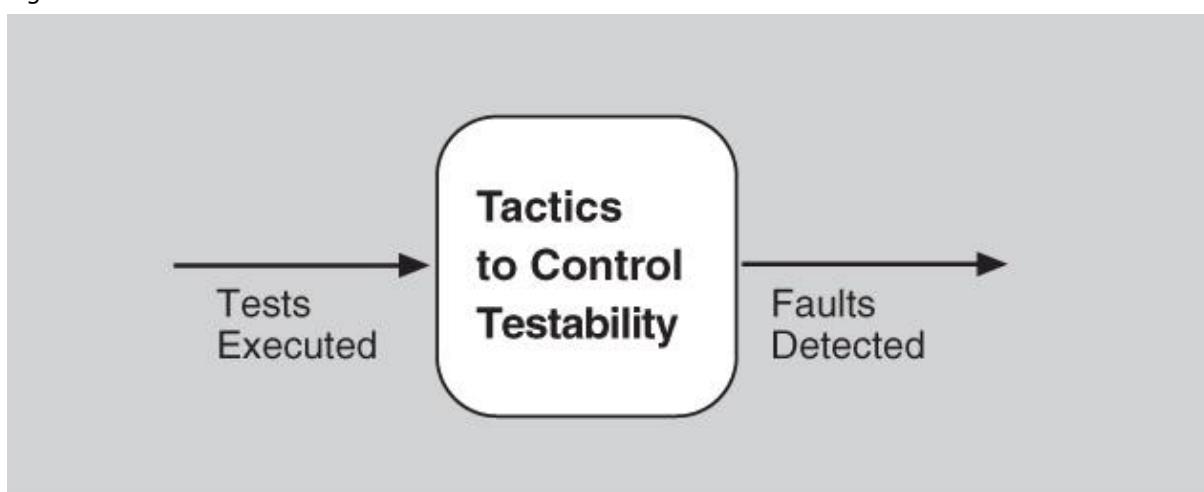


Figure 10.3. The goal of testability tactics

There are two categories of tactics for testability. The first category deals with adding controllability and observability to the system. The second deals with limiting complexity in the system's design.

Control and Observe System State

Control and observation are so central to testability that some authors even define testability in those terms. The two go hand-in-hand; it makes no sense to control something if you can't observe what happens when you do. The simplest form of control and observation is to provide a software component with a set of inputs, let it do its work, and then observe its outputs. However, the control and observe system state category of testability tactics provides insight into software that goes beyond its inputs and outputs. These tactics cause a component to maintain some sort of state information, allow testers to assign a value to that state information, and/or make that information accessible to testers on demand. The state information might be an operating state, the value of some key variable, performance load, intermediate process steps, or anything else useful to re-creating component behavior. Specific tactics include the following:

- *Specialized interfaces.* Having specialized testing interfaces allows you to control or capture variable values for a component either through a test harness or through normal execution. Examples of specialized test routines include these:
 - A *set* and *get* method for important variables, modes, or attributes (methods that might otherwise not be available except for testing purposes)
 - A *report* method that returns the full state of the object
 - A *reset* method to set the internal state (for example, all the attributes of a class) to a specified internal state
 - A method to turn on verbose output, various levels of event logging, performance instrumentation, or resource monitoring

Specialized testing interfaces and methods should be clearly identified or kept separate from the access methods and interfaces for required functionality, so that they can be removed if needed. (However, in performance-critical and some safety-critical systems, it is problematic to field different code than that which was tested. If you remove the test code, how will you know the code you field has the same behavior, particularly the same timing behavior, as the code you tested? For other kinds of systems, however, this strategy is effective.)

- *Record/playback.* The state that caused a fault is often difficult to re-create. Recording the state when it crosses an interface allows that state to be used to "play the system back" and to re-create the fault. Record/playback refers to both capturing information crossing an interface and using it as input for further testing.
- *Localize state storage.* To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place. By contrast, if the state is buried or distributed, this becomes difficult if not impossible. The state can be fine-grained, even bit-level, or coarse-grained to represent broad abstractions or overall operational modes. The choice of granularity depends on how the states will be used in testing. A convenient way to "externalize" state storage (that is, to make it able to be manipulated through interface features) is to use a state machine (or state machine object) as the mechanism to track and report current state.
- *Abstract data sources.* Similar to controlling a program's state, easily controlling its input data makes it easier to test. Abstracting the interfaces lets you substitute test data more easily. For example, if you have a database of customer transactions, you could design your architecture so that it is easy to point your test system at other test databases, or possibly even to files of test data instead, without having to change your functional code.
- *Sandbox.* "Sandboxing" refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment. Testing is helped by the ability to operate the system in such a way that it has no permanent consequences, or so that any consequences can be rolled back. This can be used for scenario analysis, training, and simulation. (The Spring framework, which is quite popular in the Java community, comes with a set of test utilities that support this. Tests are run as a "transaction," which is rolled back at the end.)

A common form of sandboxing is to virtualize resources. Testing a system often involves interacting with resources whose behavior is outside the control of the system. Using a sandbox, you can build a version of the resource whose behavior is under your control. For example, the system clock's behavior is typically not under our control—it increments one second each second—which means that if we want to make the system think it's midnight on the day when all of the data structures are supposed to overflow, we need a way to do that, because waiting around is a poor choice. By having the capability to abstract system time from clock time, we can allow the system (or components) to run at faster than wall-clock time, and to allow the system (or components) to be tested at critical time boundaries (such as the next shift on or off Daylight Savings Time). Similar virtualizations could be done for other resources, such as memory, battery, network, and so on. Stubs, mocks, and dependency injection are simple but effective forms of virtualization.

- *Executable assertions.* Using this tactic, assertions are (usually) hand-coded and placed at desired locations to indicate when and where a program is in a faulty state. The assertions are often designed to check that data values satisfy specified constraints. Assertions are defined in terms of specific data declarations, and they must be placed where the data values are referenced or modified. Assertions can be expressed as pre- and post-conditions for each method and also as class-level invariants. This results in increasing observability, when an assertion is flagged as having failed. Assertions systematically inserted where data values change can be seen as a manual way to produce an “extended” type. Essentially, the user is annotating a type with additional checking code. Any time an object of that type is modified, the checking code is automatically executed, and warnings are generated if any conditions are violated. To the extent that the assertions cover the test cases, they effectively embed the test oracle in the code—assuming the assertions are correct and correctly coded.

All of these tactics add capability or abstraction to the software that (were we not interested in testing) otherwise would not be there. They can be seen as replacing bare-bones, get-the-job-done software with more elaborate software that has bells and whistles for testing. There are a number of techniques for effecting this replacement. These are not testability tactics, per se, but techniques for replacing one component with a different version of itself. They include the following:

- Component replacement, which simply swaps the implementation of a component with a different implementation that (in the case of testability) has features that facilitate testing. Component replacement is often accomplished in a system’s build scripts.
- Preprocessor macros that, when activated, expand to state-reporting code or activate probe statements that return or display information, or return control to a testing console.
- Aspects (in aspect-oriented programs) that handle the cross-cutting concern of how state is reported.

Limit Complexity

Complex software is harder to test. This is because, by the definition of complexity, its operating state space is very large and (all else being equal) it is more difficult to re-create an exact state in a large state space than to do so in a small state space. Because testing is not just about making the software fail but about finding the fault that caused the failure so that it can be removed, we are often concerned with making behavior repeatable. This category has three tactics:

- *Limit structural complexity.* This tactic includes avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general (for example, reduce the number of external accesses to a module’s public data). In object-oriented systems, you can simplify the inheritance hierarchy: Limit the number of classes from which a class is derived, or the number of classes derived from a class. Limit the depth of the inheritance tree, and the number of children of a class. Limit polymorphism and dynamic calls. One structural metric that has been shown empirically to correlate to testability is called the *response* of a class. The response of class C is a count of the number of methods of C plus the number of methods of other classes that are invoked by the methods of C. Keeping this metric low can increase testability.

Having high cohesion, loose coupling, and separation of concerns—all modifiability tactics (see [Chapter 7](#))—can also help with testability. They are a form of limiting the complexity of the architectural elements by giving each element a focused task with limited interaction with other elements. Separation of concerns can help achieve controllability and observability (as well as reducing the size of the overall program’s state space). Controllability is critical to making testing tractable, as Robert Binder has noted: “A component that can act independently of others is more readily controllable. . . . With high coupling among classes it is typically more difficult to control the class under test, thus reducing testability. . . . If user interface capabilities are entwined with basic functions it will be more difficult to test each function” [\[Binder 94\]](#).

Also, systems that require complete data consistency at all times are often more complex than those that do not. If your requirements allow it, consider building your system under the “eventual consistency” model, where sooner or later (but maybe not right now) your data will reach a consistent state. This often makes system design simpler, and therefore easier to test.

Finally, some architectural styles lend themselves to testability. In a layered style, you can test lower layers first, then test higher layers with confidence in the lower layers.

- *Limit nondeterminism.* The counterpart to limiting structural complexity is limiting behavioral complexity, and when it comes to testing, nondeterminism is a very pernicious form of complex behavior. Nondeterministic systems are harder to test than deterministic systems. This tactic involves finding all the sources of nondeterminism, such as unconstrained parallelism, and weeding them out as much as possible. Some sources of nondeterminism are unavoidable—for instance, in multi-threaded systems that respond to unpredictable events—but for such systems, other tactics (such as record/playback) are available.

[Figure 10.4](#) provides a summary of the tactics used for testability.

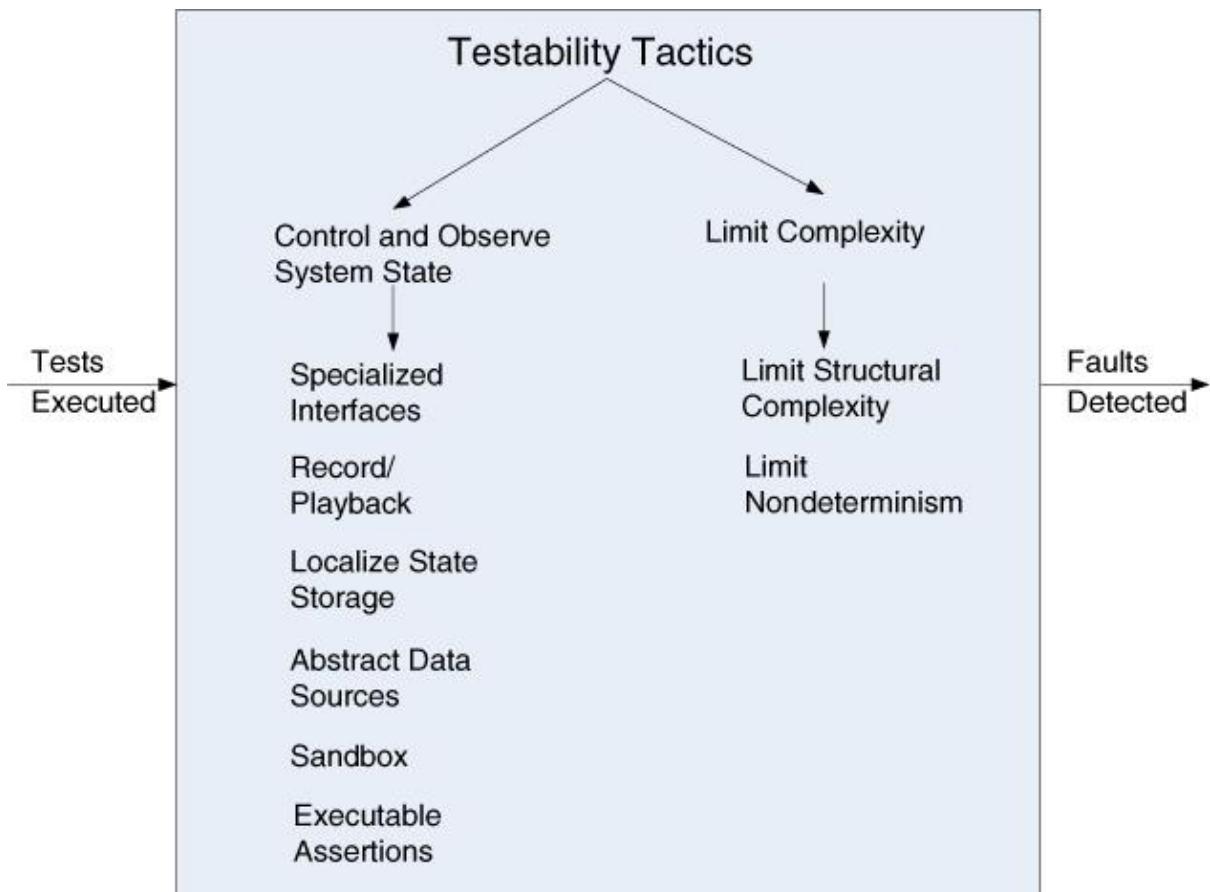


Figure 10.4. Testability tactics

10.3. A Design Checklist for Testability

[Table 10.2](#) is a checklist to support the design and analysis process for testability.

Table 10.2. Checklist to Support the Design and Analysis Process for Testability

Category	Checklist
Allocation of Responsibilities	<p>Determine which system responsibilities are most critical and hence need to be most thoroughly tested.</p> <p>Ensure that additional system responsibilities have been allocated to do the following:</p> <ul style="list-style-type: none">▪ Execute test suite and capture results (external test or self-test)▪ Capture (log) the activity that resulted in a fault <i>or</i> that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault▪ Control and observe relevant system state for testing <p>Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.</p>
Coordination Model	<p>Ensure the system's coordination and communication mechanisms:</p> <ul style="list-style-type: none">▪ Support the execution of a test suite and capture the results within a system or between systems▪ Support capturing activity that resulted in a fault within a system or between systems▪ Support injection and monitoring of state into the communication channels for use in testing, within a system or between systems▪ Do not introduce needless nondeterminism

Data Model	<p>Determine the major data abstractions that must be tested to ensure the correct operation of the system.</p> <ul style="list-style-type: none"> ▪ Ensure that it is possible to capture the values of instances of these data abstractions ▪ Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created ▪ Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured
Mapping among Architectural Elements	<p>Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, and modules to components) so that the desired test response is achieved and potential race conditions identified.</p> <p>In addition, determine whether it is possible to test for illegal mappings of architectural elements.</p>
Resource Management	<p>Ensure there are sufficient resources available to execute a test suite and capture the results. Ensure that your test environment is representative of (or better yet, identical to) the environment in which the system will run. Ensure that the system provides the means to do the following:</p> <ul style="list-style-type: none"> ▪ Test resource limits ▪ Capture detailed resource usage for analysis in the event of a failure ▪ Inject new resource limits into the system for the purposes of testing ▪ Provide virtualized resources for testing
Binding Time	<p>Ensure that components that are bound later than compile time can be tested in the late-bound context.</p> <p>Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure.</p> <p>Ensure that the full range of binding possibilities can be tested.</p>
Choice of Technology	<p>Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on?</p> <p>Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.</p> <hr/> <hr/>

Now That Your Architecture Is Set to Help You Test. . .

By Nick Rozanski, coauthor (with Eoin Woods) of Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives

In addition to architecting your system to make it amenable to testing, you will need to overcome two more specific and daunting challenges when testing very large or complex systems, namely test data and test automation.

Test Data

Your first challenge is how to create large, consistent and useful *test data sets*. This is a significant problem in my experience, particularly for integration testing (that is, testing a number of components to confirm that they work together correctly) and performance testing (confirming that the system meets its requirements for throughput, latency, and response time). For unit tests, and usually for user acceptance tests, the test data is typically created by hand.

For example, you might need 50 products, 100 customers, and 500 orders in your test database, so that you can test the functional steps involved in creating, amending, or deleting orders. This data has to be sufficiently varied to make testing worthwhile, it has to conform to all the referential integrity rules and other constraints of your data model, and you need to be able to calculate and specify the expected results of the tests.

I've seen—and been involved in—two ways of doing this: you either write a system to generate your test data, or you capture a representative data set from the production environment and anonymize it as necessary. (Anonymizing test data involves removing any sensitive information, such as personal data about people or organizations, financial details, and so on.)

Creating your own test data is the ideal, because you know what data you are using and can ensure that it covers all of your edge cases, but it is a lot of effort. Capturing data from the live environment is easier, assuming that there is a system there already, but you don't know what data and hence what coverage you're going to get, and you may have to take extra care to conform to privacy and data protection legislation.

This can have an impact on the system's architecture in a number of ways, and should be given due consideration early on by the architect. For example, the system may need to be able to capture live transactions, or take "snapshots" of live data, which can be used to generate test data. In addition, the test-data-generation system may need an architecture of its own.

Test Automation

Your second challenge is around *test automation*. In practice it is not possible to test large systems by hand because of the number of tests, their complexity, and the amount of checking of results that's required. In the ideal world, you create a test automation framework to do this automatically, which you feed with test data, and set running every night, or even run every time you check in something (the continuous integration model).

This is an area that is given too little attention on many large software development projects. It is often not budgeted for in the project plan, with an unwritten assumption that the effort needed to build it can be somehow "absorbed" into the development costs. A test automation framework can be a significantly complex thing in its own right (which raises the question of how you test it!). It should be scoped and planned like any other project deliverable.

Due consideration should be given to how the framework will invoke functions on the system under test, particularly for testing user interfaces, which is almost without exception a nightmare. (The execution of a UI test is highly dependent on the layout of the windows, the ordering of fields, and so on, which usually changes a lot in heavily user-focused systems. It is sometimes possible to execute window controls programmatically, but in the worst case you may have to record and replay keystrokes or mouse movements.)

There are lots of tools to help with this nowadays, such as Quick Test Pro, TestComplete, or Selenium for testing, and CruiseControl, Hudson, and TeamCity for continuous integration. A comprehensive list on the web can be found here:en.wikipedia.org/wiki/Test_automation.

10.4. Summary

Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system. A vehicle often used to execute the tests is the test harness. Test harnesses are software systems that encapsulate test resources such as test cases and test infrastructure so that it is easy to reapply tests across iterations and it is easy to apply the test infrastructure to new increments of the system. Another vehicle is the creation of test cases prior to the development of a component, so that developers know which tests their component must pass.

Controlling and observing the system state is a major class of testability tactics. Providing the ability to do fault injection, to record system state at key portions of the system, to isolate the system from its environment, and to abstract various resources are all different tactics to support the control and observation of a system and its components.

Complex systems are difficult to test because of the large state space in which their computations take place, and because of the larger number of interconnections among the elements of the system. Consequently, keeping the system simple is another class of tactics that supports testability.

10.5. For Further Reading

An excellent general introduction to software testing is [\[Beizer 90\]](#). For a more modern take on testing, and from the software developer's perspective rather than the tester's, Freeman and Pryce cover test-driven development in the object-oriented realm[\[Freeman 09\]](#).

Bertolino and Strigini [\[Bertolino 96\]](#) are the developers of the model of testing shown in [Figure 10.1](#).

Yin and Bieman [\[Yin 94\]](#) have written about executable assertions. Hartman [\[Hartman 10\]](#) describes a technique for using executable assertions as a means for detecting race conditions.

Bruntink and van Deursen [\[Bruntink 06\]](#) write about the impact of structure on testing.

Jeff Voas's foundational work on testability and the relationship between testability and reliability is worthwhile. There are several papers to choose from, but [\[Voas 95\]](#) is a good start that will point you to others.

10.6. Discussion Questions

1. A testable system is one that gives up its faults easily. That is, if a system contains a fault, then it doesn't take long or much effort to make that fault show up. On the other hand, fault tolerance is all about designing systems that jealously hide their faults; there, the whole idea is to make it very difficult for a system to reveal its faults. Is it possible to design a system that is both highly testable *and* highly fault tolerant, or are these two design goals inherently incompatible? Discuss.
2. "Once my system is in routine use by end users, it should *not* be highly testable, because if it still contains faults—and all systems probably do—then I don't want them to be easily revealed." Discuss.
3. Many of the tactics for testability are also useful for achieving modifiability. Why do you think that is?

- 4.** Write some concrete testability scenarios for an automatic teller machine. How would you modify your design for the automatic teller machine to accommodate these scenarios?
- 5.** What other quality attributes do you think testability is most in conflict with? What other quality attributes do you think testability is most compatible with?
- 6.** One of our tactics is to limit nondeterminism. One method is to use locking to enforce synchronization. What impact does the use of locks have on other quality attributes?
- 7.** Suppose you're building the next great social networking system. You anticipate that within a month of your debut, you will have half a million users. You can't pay half a million people to test your system, and yet it has to be robust and easy to use when all half a million are banging away at it. What should you do? What tactics will help you? Write a testability scenario for this social networking system.
- 8.** Suppose you use executable assertions to improve testability. Make a case for, and then a case against, allowing the assertions to run in the production system as opposed to removing them after testing.

11. Usability

Any darn fool can make something complex; it takes a genius to make something simple.

—Albert Einstein

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or more precisely, the user's perception of quality).

Usability comprises the following areas:

- *Learning system features.* If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier? This might include providing help features.
- *Using a system efficiently.* What can the system do to make the user more efficient in its operation? This might include the ability for the user to redirect the system after issuing a command. For example, the user may wish to suspend one task, perform several operations, and then resume that task.
- *Minimizing the impact of errors.* What can the system do so that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly.
- *Adapting the system to user needs.* How can the user (or the system itself) adapt to make the user's task easier? For example, the system may automatically fill in URLs based on a user's past entries.
- *Increasing confidence and satisfaction.* What does the system do to give the user confidence that the correct action is being taken? For example, providing feedback that indicates that the system is performing a long-running task and the extent to which the task is completed will increase the user's confidence in the system.

11.1. Usability General Scenario

The portions of the usability general scenarios are these:

- *Source of stimulus.* The end user (who may be in a specialized role, such as a system or network administrator) is always the source of the stimulus for usability.
- *Stimulus.* The stimulus is that the end user wishes to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system.
- *Environment.* The user actions with which usability is concerned always occur at runtime or at system configuration time.
- *Artifact.* The artifact is the system or the specific portion of the system with which the user is interacting.
- *Response.* The system should either provide the user with the features needed or anticipate the user's needs.
- *Response measure.* The response is measured by task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs.

[Table 11.1](#) enumerates the elements of the general scenario that characterize usability.

Table 11.1. Usability General Scenario

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system.
Environment	Runtime or configuration time
Artifacts	System or the specific portion of the system with which the user is interacting
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response Measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs

[Figure 11.1](#) gives an example of a concrete usability scenario that you could generate using [Table 11.1](#): The user downloads a new application and is using it productively after two minutes of experimentation.

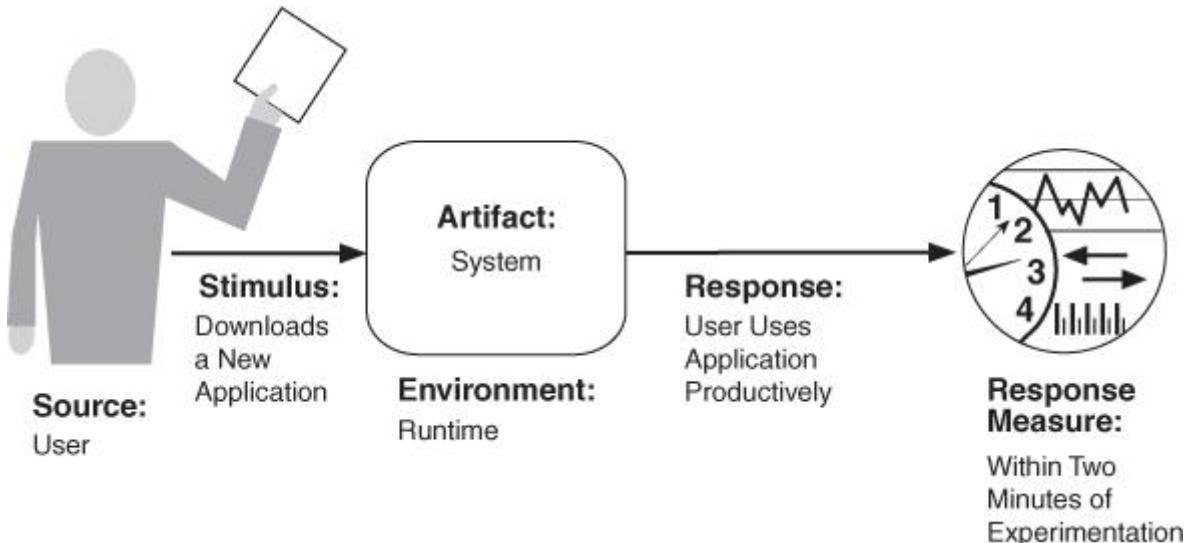


Figure 11.1. Sample concrete usability scenario

11.2. Tactics for Usability

Recall that usability is concerned with how easy it is for the user to accomplish a desired task, as well as the kind of support the system provides to the user. Researchers in human-computer interaction have used the terms *user initiative*, *system initiative*, and *mixed initiative* to describe which of the human-computer pair takes the initiative in performing certain actions and how the interaction proceeds. Usability scenarios can combine initiatives from both perspectives. For example, when canceling a command, the user issues a cancel—user initiative—and the system responds. During the cancel, however, the system may put up a progress indicator—system initiative. Thus, cancel may demonstrate mixed initiative. We use this distinction between user and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

[Figure 11.2](#) shows the goal of the set of runtime usability tactics.

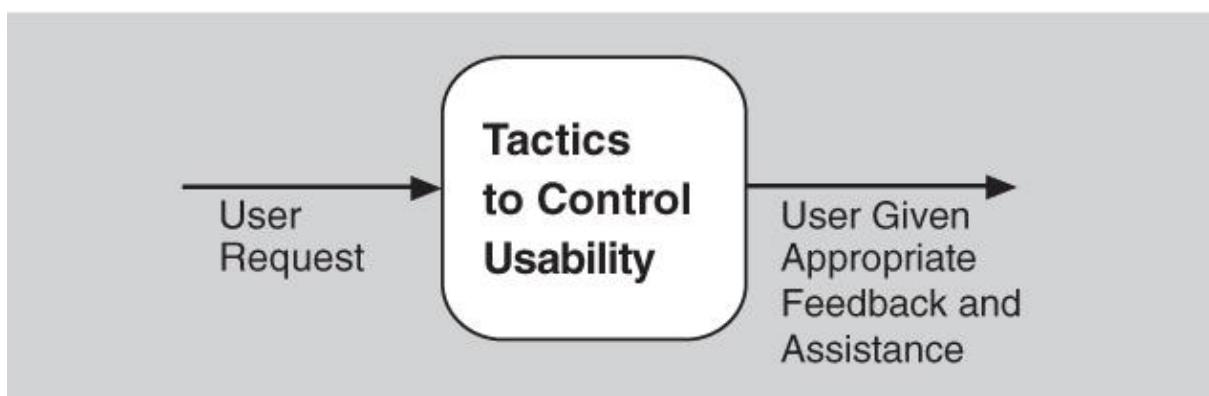


Figure 11.2. The goal of runtime usability tactics

Separate the User Interface!

One of the most helpful things an architect can do to make a system usable is to facilitate experimentation with the user interface via the construction of rapid prototypes. Building a prototype, or several prototypes, to let real users experience the interface and give their

feedback pays enormous dividends. The best way to do this is to design the software so that the user interface can be quickly changed.

Tactics for modifiability that we saw in [Chapter 7](#) support this goal perfectly well, especially these:

- Increase semantic coherence, encapsulate, and co-locate related responsibilities, which localize user interface responsibilities to a single place
- Restrict dependencies, which minimizes the ripple effect to other software when the user interface changes
- Defer binding, which lets you make critical user interface choices without having to recode

Defer binding is especially helpful here, because you can expect that your product's user interface will face pressure to change during testing and even after it goes to market.

User interface generation tools are consistent with these tactics; most produce a single module with an abstract interface to the rest of the software. Many provide the capability to change the user interface after compile time. You can do your part by restricting dependencies on the generated module, should you later decide to adopt a different tool.

Much work in different user interface separation patterns occurred in the 1980s and 90s. With the advent of the web and the modernization of the model-view-controller (MVC) pattern to reflect web interfaces, MVC has become the dominant separation pattern. Now the MVC pattern is built into a wide variety of different frameworks. (See [Chapter 14](#) for a discussion of MVC.) MVC makes it easy to provide multiple views of the data, supporting user initiative, as we discuss next.

Many times quality attributes are in conflict with each other. Usability and modifiability, on the other hand, often complement each other, because one of the best ways to make a system more usable is to make it modifiable. However, this is not always the case. In many systems business rules drive the UI—for example, specifying how to validate input. To realize this validation, the UI may need to call a server (which can negatively affect performance). To get around this performance penalty, the architect may choose to duplicate these rules in the client and the server, which then makes evolution difficult. Alas, the architect's life is never easy!

There is a connection between the achievement of usability and modifiability. The user interface design process consists of generating and then testing a user interface design. Deficiencies in the design are corrected and the process repeats. If the user interface has already been constructed as a portion of the system, then the system must be modified to reflect the latest design. Hence the connection with modifiability. This connection has resulted in standard patterns to support user interface design (see sidebar).

Support User Initiative

Once a system is executing, usability is enhanced by giving the user feedback as to what the system is doing and by allowing the user to make appropriate responses. For example, the tactics described next—*cancel*, *undo*, *pause/resume*, and *aggregate*—support the user in either correcting errors or being more efficient.

The architect designs a response for user initiative by enumerating and allocating the responsibilities of the system to respond to the user command. Here are some common examples of user initiative:

- *Cancel*. When the user issues a cancel command, the system must be listening for it (thus, there is the responsibility to have a constant listener that is not blocked by the actions of whatever is being canceled); the command being canceled must be terminated; any resources being used by the canceled command must be freed; and components that are collaborating with the canceled command must be informed so that they can also take appropriate action.
- *Undo*. To support the ability to undo, the system must maintain a sufficient amount of information about system state so that an earlier state may be restored, at the user's request. Such a record may be in the form of state "snapshots"—for example,

checkpoints—or as a set of reversible operations. Not all operations can be easily reversed: for example, changing all occurrences of the letter “a” to the letter “b” in a document cannot be reversed by changing all instances of “b” to “a”, because some of those instances of “b” may have existed prior to the original change. In such a case the system must maintain a more elaborate record of the change. Of course, some operations, such as ringing a bell, cannot be undone.

- *Pause/resume*. When a user has initiated a long-running operation—say, downloading a large file or set of files from a server—it is often useful to provide the ability to pause and resume the operation. Effectively pausing a long-running operation requires the ability to temporarily free resources so that they may be reallocated to other tasks.
- *Aggregate*. When a user is performing repetitive operations, or operations that affect a large number of objects in the same way, it is useful to provide the ability to aggregate the lower-level objects into a single group, so that the operation may be applied to the group, thus freeing the user from the drudgery (and potential for mistakes) of doing the same operation repeatedly. For example, aggregate all of the objects in a slide and change the text to 14-point font.

Support System Initiative

When the system takes the initiative, it must rely on a model of the user, the task being undertaken by the user, or the system state itself. Each model requires various types of input to accomplish its initiative. The *support system initiative* tactics are those that identify the models the system uses to predict either its own behavior or the user’s intention. Encapsulating this information will make it easier for it to be tailored or modified. Tailoring and modification can be either dynamically based on past user behavior or offline during development. These tactics are the following:

- *Maintain task model*. The task model is used to determine context so the system can have some idea of what the user is attempting and provide assistance. For example, knowing that sentences start with capital letters would allow an application to correct a lowercase letter in that position.
- *Maintain user model*. This model explicitly represents the user’s knowledge of the system, the user’s behavior in terms of expected response time, and other aspects specific to a user or a class of users. For example, maintaining a user model allows the system to pace mouse selection so that not all of the document is selected when scrolling is required. Or a model can control the amount of assistance and suggestions automatically provided to a user. A special case of this tactic is commonly found in user interface *customization*, wherein a user can explicitly modify the system’s user model.
- *Maintain system model*. Here the system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user. A common manifestation of a system model is a progress bar that predicts the time needed to complete the current activity.

[Figure 11.3](#) shows a summary of the tactics to achieve usability.

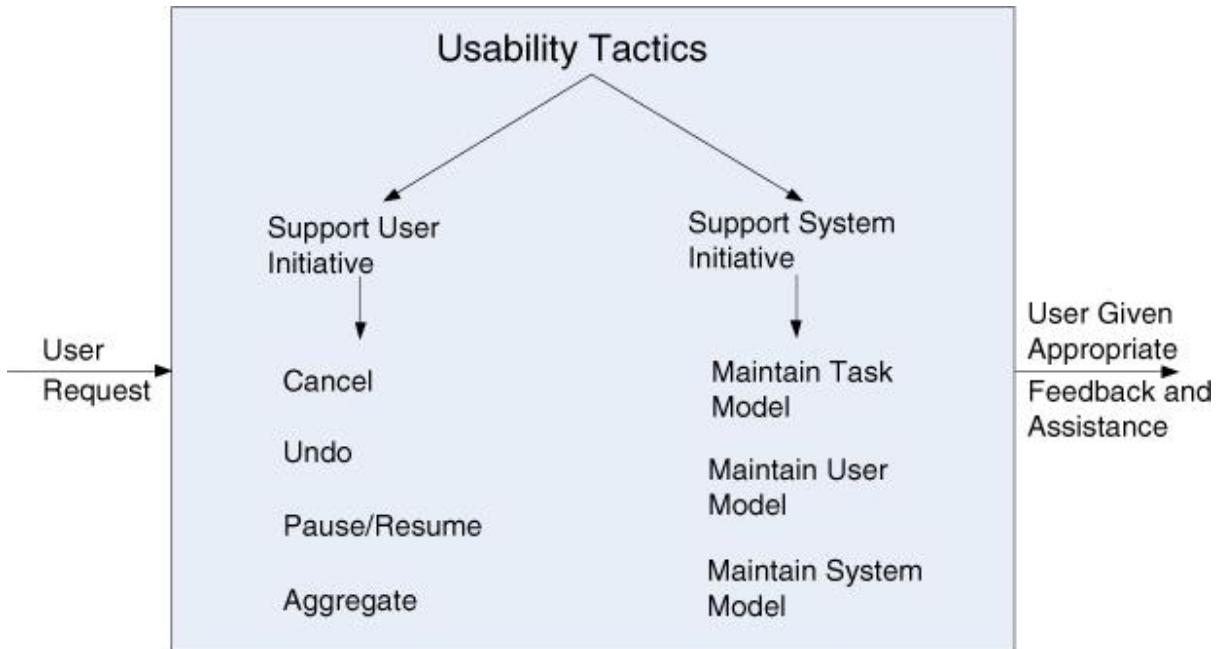


Figure 11.3. Usability tactics

11.3. A Design Checklist for Usability

[Table 11.2](#) is a checklist to support the design and analysis process for usability.

Table 11.2. Checklist to Support the Design and Analysis Process for Usability

Category	Checklist
Allocation of Responsibilities	<p>Ensure that additional system responsibilities have been allocated, as needed, to assist the user in the following:</p> <ul style="list-style-type: none">▪ Learning how to use the system▪ Efficiently achieving the task at hand▪ Adapting and configuring the system▪ Recovering from user and system errors
Coordination Model	<p>Determine whether the properties of system elements' coordination—timeliness, currency, completeness, correctness, consistency—affect how a user learns to use the system, achieves goals or completes tasks, adapts and configures the system, recovers from user and system errors, and gains increased confidence and satisfaction.</p> <p>For example, can the system respond to mouse events and give semantic feedback in real time? Can long-running events be canceled in a reasonable amount of time?</p>
Data Model	<p>Determine the major data abstractions that are involved with user-perceivable behavior. Ensure these major data abstractions, their operations, and their properties have been designed to assist the user in achieving the task at hand, adapting and configuring the system, recovering from user and system errors, learning how to use the system, and increasing satisfaction and user confidence.</p> <p>For example, the data abstractions should be designed to support <i>undo</i> and <i>cancel</i> operations: the transaction granularity should not be so great that canceling or undoing an operation takes an excessively long time.</p>
Mapping among Architectural Elements	<p>Determine what mapping among architectural elements is visible to the end user (for example, the extent to which the end user is aware of which services are local and which are remote). For those that are visible, determine how this affects the ways in which, or the ease with which, the user</p>

	will learn how to use the system, achieve the task at hand, adapt and configure the system, recover from user and system errors, and increase confidence and satisfaction.
Resource Management	Determine how the user can adapt and configure the system's use of resources. Ensure that resource limitations under all user-controlled configurations will not make users less likely to achieve their tasks. For example, attempt to avoid configurations that would result in excessively long response times. Ensure that the level of resources will not affect the users' ability to learn how to use the system, or decrease their level of confidence and satisfaction with the system.
Binding Time	Determine which binding time decisions should be under user control and ensure that users can make decisions that aid in usability. For example, if the user can choose, at runtime, the system's configuration, or its communication protocols, or its functionality via plug-ins, you need to ensure that such choices do not adversely affect the user's ability to learn system features, use the system efficiently, minimize the impact of errors, further adapt and configure the system, or increase confidence and satisfaction.
Choice of Technology	Ensure the chosen technologies help to achieve the usability scenarios that apply to your system. For example, do these technologies aid in the creation of online help, the production of training materials, and the collection of user feedback? How usable are any of your chosen technologies? Ensure the chosen technologies do not adversely affect the usability of the system (in terms of learning system features, using the system efficiently, minimizing the impact of errors, adapting/configuring the system, and increasing confidence and satisfaction).

11.4. Summary

Architectural support for usability involves both allowing the user to take the initiative—in circumstances such as canceling a long-running command or undoing a completed command—and aggregating data and commands.

To be able to predict user or system responses, the system must keep an explicit model of the user, the system, and the task.

There is a strong relationship between supporting the user interface design process and supporting modifiability; this relation is promoted by patterns that enforce separation of the user interface from the rest of the system, such as the MVC pattern.

11.5. For Further Reading

Claire Marie Karat has investigated the relation between usability and business advantage [\[Karat 94\]](#).

Jakob Nielsen has also written extensively on this topic, including a calculation on the ROI of usability [[Nielsen 08](#)].

Bonnie John and Len Bass have investigated the relation between usability and software architecture. They have enumerated around two dozen usability scenarios that have architectural impact and given associated patterns for these scenarios [[Bass 03](#)].

Greg Hartman has defined attentiveness as the ability of the system to support user initiative and allow cancel or pause/[resume](#)[[Hartman 10](#)].

Some of the patterns for separating the user interface are Arch/Slinky, Seeheim, and PAC. These are discussed in [Chapter 8](#) of *Human-Computer Interaction* [[Dix 04](#)].

11.6. Discussion Questions

- 1.** Write a concrete usability scenario for your automobile that specifies how long it takes you to set your favorite radio stations? Now consider another part of the driver experience and create scenarios that test other aspects of the response measures from the general scenario table.
- 2.** Write a concrete usability scenario for an automatic teller machine. How would your design be modified to satisfy these scenarios?
- 3.** How might usability trade off against security? How might it trade off against performance?
- 4.** Pick a few of your favorite web sites that do similar things, such as social networking or online shopping. Now pick one or two appropriate responses from the usability general scenario (such as "achieve the task at hand") and a correspondingly appropriate response measure. Using the response and response measure you chose, compare the web sites' usability.
- 5.** Specify the data model for a four-function calculator that allows undo.
- 6.** Why is it that in so many systems, the cancel button in a dialog box appears to be unresponsive? What architectural principles do you think were ignored in these systems?
- 7.** Why do you think that progress bars frequently behave erratically, moving from 10 to 90 percent in one step and then getting stuck on 90 percent?
- 8.** Research the crash of Air France Flight 296 into the forest at Habsheim, France, on June 26, 1988. The pilots said they were unable to read the digital display of the radio altimeter or hear its audible readout. If they could have, do you believe the crash would have been averted? In this context, discuss the relationship between usability and safety.

12. Other Quality Attributes

Quality is not an act, it is a habit.

—Aristotle

[Chapters 5–11](#) each dealt with a particular quality attribute important to software systems. Each of those chapters discussed how its particular quality attribute is defined, gave a general scenario for that quality attribute, and showed how to write specific scenarios to express precise shades of meaning concerning that quality attribute. And each gave a collection of techniques to achieve that quality attribute in an architecture. In short, each chapter presented a kind of portfolio for specifying and designing to achieve a particular quality attribute.

Those seven chapters covered seven of the most important quality attributes, in terms of their occurrence in modern software-reliant systems. However, as is no doubt clear, seven only begins to scratch the surface of the quality attributes that you might find needed in a software system you're working on.

Is cost a quality attribute? It is not a technical quality attribute, but it certainly affects fitness for use. We consider economic factors in[Chapter 23](#).

This chapter will give a brief introduction to a few other quality attributes—a sort of “B list” of quality attributes—but, more important, show how to build the same kind of specification or design portfolio for a quality attribute not covered in our list.

12.1. Other Important Quality Attributes

Besides the quality attributes we've covered in depth in [Chapters 5–11](#), some others that arise frequently are variability, portability, development distributability, scalability and elasticity, deployability, mobility, and monitorability. We discuss "green" computing in [Section 12.3](#).

Variability

Variability is a special form of modifiability. It refers to the ability of a system and its supporting artifacts such as requirements, test plans, and configuration specifications to support the production of a set of variants that differ from each other in a preplanned fashion. Variability is an especially important quality attribute in a software product line (this will be explored in depth in [Chapter 25](#)), where it means the ability of a core asset to adapt to usages in the different product contexts that are within the product line scope. The goal of variability in a software product line is to make it easy to build and maintain products in the product line over a period of time. Scenarios for variability will deal with the binding time of the variation and the people time to achieve it.

Portability

Portability is also a special form of modifiability. Portability refers to the ease with which software that was built to run on one platform can be changed to run on a different platform. Portability is achieved by minimizing platform dependencies in the software, isolating dependencies to well-identified locations, and writing the software to run on a "virtual machine" (such as a Java Virtual Machine) that encapsulates all the platform dependencies within. Scenarios describing portability deal with moving software to a new platform by expending no more than a certain level of effort or by counting the number of places in the software that would have to change.

Development Distributability

Development distributability is the quality of designing the software to support distributed software development. Many systems these days are developed using globally distributed teams. One problem that must be overcome when developing with distributed teams is coordinating their activities. The system should be designed so that coordination among teams is minimized. This minimal coordination needs to be achieved both for the code and for the data model. Teams working on modules that communicate with each other may need to negotiate the interfaces of those modules. When a module is used by many other modules, each developed by a different team, communication and negotiation become more complex and burdensome. Similar considerations apply for the data model. Scenarios for development distributability will deal with the compatibility of the communication structures and data model of the system being developed and the coordination mechanisms of the organizations doing the development.

Scalability

Two kinds of scalability are horizontal scalability and vertical scalability. Horizontal scalability (scaling out) refers to adding more resources to logical units, such as adding another server to a cluster of servers. Vertical scalability (scaling up) refers to adding more resources to a physical unit, such as adding more memory to a single computer. The problem that arises with either type of scaling is how to effectively utilize the additional resources. Being *effective* means that the additional resources result in a measurable improvement of some system quality, did not require undue effort to add, and did not disrupt operations. In cloud environments, horizontal scalability is called *elasticity*. Elasticity is a property that enables a customer to add or remove virtual machines from the resource pool (see [Chapter 26](#) for further discussion of such environments). These virtual machines are hosted on a large collection of upwards of 10,000 physical machines that are managed by the cloud provider. Scalability scenarios will deal with the impact of adding or removing resources, and the measures will reflect associated availability and the load assigned to existing and new resources.

Deployability

Deployability is concerned with how an executable arrives at a host platform and how it is subsequently invoked. Some of the issues involved in deploying software are: How does it arrive at its host (push, where updates are sent to users unbidden, or pull, where users must explicitly request updates)? How is it integrated into an existing system? Can this be done while the existing system is executing? Mobile systems have their own problems in terms of how they are updated, because of concerns about bandwidth. Deployment scenarios will deal with the type of update (push or pull), the form of the update (medium, such as DVD or Internet download, and packaging, such as executable, app, or plug-in), the resulting integration into an existing system, the efficiency of executing the process, and the associated risk.

Mobility

Mobility deals with the problems of movement and affordances of a platform (e.g., size, type of display, type of input devices, availability and volume of bandwidth, and battery life). Issues in mobility include battery management, reconnecting after a period of disconnection, and the number of different user interfaces necessary to support multiple platforms. Scenarios will deal with specifying the desired effects of mobility or the various affordances. Scenarios may also deal with variability, where the same software is deployed on multiple (perhaps radically different) platforms.

Monitorability

Monitorability deals with the ability of the operations staff to monitor the system while it is executing. Items such as queue lengths, average transaction processing time, and the health of various components should be visible to the operations staff so that they can take corrective action in case of potential problems. Scenarios will deal with a potential problem and its visibility to the operator, and potential corrective action.

Safety

In 2009 an employee of the Shushenskaya hydroelectric power station in Siberia sent commands over a network to remotely, and accidentally, activate an unused turbine. The offline turbine created a “water hammer” that flooded and then destroyed the plant and killed dozens of workers.

The thought that software could kill people used to belong in the realm of kitschy computers-run-amok science fiction. Sadly, it didn’t stay there. As software has come to control more and more of the devices in our lives, software safety has become a critical concern.

Safety is not purely a software concern, but a concern for any system that can affect its environment. As such it receives mention in [Section 12.3](#), where we discuss system quality attributes. But there are means to address safety that are wholly in the software realm, which is why we discuss it here as well.

Software safety is about the software’s ability to avoid entering states that cause or lead to damage, injury, or loss of life to actors in the software’s environment, and to recover and limit the damage when it does enter into bad states. Another way to put this is that safety is concerned with the prevention of and recovery from hazardous failures. Because of this, the architectural concerns with safety are almost identical to those for availability, which is also about avoiding and recovering from failures. Tactics for safety, then, overlap with those for availability to a large degree. Both comprise tactics to prevent failures and to detect and recover from failures that do occur.

Safety is not the same as reliability. A system can be reliable (consistent with its specification) but still unsafe (for example, when the specification ignores conditions leading to unsafe action). In fact, paying careful attention to the specification for safety-critical software is perhaps the most powerful thing you can do to produce safe software. Failures and hazards cannot be detected, prevented, or ameliorated if the software has not been designed with them in mind. Safety is frequently engineered by performing failure mode and effects analysis, hazard analysis, and fault tree analysis. (These techniques are discussed in [Chapter 5](#).) These techniques are intended to discover possible hazards that could result from the system’s operation and provide plans to cope with these hazards.

12.2. Other Categories of Quality Attributes

We have primarily focused on product qualities in our discussions of quality attributes, but there are other types of quality attributes that measure “goodness” of something other than the final product. Here are three:

Conceptual Integrity of the Architecture

Conceptual integrity refers to consistency in the design of the architecture, and it contributes to the understandability of the architecture and leads to fewer errors of confusion. Conceptual integrity demands that the same thing is done in the same way through the architecture. In an architecture with conceptual integrity, less is more. For example, there are countless ways that components can send information to each other: messages, data structures, signaling of events, and so forth. An architecture with conceptual integrity would feature one way only, and only provide alternatives if there was a compelling reason to do so. Similarly, components should all report and handle errors in the same way, log events or transactions in the same way, interact with the user in the same way, and so forth.

Quality in Use

ISO/IEC 25010, which we discuss in [Section 12.4](#), has a category of qualities that pertain to the use of the system by various stakeholders. For example, time-to-market is an important characteristic of a system, but it is not discernible from an examination of the product itself. Some of the qualities in this category are these:

- *Effectiveness.* This refers to the distinction between building the system correctly (the system performs according to its requirements) and building the correct system (the system performs in the manner the user wishes). Effectiveness is a measure of whether the system is correct.
- *Efficiency.* The effort and time required to develop a system. Put another way, what is the architecture’s impact on the project’s cost and schedule? Would a different set of architectural choices have resulted in a system that would be faster or cheaper to bring to fruition? Efficiency can include training time for developers; an architecture that uses technology unfamiliar to the staff on hand is less buildable. Is the architecture appropriate for the organization in terms of its experience and its available supporting infrastructure (such as test facilities or development environments)?
- *Freedom from risk.* The degree to which a product or system affects economic status, human life, health, or the environment.

A special case of efficiency is how easy it is to build (that is, compile and assemble) the system after a change. This becomes critical during testing. A recompile process that takes hours or overnight is a schedule-killer. Architects have control over this by managing dependencies among modules. If the architect doesn’t do this, then what often happens is that some bright-eyed developer writes a makefile early on, it works, and people add to it and add to it. Eventually the project ends up with a seven-hour compile step and very unhappy integrators and testers who are already behind schedule (because they always are).

Marketability

An architecture’s marketability is another quality attribute of concern. Some systems are well known by their architectures, and these architectures sometimes carry a meaning all their own, independent of what other quality attributes they bring to the system. The current craze in building cloud-based systems has taught us that the perception of an architecture can be more important than the qualities the architecture brings. Many organizations have felt they had to build cloud-based systems (or some other *technology du jour*) whether or not that was the correct technical choice.

12.3. Software Quality Attributes and System Quality Attributes

Physical systems, such as aircraft or automobiles or kitchen appliances, that rely on software embedded within are designed to meet a whole other litany of quality attributes: weight, size, electric consumption, power output, pollution output, weather resistance, battery life, and on and on. For many of these systems, safety tops the list (see the sidebar).

Sometimes the software architecture can have a surprising effect on the system's quality attributes. For example, software that makes inefficient use of computing resources might require additional memory, a faster processor, a bigger battery, or even an additional processor. Additional processors can add to a system's power consumption, weight, required cabinet space, and of course expense.

Green computing is an issue of growing concern. Recently there was a controversy about how much greenhouse gas was pumped into the atmosphere by Google's massive processor farms. Given the daily output and the number of daily requests, it is possible to estimate how much greenhouse gas *you* cause to be emitted each time *you* ask Google to perform a search. (Current estimates range from 0.2 grams to 7 grams of CO₂.) Green computing is all the rage. Eve Troeh, on the American Public Media show "Marketplace" (July 5, 2011), reports:

Two percent of all U.S. electricity now goes to data centers, according to the Environmental Protection Agency. Electricity has become the biggest cost for processing data—more than the equipment to do it, more than the buildings to house that equipment. . . . Google's making data servers that can float offshore, cooled by ocean breezes. HP has plans to put data servers near farms, and power them with methane gas from cow pies.

The lesson here is that if you are the architect for software that resides in a larger system, you will need to understand the quality attributes that are important for the containing system to achieve, and work with the system architects and engineers to see how your software architecture can contribute to achieving them.

The Vanishing Line between Software and System Qualities

This is a book about software architecture, and so we treat quality attributes from a software architect's perspective. But you may have already noticed that the quality attributes that the software architect can bring to the party are limited by the architecture of the system in which the software runs.

For example:

- The performance of a piece of software is fundamentally constrained by the performance of the computer that runs it. No matter how well you design the software, you just can't run the latest whole-earth weather forecasting models on Grampa's Commodore 64 and hope to know if it's going to rain tomorrow.
- Physical security is probably more important and more effective than software security at preventing fraud and theft. If you don't believe this, write your laptop's password on a slip of paper, tape it to your laptop, and leave it in an unlocked car with the windows down. (Actually, don't really do that. Consider this a thought experiment.)
- If we're being perfectly honest here, how usable is a device for web browsing that has a screen smaller than a credit card and keys the size of a raisin?

For me, nowhere is the barrier between software and system more nebulous than in the area of safety. The thought that software—strings of 0's and 1's—can kill or maim or destroy is still an unnatural notion. Of course, it's not the 0's and 1's that wreak havoc. At least, not directly. It's what they're connected to. Software, and the system in which it runs, has to be connected to the outside world in some way before it can do damage. That's the good news. The bad news is that the good news isn't all that good. Software *is* connected to the outside world, always. If your program has no effect whatsoever that is observable outside of itself, it probably serves no purpose.

There are notorious examples of software-related failures. The Siberian hydroelectric plant catastrophe mentioned in the text, the Therac-25 fatal radiation overdose, the Ariane 5 explosion, and a hundred lesser known accidents all caused harm because the *software* was part of a *system* that included a turbine, an X-ray emitter, or a rocket's steering controls, in the examples just cited. In these cases, flawed software commanded some hardware in the system to take a disastrous action, and the hardware simply obeyed. Actuators are devices that connect hardware to software; they are the bridge between the world of 0's and 1's and the world of motion and control. Send a digital value to an actuator (or write a bit string in the hardware register corresponding to the actuator) and that value is translated to some mechanical action, for better or worse.

But connection to an actuator is not required for software-related disasters. Sometimes all the computer has to do is send erroneous information to its human operators. In September 1983, a Soviet satellite sent data to its ground system computer, which interpreted that data as a missile launched from the United States aimed at Moscow. Seconds later, the computer reported a second missile in flight. Soon, a third, then a fourth, and then a fifth appeared. Soviet Strategic Rocket Forces lieutenant colonel Stanislav Yevgrafovich Petrov made the astonishing decision to ignore the warning system, believing it to be in error. He thought it extremely unlikely that the U.S. would have fired just a few missiles, thereby inviting total retaliatory destruction. He decided to wait it out, to see if the missiles were real—that is, to see if his country's capital city was going to be incinerated. As we know, it wasn't. The Soviet system had mistaken a rare sunlight condition for missiles in flight. Similar mistakes have occurred on the U.S. side.

Of course, the humans don't always get it right. On the dark and stormy night of June 1, 2009, Air France flight 447 from Rio de Janeiro to Paris plummeted into the Atlantic Ocean, killing all on board. The Airbus A-330's flight recorders were not recovered until May 2011, and as this book goes to publication it appears that the pilots never knew that the aircraft had entered a high-altitude stall. The sensors that measure airspeed had become clogged with ice and therefore unreliable. The software was required to disengage the autopilot in this situation, which it did. The human pilots thought the aircraft was going too fast (and in danger of structural failure) when in fact it was going too slow (and falling). During the entire three-minute-plus plunge from 38,000 feet, the pilots kept trying to pull the nose up and throttles back to lower the speed. It's a good bet that adding to the confusion was the way the A-330's stall warning system worked. When the system detects a stall, it emits a loud audible alarm. The computers deactivate the stall warning when they "think" that the angle of attack measurements are invalid. This can occur when the airspeed readings are very low. That is exactly what happened with Air France 447: Its forward speed dropped below 60 knots, and the angle of attack was extremely high. As a consequence of a rule in the flight control *software*, the stall warning stopped and started several times. Worse, it came on whenever the pilot let the nose fall a bit (increasing the airspeed and taking the readings into the "valid" range, but still in stall) and then stopped when he pulled back. That is, doing the right thing resulted in the wrong feedback and vice versa.

Was this an unsafe system, or a safe system unsafely operated? Ultimately the courts will decide.

Software that can physically harm us is a fact of our modern life. Sometimes the link between software and physical harm is direct, as in the Ariane example, and sometimes it's much more tenuous, as in the Air France 447 example. But as software professionals, we cannot take refuge in the fact that our software can't actually inflict harm any more than the person who shouts "Fire!" in a crowded theater can claim it was the stampede, not the shout, that caused injury.

—PCC

12.4. Using Standard Lists of Quality Attributes—or Not

Architects have no shortage of lists of quality attributes for software systems at their disposal. The standard with the pause-and-take-a-breath title of “ISO/IEC FCD 25010: Systems and software engineering—Systems and software product Quality Requirements and Evaluation (SQuaRE)—System and software quality models,” is a good example. The standard divides quality attributes into those supporting a “quality in use” model and those supporting a “product quality” model. That division is a bit of a stretch in some places, but nevertheless begins a divide-and-conquer march through a breathtaking array of qualities. See [Figure 12.1](#) for this array.

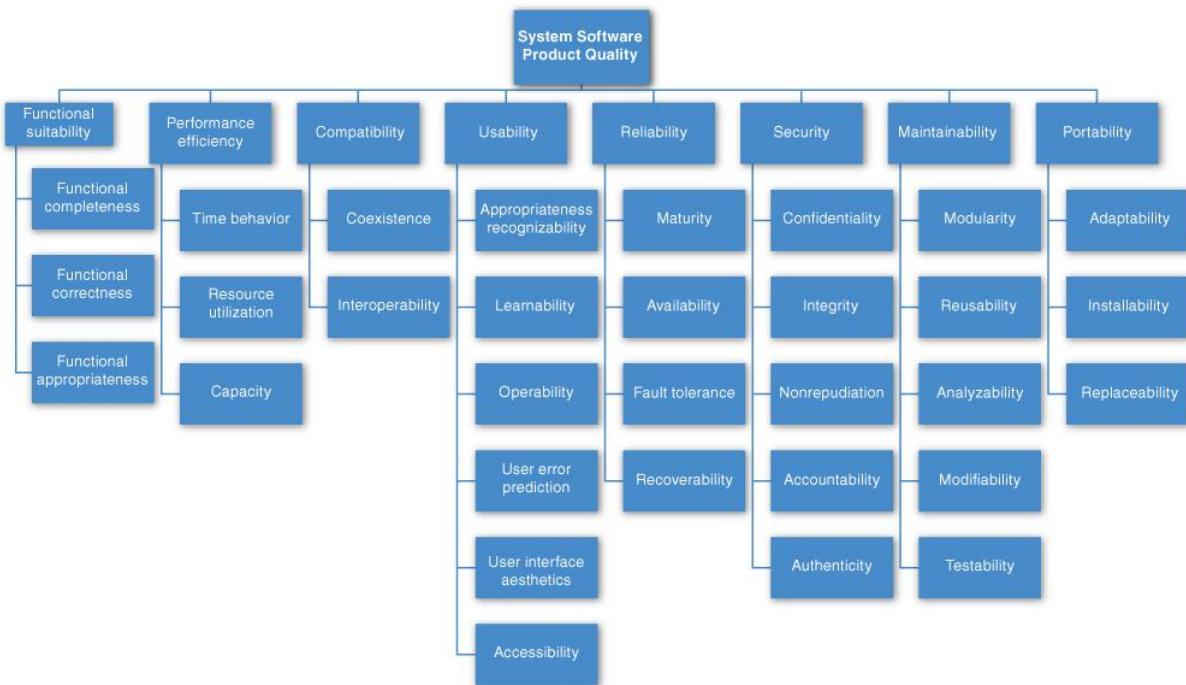


Figure 12.1. The ISO/IEC FCD 25010 product quality standard

The standard lists the following quality attributes that deal with product quality:

- *Functional suitability*. The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- *Performance efficiency*. Performance relative to the amount of resources used under stated conditions
- *Compatibility*. The degree to which a product, system, or component can exchange information with other products, systems, or components, and/or perform its required functions, while sharing the same hardware or software environment
- *Usability*. The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use
- *Reliability*. The degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time
- *Security*. The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
- *Maintainability*. The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- *Portability*. The degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another

In ISO 25010, these “quality characteristics” are each composed of “quality subcharacteristics” (for example, nonrepudiation is a subcharacteristic of security). The standard slogs through almost five dozen separate descriptions of quality subcharacteristics in this way. It defines for us the qualities of “pleasure” and “comfort.” It distinguishes among “functional correctness” and “functional completeness,” and then adds “functional appropriateness” for good measure. To exhibit “compatibility,” systems must either have “interoperability” or just plain “coexistence.” “Usability” is a product quality, not a quality-in-use quality, although it includes “satisfaction,” which *is* a quality-in-use quality. “Modifiability” and “testability” are both part of “maintainability.” So is “modularity,” which is a strategy for achieving a quality rather than a goal in its own right. “Availability” is part of “reliability.” “Interoperability” is part of “compatibility.” And “scalability” isn’t mentioned at all.

Got all that?

Lists like these—and there are many—do serve a purpose. They can be helpful checklists to assist requirements gatherers in making sure that no important needs were overlooked. Even more useful than standalone lists, they can serve as the basis for creating your own checklist that contains the quality attributes of concern in your domain, your industry, your organization, and your products. Quality attribute lists can also serve as the basis for establishing measures. If “pleasure” turns out to be an important concern in your system, how do you measure it to know if your system is providing enough of it?

However, general lists like these also have drawbacks. First, no list will ever be complete. As an architect, you will be called upon to design a system to meet a stakeholder concern not foreseen by any list-maker. For example, some writers speak of “manageability,” which expresses how easy it is for system administrators to manage the application. This can be achieved by inserting useful instrumentation for monitoring operation and for debugging and performance tuning. We know of an architecture that was designed with the conscious goal of retaining key staff and attracting talented new hires to a quiet region of the American Midwest. That system’s architects spoke of imbuing the system with “Iowability.” They achieved it by bringing in state-of-the-art technology and giving their development teams wide creative latitude. Good luck finding “Iowability” in any standard list of quality attributes, but that QA was as important to that organization as any other.

Second, lists often generate more controversy than understanding. You might argue persuasively that “functional correctness” should be part of “reliability,” or that “portability” is just a kind of “modifiability,” or that “maintainability” is a kind of “modifiability” (not the other way around). The writers of ISO 25010 apparently spent time and effort deciding to make security its own characteristic, instead of a subcharacteristic of functionality, which it was in a previous version. We believe that effort in making these arguments could be better spent elsewhere.

Third, these lists often purport to be *taxonomies*, which are lists with the special property that every member can be assigned to exactly one place. Quality attributes are notoriously squishy in this regard. We discussed denial of service as being part of security, availability, performance, and usability in [Chapter 4](#).

Finally, these lists force architects to pay attention to every quality attribute on the list, even if only to finally decide that the particular quality attribute is irrelevant to their system. Knowing how to quickly decide that a quality attribute is irrelevant to a specific system is a skill gained over time.

These observations reinforce the lesson introduced in [Chapter 4](#) that quality attribute names, by themselves, are largely useless and are at best invitations to begin a conversation; that spending time worrying about what qualities are subqualities of what other qualities is also almost useless; and that scenarios provide the best way for us to specify precisely what we mean when we speak of a quality attribute.

Use standard lists of quality attributes to the extent that they are helpful as checklists, but don’t feel the need to slavishly adhere to their terminology.

12.5. Dealing with “X-ability”: Bringing a New Quality Attribute into the Fold

Suppose, as an architect, you must deal with a quality attribute for which there is no compact body of knowledge, no “portfolio” like [Chapters 5–11](#) provided for those seven QAs? Suppose you find yourself having to deal with a quality attribute like “green computing” or “manageability” or even “Iowability”? What do you do?

Capture Scenarios for the New Quality Attribute

The first thing to do is interview the stakeholders whose concerns have led to the need for this quality attribute. You can work with them, either individually or as a group, to build a set of attribute characterizations that refine what is meant by the QA. For example, security is often decomposed into concerns such as confidentiality, integrity, availability, and others. After that refinement, you can work with the stakeholders to craft a set of specific scenarios that characterize what is meant by that QA.

Once you have a set of specific scenarios, then you can work to generalize the collection. Look at the set of stimuli you’ve collected, the set of responses, the set of response measures, and so on. Use those to construct a general scenario by making each part of the general scenario a generalization of the specific instances you collected.

In our experience, the steps described so far tend to consume about half a day.

Assemble Design Approaches for the New Quality Attribute

After you have a set of guiding scenarios for the QA, you can assemble a set of design approaches for dealing with it. You can do this by

1. Revisiting a body of patterns you’re familiar with and asking yourself how each one affects the QA of interest.
2. Searching for designs that have had to deal with this QA. You can search on the name you’ve given the QA itself, but you can also search for the terms you chose when you refined the QA into subsidiary attribute characterizations (such as “confidentiality” for the QA of security).
3. Finding experts in this area and interviewing them or simply writing and asking them for advice.
4. Using the general scenario to try to catalog a list of design approaches to produce the responses in the response category.
5. Using the general scenario to catalog a list of ways in which a problematic architecture would fail to produce the desired responses, and thinking of design approaches to head off those cases.

Model the New Quality Attribute

If you can build a conceptual model of the quality attribute, this can be helpful in creating a set of design approaches for it. By “model,” we don’t mean anything more than understanding the set of parameters to which the quality attribute is sensitive. For example, a model of modifiability might tell us that modifiability is a function of how many places in a system have to be changed in response to a modification, and the interconnectedness of those places. A model for performance might tell us that throughput is a function of transactional workload, the dependencies among the transactions, and the number of transactions that can be processed in parallel.

Once you have a model for your QA, then you can work to catalog the architectural approaches (tactics and patterns) open to you for manipulating each of the relevant parameters in your favor.

Assemble a Set of Tactics for the New Quality Attribute

There are two sources that can be used to derive tactics for any quality attribute: models and experts.

[Figure 12.2](#) shows a queuing model for performance. Such models are widely used to analyze the latency and throughput of various types of queuing systems, including manufacturing and service environments, as well as computer systems.

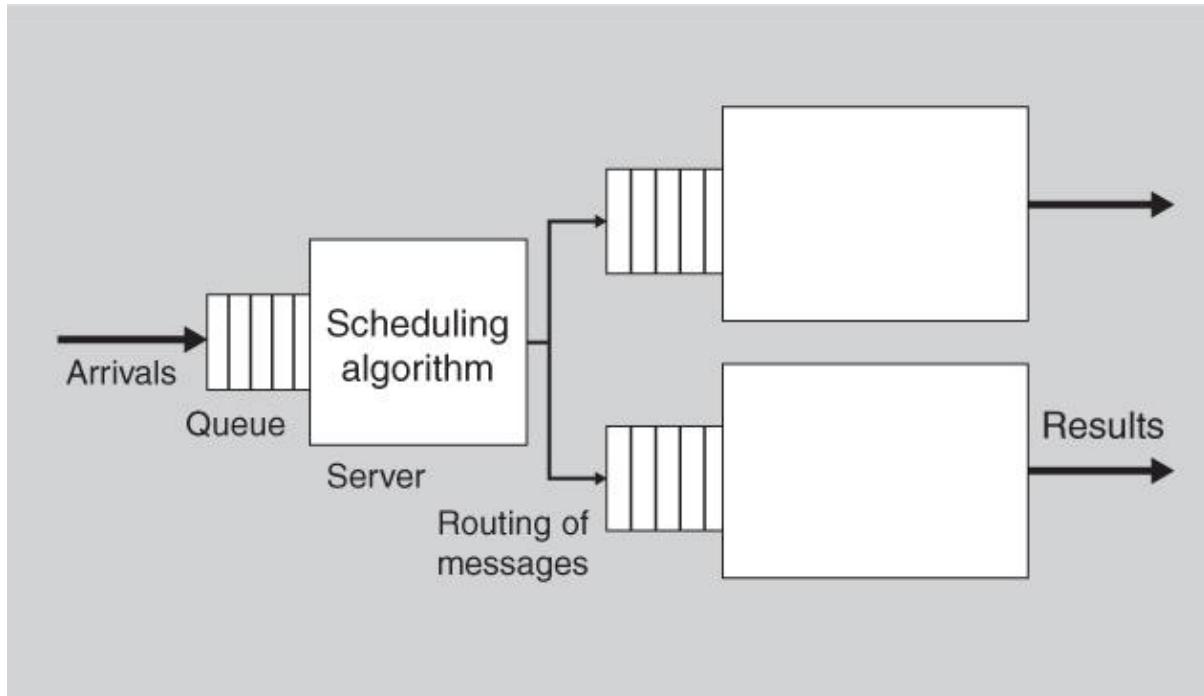


Figure 12.2. A generic queuing model

Within this model, there are seven parameters that can affect the latency that the model predicts:

- Arrival rate
- Queuing discipline
- Scheduling algorithm
- Service time
- Topology
- Network bandwidth
- Routing algorithm

These are the *only* parameters that can affect latency within this model. This is what gives the model its power. Furthermore, each of these parameters can be affected by various architectural decisions. This is what makes the model useful for an architect. For example, the routing algorithm can be fixed or it could be a load-balancing algorithm. A scheduling algorithm must be chosen. The topology can be affected by dynamically adding or removing new servers. And so forth.

The process of generating tactics based on a model is this:

- Enumerate the parameters of the model
- For each parameter, enumerate the architectural decisions that can affect this parameter

What results is a list of tactics to, in the example case, control performance and, in the more general case, to control the quality attribute that the model is concerned with. This makes the design problem seem much more tractable. This list of tactics is finite and reasonably small, because the number of parameters of the model is bounded, and for each parameter, the number of architectural decisions to affect the parameter is limited.

Deriving tactics from models is fine as long as the quality attribute in question has a model. Unfortunately, the number of such models is limited and is a subject of active research. There are no good *architectural* models for usability or security, for example. In the cases where we had no model to work from, we did four things to catalog the tactics:

1. We interviewed experts in the field, asking them what they do as architects to improve the quality attribute response.
2. We examined systems that were touted as having high usability (or testability, or whatever tactic we were focusing on).
3. We scoured the relevant design literature looking for common themes in design.
4. We examined documented architectural patterns to look for ways they achieved the quality attribute responses touted for them.

Construct Design Checklists for the New Quality Attribute

Finally, examine the seven categories of design decisions in [Chapter 4](#) and ask yourself (or your experts) how to specialize your new quality of interest to these categories. In particular, think about reviewing a software architecture and trying to figure out how well it satisfies your new qualities in these seven categories. What questions would you ask the architect of that system to understand how the design attempts to achieve the new quality? These are the basis for the design checklist.

12.6. For Further Reading

For most of the quality attributes we discussed in this chapter, the Internet is your friend. You can find reasonable discussions of scalability, portability, and deployment strategies using your favorite search engine. Mobility is harder to find because it has so many meanings, but look under “mobile computing” as a start.

Distributed development is a topic covered in the International Conference on Global Software Engineering, and looking at the proceedings of this conference will give you access to the latest research in this area (www.icgse.org).

Release It! [\[Nygard 07\]](#) has a good discussion of monitorability (which he calls transparency) as well as potential problems that are manifested after extended operation of a system. The book also includes various patterns for dealing with some of the problems.

To gain an appreciation for the importance of software safety, we suggest reading some of the disaster stories that arise when software fails. A venerable source is the ACM Risks Forum newsgroup, known as comp.risks in the USENET community, available at www.risks.org. This list has been moderated by Peter Neumann since 1985 and is still going strong.

Nancy Leveson is an undisputed thought leader in the area of software and safety. If you’re working in safety-critical systems, you should become familiar with her work. You can start small with a paper like [\[Leveson 04\]](#), which discusses a number of software-related factors that have contributed to spacecraft accidents. Or you can start at the top with [\[Leveson 11\]](#), a book that treats safety in the context of today’s complex, sociotechnical, software-intensive systems.

The Federal Aviation Administration is the U.S. government agency charged with oversight of the U.S. airspace system, and the agency is extremely concerned about safety. Their 2000 System Safety Handbook is a good practical overview of the topic [\[FAA 00\]](#).

IEEE STD-1228-1994 (“Software Safety Plans”) defines best practices for conducting software safety hazard analyses, to help ensure that requirements and attributes are specified for safety-critical software [\[IEEE 94\]](#). The aeronautical standard DO-178B (due to be replaced by DO-178C as this book goes to publication) covers software safety requirements for aerospace applications.

A discussion of safety tactics can be found in the work of Wu and Kelly [\[Wu 06\]](#).

In particular, interlocks are an important tactic for safety. They enforce some safe sequence of events, or ensure that a safe condition exists before an action is taken. Your microwave oven shuts off when you open the door because of a hardware interlock. Interlocks can be implemented in software also. For an interesting case study of this, see [\[Wozniak 07\]](#).

12.7. Discussion Questions

1. The Kingdom of Bhutan measures the happiness of its population, and government policy is formulated to increase Bhutan's GNH (gross national happiness). Go read about how the GNH is measured (try www.grossnationalhappiness.com) and then sketch a general scenario for the quality attribute of *happiness* that will let you express concrete happiness requirements for a software system.
2. Choose a quality attribute not described in [Chapters 5–11](#). For that quality attribute, assemble a set of specific scenarios that describe what you mean by it. Use that set of scenarios to construct a general scenario for that quality attribute.
3. For the QA you chose for discussion question 2, assemble a set of design approaches (patterns and tactics) that help you achieve it.
4. For the QA you chose for discussion question 2, develop a design checklist for that quality attribute using the seven categories of guiding quality design decisions outlined in [Chapter 4](#).
5. What might cause you to add a tactic or pattern to the sets of quality attributes already described in [Chapters 5–11](#) (or any other quality attribute, for that matter)?
6. According to slate.com and other sources, a teenage girl in Germany "went into hiding after she forgot to set her Facebook birthday invitation to private and accidentally invited the entire Internet. After 15,000 people confirmed they were coming, the girl's parents canceled the party, notified police, and hired private security to guard their home." Fifteen hundred people showed up anyway; several minor injuries ensued. Is Facebook "unsafe"? Discuss.
7. Author James Gleick ("A Bug and a Crash," www.around.com/ariane.html) writes that "It took the European Space Agency 10 years and \$7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch... . All it took to explode that rocket less than a minute into its maiden voyage. . . was a small computer program trying to stuff a 64-bit number into a 16-bit space. One bug, one crash. Of all the careless lines of code recorded in the annals of computer science, this one may stand as the most devastatingly efficient." Write a safety scenario that addresses the Ariane 5 disaster and discuss tactics that might have prevented it.
8. Discuss how you think development distributability tends to "trade off" against the quality attributes of performance, availability, modifiability, and interoperability.
Extra Credit: Close your eyes and, without peeking, spell "distributability." Bonus points for successfully saying "development distributability" three times as fast as you can.
9. What is the relationship between mobility and security?
10. Relate monitorability to observability and controllability, the two parts of testability. Are they the same? If you want to make your system more of one, can you just optimize for the other?

13. Architectural Tactics and Patterns

I have not failed. I've just found 10,000 ways that won't work.

—Thomas Edison

There are many ways to do design badly, and just a few ways to do it well. Because success in architectural design is complex and challenging, designers have been looking for ways to capture and reuse hard-won architectural knowledge. Architectural patterns and tactics are ways of capturing proven good design structures, so that they can be reused.

Architectural patterns have seen increased interest and attention, from both software practitioners and theorists, over the past 15 years or more. An architectural pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a *class* of architectures.

Because patterns are (by definition) found in practice, one does not invent them; one discovers them. Cataloging patterns is akin to the job of a Linnaean botanist or zoologist: “discovering” patterns and describing their shared characteristics. And like the botanist, zoologist, or ecologist, the pattern cataloger strives to understand how the characteristics lead to different behaviors and different responses to environmental conditions. For this reason there will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions, and as long as those conditions change, new patterns will emerge.

Architectural design seldom starts from first principles. Experienced architects typically think of creating an architecture as a process of selecting, tailoring, and combining patterns. The software architect must decide how to instantiate a pattern—how to make it fit with the specific context and the constraints of the problem.

In [Chapters 5–11](#) we have seen a variety of architectural tactics. These are simpler than patterns. Tactics typically use just a single structure or computational mechanism, and they are meant to address a single architectural force. For this reason they give more precise control to an architect when making design decisions than patterns, which typically combine multiple design decisions into a package. Tactics are the “building blocks” of design, from which architectural patterns are created. Tactics are atoms and patterns are molecules. Most patterns consist of (are constructed from) several different tactics. For this reason we say that patterns package tactics.

In this chapter we will take a very brief tour through the patterns universe, touching on some of the most important and most commonly used patterns for architecture, and we will then look at the relationships between patterns and tactics: showing how a pattern is constructed from tactics, and showing how tactics can be used to tailor patterns when the pattern that you find in a book or on a website doesn’t quite address your design needs.

13.1. Architectural Patterns

An architectural pattern establishes a relationship between:

- *A context.* A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context. The pattern description outlines the problem and its variants, and describes any complementary or opposing forces. The description of the problem often includes quality attributes that must be met.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution describes the architectural structures that solve the problem, including how to balance the many forces at work. The solution will describe the responsibilities of and static relationships among elements (using a module structure), or it will describe the runtime behavior of and interaction between elements (laying out a component-and-connector or allocation structure). The solution for a pattern is determined and described by:
 - A set of element types (for example, data repositories, processes, and objects)
 - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
 - A topological layout of the components
 - A set of semantic constraints covering topology, element behavior, and interaction mechanisms

The solution description should also make clear what quality attributes are provided by the static and runtime configurations of elements.

This *{context, problem, solution}* form constitutes a template for documenting a pattern.

Complex systems exhibit multiple patterns at once. A web-based system might employ a three-tier client-server architectural pattern, but within this pattern it might also use replication (mirroring), proxies, caches, firewalls, MVC, and so forth, each of which may employ more patterns and tactics. And all of these parts of the client-server pattern likely employ layering to internally structure their software modules.

13.2. Overview of the Patterns Catalog

In this section we list an assortment of useful and widely used patterns. This catalog is not meant to be exhaustive—in fact no such catalog is possible. Rather it is meant to be representative. We show patterns of runtime elements (such as broker or client-server) and of design-time elements (such as layers). For each pattern we list the *context, problem, and solution*. As part of the solution, we briefly describe the *elements, relations, and constraints* of each pattern.

Applying a pattern is not an all-or-nothing proposition. Pattern definitions given in catalogs are strict, but in practice architects may choose to violate them in small ways when there is a good design tradeoff to be had (sacrificing a little of whatever the violation cost, but gaining something that the deviation gained). For example, the layered pattern expressly forbids software in lower layers from using software in upper layers, but there may be cases (such as to gain some performance) when an architecture might allow a few specific exceptions.

Patterns can be categorized by the dominant type of elements that they show: module patterns show modules, component-and-connector (C&C) patterns show components and connectors, and allocation patterns show a combination of software elements (modules, components, connectors) and nonsoftware elements. Most published patterns are C&C patterns, but there are module patterns and allocation patterns as well. We'll begin with the granddaddy of module patterns, the layered pattern.

Module Patterns

Layered Pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented

separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Solution: To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is 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. Layers completely partition a set of software, and each partition is exposed through a public interface. The layers are created to interact according to a strict ordering relation. 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. In some cases, modules in one layer might be required to directly use modules in a nonadjacent lower layer; normally only next-lower-layer uses are allowed. This case of software in a higher layer using modules in a nonadjacent lower layer is called *layer bridging*. If many instances of layer bridging occur, the system may not meet its portability and modifiability goals that strict layering helps to achieve. Upward usages are not allowed in this pattern.

Of course, none of this comes for free. Someone must design and build the layers, which can often add up-front cost and complexity to a system. Also, if the layering is not designed correctly, it may actually get in the way, by not providing the lower-level abstractions that programmers at the higher levels need. And layering always adds a performance penalty to a system. If a call is made to a function in the top-most layer, this may have to traverse many lower layers before being executed by the hardware. Each of these layers adds some overhead of their own, at minimum in the form of context switching.

[Table 13.1](#) summarizes the solution of the layered pattern.

Table 13.1. Layered Pattern Solution

Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional <i>allowed-to-use</i> relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	<i>Layer</i> , a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	<i>Allowed to use</i> , which is a specialization of a more generic <i>depends-on</i> relation. The design should define what the layer usage rules are (e.g., “a layer is allowed to use any lower layer” or “a layer is allowed to use only the layer immediately below it”) and any allowable exceptions.
Constraints	<ul style="list-style-type: none">▪ Every piece of software is allocated to exactly one layer.▪ There are at least two layers (but usually there are three or more).▪ The <i>allowed-to-use</i> relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	<ul style="list-style-type: none">▪ The addition of layers adds up-front cost and complexity to a system.▪ Layers contribute a performance penalty.

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 13.1](#).

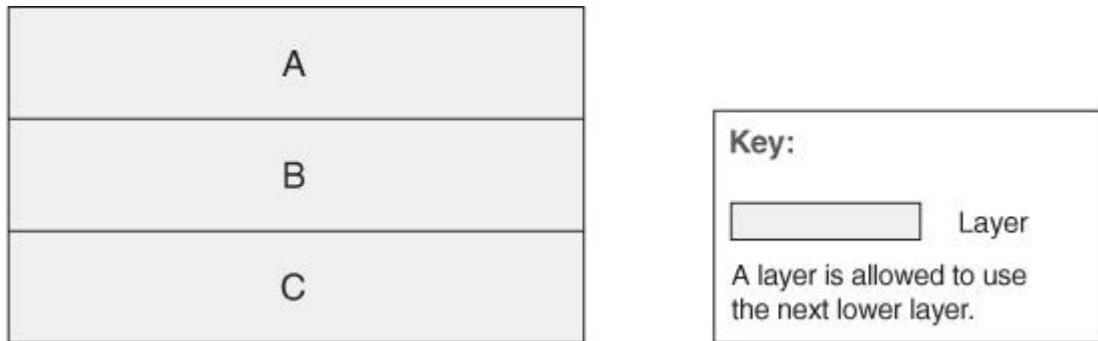


Figure 13.1. Stack-of-boxes notation for layered designs

Some Finer Points of Layers

A layered architecture is one of the few places where connections among components can be shown by adjacency, and where “above” and “below” matter. If you turn [Figure 13.1](#) upside-down so that C is on top, this would represent a completely different design. Diagrams that use arrows among the boxes to denote relations retain their semantic meaning no matter the orientation.

The layered pattern is one of the most commonly used patterns in all of software engineering, but I’m often surprised by how many people still get it wrong.

First, it is impossible to look at a stack of boxes and tell whether layer bridging is allowed or not. That is, can a layer use any lower layer, or just the next lower one? It is the easiest thing in the world to resolve this; all the architect has to do is include the answer in the key to the diagram’s notation (something we recommend for all diagrams). For example, consider the layered pattern presented in [Figure 13.2](#) on the next page.

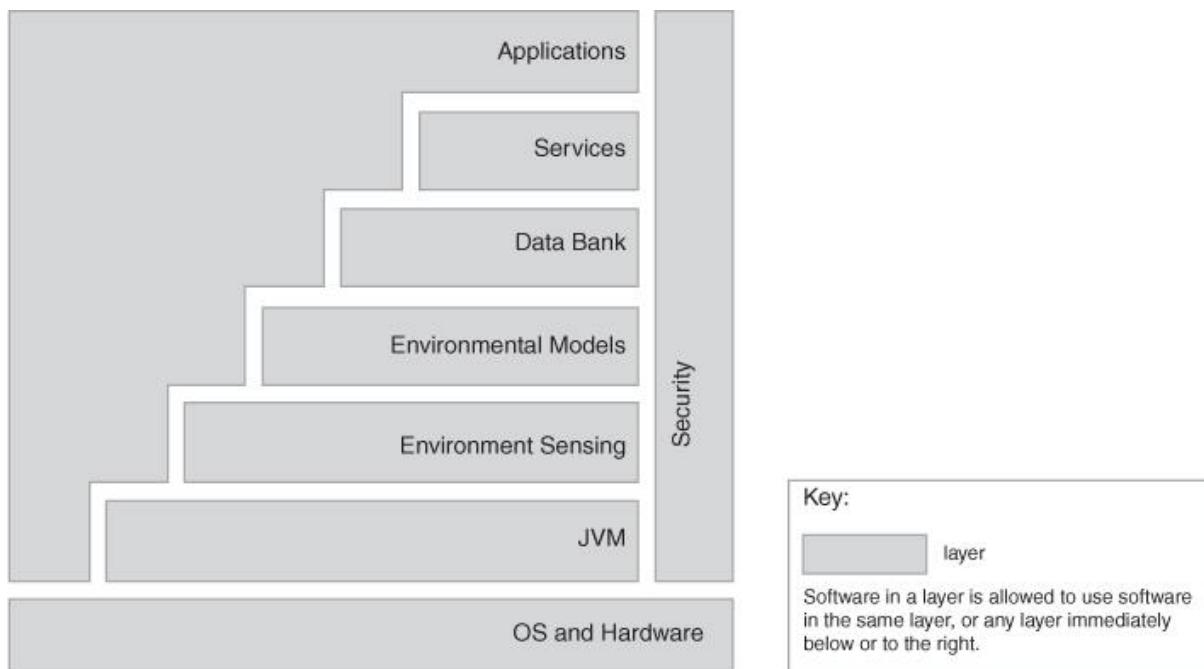


Figure 13.2. A simple layer diagram, with a simple key answering the uses question

But I’m still surprised at how few architects actually bother to do this. And if they don’t, their layer diagrams are ambiguous.

Second, any old set of boxes stacked on top of each other does not constitute a layered architecture. For instance, look at the design shown in [Figure 13.3](#), which uses arrows

instead of adjacency to indicate the relationships among the boxes. Here, everything is allowed to use everything. This is decidedly *not* a layered architecture. The reason is that if Layer A is replaced by a different version, Layer C (which uses it in this figure) might well have to change. We don't want our virtual machine layer to change every time our application layer changes. But I'm still surprised at how many people call a stack of boxes lined up with each other "layers" (or think that layers are the same as tiers in a multi-tier architecture).

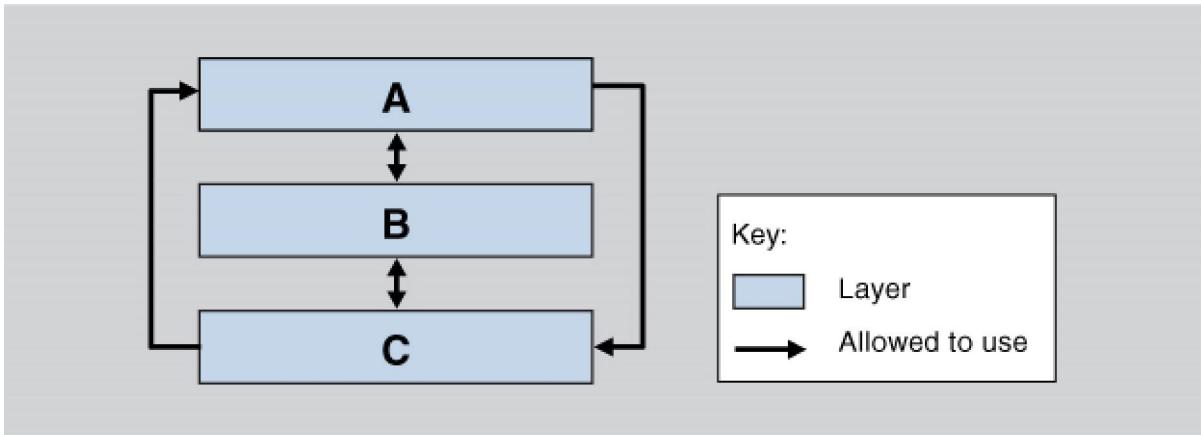


Figure 13.3. A wolf in layer's clothing

Third, many architectures that purport to be layered look something like [Figure 13.4](#). This diagram *probably* means that modules in A, B, or C can use modules in D, but without a key to tell us for sure, it could mean anything. "Sidecars" like this often contain common utilities (sometimes imported), such as error handlers, communication protocols, or database access mechanisms. This kind of diagram makes sense only in the case where no layer bridging is allowed in the main stack. Otherwise, D could simply be made the bottom-most layer in the main stack, and the "sidecar" geometry would be unnecessary. But I'm still surprised at how often I see this layout go unexplained.

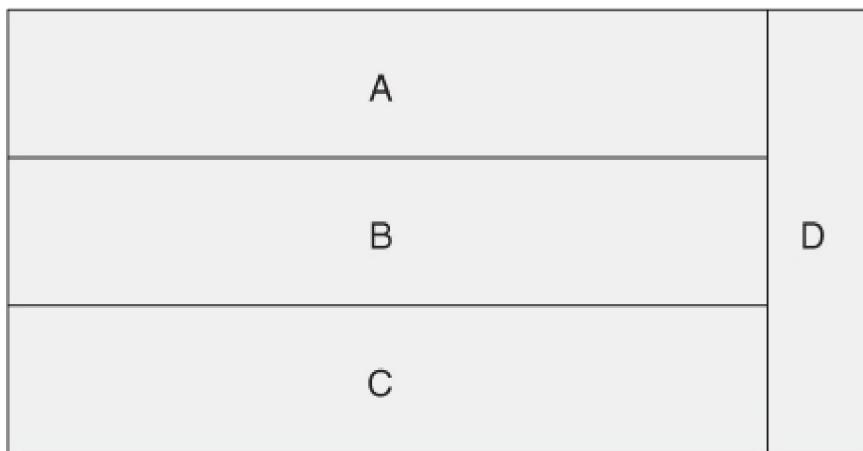


Figure 13.4. Layers with a "sidecar"

Sometimes layers are divided into segments denoting a finer-grained decomposition of the modules. Sometimes this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use* relation. When this happens, you have to specify what usage rules are in effect among the segments. Many usage rules are possible, but they must be made explicit. In [Figure 13.5](#), the top and the bottom layers are 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 different usage rules within segmented layers. Layered diagrams are often a source of hidden ambiguity because the diagram does not make explicit the *allowed-to-use* relations.

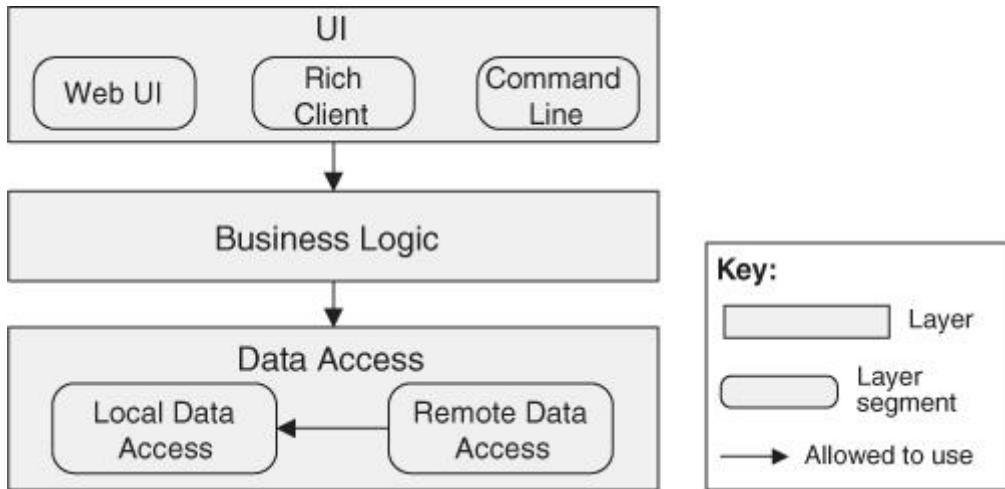


Figure 13.5. Layered design with segmented layers

Finally, the most important point about layering is that a layer isn't allowed to *use* any layer above it. A module "uses" another module when it depends on the answer it gets back. But a layer is allowed to make upward calls, as long as it isn't expecting an answer from them. This is how the common error-handling scheme of callbacks works. A program in layer A calls a program in a lower layer B, and the parameters include a pointer to an error-handling program in A that the lower layer should call in case of error. The software in B makes the call to the program in A, but cares not in the least what it does. By not depending in any way on the contents of A, B is insulated from changes in A.

—PCC

Other Module Patterns

Designers in a particular domain often publish "standard" module decompositions for systems in that domain. These standard decompositions, if put in the "context, problem, solution" form, constitute module decomposition patterns.

Similarly in the object-oriented realm, "standard" or published class/object design solutions for a class of system constitute object-oriented patterns.

Component-and-Connector Patterns

Broker Pattern

Context: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request. The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client. In this way the client remains completely ignorant of the identity, location, and characteristics of the server. Because of this separation, if a server becomes unavailable, a replacement can be dynamically chosen by the broker. If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change, and so the client is

unaffected. Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshaling and unmarshaling messages.

The down sides of brokers are that they add complexity (brokers and possibly proxies must be designed and implemented, along with messaging protocols) and add a level of indirection between a client and a server, which will add latency to their communication. Debugging brokers can be difficult because they are involved in highly dynamic environments where the conditions leading to a failure may be difficult to replicate. The broker would be an obvious point of attack, from a security perspective, and so it needs to be hardened appropriately. Also a broker, if it is not designed carefully, can be a single point of failure for a large and complex system. And brokers can potentially be bottlenecks for communication.

[Table 13.2](#) summarizes the solution of the broker pattern.

Table 13.2. Broker Pattern Solution

Overview	The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
Elements	<i>Client</i> , a requester of services <i>Server</i> , a provider of services <i>Broker</i> , an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client <i>Client-side proxy</i> , an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages <i>Server-side proxy</i> , an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
Relations	The <i>attachment</i> relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
Weaknesses	Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. The broker can be a single point of failure. A broker adds up-front complexity. A broker may be a target for security attacks. A broker may be difficult to test.

The broker is, of course, the critical component in this pattern. The pattern provides all of the modifiability benefits of the use-an-intermediary tactic (described in [Chapter 7](#)), an availability benefit (because the broker pattern makes it easy to replace a failed server with another), and a performance benefit (because the broker pattern makes it easy to assign work to the least-busy server). However, the pattern also carries with it some liabilities. For example, the use of a broker precludes performance optimizations that you might make if you knew the precise location and characteristics of the server. Also the use of this pattern adds the overhead of the intermediary and thus latency.

The original version of the broker pattern, as documented by Gamma, Helm, Johnson, and Vlissides[\[Gamma 94\]](#), is given in [Figure 13.6](#).

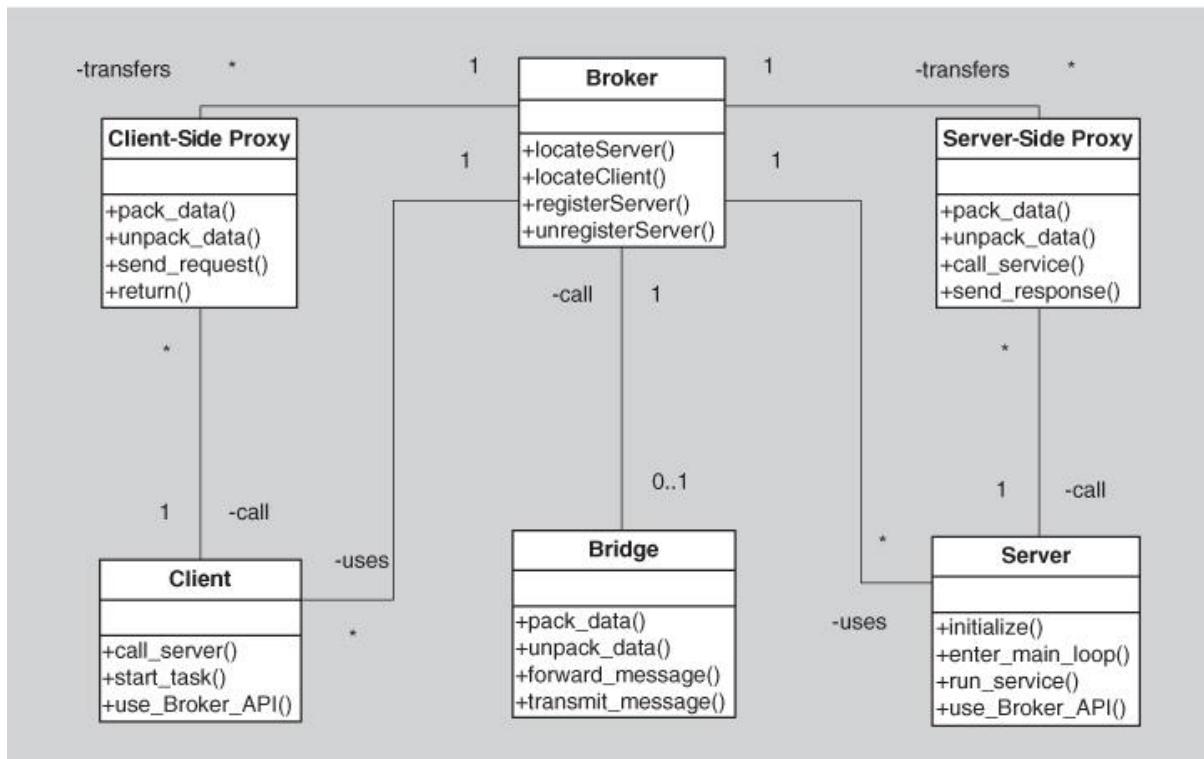


Figure 13.6. The broker pattern

The first widely used implementation of the broker pattern was in the Common Object Request Broker Architecture (CORBA). Other common uses of this pattern are found in Enterprise Java Beans (EJB) and Microsoft's .NET platform—essentially any modern platform for distributed service providers and consumers implements some form of a broker. The service-oriented architecture (SOA) approach depends crucially on brokers, most commonly in the form of an enterprise service bus.

Model-View-Controller Pattern

Context: User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications to the user interface software separate from the rest of the system. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.

Problem: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

Solution: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:

- A *model*, which contains the application's data
- A *view*, which displays some portion of the underlying data and interacts with the user
- A *controller*, which mediates between the *model* and the *view* and manages the notifications of state changes

MVC is not appropriate for every situation. The design and implementation of three distinct kinds of components, along with their various forms of interaction, may be costly, and this cost may not make sense for relatively simple user interfaces. Also, the match between the abstractions of MVC and commercial user interface toolkits is not perfect. The view and the controller split apart input and output, but these functions are often combined into individual widgets. This may result in a conceptual mismatch between the architecture and the user interface toolkit.

[Table 13.3](#) summarizes the solution of the MVC pattern.

Table 13.3. Model-View-Controller Pattern Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	<p>The <i>model</i> is a representation of the application data or state, and it contains (or provides an interface to) application logic.</p> <p>The <i>view</i> is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.</p> <p>The <i>controller</i> manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.</p>
Relations	The <i>notifies</i> relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	<p>There must be at least one instance each of model, view, and controller.</p> <p>The model component should not interact directly with the controller.</p>
Weaknesses	<p>The complexity may not be worth it for simple user interfaces.</p> <p>The model, view, and controller abstractions may not be good fits for some user interface toolkits.</p>

There may, in fact, be many views and many controllers associated with a model. For example, a set of business data may be represented as columns of numbers in a spreadsheet, as a scatter plot, or as a pie chart. Each of these is a separate view, and this view can be dynamically updated as the model changes (for example, showing live transactions in a transaction processing system). A model may be updated by different controllers; for example, a map could be zoomed and panned via mouse movements, trackball movements, keyboard clicks, or voice commands; each of these different forms of input needs to be managed by a controller.

The MVC components are connected to each other via some flavor of notification, such as events or callbacks. These notifications contain state updates. A change in the model needs to be communicated to the views so that they may be updated. An external event, such as a user input, needs to be communicated to the controller, which may in turn update the view and/or the model. Notifications may be either push or pull.

Because these components are loosely coupled, it is easy to develop and test them in parallel, and changes to one have minimal impact on the others. The relationships between the components of MVC are shown in [Figure 13.7](#).

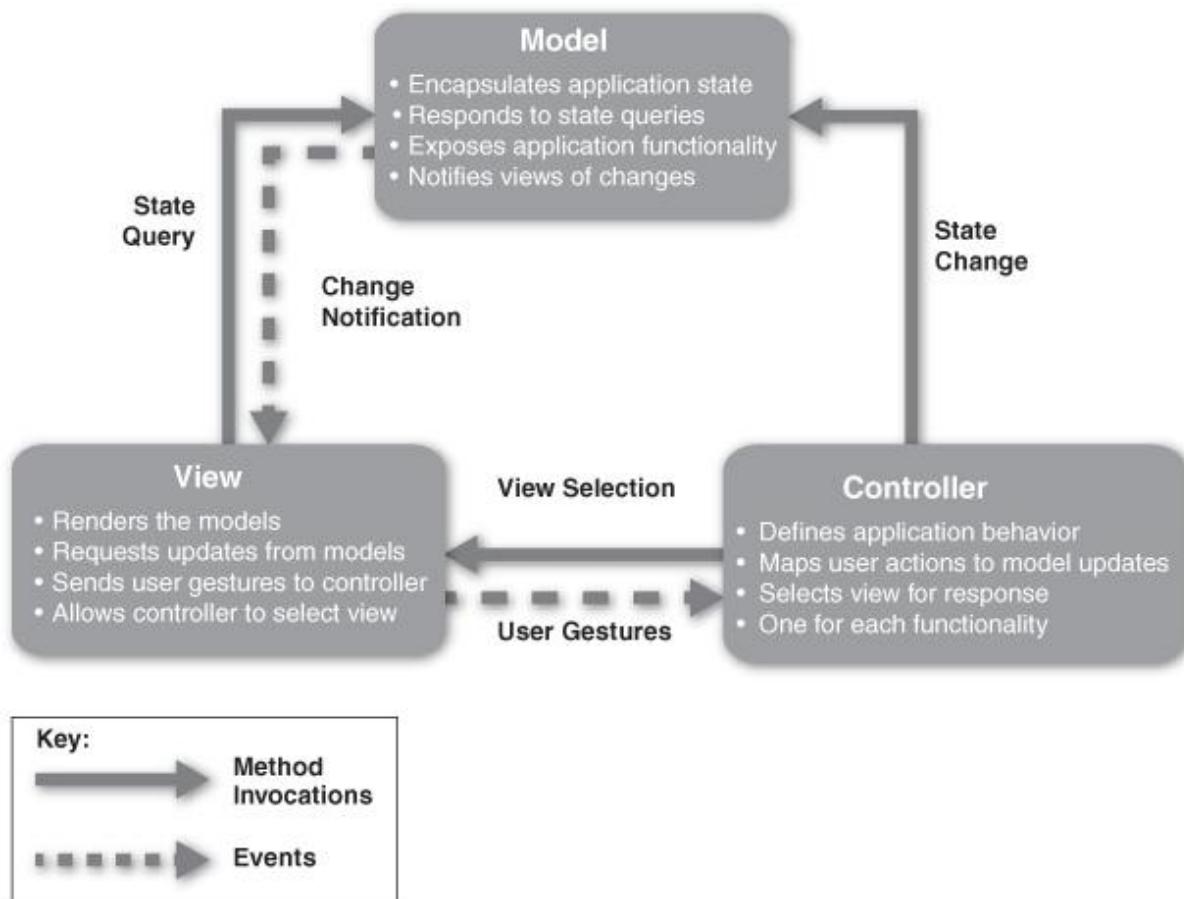


Figure 13.7. The model-view-controller pattern

The MVC pattern is widely used in user interface libraries such as Java's Swing classes, Microsoft's ASP.NET framework, Adobe's Flex software development kit, Nokia's Qt framework, and many others. As such, it is common for a single application to contain many instances of MVC (often one per user interface object).

Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

There are several weaknesses associated with the pipe-and-filter pattern. For instance, this pattern is typically not a good choice for an interactive system, as it disallows cycles (which are important for user feedback). Also, having large numbers of independent filters can add substantial amounts of computational overhead, because each filter runs as its own thread or process. Also, pipe-and-filter systems may not be appropriate for long-running computations, without the addition of some form of checkpoint/restore functionality, as the failure of any filter (or pipe) can cause the entire pipeline to fail.

The solution of the pipe-and-filter pattern is summarized in [Table 13.4](#).

Table 13.4. Pipe-and-Filter Pattern Solution

Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
Elements	<p><i>Filter</i>, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter.</p> <p><i>Pipe</i>, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.</p>
Relations	The <i>attachment</i> relation associates the output of filters with the input of pipes and vice versa.
Constraints	<p>Pipes connect filter output ports to filter input ports.</p> <p>Connected filters must agree on the type of data being passed along the connecting pipe.</p> <p>Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline.</p> <p>Other specializations may prescribe that components have certain named ports, such as the <i>stdin</i>, <i>stdout</i>, and <i>stderr</i> ports of UNIX filters.</p>
Weaknesses	<p>The pipe-and-filter pattern is typically not a good choice for an interactive system.</p> <p>Having large numbers of independent filters can add substantial amounts of computational overhead.</p> <p>Pipe-and-filter systems may not be appropriate for long-running computations.</p>

Pipes buffer data during communication. Because of this property, filters can execute asynchronously and concurrently. Moreover, a filter typically does not know the identity of its upstream or downstream filters. For this reason, pipeline pipe-and-filter systems have the property that the overall computation can be treated as the functional composition of the computations of the filters, making it easier for the architect to reason about end-to-end behavior.

Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data. The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behavior. Often such systems constitute the front end of signal-processing applications. These systems receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial processing (such as smoothing). 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.

Other systems that use pipe-and-filter include those built using UNIX pipes, the request processing architecture of the Apache web server, the map-reduce pattern (presented later in this chapter), Yahoo! Pipes for processing RSS feeds, many workflow engines, and many scientific computation systems that have to process and analyze large streams of captured data. [Figure 13.8](#) shows a UML diagram of a pipe-and-filter system.

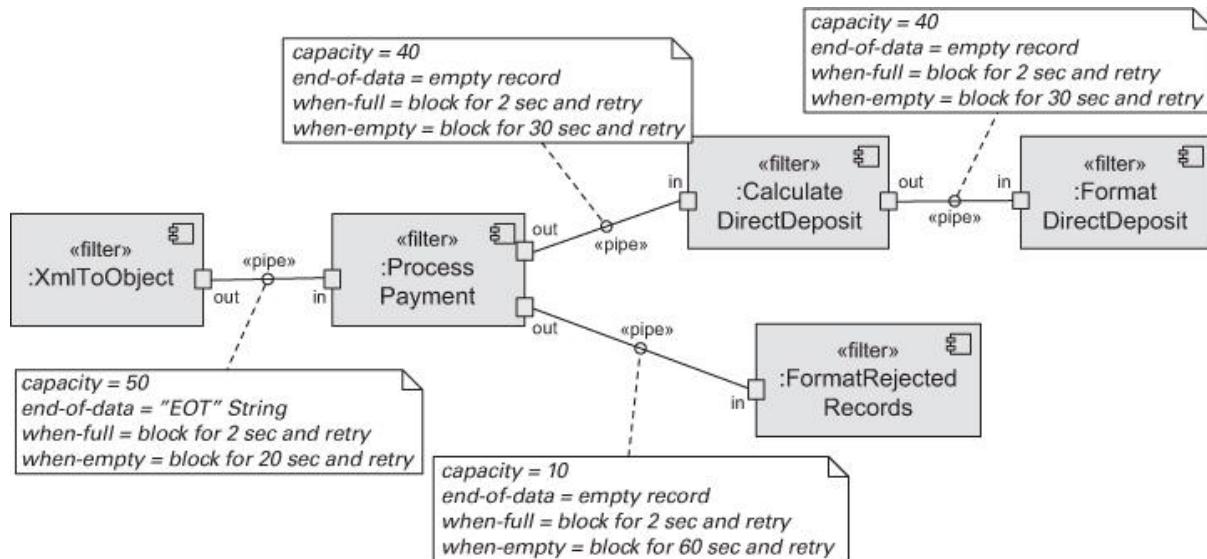


Figure 13.8. A UML diagram of a pipe-and-filter-based system

Client-Server Pattern

Context: There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

Problem: By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

Solution: Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

The client-server pattern solution is summarized in [Table 13.5](#); the component types are *clients* and *servers*; the principal connector type for the client-server pattern is a data connector driven by a request/reply protocol used for invoking services.

Table 13.5. Client-Server Pattern Solution

Overview	Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
Elements	<p><i>Client</i>, a component that invokes services of a server component. Clients have ports that describe the services they require.</p> <p><i>Server</i>, a component that provides services to clients. Servers have ports that describe the services they provide. Important characteristics include information about the nature of the server ports (such as how many clients can connect) and performance characteristics (e.g., maximum rates of service invocation).</p> <p><i>Request/reply connector</i>, a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.</p>
Relations	The <i>attachment</i> relation associates clients with servers.
Constraints	<p>Clients are connected to servers through request/reply connectors.</p> <p>Server components can be clients to other servers.</p> <p>Specializations may impose restrictions:</p> <ul style="list-style-type: none">▪ Numbers of attachments to a given port▪ Allowed relations among servers <p>Components may be arranged in tiers, which are logical groupings of related functionality or functionality that will share a host computing environment (covered more later in this chapter).</p>
Weaknesses	<p>Server can be a performance bottleneck.</p> <p>Server can be a single point of failure.</p> <p>Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.</p>

Some of the disadvantages of the client-server pattern are that the server can be a performance bottleneck and it can be a single point of failure. Also, decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Some common examples of systems that use the client-server pattern are these:

- Information systems running on local networks where the clients are GUI-launched applications and the server is a database management system
- Web-based applications where the clients are web browsers and the servers are components running on an e-commerce site

The computational flow of pure client-server systems is asymmetric: clients initiate interactions by invoking services of servers. Thus, the client must know the identity of a service to invoke it, and clients initiate all interactions. In contrast, servers do not know the identity of clients in advance of a service request and must respond to the initiated client requests.

In early forms of client-server, 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. However, variants of the client-server pattern may employ more-sophisticated connector protocols. For example:

- Web browsers don't block until the data request is served up.
- In some client-server patterns, 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 a client-server interaction.

The client-server pattern separates client applications from the services they use. This pattern simplifies systems by factoring out common services, which are reusable. Because servers can be accessed by any number of clients, it is easy to add new clients to a system. Similarly, servers may be replicated to support scalability or availability.

The World Wide Web is the best-known example of a system that is based on the client-server pattern, allowing clients (web browsers) to access information from servers across the Internet using HyperText Transfer Protocol (HTTP). HTTP is a request/reply protocol. HTTP is stateless; the connection between the client and the server is terminated after each response from the server.

[Figure 13.9](#) uses an informal notation to describe the client-server view of an automatic teller machine (ATM) banking system.

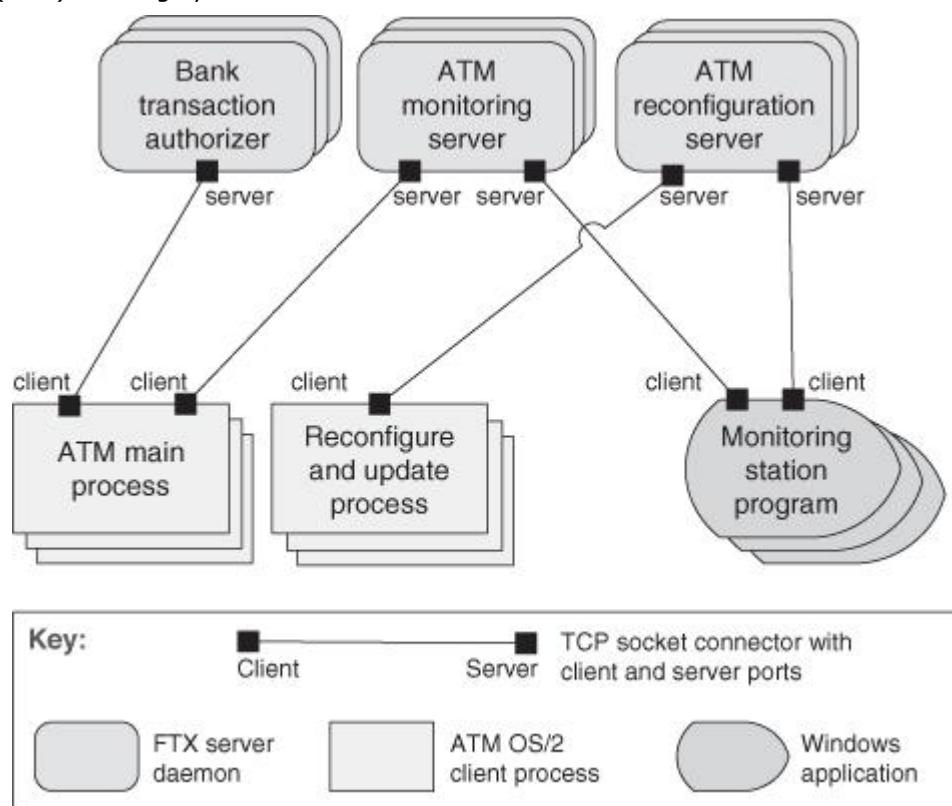


Figure 13.9. The client-server architecture of an ATM banking system

Peer-to-Peer Pattern

Context: Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.

Problem: How can a set of “equal” distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

Solution: In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal” and no peer or group of peers can be critical for the health of the system. Peer-to-peer

communication is typically a request/reply interaction without the asymmetry found in the client-server pattern. That is, any component can, in principle, interact with any other component by requesting its services. The interaction may be initiated by either party—that is, in client-server terms, each peer component is both a client and a server. Sometimes the interaction is just to forward data without the need for a reply. Each peer provides and consumes similar services and uses the same protocol. Connectors in peer-to-peer systems involve bidirectional interactions, reflecting the two-way communication that may exist between two or more peer-to-peer components.

Peers first connect to the peer-to-peer network on which they discover other peers they can interact with, and then initiate actions to achieve their computation by cooperating with other peers by requesting services. 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 specialized peer nodes (called supernodes) that have indexing or routing capabilities and allow a regular peer's search to reach a larger number of peers.

Peers can be added and removed from the peer-to-peer network with no significant impact, resulting in great scalability for the whole system. This provides flexibility for deploying the system across a highly distributed platform.

Typically multiple peers have overlapping capabilities, such as providing access to the same data or providing equivalent services. 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. There are also performance advantages: 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.

The drawbacks of the peer-to-peer pattern are strongly related to its strengths. Because peer-to-peer systems are decentralized, managing security, data consistency, data and service availability, backup, and recovery are all more complex. In many cases it is difficult to provide guarantees with peer-to-peer systems because the peers come and go; instead, the architect can, at best, offer probabilities that quality goals will be met, and these probabilities typically increase with the size of the population of peers.

[Table 13.6](#) on the next page summarizes the peer-to-peer pattern solution.

Table 13.6. Peer-to-Peer Pattern Solution

Overview	Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
Elements	<i>Peer</i> , which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability. <i>Request/reply connector</i> , which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
Relations	The relation associates peers with their connectors. Attachments may change at runtime.
Constraints	Restrictions may be placed on the following: <ul style="list-style-type: none">▪ The number of allowable attachments to any given peer▪ The number of hops used for searching for a peer▪ Which peers know about which other peers Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
Weaknesses	Managing security, data consistency, data/service availability, backup, and recovery are all more complex. Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

Peer-to-peer computing is often used in distributed computing applications such as file sharing, instant messaging, desktop grid computing, routing, and wireless ad hoc networking. Examples of peer-to-peer systems include file-sharing networks such as BitTorrent and eDonkey, and instant messaging and VoIP applications such as Skype. [Figure 13.10](#) shows an example of an instantiation of the peer-to-peer pattern.

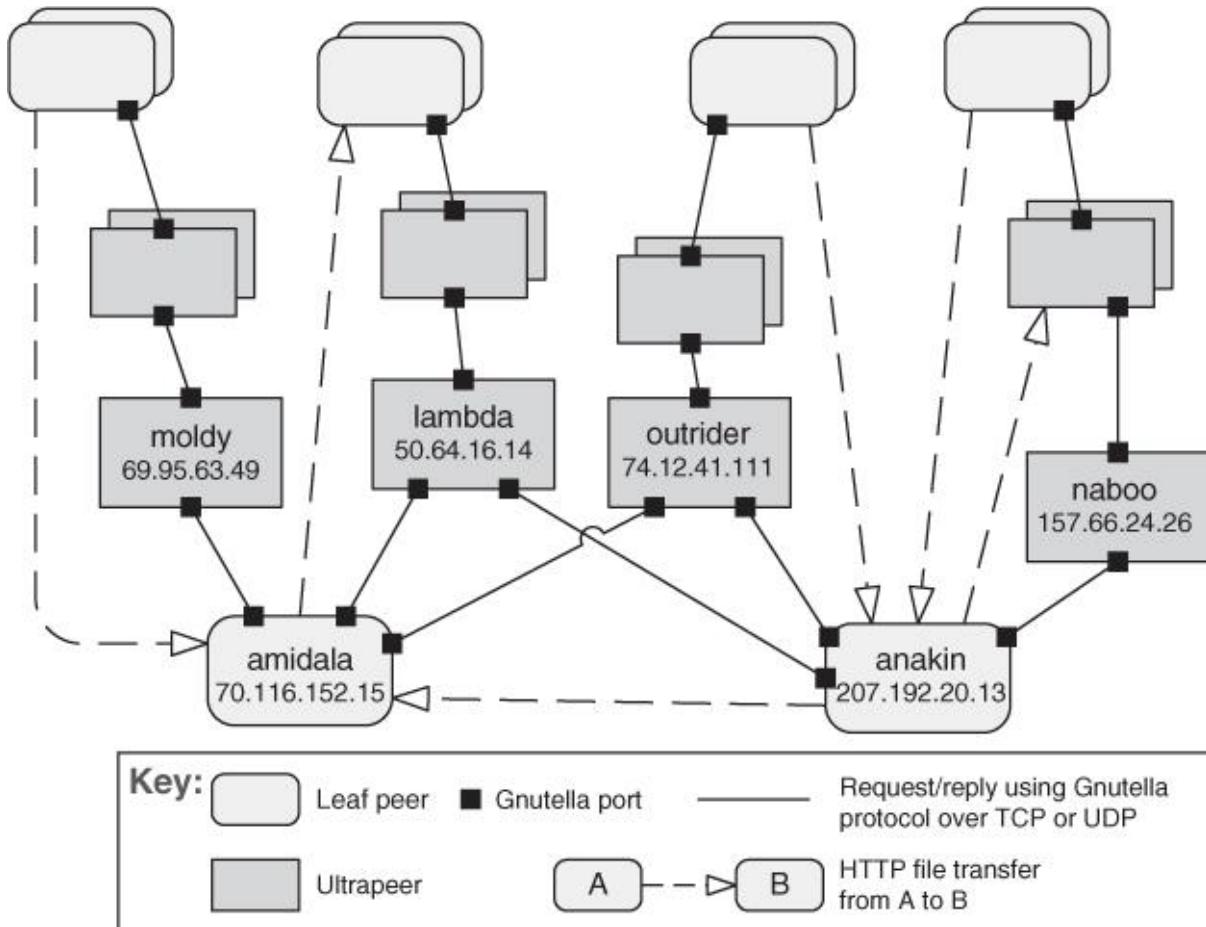


Figure 13.10. A peer-to-peer view of a Gnutella network using an informal C&C notation. For brevity, only a few peers are identified. Each of the identified leaf peers uploads and downloads files directly from other peers.

Service-Oriented Architecture Pattern

Context: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet? How can we locate services and combine (and dynamically recombine) them into meaningful coalitions while achieving reasonable performance, security, and availability?

Solution: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services. In an 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. Components have interfaces that describe the services they request from other components and the services they provide. A service's quality attributes can be specified and guaranteed with a service-level agreement (SLA). In some cases, these are legally binding. Components achieve their computation by requesting services from one another.

The elements in this pattern include service providers and service consumers, 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 service consumer components, an SOA application may use specialized components that act as intermediaries and provide infrastructure services:

- 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 (e.g., format, content, splitting, merging), perform security checks, and manage transactions. Using an ESB promotes interoperability, security, and modifiability. Of course, communicating through an ESB adds overhead thereby lowering performance, and introduces an additional point of failure. When an ESB is not in place, service providers and consumers communicate with each other in a point-to-point fashion.
- To improve the independence of service providers, a *service registry* can be used in SOA architectures. The registry is a component that allows services to be registered at runtime. This enables runtime discovery of services, which increases system modifiability by hiding the location and identity of the service provider. A registry can even permit multiple live versions of the same service.
- An *orchestration server* (or orchestration engine) orchestrates the interaction among various service consumers and providers in an SOA system. It executes scripts upon the occurrence of a specific event (e.g., a purchase order request arrived). Applications with well-defined business processes or workflows that involve interactions with distributed components or systems gain in modifiability, interoperability, and reliability by using an orchestration server. Many commercially available orchestration servers support various workflow or business process language standards.

The basic types of connectors used in SOA are these:

- *SOAP*. The standard protocol for communication in the web services technology. Service consumers and providers interact by exchanging request/reply XML messages typically on top of HTTP.
- *Representational State Transfer (REST)*. A service consumer sends nonblocking HTTP requests. These requests rely on the four basic HTTP commands (POST, GET, PUT, DELETE) to tell the service provider to create, retrieve, update, or delete a resource.
- *Asynchronous messaging*, a “fire-and-forget” information exchange. Participants do not have to wait for an acknowledgment of receipt, because the infrastructure is assumed to have delivered the message successfully. The messaging connector can be point-to-point or publish-subscribe.

In practice, SOA environments may involve a mix of the three connectors just listed, along with legacy protocols and other communication alternatives (e.g., SMTP). Commercial products such as IBM’s WebSphere MQ, Microsoft’s MSMQ, or Apache’s ActiveMQ are infrastructure components that provide asynchronous messaging. SOAP and REST are described in more detail in [Chapter 6](#).

As you can see, the SOA pattern can be quite complex to design and implement (due to dynamic binding and the concomitant use of metadata). Other potential problems with this pattern include the performance overhead of the middleware that is interposed between services and clients and the lack of performance guarantees (because services are shared and, in general, not under control of the requester). These weaknesses are all shared with the broker pattern, which is not surprising because the SOA pattern shares many of the design concepts and goals of broker. In addition, because you do not, in general, control the evolution of the services that you use, you may have to endure high and unplanned-for maintenance costs.

[Table 13.7](#) summarizes the SOA pattern.

Table 13.7. Service-Oriented Architecture Pattern Solution

Overview	Computation is achieved by a set of cooperating components that provide and/or consume services over a network. The computation is often described using a workflow language.
Elements	<p>Components:</p> <ul style="list-style-type: none">▪ <i>Service providers</i>, which provide one or more services through published interfaces. Concerns are often tied to the chosen implementation technology, and include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement.▪ <i>Service consumers</i>, which invoke services directly or through an intermediary.▪ <i>Service providers</i> may also be service consumers.▪ <i>ESB</i>, which is an intermediary element that can route and transform messages between service providers and consumers.▪ <i>Registry of services</i>, which may be used by providers to register their services and by consumers to discover services at runtime.▪ <i>Orchestration server</i>, which coordinates the interactions between service consumers and providers based on languages for business processes and workflows. <p>Connectors:</p> <ul style="list-style-type: none">▪ <i>SOAP connector</i>, which uses the SOAP protocol for synchronous communication between web services, typically over HTTP.▪ <i>REST connector</i>, which relies on the basic request/reply operations of the HTTP protocol.▪ <i>Asynchronous messaging connector</i>, which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
Relations	Attachment of the different kinds of components available to the respective connectors
Constraints	Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
Weaknesses	SOA-based systems are typically complex to build. You don't control the evolution of independent services. There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

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 a variety of systems, including legacy systems. SOA 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.

[Figure 13.11](#) shows the SOA view of a system called Adventure Builder. Adventure Builder allows a customer on the web to assemble a vacation by choosing an activity and lodging at and transportation to a destination. The Adventure Builder system interacts with external service providers to construct the vacation, and with bank services to process payment. The central OPC (Order Processing Center) component coordinates the interaction with internal and external service consumers and providers. Note that the external providers can be legacy mainframe systems, Java systems, .NET systems, and so on. The nature of these external components is transparent because SOAP provides the necessary interoperability.

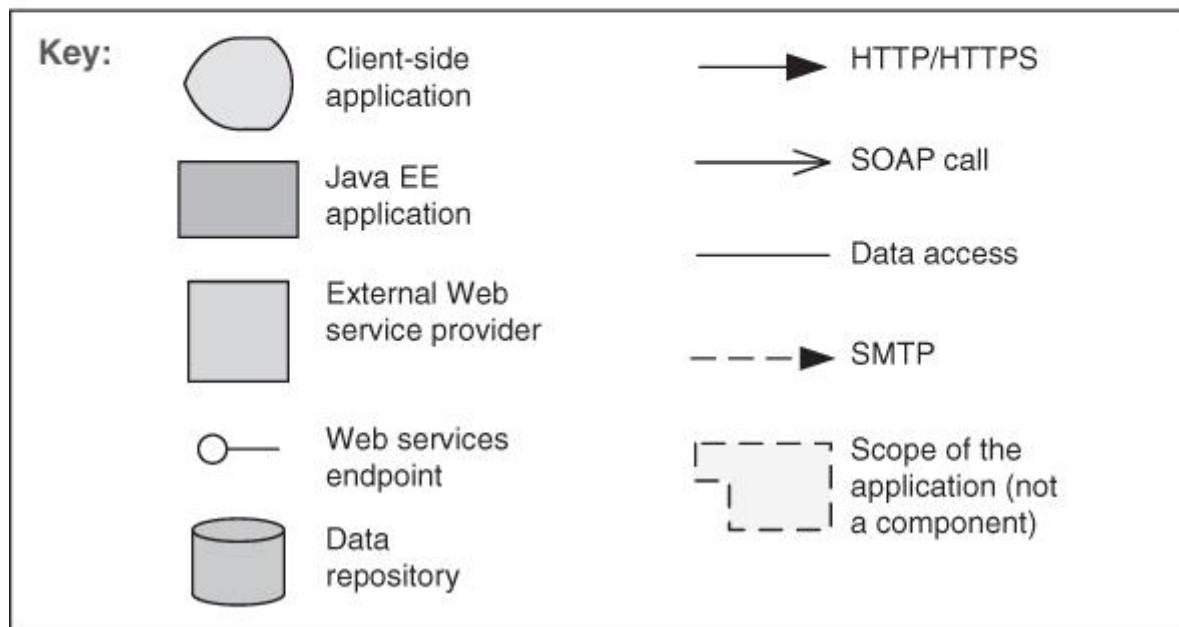
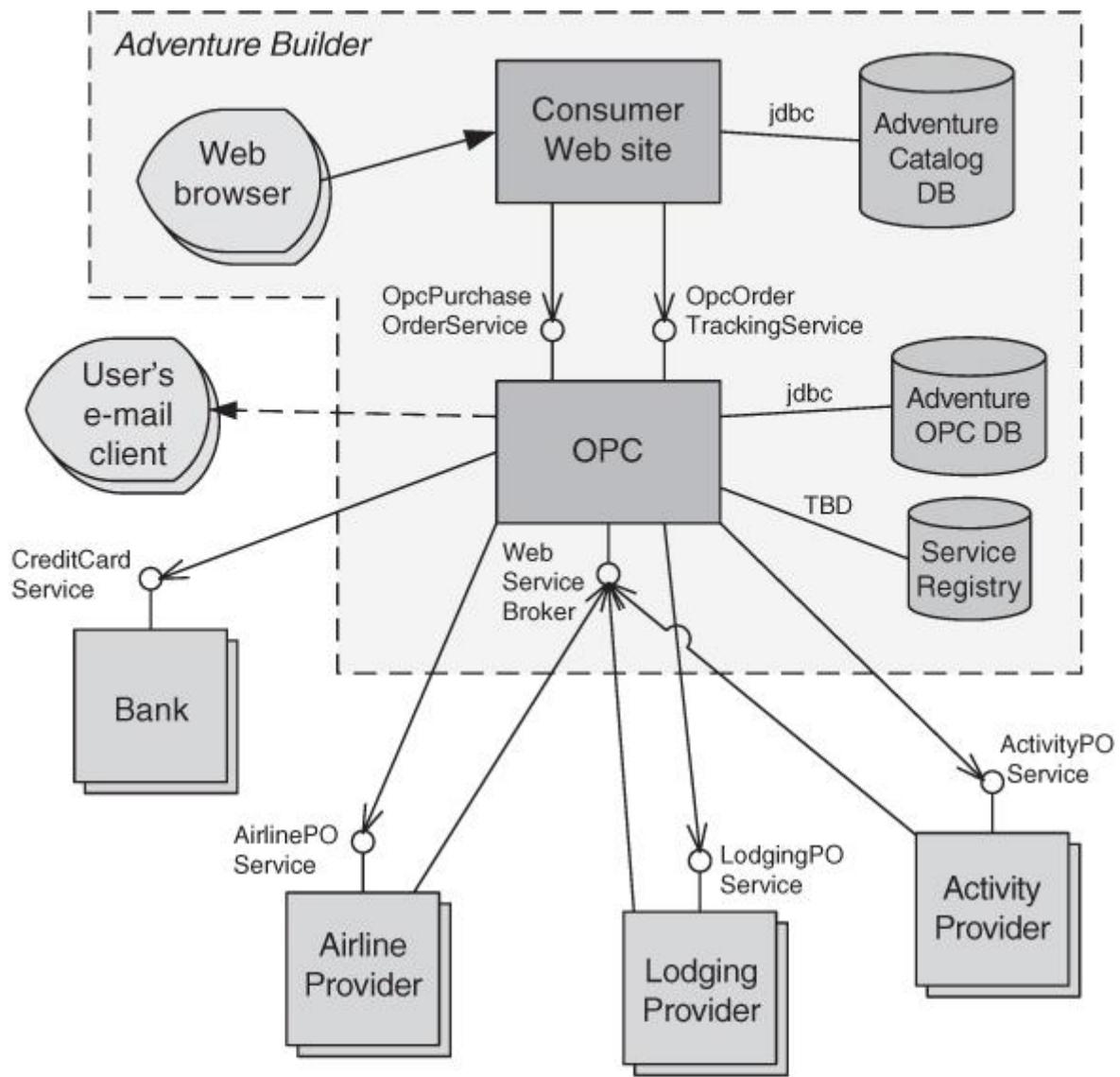


Figure 13.11. Diagram of the SOA view for the Adventure Builder system. OPC stands for “Order Processing Center.”

Publish-Subscribe Pattern

Context: There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

Solution: In the publish-subscribe pattern, summarized in [Table 13.8](#), components interact via announced messages, or 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 these patterns is an *event bus*. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events. Any component may be both a publisher and a subscriber.

Table 13.8. Publish-Subscribe Pattern Solution

Overview	Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
Elements	<i>Any C&C component</i> with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events. <i>The publish-subscribe connector</i> , which will have <i>announce</i> and <i>listen</i> roles for components that wish to publish and subscribe to events.
Relations	The <i>attachment</i> relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.
Constraints	All components are connected to an event distributor that may be viewed as either a bus—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.
Weaknesses	Typically increases latency and has a negative effect on scalability and predictability of message delivery time. Less control over ordering of messages, and delivery of messages is not guaranteed.

Publish-subscribe adds a layer of indirection between senders and receivers. This has a negative effect on latency and potentially scalability, depending on how it is implemented. One would typically not want to use publish-subscribe in a system that had hard real-time deadlines to meet, as it introduces uncertainty in message delivery times.

Also, the publish-subscribe pattern suffers in that it provides less control over ordering of messages, and delivery of messages is not guaranteed (because the sender cannot know if a receiver is listening). This can make the publish-subscribe pattern inappropriate for complex interactions where shared state is critical.

There are some specific refinements of this pattern that are in common use. We will describe several of these later in this section.

The computational model for the publish-subscribe pattern 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. An example of the publish-subscribe pattern, implemented on top of the Eclipse platform, is shown in [Figure 13.12](#).

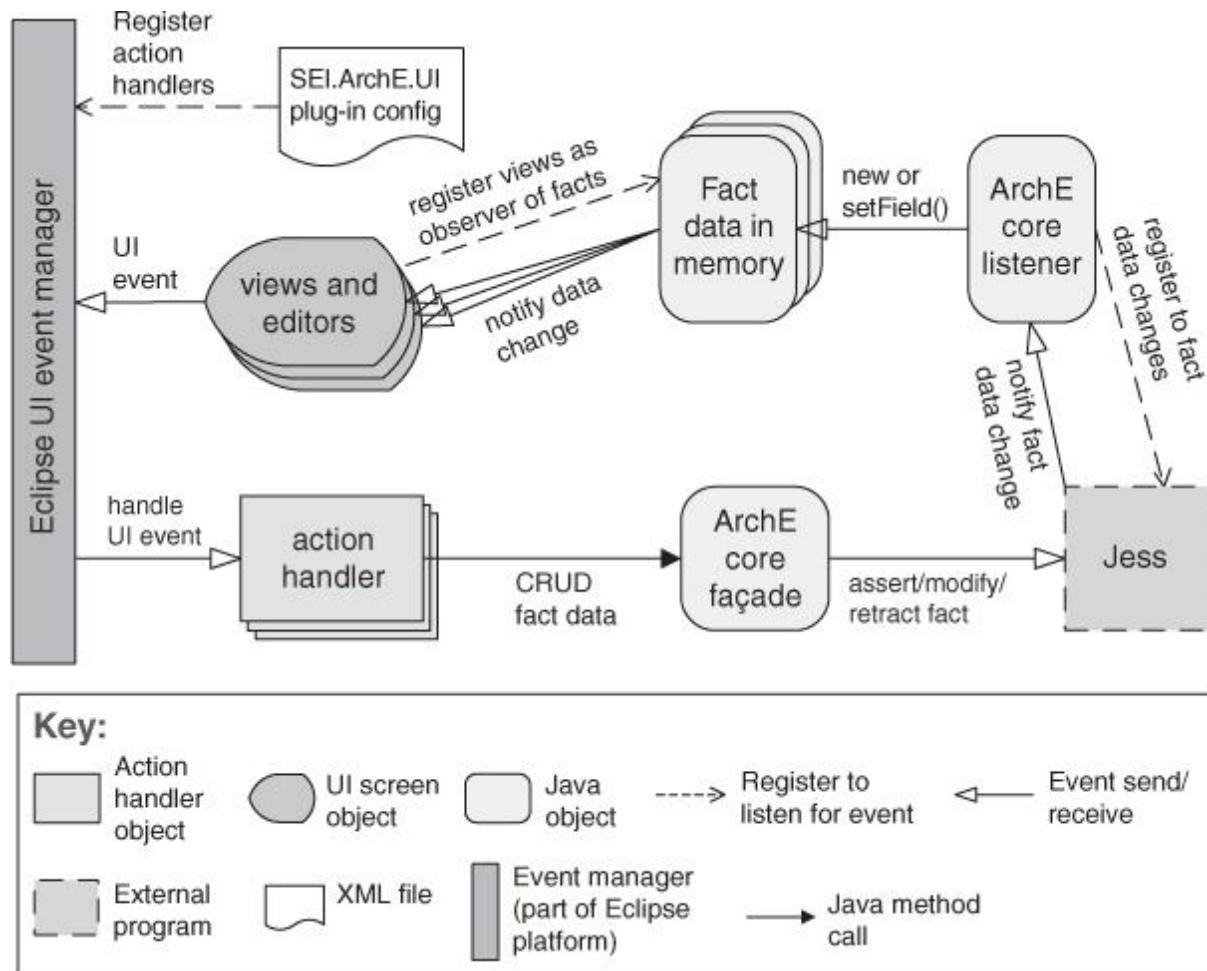


Figure 13.12. A typical publish-subscribe pattern realization

Typical examples of systems that employ the publish-subscribe pattern are the following:

- Graphical user interfaces, in which a user's low-level input actions are treated as events that are routed to appropriate input handlers
- MVC-based applications, in which view components are notified when the state of a model object changes
- Enterprise resource planning (ERP) systems, which integrate many components, each of which is only interested in a subset of system events
- 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 website

The publish-subscribe pattern 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, in general, depend on those recipients. Thus, new recipients can be added without modification to the producers.

Having components be ignorant of each other's identity results in easy modification of the system (adding or removing producers and consumers of data) but at the cost of runtime performance, because the publish-subscribe infrastructure is a kind of indirection, which adds latency. In addition, if the publish-subscribe connector fails completely, this is a single point of failure for the entire system.

The publish-subscribe pattern can take several forms:

- *List-based publish-subscribe* is a realization of the pattern where every publisher maintains a subscription list—a list of subscribers that have registered an interest in receiving the event. This version of the pattern is less decoupled than others, as we shall see below, and hence it does not provide as much modifiability, but it can be quite efficient in terms of runtime overhead. Also, if the components are distributed, there is no single point of failure.
- *Broadcast-based publish-subscribe* differs from list-based publish-subscribe in that publishers have less (or no) knowledge of the subscribers. Publishers simply publish events, which are then broadcast. Subscribers (or in a distributed system, services that act on behalf of the subscribers) examine each event as it arrives and determine whether the published event is of interest. This version has the potential to be very inefficient if there are lots of messages and most messages are not of interest to a particular subscriber.
- *Content-based publish-subscribe* is distinguished from the previous two variants, which are broadly categorized as "topic-based." Topics are predefined events, or messages, and a component subscribes to all events within the topic. Content, on the other hand, is much more general. Each event is associated with a set of attributes and is delivered to a subscriber only if those attributes match subscriber-defined patterns.

In practice the publish-subscribe pattern is typically realized by some form of message-oriented middleware, where the middleware is realized as a broker, managing the connections and channels of information between producers and consumers. This middleware is often responsible for the transformation of messages (or message protocols), in addition to routing and sometimes storing the messages. Thus the publish-subscribe pattern inherits the strengths and weaknesses of the broker pattern.

Shared-Data Pattern

Context: Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.

Problem: How can systems store and manipulate persistent data that is accessed by multiple independent components?

Solution: In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. 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 pattern differ with respect to the nature of the stored data—existing approaches include relational, object structures, layered, and hierarchical structures.

Although the sharing of data is a critical task for most large, complex systems, there are a number of potential problems associated with this pattern. For one, the shared-data store may be a performance bottleneck. For this reason, performance optimization has been a common theme in database research. The shared-data store is also potentially a single point of failure. Also, the producers and consumers of the shared data may be tightly coupled, through their knowledge of the structure of the shared data.

The shared-data pattern solution is summarized in [Table 13.9](#).

Table 13.9. Shared-Data Pattern Solution

Overview	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.
Elements	<i>Shared-data store.</i> Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted. <i>Data accessor component.</i> <i>Data reading and writing connector.</i> An important choice here is whether the connector is transactional or not, as well as the read/write language, protocols, and semantics.
Relations	<i>Attachment</i> relation determines which data accessors are connected to which data stores.
Constraints	Data accessors interact with the data store(s).
Weaknesses	The shared-data store may be a performance bottleneck. The shared-data store may be a single point of failure. Producers and consumers of data may be tightly coupled.

The shared-data pattern is useful whenever various data items are persistent and have multiple accessors. Use of this pattern has the effect of decoupling the producer of the data from the consumers of the data; hence, this pattern supports modifiability, as the producers do not have direct knowledge of the consumers. Consolidating the data in one or more locations and accessing it in a common fashion facilitates performance tuning. Analyses associated with this pattern usually center on qualities such as data consistency, performance, security, privacy, availability, scalability, and compatibility with, for example, existing repositories and their data.

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 or availability through redundancy. Such choices can strongly affect the qualities noted above.

[Figure 13.13](#) shows the diagram of a shared-data view of an enterprise access management system. There are three types of accessor components: Windows applications, web applications, and headless programs (programs or scripts that run in background and don't provide any user interface).

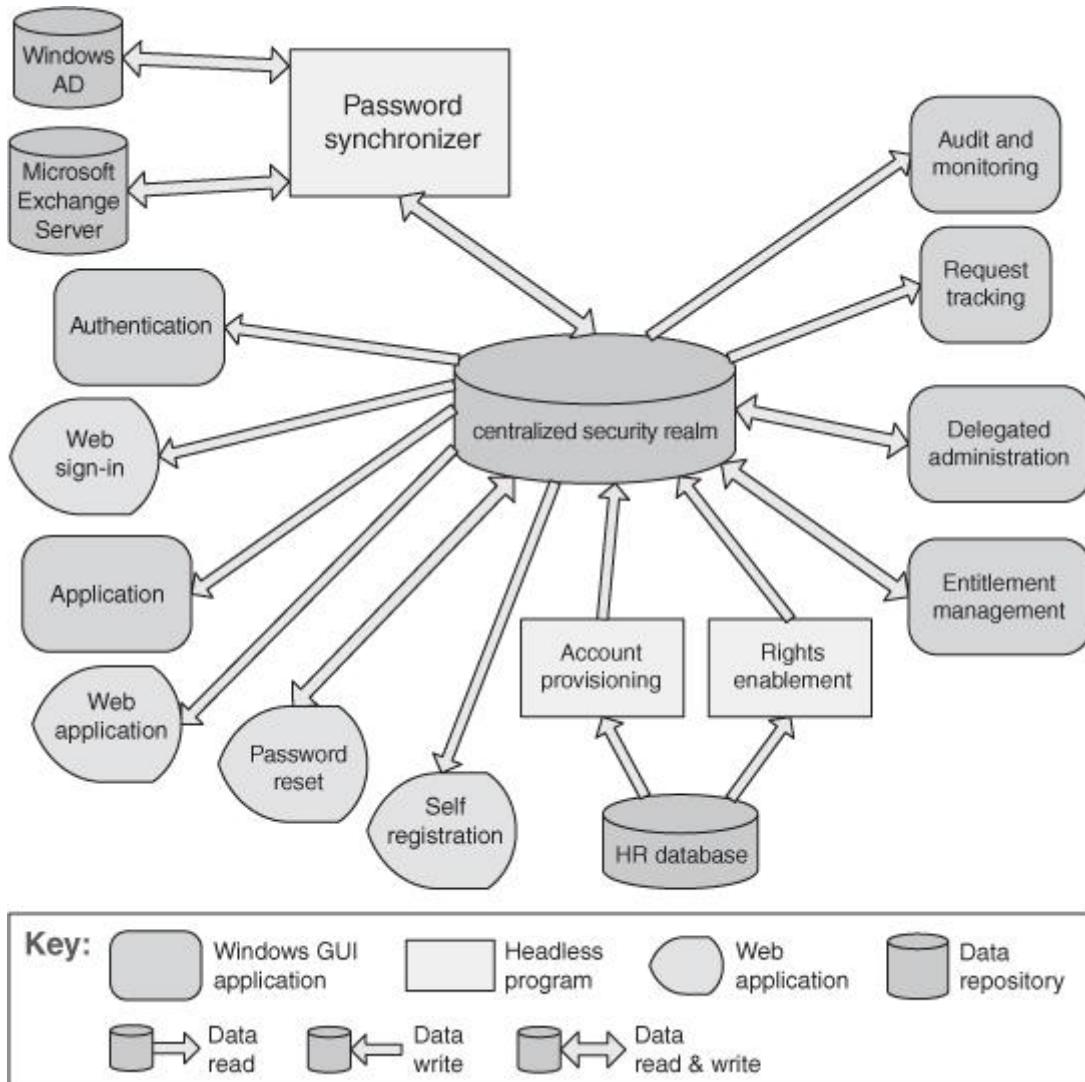


Figure 13.13. The shared-data diagram of an enterprise access management system

Allocation Patterns

Map-Reduce Pattern

Context: Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale. Examples include logs of interactions in a social network site, massive document or data repositories, and pairs of <source, target> web links for a search engine. Programs for the analysis of this data should be easy to write, run efficiently, and be resilient with respect to hardware failure.

Problem: For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.

Solution: The map-reduce pattern requires three parts: First, a specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed. A node may be a standalone processor or a core in a multi-core chip. Second and third are two programmer-coded functions called, predictably enough, *map* and *reduce*.

The map function takes as input a key (key1) and a data set. The purpose of the map function is to filter and sort the data set. All of the heavy analysis takes place in the reduce function. The

input key in the map function is used to filter the data. Whether a data record is to be involved in further processing is determined by the map function. A second key (key2) is also important in the map function. This is the key that is used for sorting. The output of the map function consists of a $\langle \text{key2}, \text{value} \rangle$ pair, where the key2 is the sorting value and the value is derived from the input record.

Sorting is performed by a combination of the map and the infrastructure. Each record output by map is hashed by key2 into a disk partition. The infrastructure maintains an index file for key2 on the disk partition. This allows for the values on the disk partition to be retrieved in key2 order.

The performance of the map phase of map-reduce is enhanced by having multiple map instances, each processing a different portion of the disk file being processed. [Figure 13.14](#) shows how the map portion of map-reduce processes data. An input file is divided into portions, and a number of map instances are created to process each portion. The map function processes its portion into a number of partitions, based on programmer-specified logic.

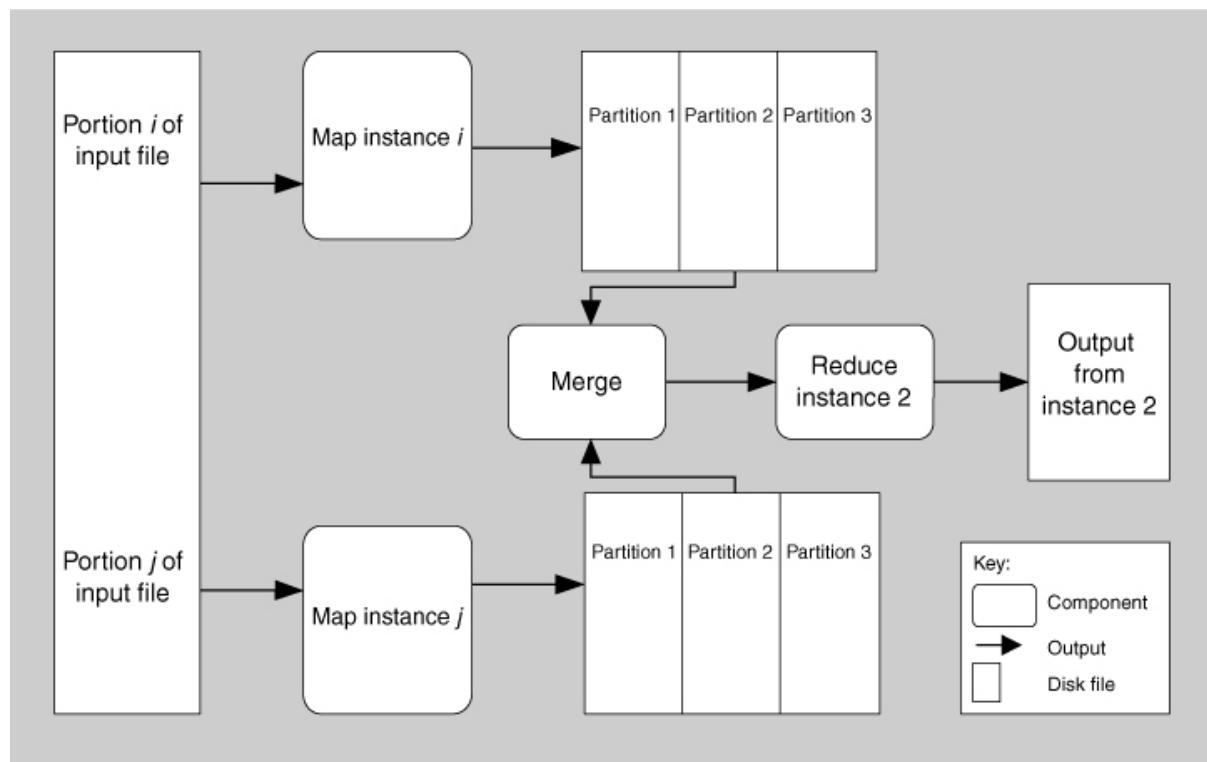


Figure 13.14. A component-and-connector view of map-reduce showing how the data processed by map is partitioned and subsequently processed by reduce

The reduce function is provided with all the sets of $\langle \text{key2}, \text{value} \rangle$ pairs emitted by all the map instances in sorted order. Reduce does some programmer-specified analysis and then emits the results of that analysis. The output set is almost always much smaller than the input sets, hence the name “reduce.” The term “load” is sometimes used to describe the final set of data emitted. [Figure 13.14](#) also shows one instance (of many possible instances) of the reduce processing, called Reduce Instance 2. Reduce Instance 2 is receiving data from all of the Partition 2s produced by the various map instances. It is possible that there are several iterations of reduce for large files, but this is not shown in [Figure 13.14](#).

A classic teaching problem for map-reduce is counting word occurrences in a document. This example can be carried out with a single map function. The document is the data set. The map function will find every word in the document and output a $\langle \text{word}, 1 \rangle$ pair for each. For example, if the document begins with the words “Having a whole book ...,” then the first results of map will be

```
<Having, 1>
<a, 1>
```

```
<whole, 1>
<book, 1>
```

In practice, the “a” would be one of the words filtered by map.

Pseudocode for map might look like this:

```
map(String key, String value) :
// key: document name
// value: document contents
for each word w in value:
Emit (w, "1");
```

The reduce function will take that list in sorted order, add up the 1s for each word to get a count, and output the result.

The corresponding reduce function would look like this:

```
reduce(List <key, value>):
// key: a word
// value: an integer
int result = 0;
sort input
for each input value:
for each input pair with same word
result ++ ;
Emit (word, result)
result = 0
```

Larger data sets lead to a much more interesting solution. Suppose we want to continuously analyze Twitter posts over the last hour to see what topics are currently “trending.” This is analogous to counting word occurrences in millions of documents. In that case, each document (tweet) can be assigned to its own instance of the map function. (If you don’t have millions of processors handy, you can break the tweet collection into groups that match the number of processors in your processor farm, and process the collection in waves, one group after the other.) Or we can use a dictionary to give us a list of words, and each map function can be assigned its own word to look for across all tweets.

There can also be multiple instances of reduce. These are usually arranged so that the reduction happens in stages, with each stage processing a smaller list (with a smaller number of reduce instances) than the previous stage. The final stage is handled by a single reduce function that produces the final output.

Of course, the map-reduce pattern is not appropriate in all instances. Some considerations that would argue against adopting this pattern are these:

- If you do not have large data sets, then the overhead of map-reduce is not justified.
- If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.
- If you have operations that require multiple reduces, this will be complex to orchestrate.

Commercial implementations of map-reduce provide infrastructure that takes care of assignment of function instances to hardware, recovery and reassignment in case of hardware failure (a common occurrence in massively parallel computing environments), and utilities like sorting of the massive lists that are produced along the way.

[Table 13.10](#) summarizes the solution of the map-reduce pattern.

Table 13.10. Map-Reduce Pattern Solution

Overview	The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the <i>extract</i> and <i>transform</i> portions of the analysis and the reduce performs the <i>loading</i> of the results. (<i>Extract-transform-load</i> is sometimes used to describe the functions of the map and reduce.)
Elements	<p><i>Map</i> is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis.</p> <p><i>Reduce</i> is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.</p> <p>The <i>infrastructure</i> is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.</p>
Relations	<p><i>Deploy on</i> is the relation between an instance of a map or reduce function and the processor onto which it is installed.</p> <p><i>Instantiate, monitor, and control</i> is the relation between the infrastructure and the instances of map and reduce.</p>
Constraints	<p>The data to be analyzed must exist as a set of files.</p> <p>The map functions are stateless and do not communicate with each other.</p> <p>The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs.</p>
Weaknesses	<p>If you do not have large data sets, the overhead of map-reduce is not justified.</p> <p>If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.</p> <p>Operations that require multiple reduces are complex to orchestrate.</p>

Map-reduce is a cornerstone of the software of some of the most familiar names on the web, including Google, Facebook, eBay, and Yahoo!

Multi-tier Pattern

The multi-tier pattern is a C&C pattern or an allocation pattern, depending on the criteria used to define the tiers. Tiers can be created to group components of similar functionality, in which case it is a C&C pattern. However, in many, if not most, cases tiers are defined with an eye toward the computing environment on which the software will run: A client tier in an enterprise system will not be running on the computer that hosts the database. That makes it an allocation pattern, mapping software elements—perhaps produced by applying C&C patterns—to computing elements. Because of that reason, we have chosen to list it as an allocation pattern.

Context: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets. This may be for operational or business reasons (for example, different parts of the infrastructure may belong to different organizations).

Problem: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media? This is done to provide specific server environments optimized for operational requirements and resource usage.

Solution: 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.

The use of tiers may be applied to any collection (or pattern) of runtime components, although in practice it is most often used in the context of client-server patterns. 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 pattern found in many Java EE and Microsoft .NET applications is an example of organization in tiers derived from the client-server pattern.

Additionally, tiers may constrain the *kinds* of communication that can take place across adjacent tiers. For example, some tiered patterns require call-return communication in one direction but event-based notification in the other.

The main weakness with the multi-tier architecture is its cost and complexity. For simple systems, the benefits of the multi-tier architecture may not justify its up-front and ongoing costs, in terms of hardware, software, and design and implementation complexity.

Tiers are not components, but rather logical groupings of components. Also, don't confuse tiers with layers! Layering is a pattern of modules (a unit of implementation), while tiers applies only to runtime entities.

[Table 13.11](#) summarizes the solution part of the multi-tier pattern.

Table 13.11. Multi-tier Pattern Solution

Overview	The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a <i>tier</i> . 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.
Elements	<i>Tier</i> , which is a logical grouping of software components. Tiers may be formed on the basis of common computing platforms, in which case those platforms are also elements of the pattern.
Relations	<i>Is part of</i> , to group components into tiers. <i>Communicates with</i> , to show how tiers and the components they contain interact with each other. <i>Allocated to</i> , in the case that tiers map to computing platforms.
Constraints	A software component belongs to exactly one tier.
Weaknesses	Substantial up-front cost and complexity.

Tiers make it easier to ensure security, and to optimize performance and availability in specialized ways. They also enhance the modifiability of the system, as the computationally independent subgroups need to agree on protocols for interaction, thus reducing their coupling.

[Figure 13.15](#) uses an informal notation to describe the multi-tier architecture of the Consumer Website Java EE application. This application is part of the Adventure Builder system. Many component-and-connector types are specific to the supporting platform, which is Java EE in this case.

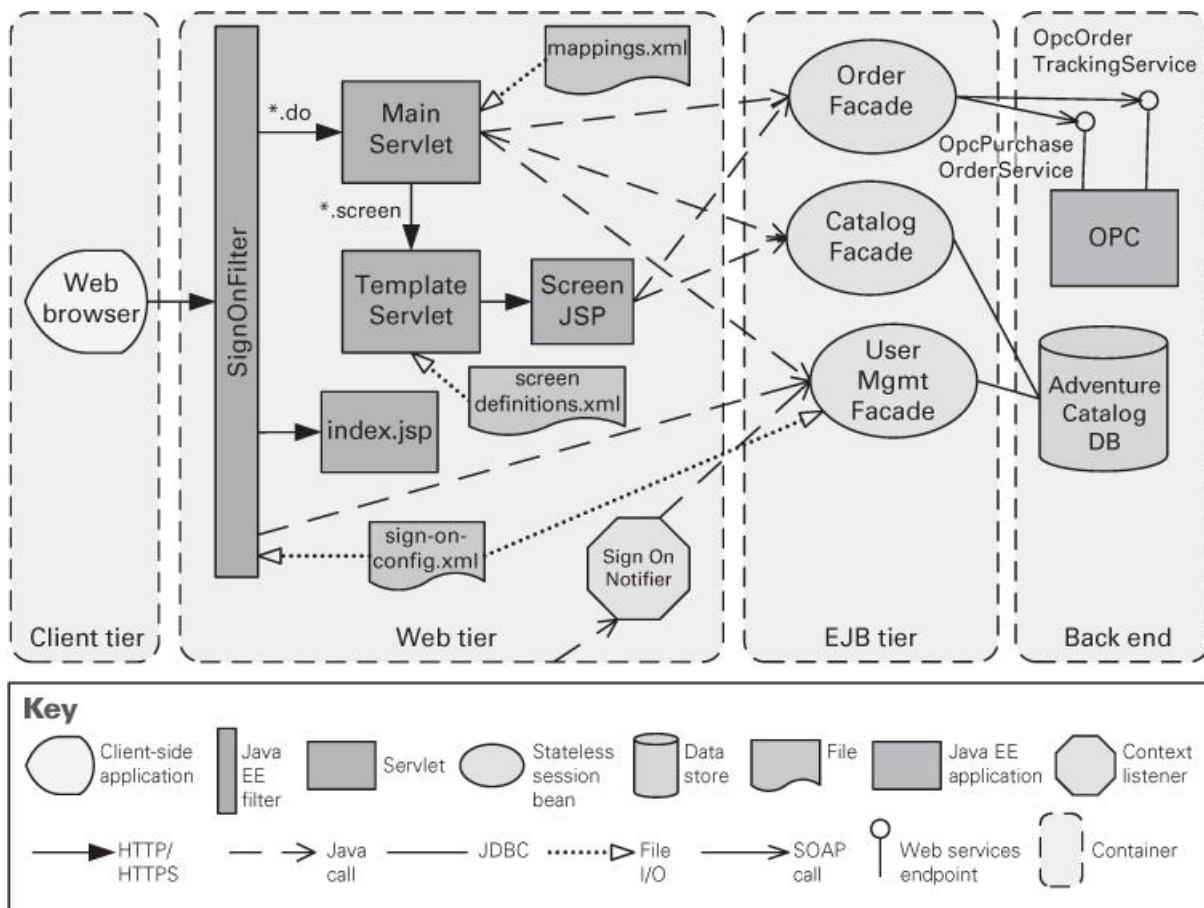


Figure 13.15. A multi-tier view of the Consumer Website Java EE application, which is part of the Adventure Builder system

Other Allocation Patterns

There are several published deployment styles. 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. 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 patterns) 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.”

There are also published work assignment patterns. These take the form of often-used team structures. For example, patterns for globally distributed Agile projects include these:

- *Platform.* In 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.* 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.* 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.

13.3. Relationships between Tactics and Patterns

Patterns and tactics together constitute the software architect’s primary tools of the trade. How do they relate to each other?

Patterns Comprise Tactics

As we said in the introduction to this chapter, tactics are the “building blocks” of design from which architectural patterns are created. Tactics are atoms and patterns are molecules. Most patterns consist of (are constructed from) several different tactics, and although these tactics might all serve a common purpose—such as promoting modifiability, for example—they are often chosen to promote *different* quality attributes. For example, a tactic might be chosen that makes an availability pattern more secure, or that mitigates the performance impact of a modifiability pattern.

Consider the example of the layered pattern, the most common pattern in all of software architecture (virtually all nontrivial systems employ layering). The layered pattern can be seen as the amalgam of several tactics—increase semantic coherence, abstract common services, encapsulate, restrict communication paths, and use an intermediary. For example:

- *Increase semantic coherence.* The goal of ensuring that a layer’s responsibilities all work together without excessive reliance on other layers is achieved by choosing responsibilities that have semantic coherence. Doing so binds responsibilities that are likely to be affected by a change. For example, responsibilities that deal with hardware should be allocated to a hardware layer and not to an application layer; a hardware responsibility typically does not have semantic coherence with the application responsibilities.
- *Restrict dependencies.* Layers define an ordering and only allow a layer to use the services of its adjacent lower layer. The possible communication paths are reduced to the number of layers minus one. This limitation has a great influence on the dependencies between the layers and makes it much easier to limit the side effects of replacing a layer.

Without any one of its tactics, the pattern might be ineffective. For example, if the restrict dependencies tactic is not employed, then any function in any layer can call any other function in any other layer, destroying the low coupling that makes the layering pattern effective. If the increase semantic coherence tactic is not employed, then functionality could be randomly sprinkled throughout the layers, destroying the separation of concerns, and hence ease of modification, which is the prime motivation for employing layers in the first place.

[Table 13.12](#) shows a number of the architectural patterns described in the book *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, by Buschmann et al., and shows which modifiability tactics they employ.

Table 13.12. Architecture Patterns and Corresponding Tactics ([Bachmann 07])

Using Tactics to Augment Patterns

A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific. A documented pattern is therefore underspecified with respect to applying it in a specific situation.

To make a pattern work in a given architectural context, we need to examine it from two perspectives:

- The inherent quality attribute tradeoffs that the pattern makes. Patterns exist to achieve certain quality attributes, and we need to compare the ones they promote (and the ones they diminish) with our needs.
- Other quality attributes that the pattern isn’t directly concerned with, but which it nevertheless affects, and which are important in our application.

To illustrate these concerns in particular, and how to use tactics to augment patterns in general, we’ll use the broker pattern as a starting point.

The broker pattern is widely used in distributed systems and dates back at least to its critical role in CORBA-based systems. Broker is a crucial component of any large-scale, dynamic, service-oriented architecture.

Using this pattern, a client requesting some information from a server does not need to know the location or APIs of the server. The client simply contacts the broker (typically through a client-side proxy); this is illustrated in the UML sequence diagram in [Figure 13.16](#).

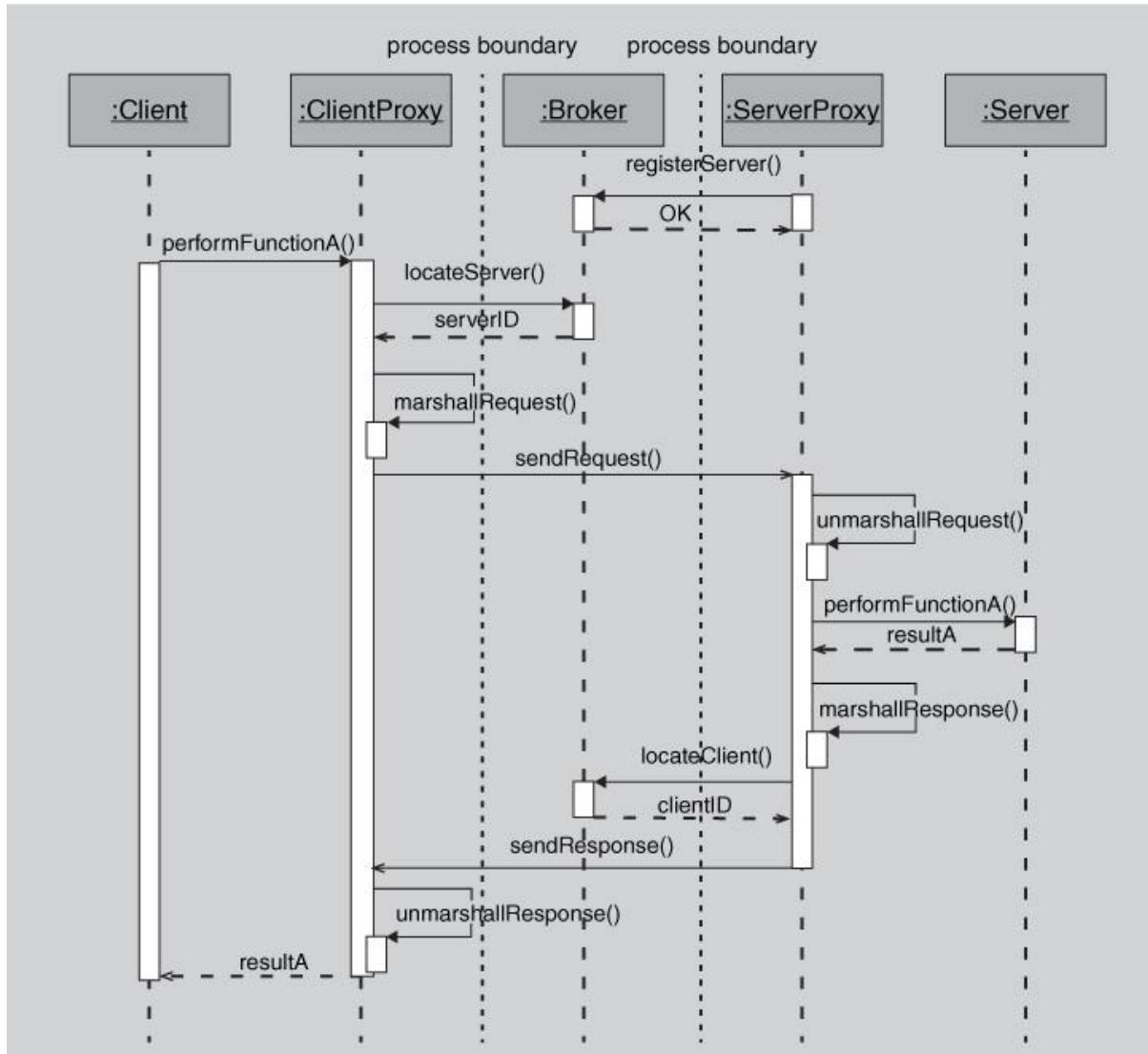


Figure 13.16. A sequence diagram showing a typical client-server interaction mediated by a broker

Weaknesses of the Broker Pattern

In [Section 13.2](#) we enumerated several weaknesses of the broker pattern. Here we will examine these weaknesses in more detail. The broker pattern has several weaknesses with respect to certain quality attributes. For example:

- **Availability.** The broker, if implemented as suggested in [Figure 13.6](#), is a single point of failure. The liveness of servers, the broker, and perhaps even the clients need to be monitored, and repair mechanisms must be provided.
- **Performance.** The levels of indirection between the client (requesting the information or service) and the server (providing the information or service) add overhead, and hence add latency. Also, the broker is a potential performance bottleneck if direct communication between the client and server is not desired (for example, for security reasons).
- **Testability.** Brokers are employed in complex multi-process and multi-processor systems. Such systems are typically highly dynamic. Requests and responses are typically

asynchronous. All of this makes testing and debugging such systems extremely difficult. But the description of the broker pattern provides no testing functionality, such as testing interfaces, state or activity capture and playback capabilities, and so forth.

- **Security.** Because the broker pattern is primarily used when the system spans process and processor boundaries—such as on web-based systems—security is a legitimate concern. However, the broker pattern as presented does not offer any means to authenticate or authorize clients or servers, and provides no means of protecting the communication between clients and servers.

Of these quality attributes, the broker pattern is mainly associated with poor performance (the well-documented price for the loose coupling it brings to systems). It is largely unconcerned with the other quality attributes in this list; they aren't mentioned in most published descriptions. But as the other bullets show, they can be unacceptable “collateral damage” that come with the broker's benefits.

Improving the Broker Pattern with Tactics

How can we use tactics to plug the gaps between the “out of the box” broker pattern and a version of it that will let us meet the requirements of a demanding distributed system? Here are some options:

- The increase available resources performance tactic would lead to multiple brokers, to help with performance and availability.
- The maintain multiple copies tactic would allow each of these brokers to share state, to ensure that they respond identically to client requests.
- Load balancing (an application of the scheduling resources tactic) would ensure that one broker is not overloaded while another one sits idle.
- Heartbeat, exception detection, or ping/echo would give the replicated brokers a way of notifying clients and notifying each other when one of them is out of service, as a means of detecting faults.

Of course, each of these tactics brings a tradeoff. Each complicates the design, which will now take longer to implement, be more costly to acquire, and be more costly to maintain. Load balancing introduces indirection that will add latency to each transaction, thus giving back some of the performance it was intended to increase. And the load balancer is a single point of failure, so it too must be replicated, further increasing the design cost and complexity.

13.4. Using Tactics Together

Tactics, as described in [Chapters 5–11](#), are design primitives aimed at managing a single quality attribute response. Of course, this is almost never true in practice; every tactic has its main effect—to manage modifiability or performance or safety, and so on—and it has its side effects, its tradeoffs. On the face of it, the situation for an architect sounds hopeless. Whatever you do to improve one quality attribute endangers another. We are able to use tactics profitably because we can gauge the direct and side effects of a tactic, and when the tradeoff is acceptable, we employ the tactic. In doing so we gain some benefit in our quality attribute of interest while giving up something else (with respect to a different quality attribute and, we hope, of a much smaller magnitude).

This section will walk through an example that shows how applying tactics to a pattern can produce negative effects in one area, but how adding other tactics can bring relief and put you back in an acceptable design space. The point is to show the interplay between tactics that you can use to your advantage. Just as some combinations of liquids are noxious whereas others yield lovely things like strawberry lemonade, tactics can either make things worse or put you in a happy design space. Here, then, is a walkthrough of tactic mixology.

Consider a system that needs to detect faults in its components. A common tactic for detecting faults is ping/echo. Let us assume that the architect has decided to employ ping/echo as a way to detect failed components in the system. Every tactic has one or more side effects, and ping/echo is no different. Common considerations associated with ping/echo are these:

- **Security.** How to prevent a ping flood attack?
- **Performance.** How to ensure that the performance overhead of ping/echo is small?

- *Modifiability.* How to add ping/echo to the existing architecture?

We can represent the architect's reasoning and decisions thus far as shown in [Figure 13.17](#).

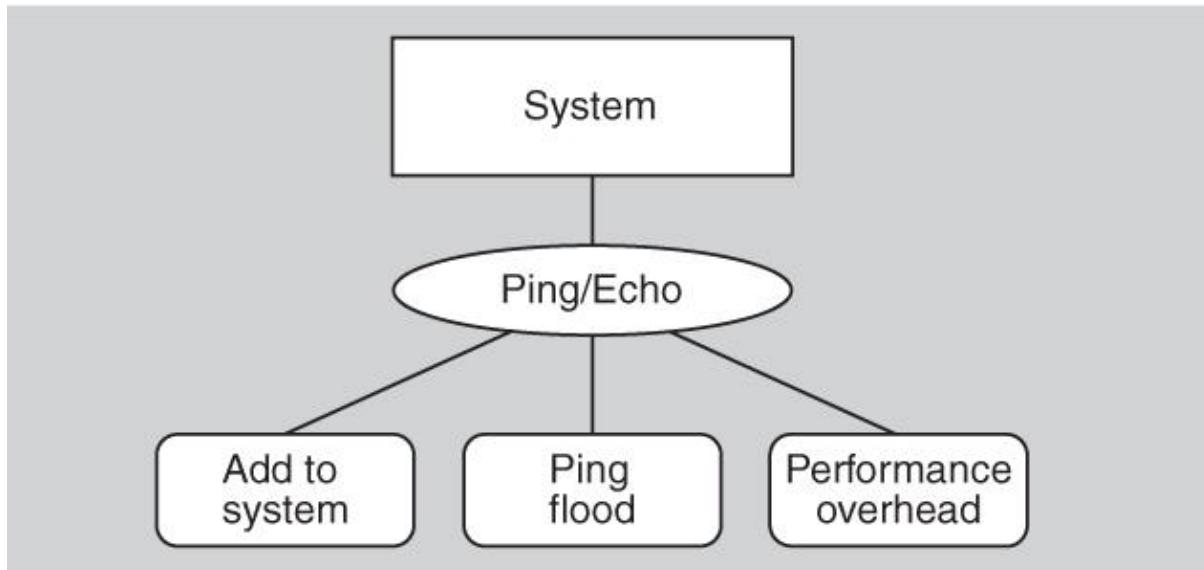


Figure 13.17. Partial availability decisions

Suppose the architect determines that the performance tradeoff (the overhead of adding ping/echo to the system) is the most severe. A tactic to address the performance side effect is *increase available resources*. Considerations associated with increase available resources are these:

- *Cost.* Increased resources cost more.
- *Performance.* How to utilize the increased resources efficiently?

This set of design decisions can now be represented as shown in [Figure 13.18](#).

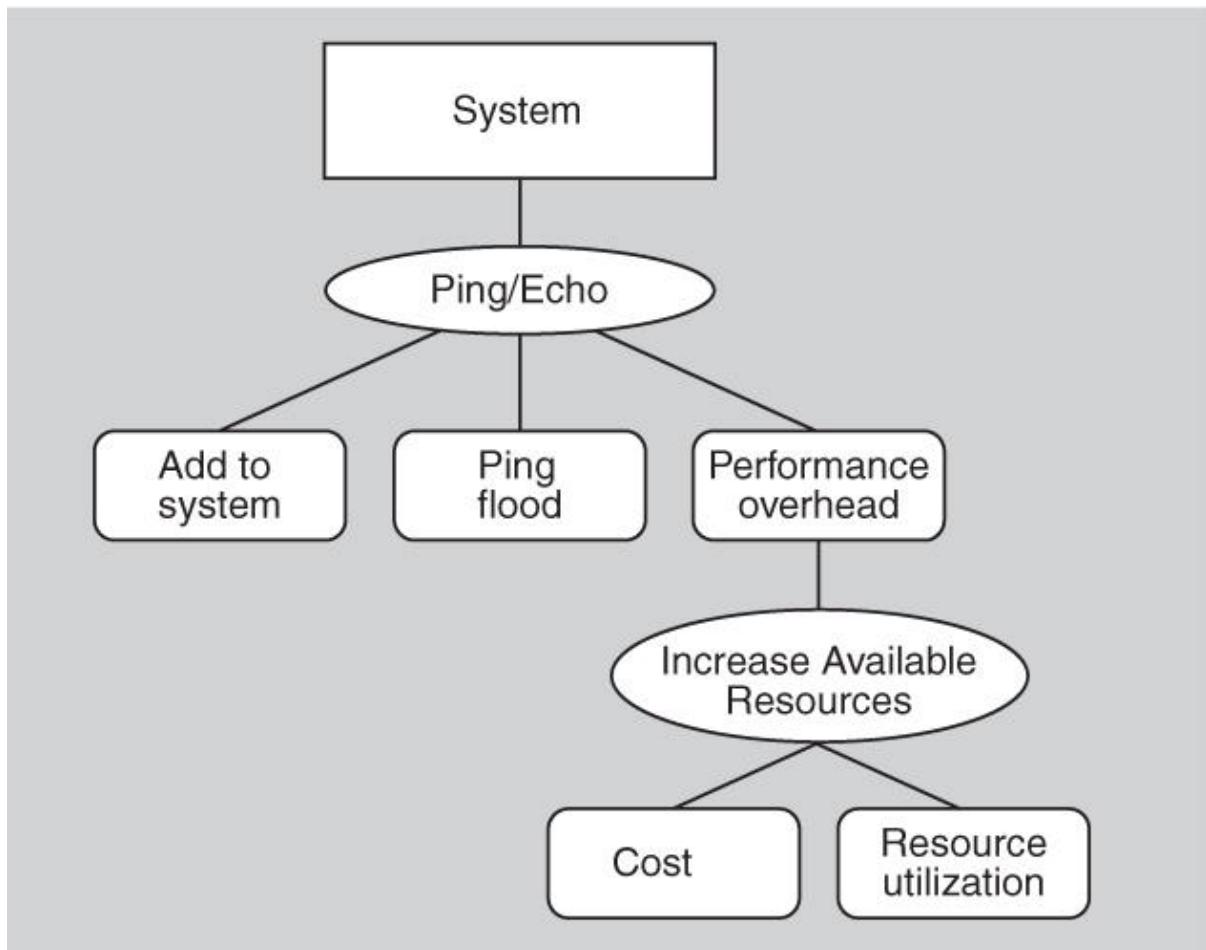


Figure 13.18. More availability decisions

Now the architect chooses to deal with the resource utilization consequence of employing increase available resources. These resources must be used efficiently or else they are simply adding cost and complexity to the system. A tactic that can address the efficient use of resources is the employment of a *scheduling policy*. Considerations associated with the scheduling policy tactic are these:

- *Modifiability.* How to add the scheduling policy to the existing architecture?
- *Modifiability.* How to change the scheduling policy in the future?

The set of design decisions that includes the scheduling policy tactic can now be represented as in [Figure 13.19](#).

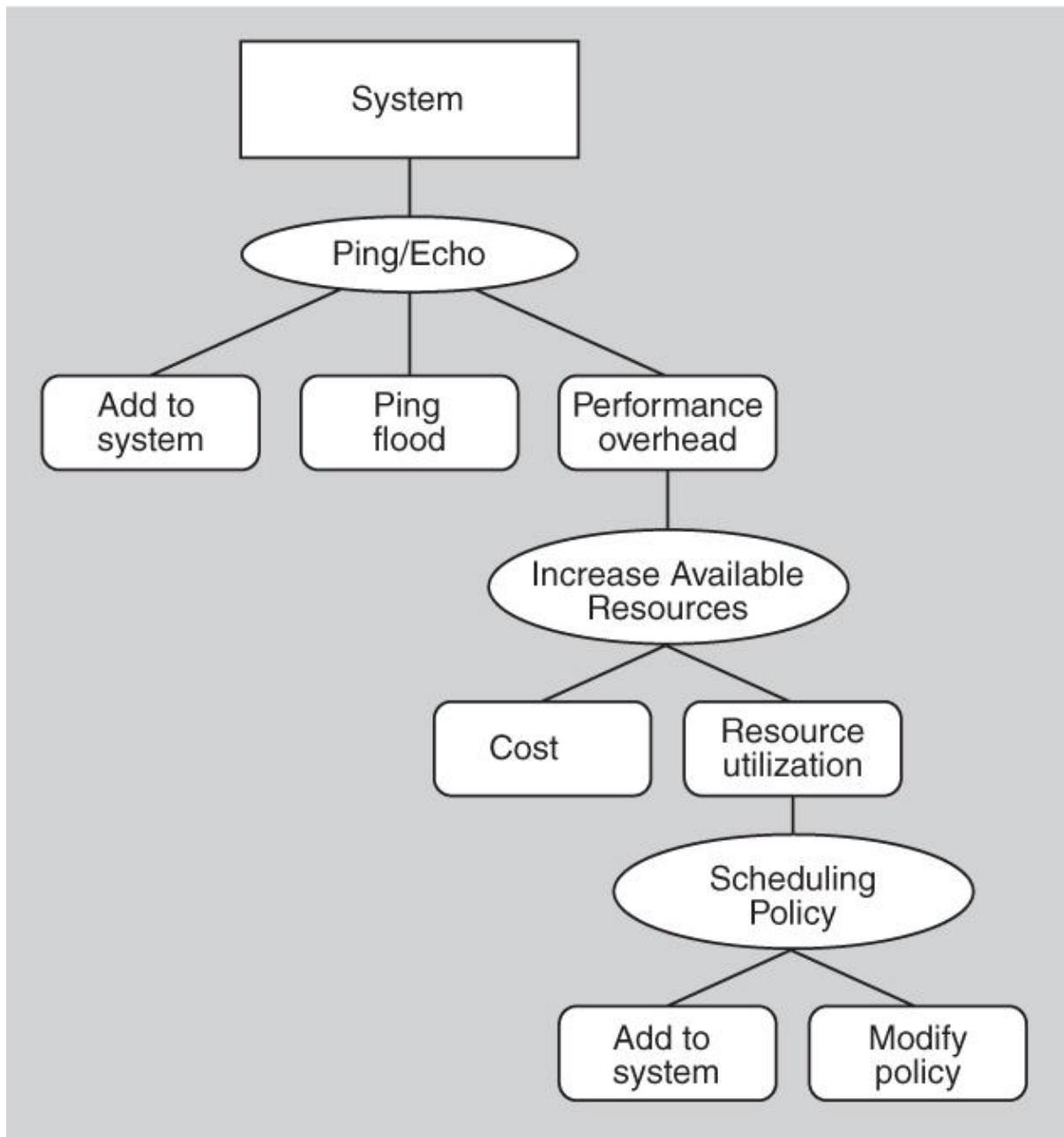


Figure 13.19. Still more availability decisions

Next the architect chooses to deal with the modifiability consequence of employing a scheduling policy tactic. A tactic to address the addition of the scheduler to the system is to *use an intermediary*, which will insulate the choice of scheduling policy from the rest of the system. One consideration associated with use an intermediary is this:

- *Modifiability.* How to ensure that all communication passes through the intermediary?

We can now represent the tactics-based set of architectural design decisions made thus far as in [Figure 13.20](#).

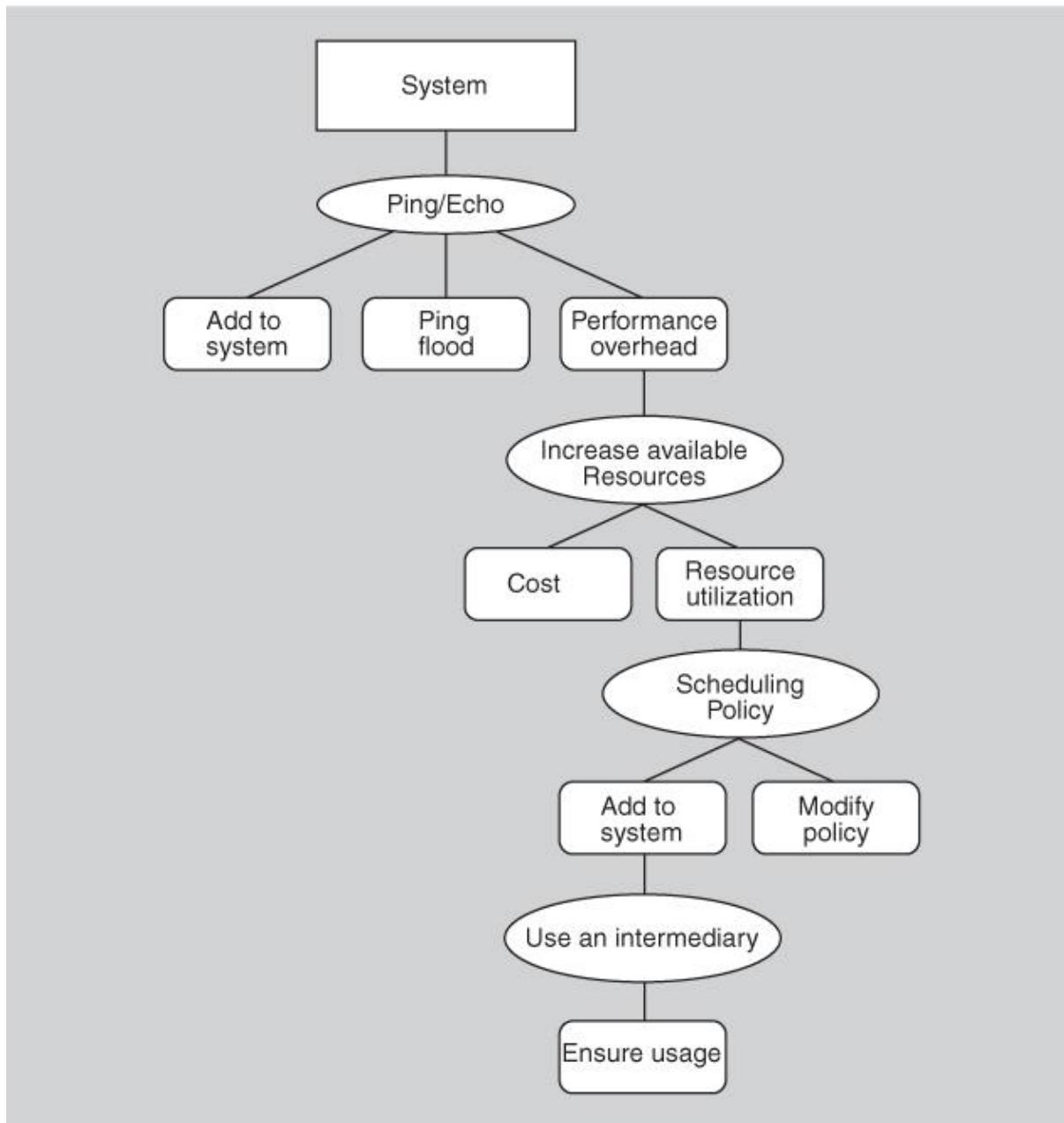


Figure 13.20. As far as we go with availability decisions

A tactic to address the concern that all communication passes through the intermediary is *restrict dependencies*. One consideration associated with the restrict dependencies tactic is this:

- *Performance*. How to ensure that the performance overhead of the intermediary is not excessive?

This design problem has now become recursive! At this point (or in fact, at any point in the tree of design decisions that we have described) the architect might determine that the performance overhead of the intermediary is small enough that no further design decisions need to be made.

Applying successive tactics is like moving through a game space, and it's a little like chess: Good players are able to see the consequences of the move they're considering, and the very good players are able to look several moves ahead. In [Chapter 17](#) we'll see the activity of design treated as an exercise of "generate and test": propose a design and test it to see if it's satisfactory. Applying tactics to an existing design solution, such as a pattern, is one technique for generating a design for subsequent testing.

13.5. Summary

An architectural pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a *class* of architectures.

Because patterns are (by definition) found repeatedly in practice, one does not *invent* them; one *discovers* them.

Tactics are simpler than patterns. Tactics typically use just a single structure or computational mechanism, and they are meant to address a single architectural force. For this reason they give more precise control to an architect when making design decisions than patterns, which typically combine multiple design decisions into a package. Tactics are the “building blocks” of design from which architectural patterns are created. Tactics are atoms and patterns are molecules.

An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem*. The problem, appropriately generalized, that arises in the given context.
- *A solution*. A successful architectural resolution to the problem, appropriately abstracted.

Complex systems exhibit multiple patterns at once.

Patterns can be categorized by the dominant type of elements that they show: module patterns show modules, component-and-connector patterns show components and connectors, and allocation patterns show a combination of software elements (modules, components, connectors) and nonsoftware elements. Most published patterns are C&C patterns, but there are module patterns and allocation patterns as well. This chapter showed examples of each type.

A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific. A documented pattern is therefore underspecified with respect to applying it in a specific situation. We can make a pattern more specific to our problem by augmenting it with tactics. Applying successive tactics is like moving through a game space, and is a little like chess: the consequences of the next move are important, and looking several moves ahead is helpful.

13.6. For Further Reading

There are many existing repositories of patterns and books written about patterns. The original and most well-known work on object-oriented design patterns is by the “Gang of Four” [\[Gamma 94\]](#).

The Gang of Four’s discussion of patterns included patterns at many levels of abstraction. In this chapter we have focused entirely on architectural patterns. The patterns that we have presented here are intended as representative examples. This chapter’s inventory of patterns is in no way meant to be exhaustive. For example, while we describe the SOA pattern, entire repositories of SOA patterns (refinements of the basic SOA pattern) have been created. A good place to start is www.soappatterns.org.

Some good references for pattern-oriented architecture are [\[Buschmann 96\]](#), [\[Hanmer 07\]](#), [\[Schmidt 00\]](#), and [\[Kircher 03\]](#).

A good place to learn more about the map-reduce pattern is Google’s foundational paper on it [\[Dean 04\]](#).

Map-reduce is the tip of the spear of the so-called “NoSQL” movement, which seeks to displace the relational database from its venerable and taken-for-granted status in large data-processing systems. The movement has some of the revolutionary flavor of the Agile movement, except that NoSQL advocates are claiming a better (for them) technology, as opposed to a better process. You can easily find NoSQL podcasts, user forums, conferences, and blogs; it’s also discussed in [Chapter 26](#).

[\[Bachmann 07\]](#) discusses the use of tactics in the layered pattern and is the source for some of our discussion of that.

The passage in this chapter about augmenting ping/echo with other tactics to achieve the desired combination of quality attributes is based on the work of Kiran Kumar and TV Prabhakar [[Kumar 10a](#)] and [[Kumar 10b](#)].

[[Urdangarin 08](#)] is the source of the work assignment patterns described in [Section 13.2](#).

The Adventure Builder system shown in [Figures 13.11](#) and [13.15](#) comes from [[AdvBuilder 10](#)].

13.7. Discussion Questions

1. What's the difference between an *architectural* pattern, such as those described in this chapter and in the Pattern-Oriented Software Architecture series of books, and *design* patterns, such as those collected by the Gang of Four in 1994 and many other people subsequently? Given a pattern, how would you decide whether it was an architectural pattern, a design pattern, a code pattern, or something else?
2. SOA systems feature dynamic service registration and discovery. Which quality attributes does this capability enhance and which does it threaten? If you had to make a recommendation to your boss about whether your company's SOA system should use external services it discovers at runtime, what would you say?
3. Write a complete pattern description for the "competence center" work assignment pattern mentioned in [Section 13.2](#).
4. For a data set that is a set of web pages, sketch a map function and a reduce function that together provide a basic search engine capability.
5. Describe how the layered pattern makes use of these tactics: abstract common services, encapsulate, and use an intermediary.

14. Quality Attribute Modeling and Analysis

Do not believe in anything simply because you have heard it. . . Do not believe in anything merely on the authority of your teachers and elders. Do not believe in traditions because they have been handed down for many generations. But after observation and analysis, when you find that anything agrees with reason and is conducive to the good and benefit of one and all, then accept it and live up to it.

—Prince Gautama Siddhartha

In [Chapter 2](#) we listed thirteen reasons why architecture is important, worth studying, and worth practicing. Reason 6 is that the analysis of an architecture enables early prediction of a system's qualities. This is an extraordinarily powerful reason! Without it, we would be reduced to building systems by choosing various structures, implementing the system, measuring the system for its quality attribute responses, and all along the way hoping for the best. Architecture lets us do better than that, much better. We can analyze an architecture to see how the system or systems we build from it will perform with respect to their quality attribute goals, even before a single line of code has been written. This chapter will explore how.

The methods available depend, to a large extent, on the quality attribute to be analyzed. Some quality attributes, especially performance and availability, have well-understood and strongly validated analytic modeling techniques. Other quality attributes, for example security, can be analyzed through checklists. Still others can be analyzed through back-of-the-envelope calculations and thought experiments.

Our topics in this chapter range from the specific, such as creating models and analyzing checklists, to the general, such as how to generate and carry out the thought experiments to perform early (and necessarily crude) analysis. Models and checklists are focused on particular quality attributes but can aid in the analysis of any system with respect to those attributes. Thought experiments, on the other hand, can consider multiple quality attributes simultaneously but are only applicable to the specific system under consideration.

14.1. Modeling Architectures to Enable Quality Attribute Analysis

Some quality attributes, most notably performance and availability, have well-understood, time-tested analytic models that can be used to assist in an analysis. By *analytic model*, we mean one that supports quantitative analysis. Let us first consider performance.

Analyzing Performance

In [Chapter 12](#) we discussed the fact that models have parameters, which are values you can set to predict values about the entity being modeled (and in [Chapter 12](#) we showed how to use the parameters to help us derive tactics for the quality attribute associated with the model). As an example we showed a queuing model for performance as [Figure 12.2](#), repeated here as [Figure 14.1](#). The parameters of this model are the following:

- The arrival rate of events
- The chosen queuing discipline
- The chosen scheduling algorithm
- The service time for events
- The network topology
- The network bandwidth
- The routing algorithm chosen

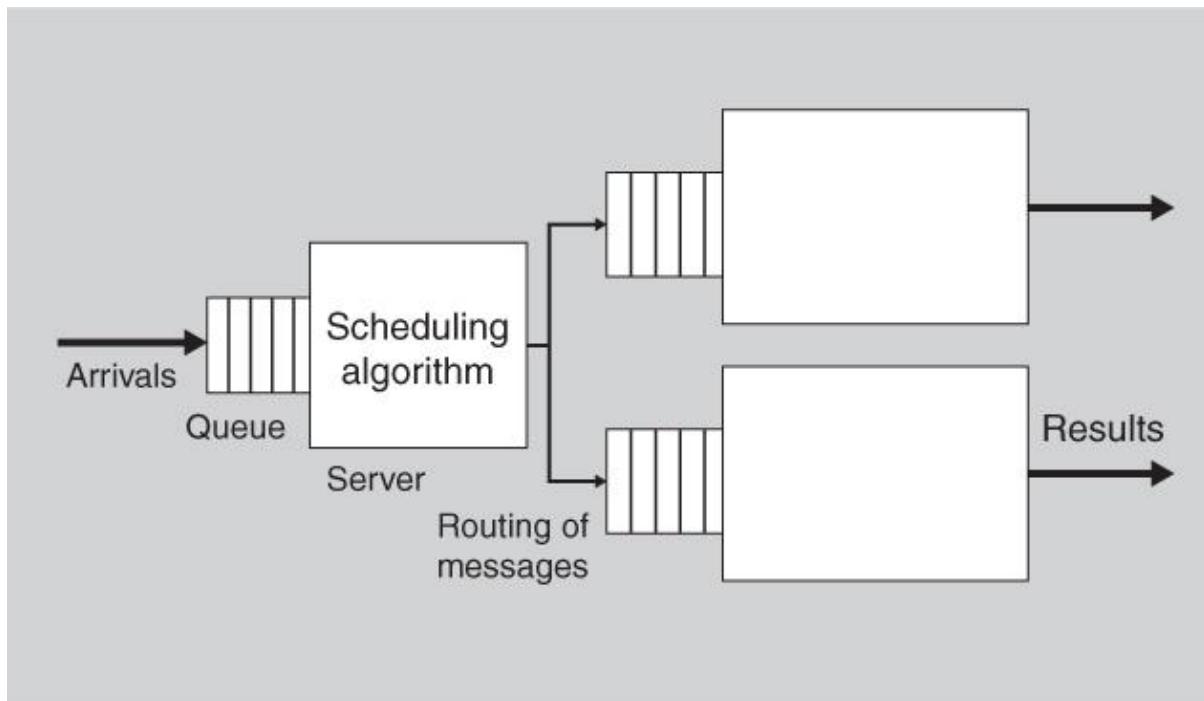


Figure 14.1. A queuing model of performance

In this section, we discuss how such a model can be used to understand the latency characteristics of an architectural design.

To apply this model in an analytical fashion, we also need to have previously made some architecture design decisions. We will use model-view-controller as our example here. MVC, as presented in [Section 13.2](#), says nothing about its deployment. That is, there is no specification of how the model, the view, and the controller are assigned to processes and processors; that's not part of the pattern's concern. These and other design decisions have to be made to transform a pattern into an architecture. Until that happens, one cannot say anything with authority about how an MVC-based implementation will perform. For this example we will assume that there is one

instance each of the model, the view, and the controller, and that each instance is allocated to a separate processor. [Figure 14.2](#) shows MVC following this allocation scheme.

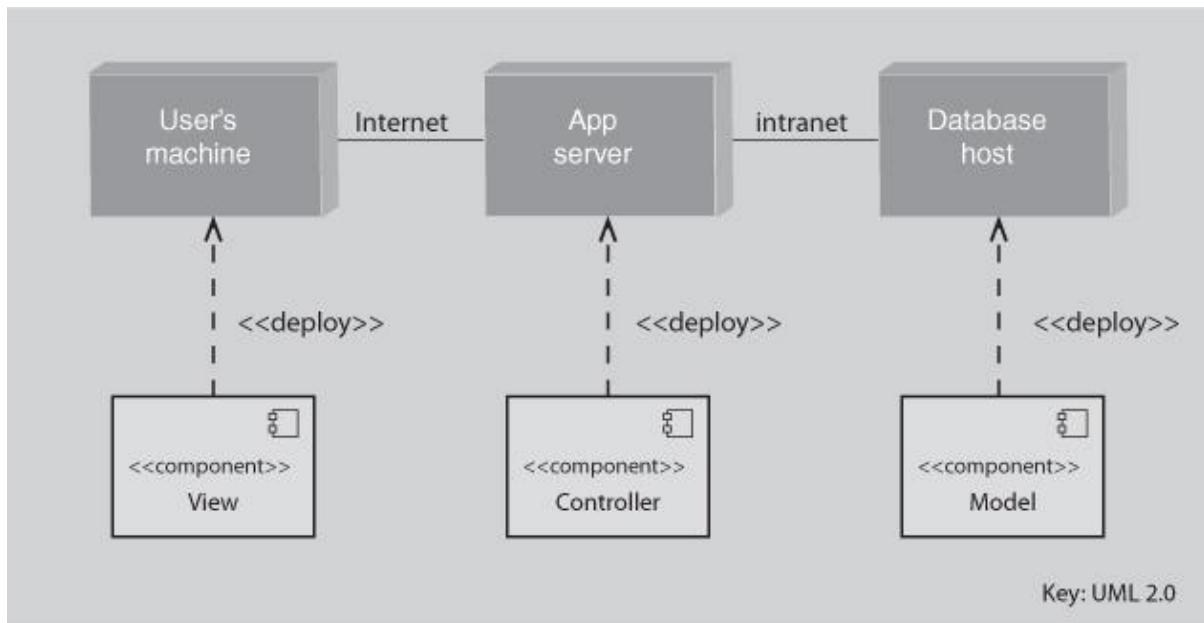


Figure 14.2. An allocation view, in UML, of a model-view-controller architecture

Given that quality attribute models such as the performance model shown in [Figure 14.1](#) already exist, the problem becomes how to map these allocation and coordination decisions onto [Figure 14.1](#). Doing this yields [Figure 14.3](#). There are requests coming from users outside the system—labeled as 1 in [Figure 14.3](#)—arriving at the view. The view processes the requests and sends some transformation of the requests on to the controller—labeled as 2. Some actions of the controller are returned to the view—labeled as 3. The controller sends other actions on to the model—labeled 4. The model performs its activities and sends information back to the view—labeled 5.

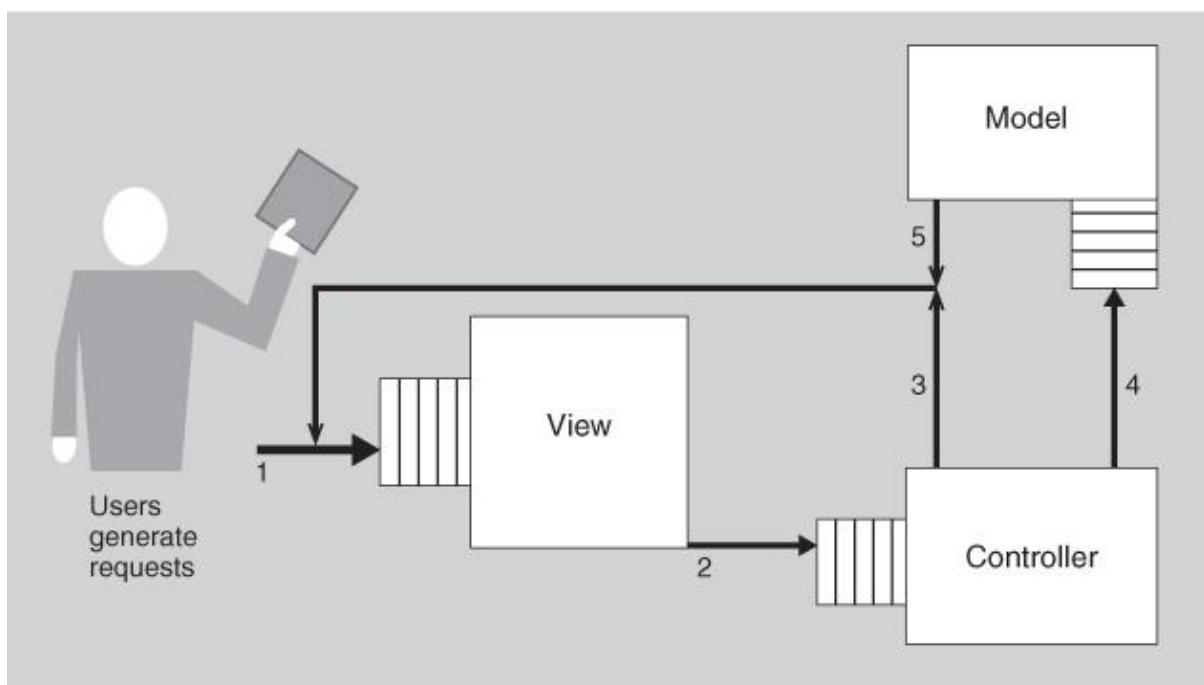


Figure 14.3. A queuing model of performance for MVC

To analyze the model in [Figure 14.3](#), a number of items need to be known or estimated:

- The frequency of arrivals from outside the system
- The queuing discipline used at the view queue
- The time to process a message within the view
- The number and size of messages that the view sends to the controller
- The bandwidth of the network that connects the view and the controller
- The queuing discipline used by the controller
- The time to process a message within the controller
- The number and size of messages that the controller sends back to the view
- The bandwidth of the network used for messages from the controller to the view
- The number and size of messages that the controller sends to the model
- The queuing discipline used by the model
- The time to process a message within the model
- The number and size of messages the model sends to the view
- The bandwidth of the network connecting the model and the view

Given all of these assumptions, the latency for the system can be estimated. Sometimes well-known formulas from queuing theory apply. For situations where there are no closed-form solutions, estimates can often be obtained through simulation. Simulations can be used to make more-realistic assumptions such as the distribution of the event arrivals. The estimates are only as good as the assumptions, but they can serve to provide rough values that can be used either in design or in evaluation; as better information is obtained, the estimates will improve.

A reasonably large number of parameters must be known or estimated to construct the queuing model shown in [Figure 14.3](#). The model must then be solved or simulated to derive the expected latency. This is the cost side of the cost/benefit of performing a queuing analysis. The benefit side is that as a result of the analysis, there is an estimate for latency, and “what if” questions can be easily answered. The question for you to decide is whether having an estimate of the latency and the ability to answer “what if” questions is worth the cost of performing the analysis. One way to answer this question is to consider the importance of having an estimate for the latency prior to constructing either the system or a prototype that simulates an architecture under an assumed load. If having a small latency is a crucial requirement upon which the success of the system relies, then producing an estimate is appropriate.

Performance is a well-studied quality attribute with roots that extend beyond the computer industry. For example, the queuing model given in [Figure 14.1](#) dates from the 1930s. Queuing theory has been applied to factory floors, to banking queues, and to many other domains. Models for real-time performance, such as rate monotonic analysis, also exist and have sophisticated analysis techniques.

Analyzing Availability

Another quality attribute with a well-understood analytic framework is availability.

Modeling an architecture for availability—or to put it more carefully, modeling an architecture to determine the availability of a system based on that architecture—is a matter of determining the failure rate and the recovery time. As you may recall from [Chapter 5](#), availability can be expressed as

$$\frac{MTBF}{(MTBF + MTTR)}$$

This models what is known as steady-state availability, and it is used to indicate the uptime of a system (or component of a system) over a sufficiently long duration. In the equation, *MTBF* is the *mean time between failure*, which is derived based on the expected value of the implementation’s failure probability density function (PDF), and *MTTR* refers to the *mean time to repair*.

Just as for performance, to model an architecture for availability, we need an architecture to analyze. So, suppose we want to increase the availability of a system that uses the broker pattern,

by applying redundancy tactics. [Figure 14.4](#) illustrates three well-known redundancy tactics from [Chapter 5](#): *active redundancy*, *passive redundancy*, and *cold spare*. Our goal is to analyze each redundancy option for its availability, to help us choose one.

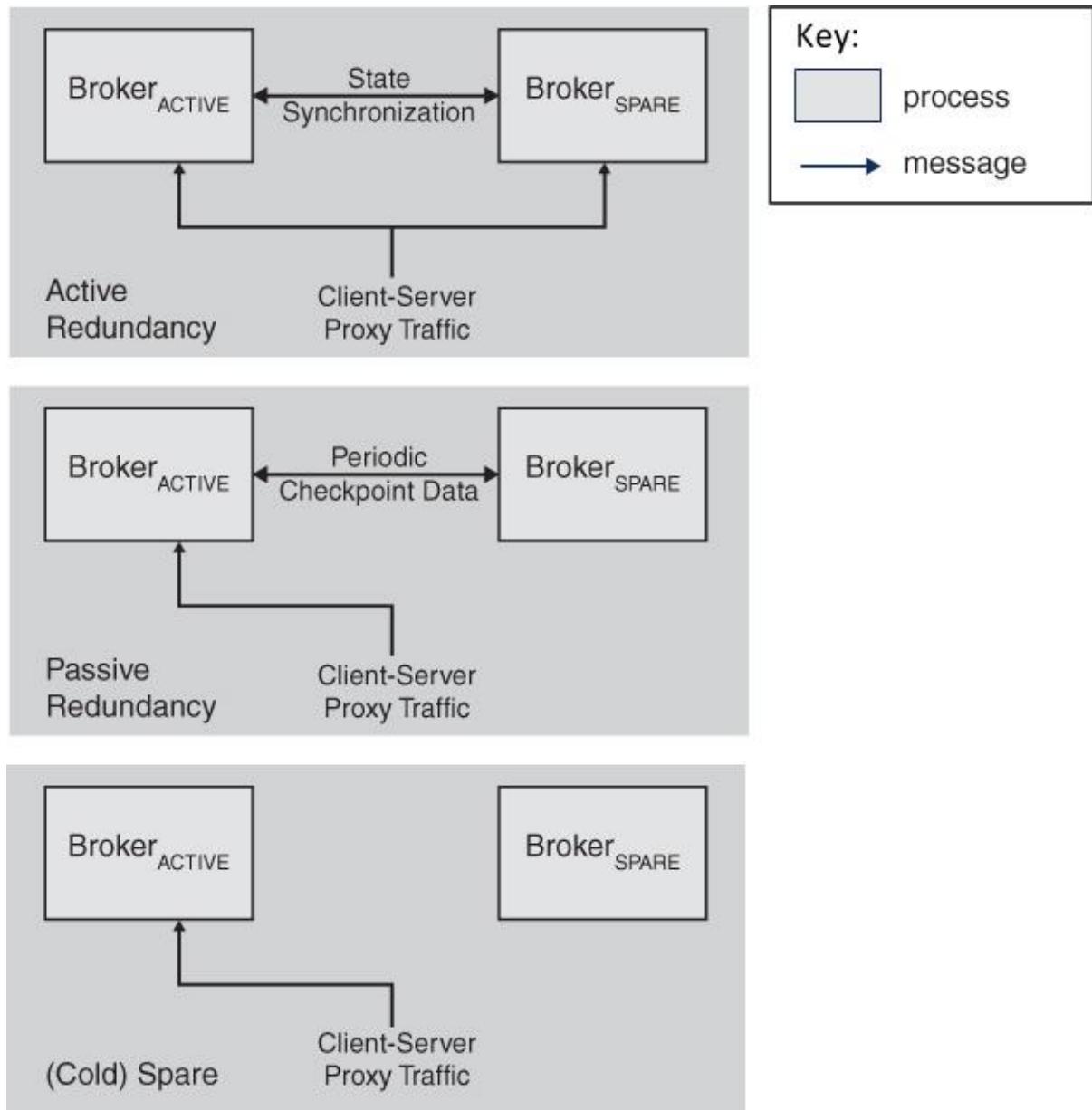


Figure 14.4. Redundancy tactics, as applied to a broker pattern

As you recall, each of these tactics introduces a backup copy of a component that will take over in case the primary component suffers a failure. In our case, a broker replica is employed as the redundant spare. The difference among them is how up to date with current events each backup keeps itself:

- In the case of active redundancy, the active and redundant brokers both receive identical copies of the messages received from the client and server proxies. The internal broker state is synchronously maintained between the active and redundant spare in order to facilitate rapid failover upon detection of a fault in the active broker.
- For the passive redundancy implementation, only the active broker receives and processes messages from the client and server proxies. When using this tactic, checkpoints of internal broker state are periodically transmitted from the active broker process to the redundant spare, using the checkpoint-based rollback tactic.

- Finally, when using the cold spare tactic, only the active broker receives and processes messages from the client and server proxies, because the redundant spare is in a dormant or even powered-off state. Recovery strategies using this tactic involve powering up, booting, and loading the broker implementation on the spare. In this scenario, the internal broker state is rebuilt organically, rather than via synchronous operation or checkpointing, as described for the other two redundancy tactics.

Suppose further that we will detect failure with the *heartbeat* tactic, where each broker (active and spare) periodically transmits a heartbeat message to a separate process responsible for fault detection, correlation, reporting, and recovery. This fault manager process is responsible for coordinating the transition of the active broker role from the failed broker process to the redundant spare.

You can now use the steady state model of availability to assign values for *MTBF* and *MTTR* for each of the three redundancy tactics we are considering. Doing so will be an exercise left to the reader (as you'll see when you reach the discussion questions for this chapter). Because the three tactics differ primarily in how long it takes to bring the backup copy up to speed, *MTTR* will be where the difference among the tactics shows up.

More sophisticated models of availability exist, based on probability. In these models, we can express a probability of failure during a period of time. Given a particular *MTBF* and a time duration *T*, the probability of failure *R* is given by

$$R(T) = \exp\left(\frac{-T}{MTBF}\right)$$

You will recall from Statistics 101 that:

- When two events A and B are independent, the probability that A or B will occur is the sum of the probability of each event: $P(A \text{ or } B) = P(A) + P(B)$.
- When two events A and B are independent, the probability of both occurring is $P(A \text{ and } B) = P(A) \cdot P(B)$.
- When two events A and B are dependent, the probability of both occurring is $P(A \text{ and } B) = P(A) \cdot P(B|A)$, where the last term means "the probability of B occurring, given that A occurs."

We can apply simple probability arithmetic to an architecture pattern for availability to determine the probability of failure of the pattern given the probability of failure of the individual components (and an understanding of their dependency relations). For example, in an architecture pattern employing the passive redundancy tactic, let's assume that the failure of a component (which at any moment might be acting as either the primary or backup copy) is independent of a failure of its counterpart, and that the probability of failure of either is the same. Then the probability that both will fail is $1 - P(F)^2$.

Still other models take into account different levels of failure severity and degraded operating states of the system. Although the derivation of these formulas is outside the scope of this chapter, you end up with formulas that look like the following for the three redundancy tactics we've been discussing, where the values C2 through C5 are references to the MTBF column of [Table 14.1](#), D2 through D4 refer to the Active column, E2 through E3 refer to the Passive column, and F2 through F3 refer to the Spare column.

- Active redundancy:
 - Availability(MTTR): $1 - ((\text{SUM}(C2:C5) + D3) \times D2) / ((C2 \times (C2 + C4 + D3)) + ((C2 + C4 + D2) \times (C3 + C5)) + ((C2 + C4) \times (C2 + C4 + D3)))$
 - $P(\text{Degraded}) = ((C3 + C5) \times D2) / ((C2 \times (C2 + C4 + D3)) + ((C2 + C4 + D2) \times (C3 + C5)) + ((C2 + C4) \times (C2 + C4 + D3)))$
- Passive redundancy:
 - Availability(MTTR_passive) = $1 - ((\text{SUM}(C2:C5) + E3) \times E2) / ((C2 \times (C2 + C4 + E3)) + ((C2 + C4 + E2) \times (C3 + C5)) + ((C2 + C4) \times (C2 + C4 + E3)))$
 - $P(\text{Degraded}) = ((C3 + C5) \times E2) / ((C2 \times (C2 + C4 + E3)) + ((C2 + C4 + E2) \times (C3 + C5)) + ((C2 + C4) \times (C2 + C4 + E3)))$
- Spare:

- Availability(MTTR) = $1 - ((\text{SUM}(\text{C2:C5}) + \text{F3}) \times \text{F2}) / ((\text{C2} \times (\text{C2} + \text{C4} + \text{F3}) + ((\text{C2} + \text{C4} + \text{F2}) \times (\text{C3} + \text{C5})) + ((\text{C2} + \text{C4}) \times (\text{C2} + \text{C4} + \text{F3})))$
- $P(\text{Degraded}) = ((\text{C3} + \text{C5}) \times \text{F2}) / ((\text{C2} \times (\text{C2} + \text{C4} + \text{F3}) + ((\text{C2} + \text{C4} + \text{F2}) \times (\text{C3} + \text{C5})) + ((\text{C2} + \text{C4}) \times (\text{C2} + \text{C4} + \text{F3})))$

Table 14.1. Calculated Availability for an Availability-Enhanced Broker Implementation

Function	Failure Severity	MTBF (Hours)	MMTR (Seconds)		
			Active Redundancy (Hot Spare)	Passive Redundancy (Warm Spare)	Spare (Cold Spare)
Hardware	1	250,000	5	5	300
	2	50,000	30	30	30
Software	1	50,000	5	5	300
	2	10,000	30	30	30
Availability			0.9999998	0.999990	0.9994

Plugging in these values for the parameters to the equations listed above results in a table like [Table 14.1](#), which can be easily calculated using any spreadsheet tool. Such a calculation can help in the selection of tactics.

The Analytic Model Space

As we discussed in the preceding sections, there are a growing number of analytic models for some aspects of various quality attributes. One of the quests of software engineering is to have a sufficient number of analytic models for a sufficiently large number of quality attributes to enable prediction of the behavior of a designed system based on these analytic models. [Table 14.2](#) shows our current status with respect to this quest for the seven quality attributes discussed in [Chapters 5–11](#).

Table 14.2. A Summary of the Analytic Model Space

Quality Attribute	Intellectual Basis	Maturity/Gaps
Availability	Markov models; statistical models	Moderate maturity; mature in the hardware reliability domain, less mature in the software domain. Requires models that speak to state recovery and for which failure percentages can be attributed to software.
Interoperability	Conceptual framework	Low maturity; models require substantial human interpretation and input.
Modifiability	Coupling and cohesion metrics; cost models	Substantial research in academia; still requires more empirical support in real-world environments.
Performance	Queuing theory; real-time scheduling theory	High maturity; requires considerable education and training to use properly.
Security	No architectural models	
Testability	Component interaction metrics	Low maturity; little empirical validation.
Usability	No architectural models	

As the table shows, the field still has a great deal of work to do to achieve the quest for well-validated analytic models to predict behavior, but there is a great deal of activity in this area (see the “[For Further Reading](#)” section for additional papers). The remainder of this chapter deals with techniques that can be used *in addition to* analytic models.

14.2. Quality Attribute Checklists

For some quality attributes, checklists exist to enable the architect to test compliance or to guide the architect when making design decisions. Quality attribute checklists can come from industry consortia, from government organizations, or from private organizations. In large organizations they may be developed in house.

These checklists can be specific to one or more quality attributes; checklists for safety, security, and usability are common. Or they may be focused on a particular domain; there are security checklists for the financial industry, industrial process control, and the electric energy sector. They may even focus on some specific aspect of a single quality attribute: cancel for usability, as an example.

For the purposes of certification or regulation, the checklists can be used by auditors as well as by the architect. For example, two of the items on the checklist of the Payment Card Industry (PCI) are to only persist credit card numbers in an encrypted form and to never persist the security code from the back of the credit card. An auditor can ask to examine stored credit card data to determine whether it has been encrypted. The auditor can also examine the schema for data being stored to see whether the security code has been included.

This example reveals that design and analysis are often two sides of the same coin. By considering the kinds of analysis to which a system will be subjected (in this case, an audit), the architect will be led into making important early architectural decisions (making the decisions the auditors will want to find).

Security checklists usually have heavy process components. For example, a security checklist might say that there should be an explicit security policy within an organization, and a cognizant security officer to ensure compliance with the policy. They also have technical components that the architect needs to examine to determine the implications on the architecture of the system being designed or evaluated. For example, the following is an item from a security checklist generated by a group chartered by an organization of electric producers and distributors. It pertains to embedded systems delivering electricity to your home:

A designated system or systems shall daily or on request obtain current version numbers, installation date, configuration settings, patch level on all elements of a [portion of the electric distribution] system, compare these with inventory and configuration databases, and log all discrepancies.

In Search of a Grand Unified Theory for Quality Attributes

How do we create analytic models for those quality attribute aspects for which none currently exist? I do not know the answer to this question, but if we had a basis set for quality attributes, we would be in a better position to create and validate quality attribute models. By *basis set* I mean a set of orthogonal concepts that allow one to define the existing set of quality attributes. Currently there is much overlap among quality attributes; a basis set would enable discussion of tradeoffs in terms of a common set of fundamental and possibly quantifiable concepts. Once we have a basis set, we could develop analytic models for each of the elements of the set, and then an analytic model for a particular quality attribute becomes a composition of the models of the portions of the basis set that make up that quality attribute.

What are some of the elements of this basis set? Here are some of my candidates:

- *Time*. Time is the basis for performance, some aspects of availability, and some aspects of usability. Time will surely be one of the fundamental concepts for defining quality attributes.
- *Dependencies among structural elements*. Modifiability, security, availability, and performance depend in some form or another on the strength of connections among various structural elements. Coupling is a form of dependency. Attacks depend on being able to move from one compromised element to a currently uncompromised element through some dependency. Fault propagation depends on dependencies. And one of the key elements of performance analysis is the dependency of one computation on another. Enumeration of the fundamental forms of dependency and their properties will enable better understanding of many quality attributes and their interaction.

- **Access.** How does a system promote or deny access through various mechanisms? Usability is concerned with allowing smooth access for humans; security is concerned with allowing smooth access for some set of requests but denying access to another set of requests. Interoperability is concerned with establishing connections and accessing information. Race conditions, which undermine availability, come about through unmediated access to critical computations.

These are some of my candidates. I am sure there are others. The general problem is to define a set of candidates for the basis set and then show how current definitions of various quality attributes can be recast in terms of the elements of the basis set. I am convinced that this is a problem that needs to be solved prior to making substantial progress in the quest for a rich enough set of analytic models to enable prediction of system behavior across the quality attributes important for a system.

—LB

This kind of rule is intended to detect malware masquerading as legitimate components of a system. The architect will look at this item and conclude the following:

- The embedded portions of the system should be able to report their version number, installation date, configuration settings, and patch levels. One technique for doing this is to use “reflection” for each component in the system. Reflection now becomes one of the important patterns used in this system.
- Each software update or patch should maintain this information. One technique for doing this is to have automated update and patch mechanisms. The architecture could also realize this functionality through reflection.
- A system must be designated to query the embedded components and persist the information. This means
 - There must be overall inventory and configuration databases.
 - Logs of discrepancies between current values and overall inventory must be generated and sent to appropriate recipients.
 - There must be network connections to the embedded components. This affects the network topology.

The creation of quality attribute checklists is usually a time-consuming activity, undertaken by multiple individuals and typically refined and evolved over time. Domain specialists, quality attribute specialists, and architects should all contribute to the development and validation of these checklists.

The architect should treat the items on an applicable checklist as requirements, in that they need to be understood and prioritized. Under particular circumstances, an item in a checklist may not be met, but the architect should have a compelling case as to why it is not.

14.3. Thought Experiments and Back-of-the-Envelope Analysis

A thought experiment is a fancy name for the kinds of discussions that developers and architects have on a daily basis in their offices, in their meetings, over lunch, over whiteboards, in hallways, and around the coffee machine. One of the participants might draw two circles and an arrow on the whiteboard and make an assertion about the quality attribute behavior of these two circles and the arrow in a particular context; a discussion ensues. The discussion can last for a long time, especially if the two circles are augmented with a third and one more arrow, or if some of the assumptions underlying a circle or an arrow are still in flux. In this section, we describe this process somewhat more formally.

The level of formality one would use in performing a thought experiment is, as with most techniques discussed in this book, a question of context. If two people with a shared understanding of the system are performing the thought experiment for their own private purposes, then circles and lines on a whiteboard are adequate, and the discussion proceeds in a kind of shorthand. If a third person is to review the results and the third person does not share the common understanding, then sufficient details must be captured to enable the third person to understand the argument—perhaps a quick legend and a set of properties need to be added to the diagram. If the results are to be included in documentation as design rationale, then even more detail must be captured, as discussed in [Chapter 18](#). Frequently such thought experiments are accompanied by rough analyses—back-of-the-envelope analyses—based on the best data available, based on past experiences, or even based on the guesses of the architects, without too much concern for precision.

The purpose of thought experiments and back-of-the-envelope analysis is to find problems or confirmation of the nonexistence of problems in the quality attribute requirements as applied to sunny-day use cases. That is, for each use case, consider the quality attribute requirements that pertain to that use case and analyze the use case with the quality attribute requirements in mind. Models and checklists focus on one quality attribute. To consider other quality attributes, one must model or have a checklist for the second quality attribute and understand how those models interact. A thought experiment may consider several of the quality attribute requirements simultaneously; typically it will focus on just the most important ones.

The process of creating a thought experiment usually begins with listing the steps associated with carrying out the use case under consideration; perhaps a sequence diagram is employed. At each step of the sequence diagram, the (mental) question is asked: What can go wrong with this step with respect to any of the quality attribute requirements? For example, if the step involves user input, then the possibility of erroneous input must be considered. Also the user may not have been properly authenticated and, even if authenticated, may not be authorized to provide that particular input. If the step involves interaction with another system, then the possibility that the input format will change after some time must be considered. The network passing the input to a processor may fail; the processor performing the step may fail; or the computation to provide the step may fail, take too long, or be dependent on another computation that may have had problems. In addition, the architect must ask about the frequency of the input, and the anticipated distribution of requests (e.g., Are service requests regular and predictable or irregular and “bursty”?), other processes that might be competing for the same resources, and so forth. These questions go on and on.

For each possible problem with respect to a quality attribute requirement, the follow-on questions consist of things like these:

- Are there mechanisms to detect that problem?
- Are there mechanisms to prevent or avoid that problem?
- Are there mechanisms to repair or recover from that problem if it occurs?
- Is this a problem we are willing to live with?

The problems hypothesized are scrutinized in terms of a cost/benefit analysis. That is, what is the cost of preventing this problem compared to the benefits that accrue if the problem does not occur?

As you might have gathered, if the architects are being thorough and if the problems are significant (that is, they present a large risk for the system), then these discussions can continue for a long time. The discussions are a normal portion of design and analysis and will naturally occur, even if only in the mind of a single designer. On the other hand, the time spent performing a particular thought experiment should be bounded. This sounds obvious, but every grey-haired architect can tell you war stories about being stuck in endless meetings, trapped in the purgatory of “analysis paralysis.”

Analysis paralysis can be avoided with several techniques:

- “Time boxing”: setting a deadline on the length of a discussion.
- Estimating the cost if the problem occurs and not spending more than that cost in the analysis. In other words, do not spend an inordinate amount of time in discussing minor or unlikely potential problems.

Prioritizing the requirements will help both with the cost estimation and with the time estimation.

14.4. Experiments, Simulations, and Prototypes

In many environments it is virtually impossible to do a purely top-down architectural design; there are too many considerations to weigh at once and it is too hard to predict all of the relevant technological barriers. Requirements may change in dramatic ways, or a key assumption may not be met: We have seen cases where a vendor-provided API did not work as specified, or where an API exposing a critical function was simply missing.

Finding the sweet spot within the enormous architectural design space of complex systems is not feasible by reflection and mathematical analysis alone; the models either aren't precise enough to deal with all of the relevant details or are so complicated that they are impractical to analyze with tractable mathematical techniques.

The purpose of *experiments*, *simulations*, and *prototypes* is to provide alternative ways of analyzing the architecture. These techniques are invaluable in resolving tradeoffs, by helping to turn unknown architectural parameters into constants or ranges. For example, consider just a few of the questions that might occur when creating a web-conferencing system—a distributed client-server infrastructure with real-time constraints:

- Would moving to a distributed database from local flat files negatively impact feedback time (latency) for users?
- How many participants could be hosted by a single conferencing server?
- What is the correct ratio between database servers and conferencing servers?

These sorts of questions are difficult to answer analytically. The answers to these questions rely on the behavior and interaction of third-party components such as commercial databases, and on performance characteristics of software for which no standard analytical models exist. The approach used for the web-conferencing architecture was to build an extensive testing infrastructure that supported simulations, experiments, and prototypes, and use it to compare the performance of each incremental modification to the code base. This allowed the architect to determine the effect of each form of improvement before committing to including it in the final system. The infrastructure includes the following:

- A client simulator that makes it appear as though tens of thousands of clients are simultaneously interacting with a conferencing server.
- Instrumentation to measure load on the conferencing server and database server with differing numbers of clients.

The lesson from this experience is that experimentation can often be a critical precursor to making significant architectural decisions. Experimentation must be built into the development process: building experimental infrastructure can be time-consuming, possibly requiring the development of custom tools. Carrying out the experiments and analyzing their results can require significant time. These costs must be recognized in project schedules.

14.5. Analysis at Different Stages of the Life Cycle

Depending on your project's state of development, different forms of analysis are possible. Each form of analysis comes with its own costs. And there are different levels of confidence associated with each analysis technique. These are summarized in [Table 14.3](#).

Table 14.3. Forms of Analysis, Their Life-Cycle Stage, Cost, and Confidence in Their Outputs

Life-Cycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	Low–High
Requirements	Back-of-the-envelope	Low	Low–Medium
Architecture	Thought experiment	Low	Low–Medium
Architecture	Checklist	Low	Medium
Architecture	Analytic model	Low–Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium–High
Implementation	Experiment	Medium–High	Medium–High
Fielded System	Instrumentation	Medium–High	High

The table shows that analysis performed later in the life cycle yields results that merit high confidence. However, this confidence comes at a price. First, the cost of performing the analysis also tends to be higher. But the cost of changing the system to fix a problem uncovered by analysis skyrockets later in the life cycle.

Choosing an appropriate form of analysis requires a consideration of all of the factors listed in [Table 14.3](#): What life-cycle stage are you currently in? How important is the achievement of the quality attribute in question and how worried are you about being able to achieve the goals for this attribute? And finally, how much budget and schedule can you afford to allocate to this form of risk mitigation? Each of these considerations will lead you to choose one or more of the analysis techniques described in this chapter.

14.6. Summary

Analysis of an architecture enables early prediction of a system's qualities. We can analyze an architecture to see how the system or systems we build from it will perform with respect to their quality attribute goals. Some quality attributes, most notably performance and availability, have well-understood, time-tested analytic models that can be used to assist in quantitative analysis. Other quality attributes have less sophisticated models that can nevertheless help with predictive analysis.

For some quality attributes, checklists exist to enable the architect to test compliance or to guide the architect when making design decisions. Quality attribute checklists can come from industry consortia, from government organizations, or from private organizations. In large organizations they may be developed in house. The architect should treat the items on an applicable checklist as requirements, in that they need to be understood and prioritized.

Thought experiments and back-of-the-envelope analysis can often quickly help find problems or confirm the nonexistence of problems with respect to quality attribute requirements. A thought experiment may consider several of the quality attribute requirements simultaneously; typically it will focus on just the most important ones. Experiments, simulations, and prototypes allow the exploration of tradeoffs, by helping to turn unknown architectural parameters into constants or ranges whose values may be measured rather than estimated.

Depending on your project's state of development, different forms of analysis are possible. Each form of analysis comes with its own costs and its own level of confidence associated with each analysis technique.

14.7. For Further Reading

There have been many papers and books published describing how to build and analyze architectural models for quality attributes. Here are just a few examples.

Availability

Many availability models have been proposed that operate at the architecture level of analysis. Just a few of these are [\[Gokhale 05\]](#) and [\[Yacoub 02\]](#).

A discussion and comparison of different black-box and white-box models for determining software reliability can be found in [\[Chandran 10\]](#).

A book relating availability to disaster recovery and business recovery is [\[Schmidt 10\]](#).

Interoperability

An overview of interoperability activities can be found in [\[Brownsword 04\]](#).

Modifiability

Modifiability is typically measured through complexity metrics. The classic work on this topic is [\[Chidamber 94\]](#).

More recently, analyses based on design structure matrices have begun to appear [\[MacCormack 06\]](#).

Performance

Two of the classic works on software performance evaluation are [\[Smith 01\]](#) and [\[Klein 93\]](#).

A broad survey of architecture-centric performance evaluation approaches can be found in [\[Koziolek 10\]](#).

Security

Checklists for security have been generated by a variety of groups for different domains. See for example:

- Credit cards, generated by the Payment Card Industry: www.pcisecuritystandards.org/security_standards/
- Information security, generated by the National Institute of Standards and Technology (NIST): [\[NIST 09\]](#).
- Electric grid, generated by Advanced Security Acceleration Project for the Smart Grid: www.smartgridipedia.org/index.php/ASAP-SG
- Common Criteria. An international standard (ISO/IEC 15408) for computer security certification: www.commoncriteriaportal.org

Testability

Work in measuring testability from an architectural perspective includes measuring testability as the measured complexity of a class dependency graph derived from UML class diagrams, and identifying class diagrams that can lead to code that is difficult to test [\[Baudry 05\]](#); and measuring controllability and observability as a function of data flow [\[Le Traon 97\]](#).

Usability

A checklist for usability can be found at www.stcsig.org/usability/topics/articles/he-checklist.html

Safety

A checklist for safety is called the Safety Integrity Level: en.wikipedia.org/wiki/Safety_Integrity_Level

Applications of Modeling and Analysis

For a detailed discussion of a case where quality attribute modeling and analysis played a large role in determining the architecture as it evolved through a number of releases, see [\[Graham 07\]](#).

14.8. Discussion Questions

1. Build a spreadsheet for the steady-state availability equation $MTBF / (MTBF + MTTR)$. Plug in different but reasonable values for $MTBF$ and $MTTR$ for each of the active redundancy, passive redundancy, and cold spare tactics. Try values for $MTBF$ that are very large compared to $MTTR$, and also try values for $MTBF$ that are much closer in size to $MTTR$. What do these tell you about which tactics you might want to choose for availability?
2. Enumerate as many responsibilities as you can that need to be carried out for providing a “cancel” operation in a user interface. *Hint:* There are at least 21 of them, as indicated in a publication by (*strong hint!*) one of the authors of this book whose last name (*unbelievably strong hint!*) begins with “B.”
3. The M/M/1 (look it up!) queuing model has been employed in computing systems for decades. Where in your favorite computing system would this model be appropriate to use to predict latency?
4. Suppose an architect produced [Figure 14.5](#) while you were sitting watching him. Using thought experiments, how can you determine the performance and availability of this system? What assumptions are you making and what conclusions can you draw? How definite are your conclusions?

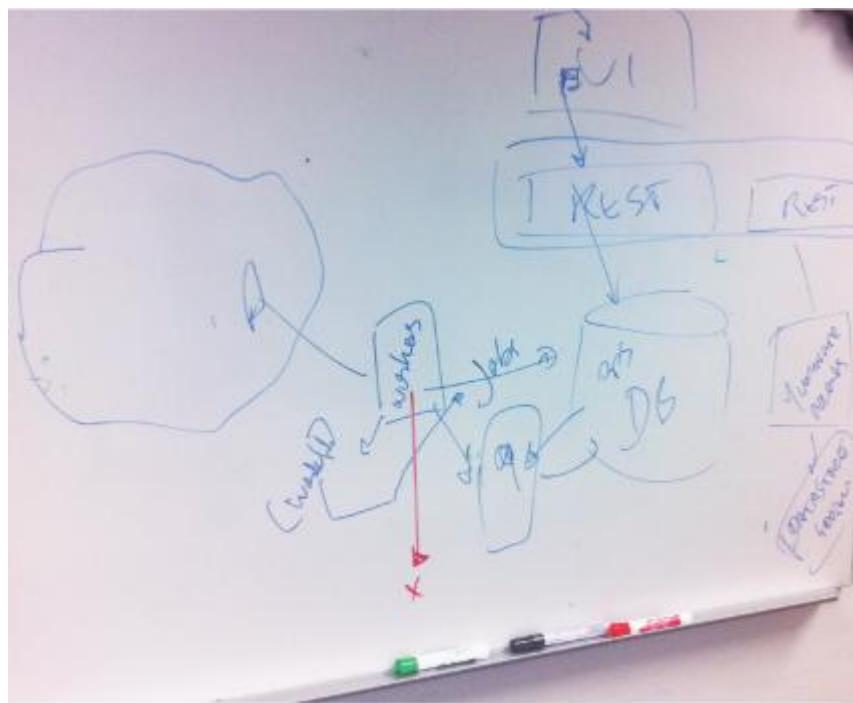


Figure 14.5. Capture of a whiteboard sketch from an architect

Part Three. Architecture in the Life Cycle

[Part I](#) of this book introduced architecture and the various contextual lenses through which it could be viewed. To recap from [Chapter 3](#), those contexts include the following:

- *Technical*. What technical role does the software architecture play in the system or systems of which it's a part? Part of the answer to this is what [Part II](#) of our book is about and the rest is included in [Part IV](#). [Part II](#) describes how decisions are made, and [Part IV](#) describes the environment that determines whether the results of the decisions satisfy the needs of the organization.
- *Project life cycle*. How does a software architecture relate to the other phases of a software development life cycle? The answer to this is what Part III of our book is about.
- *Business*. How does the presence of a software architecture affect an organization's business environment? The answer to this is what [Part IV](#) of our book is about.
- *Professional*. What is the role of a software architect in an organization or a development project? The answer to this is threaded throughout the entire book, but especially in [Chapter 24](#), where we treat the duties, skills, and knowledge of software architects.

[Part II](#) concentrated on the *technical* context of software architecture. In our philosophy, this is tantamount to understanding quality attributes. If you have a deep understanding of how architecture affects quality attributes, then you have mastered most of what you need to know about making design decisions.

Here in Part III we turn our attention to how to constructively apply that knowledge within the context of a particular software development project. Here is where software architecture meets software engineering: How do architecture concerns affect the gathering of requirements, the carrying out of design decisions, the validation and capturing of the design, and the transformation of design into implementation? In Part III, we'll find out.

A Word about Methods

Because this is a book about software architecture in *practice*, we've tried to spell out specific methods in enough detail so that you can emulate them. You'll see PALM, a method for eliciting business goals that an architecture should accommodate. You'll see Views and Beyond, an approach for documenting architecture in a set of views that serve stakeholders and their concerns. You'll see ATAM, a method for evaluating an architecture against stakeholders' ideas of what quality attributes it should provide. You'll see CBAM, a method for assessing which evolutionary path of an architecture will best serve stakeholders' needs.

All of these methods rely in some way or another on tapping stakeholders' knowledge about what an architecture under development should provide. As presented in their respective chapters, each of these methods includes a similar process of identifying the relevant stakeholders, putting them in a room together, presenting a briefing about the method that the stakeholders have been assembled to participate in, and then launching into the method.

So why is it necessary to put all of the stakeholders in the same room? The short answer is that it isn't. There are (at least) three major engagement models for conducting an architecture-focused method. Why three? Because we have identified two important factors, each of which has two values, that describe four potential engagement models for gathering information from stakeholders. These two factors are

1. Location (co-located or distributed)
2. Synchronicity (synchronous or asynchronous)

One option (co-located and asynchronous) makes no sense, and so we are left with three *viable* engagement models. The advantages and disadvantages we've observed of each engagement model follow.

Why has the big-meeting format (co-located, synchronous) tended to prevail? There are several reasons:

- It compresses the time required for the method. Time on site for remote participants is minimized, although as we will see, travel time is not considered in this argument. All of the stakeholders are available with minimal external distractions.
- It emphasizes the importance of the method. Any meeting important enough to bring multiple people together for an extended time must be judged by management to be important.
- It benefits from the helpful group mentality that emerges when people are in the same room working toward a common goal. The group mentality fosters buy-in to the architecture and buy-in to the reasons it exists. Putting stakeholders in the same room lets them open communication paths with the architect and with each other, paths that will often remain open long after the meeting has run its course. We always enjoy seeing business cards exchanged with handshakes when stakeholders meet each other for the first time. Putting the architect in a room full of stakeholders for a couple of days is a very healthy thing for any project.

Model	Advantages	Disadvantages
All stakeholders in the same room for the duration of the exercise (co-located and synchronous).	<p>All stakeholders participate equally.</p> <p>Group mentality produces buy-in for architecture and the results of the exercise.</p> <p>Enduring communication paths are opened among stakeholders.</p> <p>This option takes the shortest calendar time.</p>	<p>Scheduling can be problematic.</p> <p>Some stakeholders might not be forthcoming in a crowd.</p> <p>Stakeholders might incur substantial travel costs to attend.</p>
Some stakeholders participate in exercise remotely (distributed and synchronous).	<p>Saves travel costs for remote participants; this option might permit participation by stakeholders who otherwise would not be able to contribute.</p>	<p>Technology is a limiting factor; remote participants almost always are second-class citizens in terms of their participation and after-exercise “connection” to other participants.</p>
Facilitators interviewing stakeholders individually or in small groups (distributed and asynchronous).	<p>Allows for in-depth interaction between facilitators and stakeholders.</p> <p>Eliminates group factors that might inhibit a stakeholder from speaking in public.</p>	<p>If stakeholders are widely distributed, increased travel costs incurred by facilitator(s).</p> <p>Reduced group buy-in.</p> <p>Reduced group mentality.</p> <p>Reduced after-exercise communication among stakeholders.</p> <p>Exercise stretched out over a longer period of calendar time.</p>

But there are, as ever, tradeoffs. The big-meeting format can be costly and difficult to fit into an already crowded project schedule. Often the hardest aspect of executing any of our methods is finding two contiguous days when all the important stakeholders are available. Also, the travel costs associated with a big meeting can be substantial in a distributed organization. And some stakeholders might not be as forthcoming as we would like if they are in a room surrounded by strong-willed peers or higher-ups (although our methods use facilitation techniques to try to correct for this).

So which model is best? You already know the answer: It depends. You can see the tradeoffs among the different approaches. Pick the one that does the best job for your organization and its particularities.

Conclusion

As you read Part III and learn about architecture methods, remember that the form of the method we present is the one in which the most practical experience resides. But:

1. You can always adjust the engagement model to be something other than everybody-in-the-same-room if that will work better for you.
2. Whereas the steps of a method are nominally carried out in sequential order according to a set agenda, sometimes there must be dynamic modifications to the schedule to accommodate personnel availability or architectural information. Every situation is unique, and there may be times when you need to return briefly to an earlier step, jump forward to a later step, or iterate among steps, as the need dictates.

P.S.: We do provide one example of a shortened version of one of our methods—the ATAM. We call this *Lightweight Architecture Evaluation*, and it is described in [Chapter 21](#).

15. Architecture in Agile Projects

It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.

—Charles Darwin

Since their first appearance over a decade ago, the various flavors of Agile methods and processes have received increasing attention and adoption by the worldwide software community. New software processes do not just emerge out of thin air; they evolve in response to a palpable need. In this case, the software development world was responding to a need for projects to be more responsive to their stakeholders, to be quicker to develop functionality that users care about, to show more and earlier progress in a project's life cycle, and to be less burdened by documenting aspects of a project that would inevitably change. Is any of this inimical to the use of architecture? We emphatically say "no." In fact, the question for a software project is not "Should I do Agile or architecture?", but rather questions such as "How much architecture should I do up front versus how much should I defer until the project's requirements have solidified somewhat?", "When and how should I refactor?", and "How much of the architecture should I formally document, and when?" We believe that there are good answers to all of these questions, and that Agile and architecture are not just well suited to live together but in fact critical companions for many software projects.

The Agile software movement began to receive considerable public attention approximately a decade ago, with the release of the "Agile Manifesto." Its roots extend at least a decade earlier than that, in practices such as Extreme Programming and Scrum. The Agile Manifesto, originally signed by 17 developers, was however a brilliant public relations move; it is brief, pithy, and sensible:

Manifesto for Agile Software Development

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. [agilemanifesto.org]

The authors of the Manifesto go on to describe the twelve principles that underlie their reasoning:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.

- 8.** Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- 9.** Continuous attention to technical excellence and good design enhances agility.
- 10.** Simplicity—the art of maximizing the amount of work not done—is essential.
- 11.** The best architectures, requirements, and designs emerge from self-organizing teams.
- 12.** At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

There has been considerable elaboration of the Agile Manifesto, and Agile processes, since its first release, but the basic principles have remained solid. The Agile movement (and its predecessors) have gained considerable attention and have enjoyed widespread adoption over the past two decades. These processes were initially employed on small- to medium-sized projects with short time frames and enjoyed considerable success. They were not often used for larger projects, particularly those with distributed development. This is not surprising, given the twelve principles.

In particular principles 4 and 6 imply the need for co-location or, if co-location is not possible, then at least a high level of communication among the distributed teams. Indeed, one of the core practices of Agile projects is frequent (often daily) face-to-face meetings. Principle 11 says that, for best results, teams should be self-organizing. But self-organization is a social process that is much more cumbersome if those teams are not physically co-located. In this case we believe that the creators of the twelve Agile principles got it wrong. The best teams may be self-organizing, but the best architectures still require much more than this—technical skill, deep experience, and deep knowledge.

Principle 1 argues for “early and continuous delivery of valuable software” and principle 7 claims that “Working software is the primary measure of progress.” One might argue that a focus on early and continuous release of software, where “working” is measured in terms of customer-facing features, leaves little time for addressing the kinds of cross-cutting concerns and infrastructure critical to a high-quality large-scale system.

It has been claimed by some that there is an inherent tension between being agile and doing a conscientious job of architecting. But is there truly a tension? And if so, how do you go about characterizing it and reasoning about it? In short, how much architecture is the “right” amount of architecture?

Our brief answer, in this chapter, is that there is no tension. This issue is not “Agile versus Architecture” but rather “how best to blend Agile and Architecture.”

One more point, before we dive into the details: The Agile Manifesto is itself a compromise: a pronouncement created by a committee. The fact that architecture doesn’t clearly live anywhere within it is most likely because they had no consensus opinion on this topic and not because there is any inherent conflict.

15.1. How Much Architecture?

We often think of the early software development methods that emerged in the 1970s—such as the Waterfall method—as being plan-driven and inflexible. But this inflexibility is not for nothing. Having a strong up-front plan provides for considerable predictability (as long as the requirements don’t change too much) and makes it easier to coordinate large numbers of teams. Can you imagine a large construction or aerospace project without heavy up-front planning? Agile methods and practitioners, on the other hand, often scorn planning, preferring instead teamwork, frequent face-to-face communication, flexibility, and adaptation. This enhances invention and creativity.

Garden Shed or Skyscraper?

A few years ago I built a small shed in my back yard, for holding gardening tools, the lawn mower, the fertilizer cart, and so forth. I had a plan in my head, a small team of physically co-located “developers,” and excellent access to the customer (me) for making any last-

minute decisions and for incorporating any late-breaking feature requests. What was my architecture? For sure, nothing was written down; I had an image in my head. I went to the local big-box hardware store/lumberyard and bought a bunch of building materials, primarily wood. I already owned a fine collection of hammers, saws, and drills. The boys and I started hammering and sawing and drilling. In short order I had a garden shed which has served its purpose, with the occasional repair, for quite a few years. My process was agile: I was able to accommodate the knowledge, skills, and characteristics of my developers; we were a self-organizing team; and I was able to easily accommodate feature requests that emerged late in the process.

Would I recommend this process for the construction of a 20-story office building, or even a building-code-compliant single-family house? Of course not. All of these are built using the much-maligned BDUF (Big Design Up Front) process.

My ad hoc process for building the shed was ultimately agile, but it had little analysis or forethought. It did, however, have *just enough* forethought and planning. Doing BDUF—hiring an architect, a structural engineer, and a surveyor, and doing a detailed analysis of soil conditions, potential snow loads, and options for future modifications—would have been folly; really expensive folly!

So too with software. As with everything that we recommend in this book, the amount of up-front planning and analysis should be justified by the potential risks. In the end, everything in architecture is about cost/benefit tradeoffs.

—RK

Let us consider a specific case, to illustrate the tradeoff between up-front planning and agility: the Agile technique of employing user stories. User stories are a cornerstone of the Agile approach. Each user story describes a set of features visible to the user. Implementing user stories is a way of demonstrating progress to the customer. This can easily lead to an architecture in which every feature is independently designed and implemented. In such an environment, concerns that cut across more than one feature become hard to capture. For example, suppose there is a utility function that supports multiple features. To identify this utility function, coordination is required among the teams that develop the different features, and it also requires a role in which a broad overview across all of the features is maintained. If the development team is geographically distributed and the system being developed is a large one, then emphasis on delivering features early will cause massive coordination problems. In an architecture-centric project, a layered architecture is a way to solve this problem, with features on upper layers using shared functionality of the lower layers, but that requires up-front planning and design and feature analysis.

Successful projects clearly need a successful blend of the two approaches. For the vast majority of nontrivial projects, this is not and never should be an either/or choice. Too much up-front planning and commitment can stifle creativity and the ability to adapt to changing requirements. Too much agility can be chaos. No one would want to fly in an aircraft where the flight control software had *not* been rigorously planned and thoroughly analyzed. Similarly, no one would want to spend 18 months planning an e-commerce website for their latest cell-phone model, or video game, or lipstick (all of which are guaranteed to be badly out of fashion in 18 months). What we all want is the sweet spot—what George Fairbanks calls “just enough architecture.” This is not just a matter of doing the right amount of architecture work, but also doing it at the right time. Agile projects tend to want to evolve the architecture, as needed, in real time, whereas large software projects have traditionally favored considerable up-front analysis and planning.

An Analytic Perspective on Up-front Work vs. Agility

Boehm and Turner, analyzing historical data from 161 industrial projects, examine the effects of up-front architecture and risk resolution effort. This corresponds to the COnstructive COst MOdel II (COCOMO II) scale factor called “RESL.” There are two activities that can add time to the basic project schedule:

- Up-front design work on the architecture and up-front risk identification, planning, and resolution work

- Rework due to fixing defects and addressing modification requests.

Intuitively, these two trade off against each other: The more we invest in planning, the less (we hope) rework is needed.

So Boehm and Turner synthesized a model that allowed them to plot these two values against each other. The axes of their graph ([Figure 15.1](#)) show percent of time added for RESL and percent of time added to the schedule. The amount of architecture and risk resolution effort is plotted as the dashed line, moving up and to the right from near the origin, and ranges from 5 to 50 percent of project effort. This effort is plotted against three hypothetical projects, measured in thousands of source lines of code (KSLOC):

- One project of 10 KSLOC
- One project of 100 KSLOC
- One project of 1,000 KSLOC

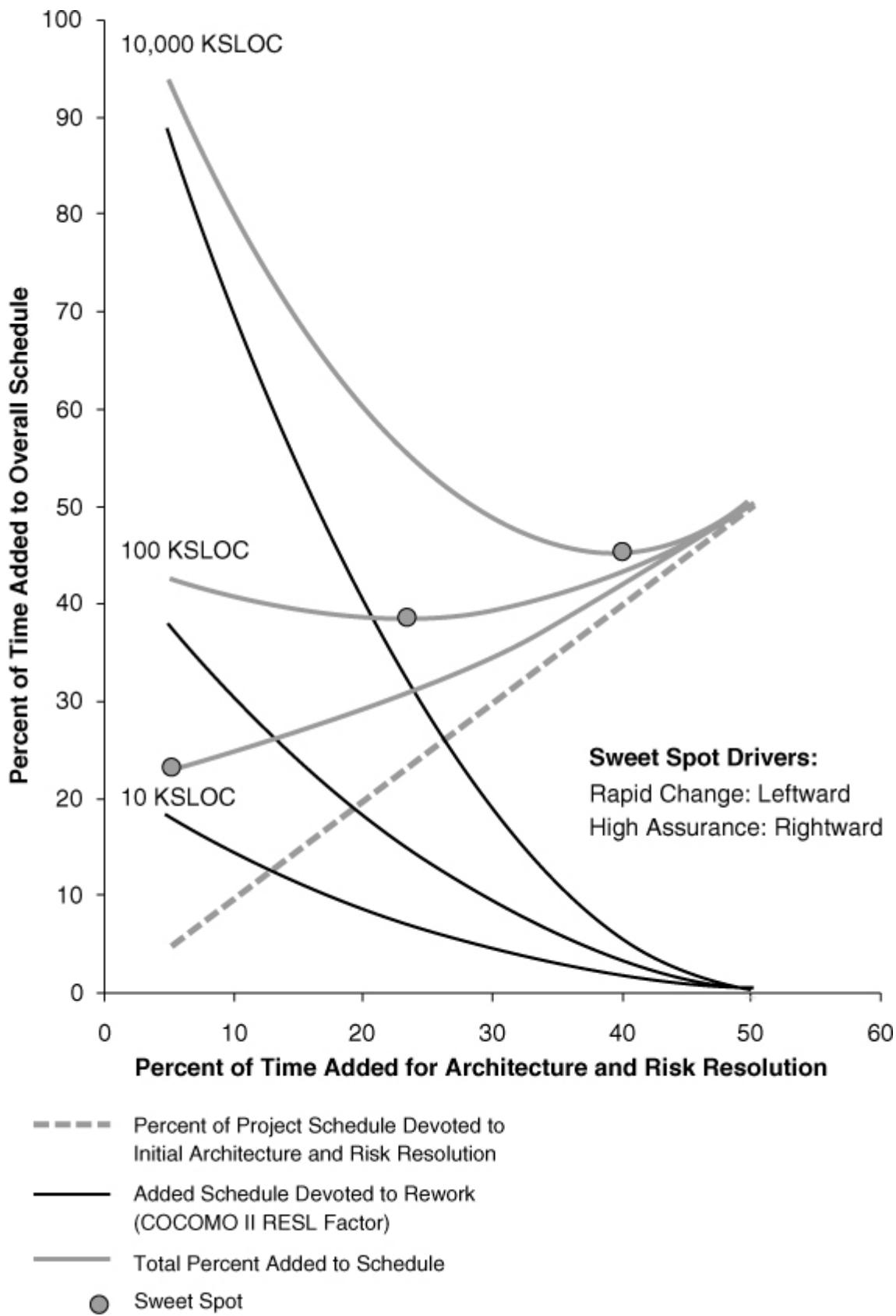


Figure 15.1. Architecture effort vs. rework

There is one line representing each of these three projects, starting near the Y axis and descending, at different rates, to the X axis at the 50 mark. This shows that adding time for up-front work reduces later rework. No surprise: that is exactly the point of doing more up-front work. However, when you sum each of those downward-trending lines (for the 10, 100, and 1,000 KSLOC projects) with the upward sloping line for the up-front (initial architecture and risk resolution) work, you get the second set of three lines, which start at the Y axis and meet the upward sloping line at the 50 mark on the X axis.

These lines show that there is a sweet spot for each project. For the 10 KSLOC project, the sweet spot is at the far left. This says that devoting much, if any, time to up-front work is a waste for a small project (assuming that the inherent domain complexity is the same for all three sets of lines). For the 100 KSLOC project, the sweet spot is at around 20 percent of the project schedule. And for the 1,000 KSLOC project, the sweet spot is at around 40 percent of the project schedule. These results are fairly intuitive. A project with a million lines of code is enormously complex, and it is difficult to imagine how Agile principles alone can cope with this complexity if there is no architecture to guide and organize the effort.

The graph shows that no one answer is appropriate for all situations, so you need methods to guide you to decide how much up-front work is right for you. Boehm and Turner's work is a start, but expected lines of code is not the only determinant for appropriateness of up-front planning. The domain, the reliability or safety required, and the experience of your development team all play a role.

The whole point of choosing how much time to budget for architecture is to reduce risk. Risk may be financial, political, operational, or reputational. Some risks might involve human life or the chance of legal action. [Chapter 22](#) covers risk management and budgets for planning in the context of architecture.

15.2. Agility and Architecture Methods

Throughout this book we emphasize methods for architecture design, analysis, and documentation. We unabashedly like methods! And so does the Agile community: dozens of books have been written on Scrum, Extreme Programming, Crystal Clear, and other Agile methods. But how should we think of architecture-centric techniques and methods in an Agile context? How well do they fit with the twelve Agile principles, for example?

We believe that they fit very well. The methods we present are based on the essential elements needed to perform the activity. If you believe that architecture needs to be designed, analyzed, and documented, then the techniques we present are essential regardless of the project in which they are embedded. The methods we present are essentially driven by the motivation to reduce risk, and by considerations of costs and benefits.

Among all of our methods—for extracting architecturally significant requirements, for architecture design, for architecture evaluation, for architecture documentation—that you'll see in subsequent chapters, one might expect the greatest Agile friction from evaluation and documentation. And so the rest of this section will examine those two practices in an Agile context.

Architecture Documentation and YAGNI

Our approach to architecture documentation is called Views and Beyond, and it will be discussed in [Chapter 18](#). Views and Beyond and Agile agree emphatically on the following point: If information isn't needed, don't spend the resources to document it. All documentation should have an intended use and audience in mind, and be produced in a way that serves both.

One of our 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. This principle is so important in Agile methods that it has been given its own name: YAGNI. YAGNI means "you ain't gonna need it," and

it refers to the idea that you should only implement or document something when you actually have the need for it. Do not spend time attempting to anticipate all possible needs.

The Views and Beyond approach uses the architectural view as the “unit” of documentation to be produced. Selecting the views to document is an example of applying this principle. The Views and Beyond approach prescribes producing a view if and only if it addresses substantial concerns of an important stakeholder community. And because documentation is not a monolithic activity that holds up all other progress until it is complete, the view selection method prescribes producing the documentation in prioritized stages to satisfy the needs of the stakeholders who need it now.

We document the portions of the architecture that we need to teach to newcomers, that embody significant potential risks if not properly managed, and that we need to change frequently. We document what we need to convey to readers so they can do their job. Although “classic” Agile emphasizes documenting the minimum amount needed to let the current team of developers make progress, our approach emphasizes that the reader might be a maintainer assigned to make a technology upgrade years after the original development team has disbanded.

Architecture Evaluation

Could an architecture evaluation work as part of an Agile process? Absolutely. In fact, doing so is perfectly Agile-consistent, because meeting stakeholders’ important concerns is a cornerstone of Agile philosophy.

Our approach to architecture evaluation is exemplified by the Architecture Tradeoff Analysis Method (ATAM) of [Chapter 21](#). It does not endeavor to analyze all, or even most, of an architecture. Rather, the focus is determined by a set of quality attribute scenarios that represent the most important (but by no means all) of the concerns of the stakeholders. “Most important” is judged by the amount of value the scenario brings to the architecture’s stakeholders, or the amount of risk present in achieving the scenario. Once these scenarios have been elicited, validated, and prioritized, they give us an evaluation agenda based on what is important to the success of the system, and what poses the greatest risk for the system’s success. Then we only delve into those areas that pose high risk for the achievement of the system’s main functions and qualities.

And as we will see in [Chapter 21](#), it is easy to tailor a lightweight architecture evaluation, for quicker and less-costly analysis and feedback whenever in the project it is called for.

15.3. A Brief Example of Agile Architecting

Our claim is that architecture and agility are quite compatible. Now we will look at a brief case study of just that. This project, which one of the authors worked on, involved the creation and evolution of a web-conferencing system. Throughout this project we practiced “agile architecting” and, we believe, hit the sweet spot between up-front planning where possible, and agility where needed.

Web-conferencing systems are complex and demanding systems. They must provide real-time responsiveness, competitive features, ease of installation and use, lightweight footprint, and much more. For example:

- They must work on a wide variety of hardware and software platforms, the details of which are not under the control of the architect.
- They must be reliable and provide low-latency response times, particularly for real-time functionality such as voice over IP (VoIP) and screen sharing.
- They must provide high security, but do so over an unknown network topology and an unknown set of firewalls and firewall policies.
- They must be easily modified and easily integrated into a wide variety of environments and applications.
- They must be highly usable and easily installed and learned by users with widely varying IT skills.

Many of the above-mentioned goals trade off against each other. Typically security (in the form of encryption) comes at the expense of real-time performance (latency). Modifiability comes at the expense of time-to-market. Availability and performance typically come at the expense of modifiability and cost.

Even if it is possible to collect, analyze, and prioritize all relevant data, functional requirements, and quality attribute requirements, the stringent time-to-market constraints that prevail in a competitive climate such as web-conferencing would have prevented us from doing this. Trying to support all possible uses is intractable, and the users themselves were poorly equipped for envisioning all possible potential uses of the system. So just asking the users what they wanted, in the fashion of a traditional requirements elicitation, was not likely to work.

This results in a classic “agility versus commitment” problem. On the one hand the architect wants to provide new capabilities quickly, and to respond to customer needs rapidly. On the other hand, long-term survival of the system and the company means that it must be designed for extensibility, modifiability, and portability. This can best be achieved by having a simple conceptual model for the architecture, based on a small number of regularly applied patterns and tactics. It was not obvious how we would “evolve” our way to such an architecture. So, how is it possible to find the “sweet spot” between these opposing forces?

The WebArrow web-conferencing system faced precisely this dilemma. It was impossible for the architect and lead designers to do purely top-down architectural design; there were too many considerations to weigh at once, and it was too hard to predict all of the relevant technological challenges. For example, they had cases where they discovered that a vendor-provided API did not work as specified—imagine that!—or that an API exposing a critical function was simply missing. In such cases, these problems rippled through the architecture, and workarounds needed to be fashioned . . . fast!

To address the complexity of this domain, the WebArrow architect and developers found that they needed to think and work in two different modes at the same time:

- Top-down—designing and analyzing architectural structures to meet the demanding quality attribute requirements and tradeoffs
- Bottom-up—analyzing a wide array of implementation-specific and environment-specific constraints and fashioning solutions to them

To compensate for the difficulty in analyzing architectural tradeoffs with any precision, the team adopted an agile architecture discipline combined with a rigorous program of experiments aimed at answering specific tradeoff questions. These experiments are what are called “spikes” in Agile terminology. And these experiments proved to be the key in resolving tradeoffs, by helping to turn unknown architectural parameters into constants or ranges. Here’s how it worked:

1. First, the WebArrow team quickly created and crudely analyzed an initial software and system architecture concept, and then they implemented and fleshed it out incrementally, starting with the most critical functionality that could be shown to a customer.
2. They adapted the architecture and refactored the design and code whenever new requirements popped up or a better understanding of the problem domain emerged.
3. Continuous experimentation, empirical evaluation, and architecture analysis were used to help determine architectural decisions as the product evolved.

For example, incremental improvement in the scalability and fault-tolerance of WebArrow was guided by significant experimentation. The sorts of questions that our experiments (spikes) were designed to answer were these:

- Would moving to a distributed database from local flat files negatively impact feedback time (latency) for users?
- What (if any) scalability improvement would result from using mod_perl versus standard Perl? How difficult would the development and quality assurance effort be to convert to mod_perl?
- How many participants could be hosted by a single meeting server?
- What was the correct ratio between database servers and meeting servers?

Questions like these are difficult to answer analytically. The answers rely on the behavior and interactions of third-party components, and on performance characteristics of software for which no standard analytic models exist. The Web-Arrow team’s approach was to build an extensive

testing infrastructure (including both simulation and instrumentation), and to use this infrastructure to compare the performance of each modification to the base system. This allowed the team to determine the effect of each proposed improvement before committing it to the final system.

The lesson here is that making architecture processes agile does not require a radical re-invention of either Agile practices or architecture methods. The Web-Arrow team's emphasis on experimentation proved the key factor; it was our way of achieving an agile form of architecture conception, implementation, and evaluation.

This approach meant that the WebArrow architecture development approach was in line with many of the twelve principles, including:

- Principle 1, providing early and continuous delivery of working software
- Principle 2, welcoming changing requirements
- Principle 3, delivering working software frequently
- Principle 8, promoting sustainable development at a constant pace
- Principle 9, giving continuous attention to technical excellence and good design

15.4. Guidelines for the Agile Architect

Barry Boehm and colleagues have developed the Incremental Commitment Model—a hybrid process model framework that attempts to find the balance between agility and commitment. This model is based upon the following six principles:

- 1.** Commitment and accountability of success-critical stakeholders
- 2.** Stakeholder “satisficing” (meeting an acceptability threshold) based on success-based negotiations and tradeoffs
- 3.** Incremental and evolutionary growth of system definition and stakeholder commitment
- 4.** Iterative system development and definition
- 5.** Interleaved system definition and development allowing early fielding of core capabilities, continual adaptation to change, and timely growth of complex systems without waiting for every requirement and subsystem to be defined
- 6.** Risk management—risk-driven anchor point milestones, which are key to synchronizing and stabilizing all of this concurrent activity

Grady Booch has also provided a set of guidelines for an agile architecture (which in turn imply some duties for the agile architect). Booch claims that all good software-intensive architectures are agile. What does he mean by this? He means that a successful architecture is resilient and loosely coupled. It is composed of a core set of well-reasoned design decisions but still contains some “wiggle room” that allows modifications to be made and refactorings to be done, without ruining the original structure.

Booch also notes that an effective agile process will allow the architecture to grow incrementally as the system is developed and matures. The key to success is to have decomposability, separation of concerns, and near-independence of the parts. (Sound familiar? These are all modifiability tactics.)

Finally, Booch notes that to be agile, the architecture should be visible and self-evident in the code; this means making the design patterns, cross-cutting concerns, and other important decisions obvious, well communicated, and defended. This may, in turn, require documentation. But whatever architectural decisions are made, the architect must make an effort to “socialize” the architecture.

Ward Cunningham has coined the term “technical debt.” Technical debt is an analogy to the normal debt that we acquire as consumers: we purchase something now and (hope to) pay for it later. In software the equivalent of “purchasing something now” is quick-and-dirty implementation. Such implementation frequently leaves technical debt that incurs penalties in the future, in terms of increased maintenance costs. When technical debt becomes unacceptably high, projects need to

pay down some of this debt, in the form of refactoring, which is a key part of every agile architecting process.

What is our advice?

1. If you are building a large and complex system with relatively stable and well-understood requirements, it is probably optimal to do a large amount of architecture work up front (see [Figure 15.1](#) for some sample values for "large").
2. On big projects with vague or unstable requirements, start by quickly designing a complete candidate architecture even if it is just a "PowerPoint architecture," even if it leaves out many details, and even if you design it in just a couple of days. Alistair Cockburn has introduced a similar idea in his Crystal Clear method, called a "walking skeleton," which is enough architecture to be able to demonstrate end-to-end functionality, linking together the major system functions. Be prepared to change and elaborate this architecture as circumstances dictate, as you perform your spikes and experiments, and as functional and quality attribute requirements emerge and solidify. This early architecture will help guide development, help with early problem understanding and analysis, help in requirements elicitation, help teams coordinate, and help in the creation of coding templates and other project standards.
3. On smaller projects with uncertain requirements, at least try to get agreement on the central patterns to be employed, but don't spend too much time on construction, documentation, or analysis up front. In [Chapter 21](#) we will show how analysis can be done in a relatively lightweight and "just-in-time" fashion.

15.5. Summary

The Agile software movement is emblemized by the Agile Manifesto and a set of principles that assign high value to close-knit teams and continuous and frequent delivery of working software. Agile processes were initially employed on small- to medium-sized projects with short time frames and enjoyed considerable success. They were not often used for larger projects, particularly those with distributed development.

Although there might appear to be an inherent tension between being agile and architecture practices of the sort prescribed in this book, the underlying philosophies are not at odds and can be married to great effect. Successful projects need a successful blend of the two approaches. Too much up-front planning and commitment can be stifling and unresponsive to customers' needs, whereas too much agility can simply result in chaos. Agile architects tend to take a middle ground, proposing an initial architecture and running with that, until its technical debt becomes too great, at which point they need to refactor.

Boehm and Turner, analyzing historical data from 161 industrial projects, examined the effects of up-front architecture and risk resolution effort. They found that projects tend to have a "sweet spot" where some up-front architecture planning pays off and is not wasteful.

Among this book's architecture methods, documentation and evaluation might seem to be where the most friction with Agile philosophies might lie. However, our approaches to these activities are risk-based and embodied in methods that help you focus effort where it will most pay off.

The WebArrow example showed how adding experimentation to the project's processes enabled it to obtain benefits from both architecture and classic Agile practices, and be responsive to ever-changing requirements and domain understanding.

15.6. For Further Reading

Agile comes in many flavors. Here are some of the better-known ones:

- Extreme Programming [[Beck 04](#)]
- Scrum [[Schwaber 04](#)]
- Feature-Driven Development [[Palmer 02](#)]
- Crystal Clear [[Cockburn 04](#)]

The journal *IEEE Software* devoted an entire special issue in 2010 to the topic of agility and architecture. The editor's introduction [[Abrahamson 10](#)] discusses many of the issues that we have raised here.

George Fairbanks in his book *Just Enough Architecture* [[Fairbanks 10](#)] provides techniques that are very compatible with Agile methods.

Barry Boehm and Richard Turner [[Boehm 04](#)] offer a data- and analysis-driven perspective on the risks and tradeoffs involved in the continuum of choices regarding agility and what they called "discipline." The choice of "agility versus discipline" in the title of the book has angered and alienated many practitioners of Agile methods, most of which are quite disciplined. While this book does not focus specifically on architecture, it does touch on the subject in many ways. This work was expanded upon in 2010, when Boehm, Lane, Koolmanojwong, and Turner described the Incremental Commitment Model and its relationship to agility and architecture [[Boehm 10](#)]. All of Boehm and colleagues' work is informed by an active attention to risk. The seminal article on software risk management [[Boehm 91](#)] was written by Barry Boehm, more than 20 years ago, and it is still relevant and compelling reading today.

Carriere, Kazman, and Ozkaya [[Carriere 10](#)] provide a way to reason about when and where in an architecture you should do refactoring—to reduce technical debt—based on an analysis of the *propagation cost* of anticipated changes.

The article by Graham, Kazman, and Walmsley [[Graham 07](#)] provides substantially more detail on the WebArrow case study of agile architecting, including a number of architectural diagrams and additional description of the experimentation performed.

Ward Cunningham first coined the term "technical debt" in 1992 [[Cunningham 92](#)]. Brown et al. [[Brown 10](#)], building in part on Cunningham's work, offer an economics-driven perspective on how to enable agility through architecture.

Robert Nord, Jim Tomayko, and Rob Wojcik [[Nord 04](#)] have analyzed the relationship between several of the Software Engineering Institute's architecture methods and Extreme Programming. Grady Booch has blogged extensively on the relationship between architecture and Agile in his blog, for example [[Booch 11](#)].

Felix Bachmann [[Bachmann 11](#)] has provided a concrete example of a lightweight version of the ATAM that fits well with Agile projects and principles.

15.7. Discussion Questions

1. How would you employ the Agile practices of pair programming, frequent team interaction, and dedicated customer involvement in a distributed development environment?
2. Suppose, as a supporter of architecture practices, you were asked to write an Architecture Manifesto that was modeled on the Agile Manifesto. What would it look like?
3. Agile projects must be budgeted and scheduled like any other. How would you do that? Does an architecture help or hinder this process?
4. What do you think are the essential skills for an architect operating in an Agile context? How do you suppose they differ for an architect working in a non-Agile project?
5. The Agile Manifesto professes to value individuals and interactions over processes and tools. Rationalize this statement in terms of the role of tools in the modern software development process: compilers, integrated development environments, debuggers, configuration managers, automatic test tools, and build and configuration tools.
6. Critique the Agile Manifesto in the context of a 200-developer, 5-million-line project with an expected lifetime of 20 years.

16. Architecture and Requirements

The two most important requirements for major success are: first, being in the right place at the right time, and second, doing something about it.

—Ray Kroc

Architectures exist to build systems that satisfy requirements. That's obvious. What may be less obvious is that, to an architect, not all requirements are created equal. Some have a much more profound effect on the architecture than others. An architecturally significant requirement (ASR) is a requirement that will have a profound effect on the architecture—that is, the architecture might well be dramatically different in the absence of such a requirement.

You cannot hope to design a successful architecture if you do not know the ASRs. ASRs often, but not always, take the form of quality attribute requirements—the performance, security, modifiability, availability, usability, and so forth, that the architecture must provide to the system. In [Chapters 5–13](#) we introduced patterns and tactics to achieve quality attributes. Each time you select a pattern or tactic to use in your architecture, you are changing the architecture as a result of the need to meet quality attribute requirements. The more difficult and important the QA requirement, the more likely it is to significantly affect the architecture, and hence to be an ASR.

Architects have to identify ASRs, usually after doing a significant bit of work to uncover candidate ASRs. Competent architects know this, and as we observe experienced architects going about their duties, we notice that the first thing they do is start talking to the important stakeholders. They're gathering the information they need to produce the architecture that will respond to the project's needs—whether or not this information has already been identified.

This chapter provides some systematic means for identifying the ASRs and other factors that will shape the architecture.

16.1. Gathering ASRs from Requirements Documents

An obvious location to look for candidate ASRs is in the requirements documents or in user stories. After all, we are looking for requirements, and requirements should be in requirements documents. Unfortunately, this is not usually the case, although as we will see, there is information in the requirements documents that can be of use.

Don't Get Your Hopes Up

Many projects don't create or maintain the kind of requirements document that professors in software engineering classes or authors of traditional software engineering books love to prescribe. Whether requirements are specified using the "MoSCoW" style (must, should, could, won't), or as a collection of "user stories," neither of these is much help in nailing down quality attributes.

Furthermore, no architect just sits and waits until the requirements are "finished" before starting work. The architect *must* begin while the requirements are still in flux. Consequently, the QA requirements are quite likely to be up in the air when the architect starts work. Even where they exist and are stable, requirements documents often fail an architect in two ways.

First, most of what is in a requirements specification does not affect the architecture. As we've seen over and over, architectures are mostly driven or "shaped" by quality attribute requirements. These determine and constrain the most important architectural decisions. And yet the vast bulk of most requirements specifications is focused on the required features and functionality of a system, which shape the architecture the least. The best software engineering practices do prescribe capturing quality attribute requirements. For example, the Software Engineering Body of Knowledge (SWEBOk) says that quality attribute requirements are like any other requirements. They must be captured if they are important, and they should be specified unambiguously and be testable.

In practice, though, we rarely see adequate capture of quality attribute requirements. How many times have you seen a requirement of the form "The system shall be modular" or "The system shall exhibit high usability" or "The system shall meet users' performance expectations"? These are not requirements, but in the best case they are invitations for the architect to begin a conversation about what the requirements in these areas really are.

Second, much of what is useful to an architect is not in even the best requirements document. Many concerns that drive an architecture do not manifest themselves at all as observables in the system being specified, and so are not the subject of requirements specifications. ASRs often derive from business goals in the development organization itself; we'll explore this in [Section 16.3](#). Developmental qualities are also out of scope; you will rarely see a requirements document that describes teaming assumptions, for example. In an acquisition context, the requirements document represents the interests of the acquirer, not that of the developer. But as we saw in [Chapter 3](#), stakeholders, the technical environment, and the organization itself all play a role in influencing architectures.

Sniffing Out ASRs from a Requirements Document

Although requirements documents won't tell an architect the whole story, they are an important source of ASRs. Of course, ASRs aren't going to be conveniently labeled as such; the architect is going to have to perform a bit of excavation and archaeology to ferret them out.

[Chapter 4](#) categorizes the design decisions that architects have to make. [Table 16.1](#) summarizes each category of architectural design decision, and it gives a list of requirements to look for that might affect that kind of decision. If a requirement affects the making of a critical architectural design decision, it is by definition an ASR.

Table 16.1. Early Design Decisions and Requirements That Can Affect Them

Design Decision Category	Look for Requirements Addressing ...
Allocation of Responsibilities	Planned evolution of responsibilities, user roles, system modes, major processing steps, commercial packages
Coordination Model	Properties of the coordination (timeliness, currency, completeness, correctness, and consistency) Names of external elements, protocols, sensors or actuators (devices), middleware, network configurations (including their security properties) Evolution requirements on the list above
Data Model	Processing steps, information flows, major domain entities, access rights, persistence, evolution requirements
Management of Resources	Time, concurrency, memory footprint, scheduling, multiple users, multiple activities, devices, energy usage, soft resources (buffers, queues, etc.) Scalability requirements on the list above
Mapping among Architectural Elements	Plans for teaming, processors, families of processors, evolution of processors, network configurations
Binding Time Decisions	Extension of or flexibility of functionality, regional distinctions, language distinctions, portability, calibrations, configurations
Choice of Technology	Named technologies, changes to technologies (planned and unplanned)

16.2. Gathering ASRs by Interviewing Stakeholders

Say your project isn't producing a comprehensive requirements document. Or it is, but it's not going to have the QAs nailed down by the time you need to start your design work. What do you do?

Architects are often called upon to help set the quality attribute requirements for a system. Projects that recognize this and encourage it are much more likely to be successful than those that don't. Relish the opportunity. Stakeholders often have no idea what QAs they want in a system, and no amount of nagging is going to suddenly instill the necessary insight. If you insist on quantitative QA requirements, you're likely to get numbers that are arbitrary, and there's a good chance that you'll find at least some of those requirements will be very difficult to satisfy.

Architects often have very good ideas about what QAs are exhibited by similar systems, and what QAs are reasonable (and reasonably straightforward) to provide. Architects can usually provide quick feedback as to which quality attributes are going to be straightforward to achieve and which are going to be problematic or even prohibitive. And architects are the only people in the room who can say, "I can actually deliver an architecture that will do better than what you had in mind—would that be useful to you?"

Interviewing the relevant stakeholders is the surest way to learn what they know and need. Once again, it behooves a project to capture this critical information in a systematic, clear, and repeatable way. Gathering this information from stakeholders can be achieved by many methods. One such method is the Quality Attribute Workshop (QAW), described in the sidebar.

The results of stakeholder interviews should include a list of architectural drivers and a set of QA scenarios that the stakeholders (as a group) prioritized. This information can be used to do the following:

- Refine system and software requirements
 - Understand and clarify the system's architectural drivers
 - Provide rationale for why the architect subsequently made certain design decisions
 - Guide the development of prototypes and simulations
 - Influence the order in which the architecture is developed
-

The Quality Attribute Workshop

The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios before the software architecture is completed. The QAW is focused on system-level concerns and specifically the role that software will play in the system. The QAW is keenly dependent on the participation of system stakeholders.¹

¹. This material was adapted from [Barbacci 03].

The QAW involves the following steps:

Step 1. QAW Presentation and Introductions. QAW facilitators describe the motivation for the QAW and explain each step of the method. Everyone introduces themselves, briefly stating their background, their role in the organization, and their relationship to the system being built.

Step 2. Business/Mission Presentation. The stakeholder representing the business concerns behind the system (typically a manager or management representative) spends about one hour presenting the system's business context, broad functional requirements, constraints, and known quality attribute requirements. The quality attributes that will be refined in later steps will be derived largely from the business/mission needs presented in this step.

Step 3. Architectural Plan Presentation. Although a detailed system or software architecture might not exist, it is possible that broad system descriptions, context drawings, or other artifacts have been created that describe some of the system's technical details. At this point in the workshop, the architect will present the system architectural plans as they stand. This lets stakeholders know the current architectural thinking, to the extent that it exists.

Step 4. Identification of Architectural Drivers. The facilitators will share their list of key architectural drivers that they assembled during steps 2 and 3, and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea is to reach a consensus on a distilled list of architectural drivers that includes overall requirements, business drivers, constraints, and quality attributes.

Step 5. Scenario Brainstorming. Each stakeholder expresses a scenario representing his or her concerns with respect to the system. Facilitators ensure that each scenario has an explicit stimulus and response. The facilitators ensure that at least one representative scenario exists for each architectural driver listed in step 4.

Step 6. Scenario Consolidation. After the scenario brainstorming, similar scenarios are consolidated where reasonable. Facilitators ask stakeholders to identify those scenarios that are very similar in content. Scenarios that are similar are merged, as long as the people who proposed them agree and feel that their scenarios will not be diluted in the process. Consolidation helps to prevent votes from being spread across several scenarios that are expressing the same concern. Consolidating almost-alike scenarios assures that the underlying concern will get all of the votes it is due.

Step 7. Scenario Prioritization. Prioritization of the scenarios is accomplished by allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios generated after consolidation. Stakeholders can allocate any number of their votes to any

scenario or combination of scenarios. The votes are counted, and the scenarios are prioritized accordingly.

Step 8. Scenario Refinement. After the prioritization, the top scenarios are refined and elaborated. Facilitators help the stakeholders put the scenarios in the six-part scenario form of source–stimulus–artifact–environment–response–response measure that we described in [Chapter 4](#). As the scenarios are refined, issues surrounding their satisfaction will emerge. These are also recorded. Step 8 lasts as long as time and resources allow.

16.3. Gathering ASRs by Understanding the Business Goals

Business goals are the *raison d'être* for building a system. No organization builds a system without a reason; rather, the organization's leaders want to further the mission and ambitions of their organization and themselves. Common business goals include making a profit, of course, but most organizations have many more concerns than simply profit, and in other organizations (e.g., nonprofits, charities, governments), profit is the farthest thing from anyone's mind.

Business goals are of interest to architects because they often are the precursor or progenitor of requirements that may or may not be captured in a requirements specification but whose achievement (or lack) signals a successful (or less than successful) architectural design. Business goals frequently lead directly to ASRs.

There are three possible relationships between business goals and an architecture:

1. *Business goals often lead to quality attribute requirements.* Or to put it another way, every quality attribute requirement—such as user-visible response time or platform flexibility or ironclad security or any of a dozen other needs—should originate from some higher purpose that can be described in terms of added value. If we ask, for example, “*Why do you want this system to have a really fast response time?*”, we might hear that this will differentiate the product from its competition and let the developing organization capture market share; or that this will make the soldier a more effective warfighter, which is the mission of the acquiring organization; or other reasons having to do with the satisfaction of some business goal.
2. *Business goals may directly affect the architecture without precipitating a quality attribute requirement at all.* In [Chapter 3](#) we told the story of the architect who designed a system without a database until the manager informed him that the database team needed work. The architecture was importantly affected without any relevant quality attribute requirement.
3. *No influence at all.* Not all business goals lead to quality attributes. For example, a business goal to “reduce cost” may be realized by lowering the facility’s thermostats in the winter or reducing employees’ salaries or pensions.

[Figure 16.1](#) illustrates the major points just described. In the figure, the arrows mean “leads to.” The solid arrows are the ones highlighting relationships of most interest to architects.

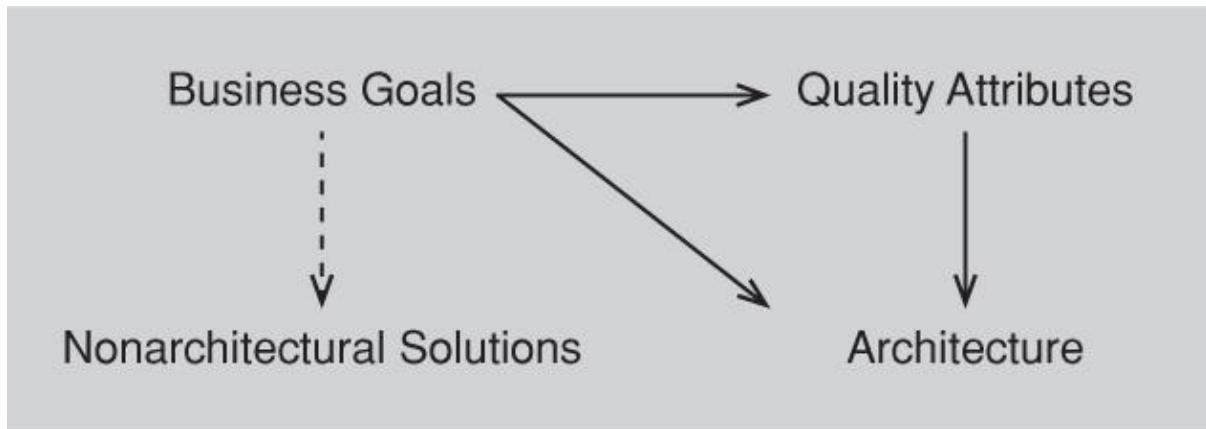


Figure 16.1. Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

Architects often become aware of an organization's business and business goals via osmosis—working, listening, talking, and soaking up the goals that are at work in an organization. Osmosis is not without its benefits, but more systematic ways are possible. We describe one such way in the sidebar "[A Method for Capturing Business Goals](#)."

A Categorization of Business Goals

Business goals are worth capturing explicitly. This is because they often imply ASRs that would otherwise go undetected until it is too late or too expensive to address them. Capturing business goals is well served by having a set of candidate business goals handy to use as conversation starters. If you know that many businesses want to gain market share, for instance, you can use that to engage the right stakeholders in your organization to ask, "What are our ambitions about market share for this product, and how could the architecture contribute to meeting them?"

Our research in business goals has led us to adopt the categories shown in [Table 16.2](#). These categories can be used as an aid to brainstorming and elicitation. By employing the list of categories, and asking the stakeholders about possible business goals in each category, some assurance of coverage is gained.

Table 16.2. A List of Standard Business Goal Categories

-
1. Contributing to the growth and continuity of the organization
 2. Meeting financial objectives
 3. Meeting personal objectives
 4. Meeting responsibility to employees
 5. Meeting responsibility to society
 6. Meeting responsibility to state
 7. Meeting responsibility to shareholders
 8. Managing market position
 9. Improving business processes
 10. Managing the quality and reputation of products
 11. Managing change in environmental factors
-

These categories are not completely orthogonal. Some business goals may fit into more than one category, and that's all right. In an elicitation method, the categories should prompt questions about the existence of organizational business goals that fall into that category. If the categories overlap, then this might cause us to ask redundant questions. This is not harmful and could well be helpful. The utility of these categories is to help identify *all* business goals, not to provide a taxonomy.

1. *Contributing to the growth and continuity of the organization.* How does the system being developed contribute to the growth and continuity of the organization? In one experience using this business goal category, the system being developed was the sole reason for the existence of the organization. If the system was not successful, the organization would cease to exist. Other topics that might come up in this category deal with market share, product lines, and international sales.
2. *Meeting financial objectives.* This category includes revenue generated or saved by the system. The system may be for sale, either in standalone form or by providing a service, in which case it generates revenue. The system may be for use in an internal process, in which case it should make those processes more effective or more efficient. Also in this category is the cost of development, deployment, and operation of the system. But this category can also include financial objectives of individuals: a manager hoping for a raise, for example, or a shareholder expecting a dividend.
3. *Meeting personal objectives.* Individuals have various goals associated with the construction of a system. They may range from "I want to enhance my reputation by the success of this system" to "I want to learn new technologies" to "I want to gain experience with a different portion of the development process than in the past." In any case, it is possible that technical decisions are influenced by personal objectives.
4. *Meeting responsibility to employees.* In this category, the employees in question are usually those employees involved in development or those involved in operation. Responsibility to employees involved in development might include ensuring that certain types of employees have a role in the development of this system, or it might include providing employees the opportunities to learn new skills. Responsibility to employees involved in operating the system might include safety, workload, or skill considerations.
5. *Meeting responsibility to society.* Some organizations see themselves as being in business to serve society. For these organizations, the system under development is helping them meet those responsibilities. But all organizations must discharge a responsibility to society by obeying relevant laws and regulations. Other topics that might come up under this category are resource usage, "green computing," ethics, safety, open source issues, security, and privacy.
6. *Meeting responsibility to state.* Government systems, almost by definition, are intended to meet responsibility to a state or country. Other topics that might come up in this category deal with export controls, regulatory conformance, or supporting government initiatives.
7. *Meeting responsibility to shareholders.* There is overlap between this category and the financial objectives category, but additional topics that might come up here are liability protection and certain types of regulatory conformance such as, in the United States, adherence to the Sarbanes-Oxley Act.
8. *Managing market position.* Topics that might come up in this category are the strategy used to increase or hold market share, various types of intellectual property protection, or the time to market.
9. *Improving business processes.* Although this category partially overlaps with meeting financial objectives, reasons other than cost reduction exist for improving business processes. It may be that improved business processes enable new markets, new products, or better customer support.
10. *Managing the quality and reputation of products.* Topics that might come up in this category include branding, recalls, types of potential users, quality of existing products, and testing support and strategies.
11. *Managing change in environmental factors.* As we said in [Chapter 3](#), the business context for a system might change. This item is intended to encourage the stakeholders to consider what might change in the business goals for a system.

Expressing Business Goals

How will you write down a business goal once you've learned it? Just as for quality attributes, a scenario makes a convenient, uniform, and clarifying way to express business goals. It helps ensure that all business goals are expressed clearly, in a consistent fashion, and contain sufficient information to enable their shared understanding by relevant stakeholders. Just as a quality attribute scenario adds precision and meaning to an otherwise vague need for, say, "modifiability," a business goal scenario will add precision and meaning to a desire to "meet financial objectives."

Our business goal scenario template has seven parts. They all relate to the system under development, the identity of which is implicit. The parts are these:

1. **Goal-source.** These are the people or written artifacts providing the goal.
2. **Goal-subject.** These are the stakeholders who own the goal and wish it to be true. Each stakeholder might be an individual or (in the case of a goal that has no one owner and has been assimilated into an organization) the organization itself. If the business goal is, for example, "Maximize dividends for the shareholders," who is it that cares about that? It is probably not the programmers or the system's end users (unless they happen to own stock). Goal-subjects can and do belong to different organizations. The developing organization, the customer organizations, subcontractors, vendors and suppliers, standards bodies, regulatory agencies, and organizations responsible for systems with which ours must interact are all potential goal-subjects.
3. **Goal-object.** These are the entities to which the goal applies. "Object" is used in the sense of the object of a verb in a sentence. All goals have goal-objects: we want something to be true about something (or someone) that (or whom) we care about. For example, for goals we would characterize as furthering one's self-interest, the goal-object can be "myself or my family." For some goals the goal-object is clearly the development organization, but for some goals the goal-object can be more refined, such as the rank-and-file employees of the organization or the shareholders of the organization. [Table 16.3](#) is a representative cross-section of goal-objects. Goal-objects in the table start small, where the goal-object is a single individual, and incrementally grow until the goal-object is society at large.

Table 16.3. Business Goals and Their Goal-Objects

Goal-Object	Business Goals That Often Have This Goal-Object	Remarks
Individual	Personal wealth, power, honor/face/reputation, game and gambling spirit, maintain or improve reputation (personal), family interests	The individual who has these goals has them for him/herself or his/her family.
System	Manage flexibility, distributed development, portability, open systems/standards, testability, product lines, integrability, interoperability, ease of installation and ease of repair, flexibility/configurability, performance, reliability/availability, ease of use, security, safety, scalability/extensibility, functionality, system constraints, internationalization, reduce time to market	These can be goals for a system being developed or acquired. The list applies to systems in general, but the quantification of any one item likely applies to a single system being developed or acquired.
Portfolio	Reduce cost of development, cost leadership, differentiation, reduce cost of retirement, smooth transition to follow-on systems, replace legacy systems, replace labor with automation, diversify operational sequence, eliminate intermediate stages, automate tracking of business events, collect/communicate/retrieve operational knowledge, improve decision making, coordinate across distance, align task and process, manage on basis of process measurements, operate effectively within the competitive environment, the technological environment, or the customer environment Create something new, provide the best quality products and services possible, be the leading innovator in the industry	These goals live on the cusp between an individual system and the entire organization. They apply either to a single system or to an organization's entire portfolio that the organization is building or acquiring to achieve its goals.
Organization's Employees	Provide high rewards and benefits to employees, create a pleasant and friendly workplace, have satisfied employees, fulfill responsibility toward employees, maintain jobs of workforce on legacy systems	Before we get to the organization as a whole, there are some goals aimed at specific subsets of the organization.
Organization's Shareholders	Maximize dividends for the shareholders	
Organization	Growth of the business, continuity of the business, maximize profits over the short run, maximize profits over the long run, survival of the organization, maximize the company's net assets and reserves, be a market leader, maximize the market share, expand or retain market share, enter new markets, maximize the company's rate of growth, keep tax payments to a minimum, increase sales growth, maintain or improve reputation, achieve business goals through financial objectives, run a stable organization	These are goals for the organization as a whole. The organization can be a development or acquisition organization, although most were undoubtedly created with the former in mind.
Nation	Patriotism, national pride, national security, national welfare	Before we get to society at large, this goal-object is specifically limited to the goal owner's own country.
Society	Run an ethical organization, responsibility toward society, be a socially responsible company, be of service to the community, operate effectively within social environment, operate effectively within legal environment	Some interpret "society" as "my society," which puts this category closer to the nation goal-object, but we are taking a broader view.

- 4. Environment.** This is the context for this goal. For example, there are social, legal, competitive, customer, and technological environments. Sometimes the political environment is key; this is as a kind of social factor. Upcoming technology may be a major factor.
- 5. Goal.** This is any business goal articulated by the goal-source.
- 6. Goal-measure.** This is a testable measurement to determine how one would know if the goal has been achieved. The goal-measure should usually include a time component, stating the time by which the goal should be achieved.
- 7. Pedigree and value.** The pedigree of the goal tells us the degree of confidence the person who stated the goal has in it, and the goal's volatility and value. The value of a goal can be expressed by how much its owner is willing to spend to achieve it or its relative importance compared to other goals. Relative importance may be given by a ranking from 1 (most important) to n (least important), or by assigning each goal a value on a fixed scale such as 1 to 10 or high-medium-low. We combine value and pedigree into one part although it certainly is possible to treat them separately. The important concern is that both are captured.

Elements 2–6 can be combined into a sentence that reads:

For the system being developed, <goal-subject> desires that <goal-object> achieve <goal> in the context of <environment> and will be satisfied if <goal-measure>.

The sentence can be augmented by the goal's source (element 1) and the goal's pedigree and value (element 7). Some sample business goal scenarios include the following:

- For MySys, the project manager has the goal that his family's stock in the company will rise by 5 percent (as a result of the success of MySys).
- For MySys, the developing organization's CEO has the goal that MySys will make it 50 percent less likely that his nation will be attacked.
- For MySys, the portfolio manager has the goal that MySys will make the portfolio 30 percent more profitable.
- For MySys, the project manager has the goal that customer satisfaction will rise by 10 percent (as a result of the increased quality of MySys).

In many contexts, the goals of different stakeholders may conflict. By identifying the stakeholder who owns the goal, the sources of conflicting goals can be identified.

A General Scenario for Business Goals

A general scenario (see [Chapter 4](#)) is a template for constructing specific or “concrete” scenarios. It uses the generic structure of a scenario to supply a list of possible values for each non-boilerplate part of a scenario. See [Table 16.4](#) for a general scenario for business goals.

Table 16.4. General Scenario Generation Table for Business Goals

2. Goal-subject	3. Goal-object	5. Goal	4. Environment	6. Goal-measure (examples, based on goal categories)	7. Value
Any stakeholder or stakeholder group identified as having a <i>legitimate</i> interest in the system	Individual System Portfolio Organization's employees Organization's shareholders Organization Nation Society	Contributing to the growth and continuity of the organization Meeting financial objectives Meeting personal objectives Meeting responsibility to employees Meeting responsibility to society Meeting responsibility to state Meeting responsibility to shareholders Managing market position Improving business processes Managing quality and reputation of products Managing change in environmental factors	Social (includes political) Legal Competitive Customer Technological	Time that business remains viable Financial performance vs. objectives Promotion or raise achieved in period Employee satisfaction; turnover rate Contribution to trade deficit/surplus Stock price, dividends Market share Time to carry out a business process Quality measures of products Technology-related problems Time window for achievement	1–n 1–10 H–M–L Resources willing to expend
... has the goal that achieves in the context of and will be satisfied if ...		

For each of these scenarios you might want to additionally capture its source (e.g., Did this come directly from the goal-subject, a document, a third party, a legal requirement?), its volatility, and its importance.

Capturing Business Goals

Business goals are worth capturing because they can hold the key to discovering ASRs that emerge in no other context. One method for eliciting and documenting business goals is the Pedigree Attribute eLicitation Method, or PALM. The word “pedigree” means that the business goal has a clear derivation or background. PALM uses the standard list of business goals and the business goal scenario format we described earlier.

PALM can be used to sniff out missing requirements early in the life cycle. For example, having stakeholders subscribe to the business goal of improving the quality and reputation of their products may very well lead to (for example) security, availability, and performance requirements that otherwise might not have been considered.

PALM can also be used to discover and carry along additional information about existing requirements. For example, a business goal might be to produce a product that outcompetes a rival's market entry. This might precipitate a performance requirement for, say, half-second turnaround when the rival features one-second turnaround. But if the competitor releases a new product with half-second turnaround, then what does our requirement become? A conventional requirements document will continue to carry the half-second requirement, but the goal-savvy architect will know that the real requirement is to beat the competitor, which may mean even faster performance is needed.

Finally, PALM can be used to examine particularly difficult quality attribute requirements to see if they can be relaxed. We know of more than one system where a quality attribute requirement proved quite expensive to provide, and only after great effort, money, and time were expended trying to meet it was it revealed that the requirement had no actual basis other than being someone's best guess or fond wish at the time.

16.4. Capturing ASRs in a Utility Tree

As we have seen, ASRs can be extracted from a requirements document, captured from stakeholders in a workshop such as a QAW, or derived from business goals. It is helpful to record

them in one place so that the list can be reviewed, referenced, used to justify design decisions, and revisited over time or in the case of major system changes.

To recap, an ASR must have the following characteristics:

- *A profound impact on the architecture.* Including this requirement will very likely result in a different architecture than if it were not included.
- *A high business or mission value.* If the architecture is going to satisfy this requirement—potentially at the expense of not satisfying others—it must be of high value to important stakeholders.

Using a single list can also help evaluate each potential ASR against these criteria, and to make sure that no architectural drivers, stakeholder classes, or business goals are lacking ASRs that express their needs.

A Method for Capturing Business Goals

PALM is a seven-step method, nominally carried out over a day and a half in a workshop attended by architects and stakeholders who can speak to the business goals of the organizations involved. The steps are these:

1. *PALM overview presentation.* Overview of PALM, the problem it solves, its steps, and its expected outcomes.
 2. *Business drivers presentation.* Briefing of business drivers by project management. What are the goals of the customer organization for this system? What are the goals of the development organization? This is normally a lengthy discussion that allows participants to ask questions about the business goals as presented by project management.
 3. *Architecture drivers presentation.* Briefing by the architect on the driving business and quality attribute requirements: the ASRs.
 4. *Business goals elicitation.* Using the standard business goal categories to guide discussion, we capture the set of important business goals for this system. Business goals are elaborated and expressed as scenarios. We consolidate almost-alike business goals to eliminate duplication. Participants then prioritize the resulting set to identify the most important goals.
 5. *Identification of potential quality attributes from business goals.* For each important business goal scenario, participants describe a quality attribute that (if architected into the system) would help achieve it. If the QA is not already a requirement, this is recorded as a finding.
 6. *Assignment of pedigree to existing quality attribute drivers.* For each architectural driver named in step 3, we identify which business goals it is there to support. If none, that's recorded as a finding. Otherwise, we establish its pedigree by asking for the source of the quantitative part. For example: Why is there a 40-millisecond performance requirement? Why not 60 milliseconds? Or 80 milliseconds?
 7. *Exercise conclusion.* Review of results, next steps, and participant feedback.
-

Architects can use a construct called a *utility tree* for all of these purposes. A utility tree begins with the word “utility” as the root node. Utility is an expression of the overall “goodness” of the system. We then elaborate this root node by listing the major quality attributes that the system is required to exhibit. (We said in [Chapter 4](#) that quality attribute names by themselves were not very useful. Never fear: we are using them only as placeholders for subsequent elaboration and refinement!)

Under each quality attribute, record a specific refinement of that QA. For example, performance might be decomposed into “data latency” and “transaction throughput.” Or it might be decomposed into “user wait time” and “time to refresh web page.” The refinements that you choose should be the ones that are relevant to your system. Under each refinement, record the appropriate ASRs (usually expressed as QA scenarios).

Some ASRs might express more than one quality attribute and so might appear in more than one place in the tree. That is not necessarily a problem, but it could be an indication that the ASR

tries to cover too much diverse territory. Such ASRs may be split into constituents that each attach to smaller concerns.

Once the ASRs are recorded and placed in the tree, you can now evaluate them against the two criteria we listed above: the business value of the candidate ASR and the architectural impact of including it. You can use any scale you like, but we find that a simple "H" (high), "M" (medium), and "L" (low) suffice for each criterion.

For business value, High designates a must-have requirement, Medium is for a requirement that is important but would not lead to project failure were it omitted. Low describes a nice requirement to have but not something worth much effort.

For architectural impact, High means that meeting this ASR will profoundly affect the architecture. Medium means that meeting this ASR will somewhat affect the architecture. Low means that meeting this candidate ASR will have little effect on the architecture.

[Table 16.5](#) shows a portion of a sample utility tree drawn from a health care application called Nightingale. Each ASR is labeled with a pair of "H," "M," and "L" values indicating (a) the ASR's business value and (b) its effect on the architecture.

Table 16.5. Tabular Form of the Utility Tree for the Nightingale ATAM Exercise

Quality Attribute	Attribute Refinement	ASR
Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 second. (H,M)
	Throughput	A user updates a patient's account in response to a change-of-address notification while the system is under double the peak load, and the transaction completes in less than 4 seconds. (L,M)
Usability	Proficiency training	At peak load, the system is able to complete 150 normalized transactions per second. (M,M)
	Normal operations	A new hire with two or more years' experience in the business becomes proficient in Nightingale's core functions in less than 1 week. (M,L)
Configurability	User-defined changes	A user in a particular context asks for help, and the system provides help for that context, within 3 seconds. (H,M)
		A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process without the system introducing delays. (M,M)
		A hospital increases the fee for a particular service. The configuration team makes the change in 1 working day; no source code needs to change. (H,L)

Maintainability	Routine changes	A maintainer encounters search- and response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H,M)
	Upgrades to commercial components	A reporting requirement requires a change to the report-generating metadata. Change is made in 4 person-hours of effort. (M,L)
Extensibility	Adding new product	The database vendor releases a new version that must be installed in less than 3 person-weeks. (H,M)
Security	Confidentiality	A physical therapist is allowed to see that part of a patient's record dealing with orthopedic treatment but not other parts nor any financial information. (H,M)
	Integrity	The system resists unauthorized intrusion and reports the intrusion attempt to authorities within 90 seconds. (H,M)
Availability	No downtime	The database vendor releases new software, which is hot-swapped into place, with no downtime. (H,L)
		The system supports 24/7 web-based account access by patients. (L,L)

Once you have a utility tree filled out, you can use it to make important checks. For instance:

- A QA or QA refinement without any ASR is not necessarily an error or omission that needs to be rectified, but it is an indication that attention should be paid to finding out for sure if there are unrecorded ASRs in that area.
- ASRs that rate a (H,H) rating are obviously the ones that deserve the most attention from you; these are the most significant of the significant requirements. A very large number of these might be a cause for concern about whether the system is achievable.
- Stakeholders can review the utility tree to make sure their concerns are addressed. (An alternative to the organization we have described here is to use stakeholder roles rather than quality attributes as the organizing rule under "Utility.")

16.5. Tying the Methods Together

How should you employ requirements documents, stakeholder interviews, Quality Attribute Workshops, PALM, and utility trees in concert with each other?

As for most complex questions, the answer to this one is "It depends." If you have a requirements process that gathers, identifies, and prioritizes ASRs, then use that and consider yourself lucky.

If you feel your requirements fall short of this ideal state, then you can bring to bear one or more of the other approaches. For example, if nobody has captured the business goals behind the system you're building, then a PALM exercise would be a good way to ensure that those goals are represented in the system's ASRs.

If you feel that important stakeholders have been overlooked in the requirements-gathering process, then it will probably behoove you to capture their concerns through interviews. A Quality Attribute Workshop is a structured method to do that and capture their input.

Building a utility tree is a good way to capture ASRs along with their prioritization—something that many requirements processes overlook.

Finally, you can blend all the methods together: PALM makes an excellent “subroutine call” from a Quality Attribute Workshop for the step that asks about business goals, and a quality attribute utility tree makes an excellent repository for the scenarios that are the workshop’s output.

It is unlikely, however, that your project will have the time and resources to support this do-it-all approach. Better to pick the approach that fills in the biggest gap in your existing requirements: stakeholder representation, business goal manifestation, or ASR prioritization.

16.6. Summary

Architectures are driven by architecturally significant requirements: requirements that will have profound effects on the architecture. Architecturally significant requirements may be captured from requirements documents, by interviewing stakeholders, or by conducting a Quality Attribute Workshop.

In gathering these requirements, we should be mindful of the business goals of the organization. Business goals can be expressed in a common, structured form and represented as scenarios. Business goals may be elicited and documented using a structured facilitation method called PALM.

A useful representation of quality attribute requirements is in a utility tree. The utility tree helps to capture these requirements in a structured form, starting from coarse, abstract notions of quality attributes and gradually refining them to the point where they are captured as scenarios. These scenarios are then prioritized, and this prioritized set defines your “marching orders” as an architect.

16.7. For Further Reading

PALM can be used to capture the business goals that conform to a business goal viewpoint; that is, you can use PALM to populate a business goal view of your system, using the terminology of ISO Standard 42010. We discuss this in *The Business Goals Viewpoint*[\[Clements 10c\]](#). Complete details of PALM can be found in CMU/SEI-2010-TN-018, *Relating Business Goals to Architecturally Significant Requirements for Software Systems*[\[Clements 10b\]](#).

The Open Group Architecture Framework, available at www.opengroup.org/togaf/, provides a very complete template for documenting a business scenario that contains a wealth of useful information. Although we believe architects can make use of a lighter-weight means to capture a business goal, it’s worth a look.

The definitive reference source for the Quality Attribute Workshop is [\[Barbacci 03\]](#).

The term *architecturally significant requirement* was created by the Software Architecture Review and Assessment (SARA) group[\[Obbink 02\]](#).

When dealing with systems of systems (SoS), the interaction and handoff between the systems can be a source of problems. The Mission Thread Workshop and Business Thread Workshop focus on a single thread of activity within the overall SoS context and identify potential problems having to do with the interaction of the disparate systems. Descriptions of these workshops can be found at [\[Klein 10\]](#) and [\[Gagliardi 09\]](#).

16.8. Discussion Questions

1. Interview representative stakeholders for your business's or university's expense recovery system. Capture the business goals that are motivating the system. Use the seven-part business goal scenario outline given in [Section 16.3](#).
2. Draw a relation between the business goals you uncovered for the previous question and ASRs.
3. Consider an automated teller machine (ATM) system. Attempt to apply the 11 categories of business goals to that system and infer what goals might have been held by various stakeholders involved in its development.
4. Create a utility tree for the ATM system above. (Interview some of your friends and colleagues if you like, to have them contribute quality attribute considerations and scenarios.) Consider a minimum of four different quality attributes. Ensure that the scenarios that you create at the leaf nodes have explicit responses and response measures.
5. Restructure the utility tree given in [Section 16.4](#) using stakeholder roles as the organizing principle. What are the benefits and drawbacks of the two representations?
6. Find a software requirements specification that you consider to be of high quality. Using colored pens (real ones if the document is printed, virtual ones if the document is online), color red all the material that you find completely irrelevant to a software architecture for that system. Color yellow all of the material that you think *might* be relevant, but not without further discussion and elaboration. Color green all of the material that you are certain is architecturally significant. When you're done, every part of the document that's not white space should be red, yellow, or green. Approximately what percentage of each color did your document end up being? Do the results surprise you?

17. Designing an Architecture

In most people's vocabularies, design means veneer. It's interior decorating. It's the fabric of the curtains or the sofa. But to me, nothing could be further from the meaning of design. Design is the fundamental soul of a human-made creation that ends up expressing itself in successive outer layers of the product or service.

—Steve Jobs

We have discussed the building blocks for designing a software architecture, which principally are locating architecturally significant requirements; capturing quality attribute requirements; and choosing, generating, tailoring, and analyzing design decisions for achieving those requirements. All that's missing is a way to pull the pieces together. The purpose of this chapter is to provide that way.

We begin by describing our strategy for designing an architecture and then present a packaging of these ideas into a method: the Attribute-Driven Design method.

17.1. Design Strategy

We present three ideas that are key to architecture design methods: decomposition, designing to architecturally significant requirements, and generate and test.

Decomposition

Architecture determines the quality attributes of a system. Hopefully, we have convinced you of that by now. The quality attributes are properties of the system as a whole. Latency, for example, is the time between the arrival of an event and the output of the processing of that event. Availability refers to the system providing services, and so forth.

Given the fact that quality attributes refer to the system as a whole, if we wish to design to achieve quality attribute requirements, we must begin with the system as a whole. As the design is decomposed, the quality attribute requirements can also be decomposed and assigned to the elements of the decomposition.

A decomposition strategy does not mean that we are assuming the design is a green-field design or that there are no constraints on the design to use particular preexisting components either externally developed or legacy. Just as when you choose a route from one point to another, you may choose to stop at various destinations along the route, constraints on the design can be accommodated by a decomposition strategy. You as the designer must keep in mind the constraints given to you and arrange the decomposition so that it will accommodate those constraints. In some contexts, the system may end up being constructed mostly from preexisting components; in others, the preexisting components may be a smaller portion of the overall system. In either case, the goal of the design activity is to generate a design that accommodates the constraints and achieves the quality and business goals for the system.

We have already talked about module decomposition, but there are other kinds of decompositions that one regularly finds in an architecture, such as the decomposition of a component in a components-and-connectors (C&C) pattern into its subcomponents. For example, a user interface implemented using the model-view-controller (MVC) pattern would be decomposed into a number of components for the model, one or more views, and one or more controllers.

Designing to Architecturally Significant Requirements

In [Chapter 16](#), we discussed architecturally significant requirements (ASRs) and gave a technique for collecting and structuring them. These are the requirements that drive the architectural design; that is why they are significant. Driving the design means that these requirements have a profound effect on the architecture. In other words, you must design to satisfy these requirements. This raises two questions: What happens to the other requirements? and Do I design for one ASR at a time or all at once?

1. *What about the non-ASR requirements?* The choice of ASRs implies a prioritization of the requirements. Once you have produced a design that satisfies the ASRs, you know that you are in good shape. However, in the real world, there are other requirements that, while not ASRs, you would like to be able to satisfy. You have three options with respect to meeting these other requirements: (a) You can still meet the other requirements. (b) You can meet the other requirements with a slight adjustment of the existing design, and this slight adjustment does not keep the higher priority requirements from being met. (c) You cannot meet the other requirements under the current design. In case (a) or (b), there is nothing more to be done. You are happy. In case (c), you now have three options: (i) If you are close to meeting the requirement, you can see if the requirement can be relaxed. (ii) You can reprioritize the requirements and revisit the design. (iii) You can report that you cannot meet the requirement. All of these latter three options involve adjusting either the requirement or its priority. Doing so may have a business impact, and it should be reported up the management chain.
2. *Design for all of the ASRs or one at a time?* The answer to this question is a matter of experience. When you learn chess, you begin by learning that the horse goes up two and over one. After you have been playing for a while, you internalize the moves of the knight and you can begin to look further ahead. The best players may look ahead a

dozen or more moves. This situation applies to when you are designing to satisfy ASRs. Left to their own devices, novice architects will likely focus on one ASR at a time. But you can do better than that. Eventually, through experience and education, you will develop an intuition for designing, and you will employ patterns to aid you in designing for multiple ASRs.

Generate and Test

One way of viewing design is as a process of “generate and test.” This generate-and-test approach views a particular design as a hypothesis: namely, the design satisfies the requirements. Testing is the process of determining whether the design hypothesis is correct. If it is not, then another design hypothesis must be generated. [Figure 17.1](#) shows this iteration.

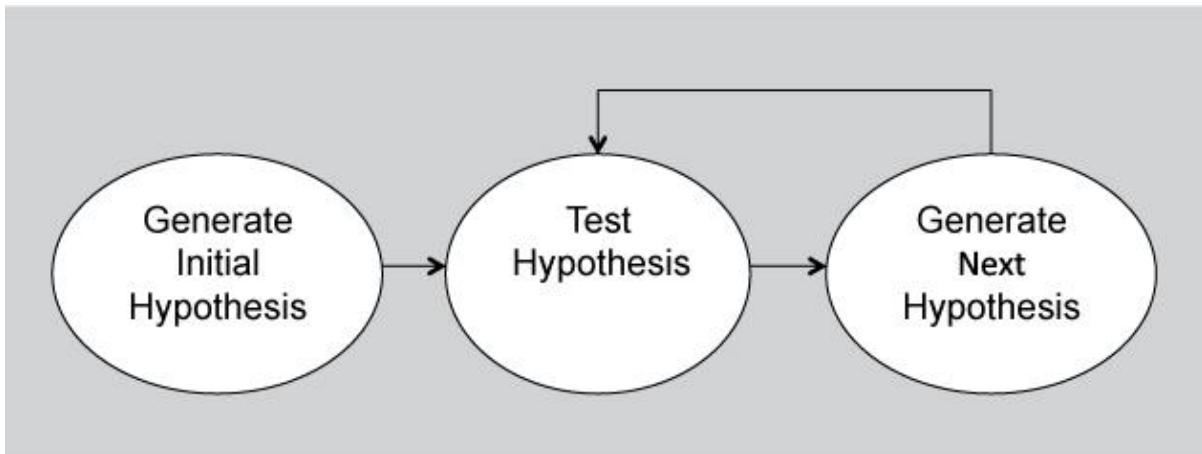


Figure 17.1. The generate-and-test process of architecture design

For this process to be effective, the generation of the next design hypothesis must build on the results of the tests. That is, the things wrong with the current design hypothesis are fixed in the next design hypothesis, and the things that are right are kept. If there is no coupling between the testing and the generation of the next design hypothesis, then this process becomes “guess and test” and that is not effective.

Generate and test as a design strategy leads to the following questions:

1. Where does the initial hypothesis come from?
2. What are the tests that are applied?
3. How is the next hypothesis generated?
4. When are you done?

We have already seen many of the elements of the answers to these questions. But now we can think about them and organize them more systematically.

Creating the Initial Hypothesis

Design solutions are created using “collateral” that is available to the project. Collateral can include existing systems, frameworks available to the project, known architecture patterns, design checklists, or a domain decomposition.

- *Existing systems.* Very few systems are completely unprecedented, even within a single organization. Organizations are in a particular business, their business leads to specialization, and specialization leads to the development of variations on a theme. It is likely that systems already exist that are similar to the system being constructed in your company.

Existing systems are likely to provide the most powerful collateral, because the business context and requirements for the existing system are likely to be similar to the business context and requirements for the new system, and many of the problems that occur have already been solved in the existing design.

A common special case is when the existing system you’re drawing on for knowledge is the same one that you’re building. This occurs when you’re evolving a system, not building one from scratch. The existing design serves as the initial design hypothesis. The “test” part of this process will reveal the parts that don’t work under the current (presumably changed) set of requirements and will therefore pinpoint the parts of the system’s design that need to change.

Another special case is when you have to combine existing legacy systems into a single system. In this case, the collection of legacy systems can be mined to determine the initial design hypothesis.

- *Frameworks*. A framework is a partial design (accompanied by code) that provides services that are common in particular domains. Frameworks exist in a great many domains, ranging from web applications to middleware systems to decision support systems. The design of the framework (especially the architectural assumptions it makes) provides the initial design hypothesis. For example, a design framework might constrain all communication to be via a broker, or via a publish-subscribe bus, or via callbacks. In each case this design framework has constrained your initial design hypothesis.
- *Patterns and tactics*. As we discussed in [Chapter 13](#), a pattern is a known solution to a common problem in a given context. Cataloged architectural patterns, possibly augmented with tactics, should be considered as candidates for the design hypothesis you’re building.
- *Domain decomposition*. Another option for the initial design hypothesis comes from performing a domain decomposition. For example, most object-oriented analysis and design processes begin this way, identifying actors and entities in the domain. This decomposition will divide the responsibilities to make certain modifications easier, but by itself it does not speak to many other quality attribute requirements.
- *Design checklists*. The design checklists that we presented in [Chapters 5–11](#) can guide an architect to making quality-attribute-targeted design choices. The point of using a checklist is to ensure completeness: Have I thought about all of the issues that might arise with respect to the many quality attribute concerns that I have? The checklist will provide guidance and confidence to an architect.

Choosing the Tests

Three sources provide the tests to be applied to the hypothesis:

1. The analysis techniques described in [Chapter 14](#).
2. The design checklists for the quality attributes that we presented in [Chapters 5–11](#) can also be used to test the design decisions already made, from the sources listed above. For the important quality attribute requirements, use the design checklists to assess whether the decisions you’ve made so far are sound and complete. For example, if testability is important for your system, the checklist says to ensure that the coordination model supports capturing the activity that led to a fault.
3. The architecturally significant requirements. If the hypothesis does not provide a solution for the ASRs, then it must be improved.

Generating the Next Hypothesis

After applying the tests, you might be done—everything looks good. On the other hand, you might still have some concerns; specifically, you might have a list of quality attribute problems associated with your analysis of the current hypothesis. This is the problem that tactics are intended to solve: to improve a design with respect to a particular quality attribute. Use the sets of tactics described in each of [Chapters 5–11](#) to help you to choose the ones that will improve your design so that you can satisfy these outstanding quality attribute requirements.

Terminating the Process

You are done with the generate-and-test process when you either have a design that satisfies the ASRs or when you exhaust your budget for producing the design. In [Chapter 22](#), we discuss how much time should be budgeted for producing the architecture.

If you do not produce such a design within budget, then you have two options depending on the set of ASRs that are satisfied. Your first option is to proceed to implementation with the best hypothesis you were able to produce, with the realization that some ASRs may not be met and may need to be relaxed or eliminated. This is the most common case. Your second option is to argue for more budget for design and analysis, potentially revisiting some of the major early design decisions and resuming generate and test from that point. If all else fails, you could suggest that the project be terminated. If all of the ASRs are critical and you were not able to produce an acceptable or nearly acceptable design, then the system you produce from the design will not be satisfactory and there is no sense in producing it.

17.2. The Attribute-Driven Design Method

The Attribute-Driven Design (ADD) method is a packaging of the strategies that we have just discussed. ADD is an iterative method that, at each iteration, helps the architect to do the following:

- Choose a part of the system to design.
- Marshal all the architecturally significant requirements for that part.
- Create and test a design for that part.

The output of ADD is not an architecture complete in every detail, but an architecture in which the main design approaches have been selected and vetted. It produces a “workable” architecture early and quickly, one that can be given to other project teams so they can begin their work while the architect or architecture team continues to elaborate and refine.

Inputs to ADD

Before beginning a design process, the requirements—functional, quality, and constraints—should be known. In reality, waiting for all of the requirements to be known means the project will never be finished, because requirements are continually arriving to a project as a result of increased knowledge on the part of the stakeholders and changes in the environment (technical, social, legal, financial, or political) over time. ADD can begin when a set of architecturally significant requirements is known.

This increases the importance of having the correct set of ASRs. If the set of ASRs changes after design has begun, then the design may well need to be reworked (a truth under any design method, not just ADD). To the extent that you have any influence over the requirements-gathering process, it would behoove you to lobby for collection of ASRs first. Although these can't all be known *a priori*, as we saw in [Chapter 16](#), quality attribute requirements are a good start.

In addition to the ASRs, input to ADD should include a context description. The context description gives you two vital pieces of information as a designer:

1. *What are the boundaries of the system being designed?* What is inside the system and what is outside the system must be known in order to constrain the problem and establish the scope of the architecture you are designing. The system's scope is unknown or unclear surprisingly often, and it will help the architecture to nail down the scope as soon as you can.
2. *What are the external systems, devices, users, and environmental conditions with which the system being designed must interact?* By “environmental conditions” here we are referring to the system's runtime environment. The system's environmental conditions are an enumeration of factors such as where the input comes from, where the output goes, what forms they take, what quality attributes they have, and what forces may affect the operation of the system. It is possible that not all of the external systems are known at design time. In this case, the system must have some discovery mechanisms, but the context description should enumerate the assumptions that can be made about the external systems even if their specifics are not yet known. An example of accommodating environment conditions can be seen in a system that must be sent into space. In addition to handling its inputs, outputs, and quality attributes, such a system must accommodate failures caused by stray gamma rays, certainly a force affecting the operation of the system.

Output of ADD

The output of ADD is a set of sketches of architectural views. The views together will identify a collection of architectural elements and their relationships or interactions. One of the views produced will be a module decomposition view, and in that view each element will have an enumeration of its responsibilities listed.

Other views will be produced according to the design solutions chosen along the way. For example, if at one point in executing the method, you choose the service-oriented architecture (SOA) pattern for part of the system, then you will capture this in an SOA view (whose scope is that part of the system to which you applied the pattern).

The interactions of the elements are described in terms of the information being passed between the elements. For example, we might specify protocol names, synchronous, asynchronous, level of encryption, and so forth.

The reason we refer to “sketches” above is that ADD does not take the design so far as to include full-blown interface specifications, or even so far as choosing the names and parameter types of interface programs (methods). That can come later. ADD does identify the information that passes through the interfaces and important characteristics of the information. If any aspects of an interface have quality attribute implications, those are captured as annotations.

When the method reaches the end, you will have a full-fledged architecture that is roughly documented as a set of views. You can then polish this collection, perhaps merging some of the views as appropriate, to the extent required by your project. In an Agile project, this set of rough sketches may be all you need for quite a while, or for the life of the project.

17.3. The Steps of ADD

ADD is a five-step method:

1. Choose an element of the system to design.
2. Identify the ASRs for the chosen element.
3. Generate a design solution for the chosen element.
4. Inventory remaining requirements and select the input for the next iteration.
5. Repeat steps 1–4 until all the ASRs have been satisfied.

Step 1: Choose an Element of the System to Design

ADD works by beginning with a part of the system that has not yet been designed, and designing it. In this section, we’ll discuss how to make that choice.

For green-field designs, the “element” to begin with is simply the entire system. The first trip through the ADD steps will yield a broad, shallow design that will produce a set of newly identified architectural elements and their interactions. These elements will almost certainly require more design decisions to flesh out what they do and how they satisfy the ASRs allocated to them; during the next iteration of ADD, those elements become candidates for the “choose an element” step.

So, nominally, the first iteration of ADD will create a collection of elements that together constitute the entire system. The second iteration will take one of these elements—what we call the “chosen element”—and design it, resulting in still finer-grained elements. The third iteration will take another element—either one of the children of the whole system or one of the children that was created from the design of one of the children of the whole system—and so forth. For example, if you choose an SOA pattern in the first iteration, you might choose child elements such as service clients, service providers, and the SOA infrastructure components. In the next iteration through the loop, you would refine one of these child elements, perhaps the infrastructure components. In the next iteration you now have a choice: refine another child of the SOA pattern, such as a service provider, or refine one of the child elements of the infrastructure components. [Figure 17.2](#) shows these choices as a decomposition tree, annotated with the ADD iteration that applies to each node. (The example components are loosely based on the Adventure Builder system, introduced in [Chapter 13](#).) [Figure 17.2](#) is a decomposition view of our hypothetical system after two iterations of ADD.

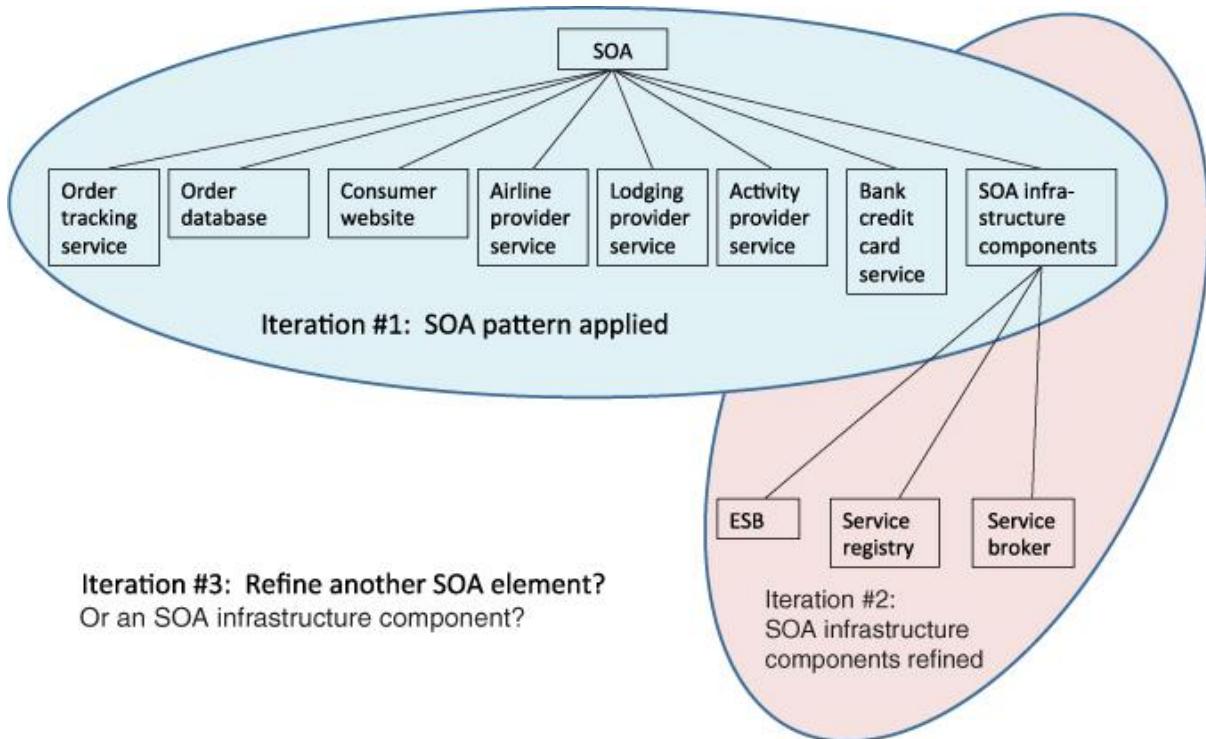


Figure 17.2. Iteration 1 applied the SOA pattern. Iteration 2 refined the infrastructure components. Where will iteration 3 take you?

There are cases when the first iteration of ADD is different. Perhaps you are not creating a system but evolving an existing one. Perhaps you are required to use a piece of software that your company already owns, and therefore must fit it into the design. There are many reasons why some of the design might already be done for you, and the first time through the steps of ADD you won't pick "whole system" as the starting point. Nevertheless, step 1 still holds: All it requires is that at least one of the elements you know about needs further design.

There are two main refinement strategies to pursue with ADD: breadth first and depth first. Breadth first means that all of the second-level elements are designed before any of the third-level elements, and so forth. Depth first means that one downward chain is completed before beginning a second downward chain. The order that you should work through ADD is influenced by the business and technical contexts within which the project is operating. Some of the important factors include the following:

- *Personnel availability may dictate a refinement strategy.* If an important group or team has a window of availability that will close soon and will work on a particular part of the system, then it behooves the architect to design that part of the system to the point where it can be handed off for implementation—depth first. But if the team is not currently available but will be available at some definite time in the future, then you can defer their part of the design until later.
- *Risk mitigation may dictate a refinement strategy.* The idea is to design the risky parts of the system to enough depth so that problems can be identified and solved early. For example, if an unfamiliar technology is being introduced on the project, prototypes using that technology will likely be developed to gain understanding of its implications. These prototypes are most useful if they reflect the design of the actual system. A depth-first strategy can provide a context for technology prototyping. Using this context you can build the prototype in a fashion that allows for its eventual integration into the architecture. On the other hand, if the risk is in how elements at the same level of the design interact with each other to meet critical quality attributes, then a breadth-first strategy is in order.
- *Deferral of some functionality or quality attribute concerns may dictate a mixed approach.* For example, suppose the system being constructed has a medium-priority availability requirement. In this case you might adopt a strategy of employing redundancy

for availability but defer detailed consideration of this redundancy strategy to allow for the rapid generation of the high-priority functionality in an intermediate release. You might therefore apply a breadth-first approach for everything but availability, and then in subsequent design iterations you revisit some of the elements to enable the addition of the responsibilities to support availability. In reality this approach will require some backtracking, where you revisit earlier decisions and refine them or modify them to accommodate this new requirement.

All else being equal, a breadth-first refinement strategy is preferred because it allows you to apportion the most work to the most teams soonest. Breadth first allows for consideration of the interaction among the elements at the same level.

Step 2: Identify the ASRs for This Element

In [Chapter 16](#) we described a number of methods for discovering the ASRs for a system. One of those methods involved building a utility tree. To support the design process, the utility tree has an advantage over the other methods: it guides the stakeholders in prioritizing the QA requirements. The two factors used to prioritize the ASRs in a utility tree are business value and architectural impact. The business value of an ASR typically will not change throughout the design process and does not need to be reconsidered.

If the chosen element for design in step 1 is the whole system, then a utility tree can be a good source for the ASRs. Otherwise, construct a utility tree specifically focused on this chosen element, using the quality attribute requirements that apply to this element (you'll see how to assign those in step 4). Those that are labeled (High, High) are the ASRs for this element. As an architect you will also need to pay attention to the (High, Medium) and (Medium, High) utility tree leaves as well. These will almost certainly also be ASRs for this element.

Step 3: Generate a Design Solution for the Chosen Element

This step is the heart of the ADD. It is the application of the generate-and-test strategy. Upon entry to this step, we have a chosen element for design and a list of ASRs that apply to it. For each ASR, we develop a solution by choosing a candidate design approach.

Your initial candidate design will likely be inspired by a pattern, possibly augmented by one or more tactics. You may then refine this candidate design by considering the design checklists that we gave for the quality attributes in [Chapters 5–11](#). For ASRs that correspond to quality attributes, you can invoke those checklists to help you instantiate or refine the major design approach (such as a pattern) that you've chosen. For example, the layered pattern is helpful for building systems in which modifiability is important, but the pattern does not tell you how many layers you should have or what each one's responsibility should be. But the checklist for the "allocation of responsibilities" design decision category for modifiability in [Chapter 7](#) will help you ask the right questions to make that determination.

Although this step is performed for each ASR in turn, the sources of design candidates outlined above—patterns, tactics, and checklists—will usually do much better than that. That is, you're likely to find design candidates that address several of your ASRs at once. This is because to the extent that the system you're building is similar to others you know about, or to the extent that the problem you are solving is similar to the problems solved by patterns, it is likely that the solutions you choose will be solving a whole collection of ASRs simultaneously. If you can bring a solution to bear that solves more than one of your ASRs at once, so much the better.

The design decisions made in this step now become constraints on all future steps of the method.

Step 4: Verify and Refine Requirements and Generate Input for the Next Iteration

It's possible that the design solution you came up with in the prior step won't satisfy all the ASRs. Step 4 of ADD is a test step that is applied to your design for the element you chose to elaborate in step 1 of this iteration. One of the possible outcomes of step 4 is "backtrack," meaning that an important requirement was not satisfied and cannot be satisfied by further elaborating this design. In this case, the design needs to be reconsidered.

The ASRs you have not yet satisfied could be related to the following:

1. A quality attribute requirement allocated to the parent element
2. A functional responsibility of the parent element
3. One or more constraints on the parent element

[Table 17.1](#) summarizes the types of problems and the actions we recommend for each.

Table 17.1. Recommended Actions for Problems with the Current Hypothesis

Type of ASR Not Met	Action Recommended
1. Quality attribute requirement	Consider applying (more) tactics to improve the design with respect to the quality attribute. For each candidate tactic, ask: <ul style="list-style-type: none">▪ Will this tactic improve the quality attribute behavior of the current design sufficiently?▪ Should this tactic be used in conjunction with another tactic?▪ What are the tradeoff considerations when applying this tactic?
2. Functional responsibility	Add responsibilities either to existing modules or to newly created modules: <ul style="list-style-type: none">▪ Assign the responsibility to a module containing similar responsibilities.▪ Break a module into portions when it is too complex.▪ Assign the responsibility to a module containing responsibilities with similar quality attribute characteristics—for example, similar timing behavior, similar security requirements, or similar availability requirements.
3. Constraint	Modify the design or try to relax the constraint: <ul style="list-style-type: none">▪ Modify the design to accommodate the constraint.▪ Relax the constraint.

In most real-world systems, requirements outstrip available time and resources. Consequently you will find yourself unable to meet some of the QA requirements, functional requirements, and constraints. These kinds of decisions are outside the scope of the ADD method, but they are clearly important drivers of the design process, and as an architect you will be continually negotiating decisions of this form.

Step 4 is about taking stock and seeing what requirements are left that still have not been satisfied by our design so far. At this point you should sequence through the quality attribute requirements, responsibilities, and constraints for the element just designed. For each one there are four possibilities:

1. *The quality attribute requirement, functional requirement, or constraint has been satisfied.* In this case, the design with respect to that requirement is complete; the next time around, when you further refine the design, this requirement will not be considered. For example, if a constraint is to use a particular middleware and the system is decomposed into elements that all use this middleware, the constraint has been satisfied and can be removed from consideration. An example of a quality attribute requirement being satisfied is a requirement to make it easy to modify elements and their interactions. If a publish-subscribe pattern can be shown to have

been employed throughout the system, then this QA requirement can be said to be satisfied.

2. *The quality attribute requirement, functional requirement, or constraint is delegated to one of the children.* For example, if a constraint is to use a particular middleware and the decomposition has a child element that acts as the infrastructure, then delegating that constraint to that child will retain the constraint and have it be reconsidered when the infrastructure element is chosen for subsequent design. Similarly, with the example we gave earlier about providing extensibility, if there is as yet no identifiable plug-in manager, then this requirement is delegated to the child where the plug-in manager is likely to appear.
3. *The quality attribute requirement, functional requirement, or constraint is distributed among the children.* For example, a constraint might be to use .NET. In this case, .NET Remoting might become a constraint on one child and ASP.NET on another. Or a quality attribute requirement that constrains end-to-end latency of a certain operation to 2 seconds could be distributed among the element's three children so that the latency requirement for one element is 0.8 seconds, the latency for a second element is 0.9 seconds, and the latency for a third is 0.3 seconds. When those elements are subsequently chosen for further design, those times will serve as constraints on them individually.
4. *The quality attribute requirement, functional requirement, or constraint cannot be satisfied with the current design.* In this case there are the same two options we discussed previously: you can either backtrack—revisit the design to see if the constraint or quality attribute requirement can be satisfied some other way—or push back on the requirement. This will almost certainly involve the stakeholders who care about that requirement, and you should have convincing arguments as to why the dropping of the requirement is necessary.

Report to the project manager that the constraint cannot be satisfied without jeopardizing other requirements. You must be prepared to justify such an assertion. Essentially, this is asking, "What's more important—the constraint or these other requirements?"

Step 5: Repeat Steps 1–4 Until Done

After the prior steps, each element has a set of responsibilities, a set of quality attribute requirements, and a set of constraints assigned to it. If it's clear that all of the requirements are satisfied, then this unequivocally ends the ADD process.

In projects in which there is a high degree of trust between you and the implementation teams, the ADD process can be terminated when only a sketch of the architecture is available. This could be as soon as two levels of breadth-first design, depending on the size of the system. In this case, you trust the implementation team to be able to flesh out the architecture design in a manner consistent with the overall design approaches you have laid out. The test for this is if you believe that you could begin implementation with the level of detail available and trust the implementation team to that extent. If you have less trust in the implementation team, then an additional level (or levels) of design may be necessary. (And, of course, you will need to subsequently ensure that the implementation is faithfully followed by the team.)

On the other hand, if there is a contractual arrangement between your organization and the implementation organization, then the specification of the portion of the system that the implementers are providing must be legally enforceable. This means that the ADD process must continue until that level of specificity has been achieved.

Finally, another condition for terminating ADD is when the project's design budget has been exhausted. This happens more often than you might think.

Choosing when to terminate ADD and when to start releasing the architecture that you've sketched out are not the same decision. You can, and in many cases should, start releasing early architectural views based on the needs of the project (such as scheduled design reviews or customer presentations) and your confidence in the design so far. The unpalatable alternative is to make everyone wait until the architecture design is finished. You-can't-have-it-until-it's-done is particularly unpalatable in Agile projects, as we discussed in [Chapter 15](#).

You should release the documentation with a caveat as to how likely you think it is to change. But even early broad-and-shallow architectural descriptions can be enormously helpful to implementers and other project staff. A first- or second-level module decomposition view, for instance, lets experts start scouring the marketplace for commercial products that provide the responsibilities of the identified modules. Managers can start making budgets and schedules for implementation that are based on the architecture and not just the requirements. Support staff can start building the infrastructure and file systems to hold project artifacts (these are often structured to mirror the module decomposition view). And early release invites early feedback.

17.4. Summary

The Attribute-Driven Design method is an application of the generate-and-test philosophy. It keeps the number of requirements that must be satisfied to a humanly achievable quantity. ADD is an iterative method that, at each iteration, helps the architect to do the following:

- Choose an element of the system to design.
- Marshal all the architecturally significant requirements for the chosen element.
- Create and test a design for that chosen element.

The output of ADD is not an architecture complete in every detail, but an architecture in which the main design approaches have been selected and validated. It produces a “workable” architecture early and quickly, one that can be given to other project teams so they can begin their work while the architect or architecture team continues to elaborate and refine.

ADD is a five-step method:

1. Choose the element of the system to design. For green-field designs, the “part” to begin with is simply the entire system. For designs that are already partially completed (either by external constraints or by previous iterations through ADD), the part is an element that is not yet designed. Choosing the next element can proceed in a breadth-first, depth-first, or mixed manner.
2. Identify the ASRs for the chosen element.
3. Generate a design solution for the chosen element, using design collateral such as existing systems, frameworks, patterns and tactics, and the design checklists from [Chapters 5–11](#).
4. Verify and refine requirements and generate input for the next iteration. Either the design in step 3 will satisfy all of the chosen element’s ASRs or it won’t. If it doesn’t, then either they can be allocated to elements that will be elaborated in future iterations of ADD, or the existing design is inadequate and we must backtrack. Furthermore, non-ASR requirements will either be satisfied, allocated to children, or indicated as not achievable.
5. Repeat steps 1–4 until all the ASRs have been satisfied, or until the architecture has been elaborated sufficiently for the implementers to use it.

17.5. For Further Reading

You can view design as the process of making decisions; this is another philosophy of design. This view of design leads to an emphasis on design rationale and tools to capture design rationale. The view of design as the process of making decisions dates to the 1940s [[Mettler 91](#)], but it has been recently applied to architecture design most prominently by Philippe Kruchten [[Kruchten 04](#)], and Hans van Vliet and Jan Bosch [[van Vliet 05](#)].

The Software Engineering Institute has produced a number of reports describing the ADD method and its application in a variety of contexts. These include [[Wojcik 06](#)], [[Kazman 04](#)], and [[Wood 07](#)].

George Fairbanks has written an engaging book that describes a risk-driven process of architecture design, entitled *Just Enough Software Architecture: A Risk-Driven Approach* [[Fairbanks 10](#)].

Tony Lattanze has created an Architecture-Centric Design Method (ACDM), described in his book *Architecting Software Intensive Systems: A Practitioners Guide* [[Lattanze 08](#)].

Ian Gorton's *Essential Architecture, Second Edition*, emphasizes the middleware aspects of a design [[Gorton 10](#)].

Woods and Rozanski have written *Software Systems Architecture, Second Edition*, which interprets the design process through the prism of different views [[Woods 11](#)].

A number of authors have compared five different industrial architecture design methods. You can find this comparison at [[Hofmeister 07](#)].

Raghvinder Sangwan and his colleagues describe the design of a building management system that was originally designed using object-oriented techniques and then was redesigned using ADD [[Sangwan 08](#)].

17.6. Discussion Questions

1. ADD does not help with the detailed design of interfaces for the architectural elements it identifies. Details of an interface include what each method does, whether you need to call a single all-encompassing method to perform the work of the element or many methods of finer-grained function, what exceptions are raised on the interface, and more. What are some examples where the specific design of an interface might bring more or less performance, security, or availability to a system? (By the way, if there are quality attribute implications to an interface, you can capture those as annotations on the element.)
2. What sets a constraint apart from other (even high-priority) requirements is that it is not negotiable. Should this consideration guide the design process? For example, would it be wise to design to satisfy all of the constraints before worrying about other ASRs?
3. In discussion question 4 of [Chapter 16](#) you were asked to create a utility tree for an ATM. Now choose the two most important ASRs from that utility tree and create a design fragment using the ADD method employing and instantiating a pattern.

18. Documenting Software Architectures

If it is not written down, it does not exist.

—Philippe Kruchten

Even the best architecture, the 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 use, build, or modify it; or (worst of all) misunderstand it and apply it incorrectly. And 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, one that you expect to stand the test of time, then you *must* go to the trouble of describing it in enough detail, without ambiguity, and organizing it so that others can quickly find and update 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.

The sad truth is that architectural documentation today, if it is done at all, 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 all 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 good 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 architects also see documentation as delivering value to themselves. Documentation serves as the receptacle to hold the results of major design decisions as they are confirmed. A well-thought-out documentation scheme can make the process of design go much more smoothly and systematically. Documentation helps the architect(s) reason about the architecture design and communicate it while the architecting is in progress, whether in a six-month design phase or a six-day Agile sprint.

18.1. Uses and Audiences for Architecture Documentation

Architecture documentation must serve varied purposes. It should be sufficiently transparent and accessible 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. When planning and reviewing documentation, you need to ensure support for all the relevant needs.

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 information to capture.

Fundamentally, architecture documentation has three uses:

1. Architecture documentation 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 documentation 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.

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 or 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 to ever be reproducible from memory alone. See the sidebar “[Schmucks and Jerks](#).”

3. Architecture documentation serves as the basis for system analysis and construction. Architecture tells implementers what to implement. Each module has interfaces that must be provided and uses interfaces from other modules. Not only does this provide instructions about the provided and used interfaces, but it also determines with what other teams the development team for the module must communicate.

During development, an architecture can be very complex, with many issues left to resolve. Documentation can serve as a receptacle for registering and communicating these issues that might otherwise be overlooked.

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. It must contain the information necessary to evaluate a variety of attributes, such as security, performance, usability, availability, and modifiability.

For system builders who use automatic code-generation tools, the documentation may incorporate the models used for generation. These models provide guidance to those who wish to understand the behavior of the module in more detail than is normally documented but in less detail than examining the code would provide.

18.2. Notations for Architecture Documentation

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

- *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 they cannot be formally analyzed. In our experience, the most common tool for informal notations is PowerPoint.
- *Semiformal notations.* Views are expressed in a standardized notation that prescribes graphical elements and rules of construction, but it 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. UML is a semiformal notation in this sense.
- *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. 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 architectural views. In others they allow many views, or even provide the ability to formally define new views. The usefulness of ADLs lies in their ability to support automation through associated tools:

automation to provide useful analysis of the architecture or assist in code generation. In practice, the use of such notations is rare.

Schmucks and Jerks

One day I was sitting in a meeting with a well-known compiler guru. He was recounting some of his favorite war stories from his long career. One of these stories particularly stuck with me. He was talking about the time that he was chasing down a very nasty and subtle bug in the code of a compiler that he was maintaining. After a long and exasperating search, he finally located and eventually fixed the bug. But the search itself had gotten him so worked up, and he was so infuriated at the irresponsible thought and programming that led to the bug, that he decided to do a bit more detective work and figure out who was the jerk responsible for that bug.

By going backward through the revision history, he found the culprit. It was him. He was the jerk. It turns out that he was the one who—eight years earlier—had written that offending piece of code. The trouble was, he had no recollection of writing the code and no recollection of the rationale for writing it the way he had done. Perhaps there was a good reason to do so at the time, but if so it was lost now.

That is why we document. The documentation helps the poor schmuck who has to maintain your code in the future, and that schmuck might very well be you!

—RK

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

Regardless of the level of formality, always remember that different notations are better (or worse) for expressing different kinds of information. Formality aside, no UML class diagram will help you reason about schedulability, nor will a sequence chart tell you very much about the system's likelihood of being delivered on time. You should choose your notations and representation languages always keeping in mind the important issues you need to capture and reason about.

18.3. Views

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. A view is a representation of a set of system elements and relations among them—not all system elements, but those of a particular type. For example, a layered view of a system would show elements of type “layer”—that is, it would show the system’s decomposition into layers—and the relations among those layers. A pure layered view would not, however, show the system’s services, or clients and servers, or data model, or any other type of element.

Thus, views let us divide the multidimensional entity that is a software architecture into a number of (we hope) interesting and manageable representations of the system. The concept of *views* gives us our most fundamental principle of architecture documentation:

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

This maxim gives our approach to documentation its name: *Views and Beyond*.

What are the relevant views? This depends entirely 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 let you reason about your system's portability, a *deployment view* will let you reason about your system's performance and reliability, and so forth.

Different views support different goals and uses. This is why we do not advocate a particular view or collection 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 relations. How many different views to represent is the result of a cost/benefit decision. Each view has a cost and a benefit, and you should ensure that the benefits of maintaining a particular view outweigh its costs.

Views may be driven by the need to document a particular pattern in your design. Some patterns are composed of modules, others of components and connectors, and still others have deployment considerations. Module views, component-and-connector (C&C) views, and allocation views are the appropriate mechanisms for representing these considerations.

Module Views

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. Example module views are decomposition, uses, and layers. 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 revision history. The relations that modules have to one another include *is part of*, *depends on*, and *is a*.

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, this 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.

[Table 18.1](#) summarizes the elements, relations, constraints, and purpose of the module views in general. Later we provide this information specific to each of a number of often used module views.

Table 18.1. Summary of the Module Views

Elements	Modules, which are implementation units of software that provide a coherent set of responsibilities.
Relations	<ul style="list-style-type: none">▪ <i>Is part of</i>, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.▪ <i>Depends on</i>, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant.▪ <i>Is a</i>, 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, such as limitations on the visibility between modules.
Usage	<ul style="list-style-type: none">▪ Blueprint for construction of the code▪ Change-impact analysis▪ Planning incremental development▪ Requirements traceability analysis▪ Communicating the functionality of a system and the structure of its code base▪ Supporting the definition of work assignments, implementation schedules, and budget information▪ Showing the structure of information that the system needs to manage

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. 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.
- *Responsibilities*. The responsibility property for 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.
- *Visibility of interface(s)*. When a module has submodules, some interfaces of the submodules are public and some may be private; that is, the interfaces are used only by the submodules within the enclosing parent module. These private interfaces are not visible outside that context.
- *Implementation information*. Modules are units of implementation. It is therefore useful to record information related to their implementation from the point of view of managing their development and building the system that contains them. This might include the following:
 - *Mapping to source code units*. This identifies the files that constitute the implementation of a module. For example, a module Account, if implemented in Java, might have several files that constitute its implementation: IAccount.java (an interface), AccountImpl.java (the 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 document. This information may simply be a pointer to the location of these artifacts.
- *Management information.* A manager may need information about the module's predicted schedule and budget. This information may simply be a pointer to the location of these artifacts.
- *Implementation constraints.* In many cases, the architect will have an implementation strategy in mind for a module or may know of constraints that the implementation must follow.
- *Revision history.* Knowing the history of a module including authors and particular changes may help when you perform maintenance activities.

Because modules partition the system, it should be possible to determine how the functional requirements of a system are supported by module responsibilities. Module views that show dependencies among modules or layers (which are groups of modules that have a specific pattern of allowed usage) provide a good basis for change-impact analysis. Modules are typically 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 to be able to create useful results.

A module view can be used to explain the system's functionality to someone not familiar with it. 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 helpful, as they explain the structure of the code base to a new developer on the team. Thus, up-to-date module views can simplify and regularize system maintenance.

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, and many other runtime qualities. For those, we rely on component-and-connector and allocation 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 for small, simple applications. More often, a single module will be replicated as part of many runtime components, and a given component could map to several modules. Module views also provide the software elements that are mapped to the diverse nonsoftware elements of the system environment in the various allocation views.

Component-and-Connector Views

Component-and-connector views show elements that have some runtime presence, such as processes, objects, clients, servers, and data stores. These elements are termed *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. Sample C&C views are service-oriented architecture (SOA), client-server, or communicating process views.

Components have interfaces called *ports*. A port defines a 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, each forming a different input or output channel at runtime. In this respect ports differ from interfaces of modules, whose interfaces are never replicated. You can annotate a port with a number or range of numbers to indicate replication; for example, "1..4" might mean that an interface could be replicated up to four times. A component's ports should be explicitly documented, by showing them in the diagram and defining them in the diagram's supporting documentation.

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 employ a different pattern than the one in which the component appears.

Connectors are the other kind of element in a C&C view. Simple examples of connectors are service invocation; asynchronous message queues; event multicast supporting publish-subscribe interactions; and pipes that represent asynchronous, order-preserving data streams. 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.

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.

Like components, complex connectors may in turn be decomposed into collections of components and connectors that describe the architectural substructure of those connectors. Connectors need not be binary. That is, they need not have exactly two roles. For example, a publish-subscribe connector 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. 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 timeouts. The protocol of interaction should be documented.

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 view. Compatibility often is defined in terms of information type and protocol. For example, in a call-return architecture, you should check to make sure that all "calls" ports are attached to some call-return connector. At a deeper semantic level, you should check to make sure that a port's protocol is consistent with the behavior expected by the role to which it is attached.

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. 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 availability.
- *Performance*. What kinds of response time will the component provide under what loads? What kind of bandwidth, latency, jitter, transaction volume, or throughput can be expected for a given connector? This property can be used with others to determine system-wide 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.

- *Modifiability*. Does the messaging structure support a structure to cater for evolving data exchanges? Can the components be adapted to process those new messages? This property can be defined to extend the functionality of a component.
- *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.

C&C views are commonly used to show to developers and other stakeholders how the system works—one can “animate” or trace through a C&C view, showing an end-to-end thread of activity. C&C views are also used to reason about runtime system quality attributes, such as performance and availability. In particular, a well-documented view allows architects to predict overall system properties such as latency or reliability, given estimates or measurements of properties of the individual elements and their interactions.

[Table 18.2](#) summarizes the elements, relations, and properties that can appear in C&C views. This table is followed by a more detailed discussion of these concepts, together with guidelines concerning their documentation.

Table 18.2. Summary of Component-and-Connector Views

Elements	<ul style="list-style-type: none"> ▪ <i>Components</i>. Principal processing units and data stores. A component has a set of <i>ports</i> through which it interacts with other components (via connectors). ▪ <i>Connectors</i>. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.
Relations	<ul style="list-style-type: none"> ▪ <i>Attachments</i>. Component ports are associated with connector roles to yield a graph of components and connectors. ▪ <i>Interface delegation</i>. In some situations component ports are associated with one or more ports in an “internal” subarchitecture. The case is similar for the roles of a connector.
Constraints	<ul style="list-style-type: none"> ▪ Components can only be attached to connectors, not directly to other components. ▪ Connectors can only be attached to components, not directly to other connectors. ▪ Attachments can only be made between compatible ports and roles. ▪ Interface delegation can only be defined between two compatible ports (or two compatible roles). ▪ Connectors cannot appear in isolation; a connector must be attached to a component.
Usage	<ul style="list-style-type: none"> ▪ Show how the system works. ▪ Guide development by specifying structure and behavior of runtime elements. ▪ Help reason about runtime system quality attributes, such as performance and availability.

Notations for C&C Views

As always, box-and-line drawings are available to represent C&C views. Although informal notations are limited in the semantics that can be conveyed, following some simple guidelines can lend rigor and depth to the descriptions. The primary guideline is simple: assign each component type and each connector type a separate visual form (symbol), and list each of the types in a key.

UML components are a good semantic match to C&C components because they permit intuitive documentation of important information like interfaces, properties, and behavioral descriptions.

UML components also distinguish between component types and component instances, which is useful when defining view-specific component types.

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 18.1](#), though this is typically only done on component types. The number of ports on component instances, as shown in the right portion of [Figure 18.1](#), is typically bound to a specific number. Components that dynamically create and manage a set of ports should retain a multiplicity descriptor on instance descriptions.



Figure 18.1. A UML representation of the ports on a C&C component type (left) and component instance (right). The Account Database component type has two types of ports, Server and Admin (noted 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.

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 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. 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.

You should represent a “rich” C&C connector using a UML component, or by annotating a line UML connector with a tag or other auxiliary documentation that explains the meaning of the complex connector.

Allocation Views

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

[Table 18.3](#) summarizes the characteristics of allocation views. Allocation views consist of software elements and 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 view.

Table 18.3. Summary of the Characteristics of Allocation Views

Elements	<ul style="list-style-type: none">▪ <i>Software element.</i> A software element has properties that are <i>required</i> of the environment.▪ <i>Environmental element.</i> An environmental element has properties that are <i>provided</i> to the software.
Relations	<i>Allocated to.</i> A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.
Constraints	Varies by view
Usage	<ul style="list-style-type: none">▪ For reasoning about performance, availability, security, and safety.▪ For reasoning about distributed development and allocation of work to teams.▪ For reasoning about concurrent access to software versions.▪ For reasoning about the form and mechanisms of system installation.

The relation in an allocation view is *allocated to*. We usually talk about allocation views in terms of a mapping from software elements to environmental elements, although the reverse mapping can also be relevant and interesting. 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, either during development or execution of the system, then the architecture is said to be dynamic with respect to that allocation. For example, processes might migrate from one processor or virtual machine to another. Similarly modules might migrate from one development team to another.

Software elements and environmental elements have properties in allocation views. 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 processing power. For another example, a computing platform might not allow a task to use more than 10 kilobytes of virtual memory. An execution model of the software element in question can be used to determine the required virtual memory usage. Similarly, if you are migrating a module from one team to another, you might want to ensure that the new team has the appropriate skills and background knowledge.

Allocation views can depict static or dynamic views. A static view depicts a fixed allocation of resources in an environment. A dynamic view depicts the conditions and the triggers for which allocation of resources changes according to loading. Some systems recruit and utilize new resources as their load increases. An example is a load-balancing system in which new processes or threads are created on another machine. In this view, the conditions under which the allocation changes, the allocation of runtime software, and the dynamic allocation mechanism need to be documented. (Recall from [Chapter 1](#) that one of the allocation structures is the work assignment structure, which allocates modules to teams for development. That relationship, too, can be allocated dynamically, depending on "load"—in this case, the load on development teams.)

Quality Views

Module, C&C, and allocation views are all structural views: They primarily show the structures that the architect has engineered into the architecture to satisfy functional and quality attribute requirements.

These views are excellent for guiding and constraining downstream developers, whose primary job it is to implement those structures. However, in systems in which certain quality attributes (or, for that matter, some other kind of stakeholder concerns) are particularly important and pervasive, structural views may not be the best way to present the architectural solution to those needs. The reason is that the solution may be spread across multiple structures that are inconvenient to combine (for example, because the element types shown in each are different).

Another kind of view, which we call a *quality view*, can be tailored for specific stakeholders or to address specific concerns. These quality views are formed by extracting the relevant pieces of structural views and packaging them together. Here are five examples:

- A *security view* can show all of the architectural measures taken to provide security. It would show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest. The view's context information would show other security measures (such as physical security) in the system's environment. The behavior part of a security view would show the operation of security protocols and where and how humans interact with the security elements. It would also capture how the system would respond to specific threats and vulnerabilities.
- A *communications view* might be especially helpful for systems that are globally dispersed and heterogeneous. This view would show all of the component-to-component channels, the various network channels, quality-of-service parameter values, and areas of concurrency. This view can be used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection). The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.
- An *exception or error-handling view* could help illuminate and draw attention to error reporting and resolution mechanisms. Such a view would show how components detect, report, and resolve faults or errors. It would help identify the sources of errors and appropriate corrective actions for each. Root-cause analysis in those cases could be facilitated by such a view.
- A *reliability view* would be one in which reliability mechanisms such as replication and switchover are modeled. It would also depict timing issues and transaction integrity.
- A *performance view* would include those aspects of the architecture useful for inferring the system's performance. Such a view might show network traffic models, maximum latencies for operations, and so forth.

These and other quality views reflect the documentation philosophy of ISO/IEC/IEEE standard 42010:2011, which prescribes creating views driven by stakeholder concerns about the architecture.

18.4. Choosing the Views

Documenting decisions during the design process (something we strongly recommend) produces views, which are the heart of an architecture document. It is most likely that these views are rough 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 broad dissemination and cosmetic polish. They are documented purely as your own memory aid.

By the time you're ready to release an architecture document, 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 given 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.

You can determine which views are required, when to create them, and how much detail to include if you know the following:

- What people, and with what skills, are available
- Which standards you have to comply with
- What budget is on hand
- What the schedule is
- What the information needs of the important stakeholders are
- What the driving quality attribute requirements are
- What the size of the system is

At a minimum, expect to have at least one module view, at least one C&C view, and for larger systems, at least one allocation view in your architecture document. Beyond that basic rule of thumb, however, there is a three-step method for choosing the views:

- **Step 1. Build a stakeholder/view table.** Enumerate the stakeholders for your project's software architecture documentation down the rows. Be as comprehensive as you can. For the columns, enumerate the views that apply to your system. (Use the structures discussed in [Chapter 1](#), the views discussed in this chapter, and the views that your design work in ADD has suggested as a starting list of candidates.) Some views (such as decomposition, uses, and work assignment) apply to every system, while others (various C&C views, the layered view) only apply to some systems. 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.

- **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 marginal views in the table: those that require only an overview, or that serve very few stakeholders. Combine each marginal view with another view that has a stronger constituency.

- **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. What you do first depends on your project, but here are some things to consider:

- The decomposition view (one of the module views) is a particularly helpful view to release early. High-level (that is, broad and shallow) decompositions are often easy to design, and with this information the project manager can start to staff development teams, put training in place, determine which parts to outsource, and start producing budgets and schedules.
- Be aware that you don't have to satisfy all the information needs of all the stakeholders to the fullest extent. Providing 80 percent of the information goes a long way, and this might be good enough so that the stakeholders can do their job. Check with the stakeholder to see if a subset of information would be sufficient. They typically prefer a product that is delivered on time and within 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.

18.5. 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 have strong associations with each other. Managing how architectural structures are associated is an important part of the architect's job, independent of whether any documentation of those structures exists.

Sometimes the most convenient way to show a strong association between two views is to collapse them into a single *combined view*, as dictated by step 2 of the three-step method just presented to choose the views. A combined view is a view that contains elements and relations that come from two or more other views. Combined views can be very useful as long as you do not try to overload them with too many mappings.

The easiest way to merge views is to create an *overlay* that combines the information that 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 understand than the two views seen separately. For an example, see the overlay of decomposition and uses sketches shown in [Figure 18.2](#). In an overlay, the elements and the relations keep the types as defined in their constituent views.

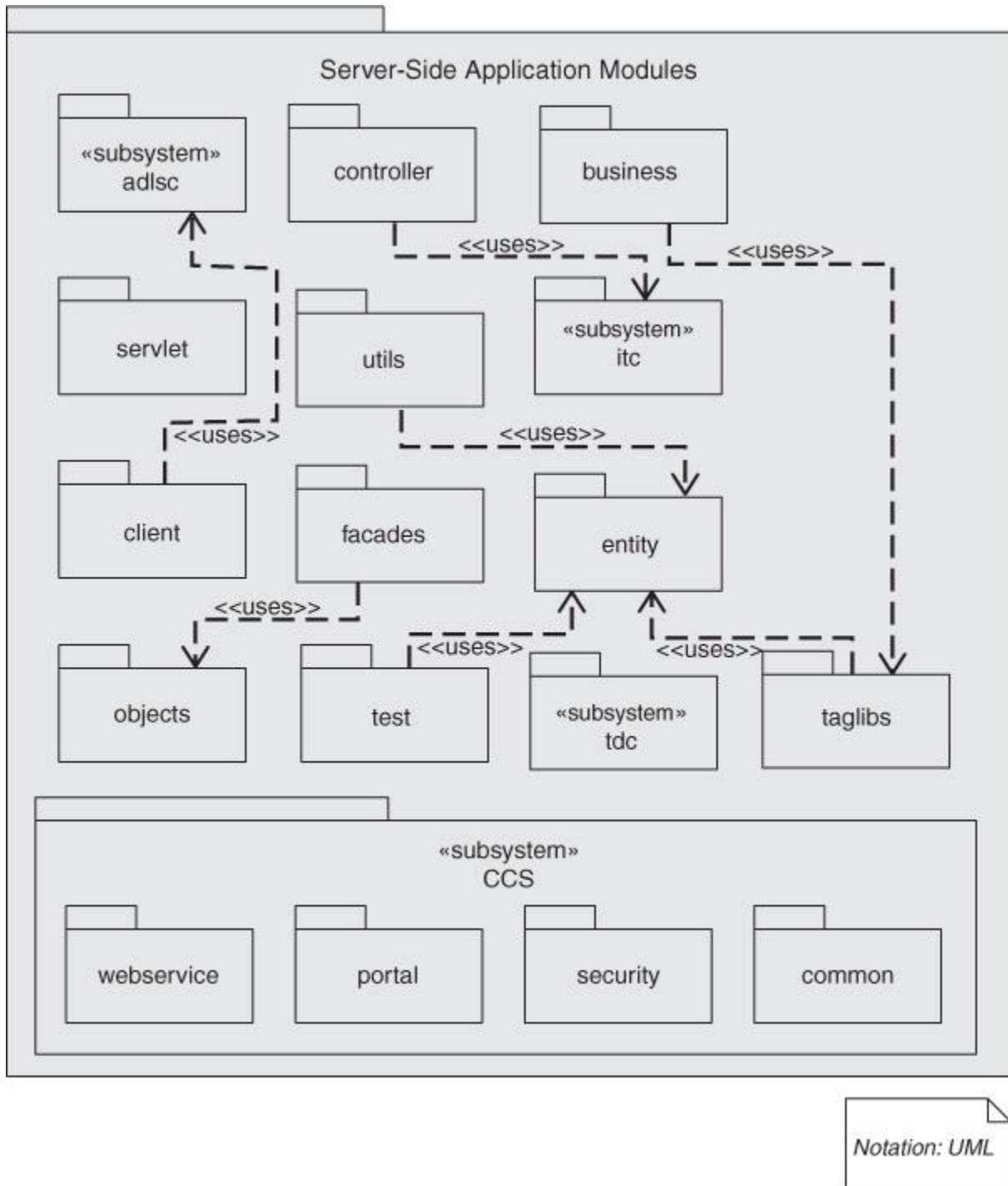


Figure 18.2. A decomposition view overlaid with “uses” information, to create a decomposition/uses overlay.

The views below often combine naturally:

- *Various C&C views.* Because C&C views all show runtime relations 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 SOA or communicating-processes views.* An SOA 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.

- *Decomposition view and any of work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses. In addition, these modules populate layers.

18.6. Building the Documentation Package

Remember the principle of architecture documentation, with which we started this chapter. This principle tells us that our task is to document the relevant views and to document the information that applies to more than one view.

Documenting a View

[Figure 18.3](#) shows a template for documenting a view.

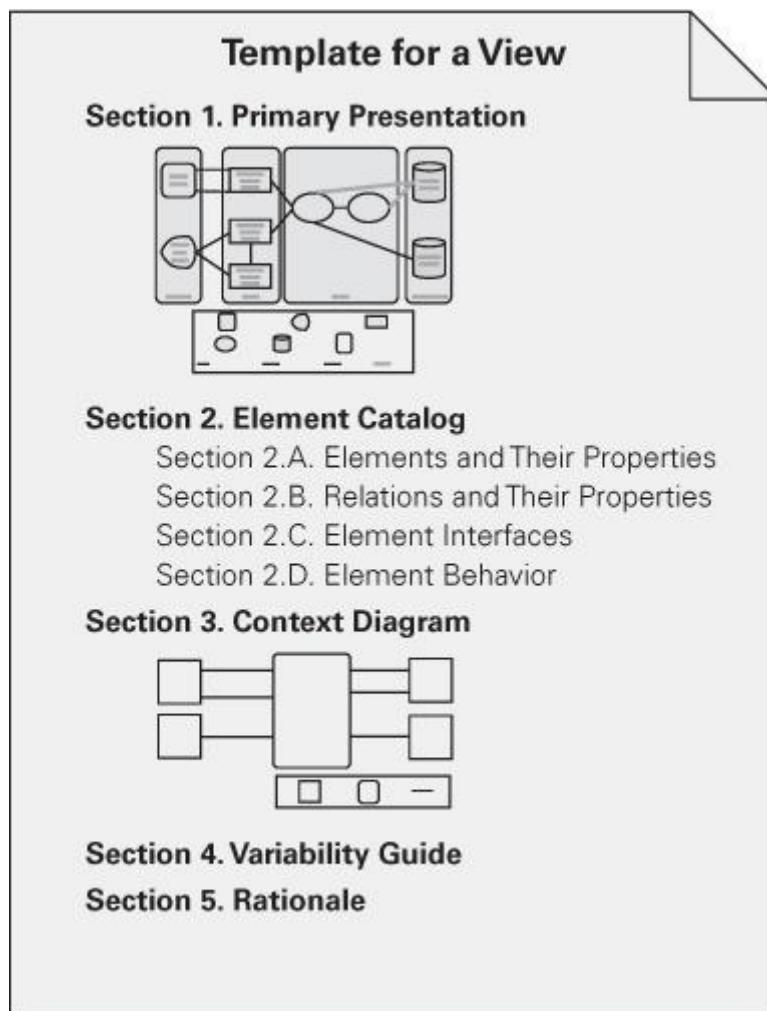


Figure 18.3. View template

No matter what the view, the documentation for a view can be placed into a standard organization consisting of these parts:

- **Section 1: The Primary Presentation.** 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. 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.

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. If your primary presentation is graphical, make sure to include a key that explains the notation. Lack of a key is the most common mistake that we see in documentation in practice.

Occasionally the primary presentation will be textual, such as a table or a list. If that text is presented according to certain stylistic rules, these rules 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.

- **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. 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:

- *Elements and their properties.* This section names each element in the view and lists the properties of that element. Each view introduced in [Chapter 1](#) listed a set of suggested properties associated with that view. 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 view chosen or the architect has introduced new ones, this is where they are documented and given values.
- *Relations and their properties.* Each view has specific relation types 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.
- *Element interfaces.* This section documents element interfaces.
- *Element behavior.* This section documents element behavior that is not obvious from the primary presentation.

- **Section 3: Context Diagram.** A *context diagram* shows how the system or portion of the system depicted in this view relates to its environment. The purpose of a **context diagram** is to depict the scope of a view. Here “context” means an environment with which the part of the system interacts. Entities in the environment may be humans, other computer systems, or physical objects, such as sensors or controlled devices.
- **Section 4: Variability Guide.** A *variability guide* shows how to exercise any variation points that are a part of the architecture shown in this view.
- **Section 5: Rationale.** *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 choice of a pattern in this view should be justified here by describing the architectural problem that the chosen pattern solves and the rationale for choosing it over another.

Documenting Information Beyond Views

As shown in [Figure 18.4](#), documentation beyond views can be divided into two parts:

1. *Overview of the architecture documentation.* This tells 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 and 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.

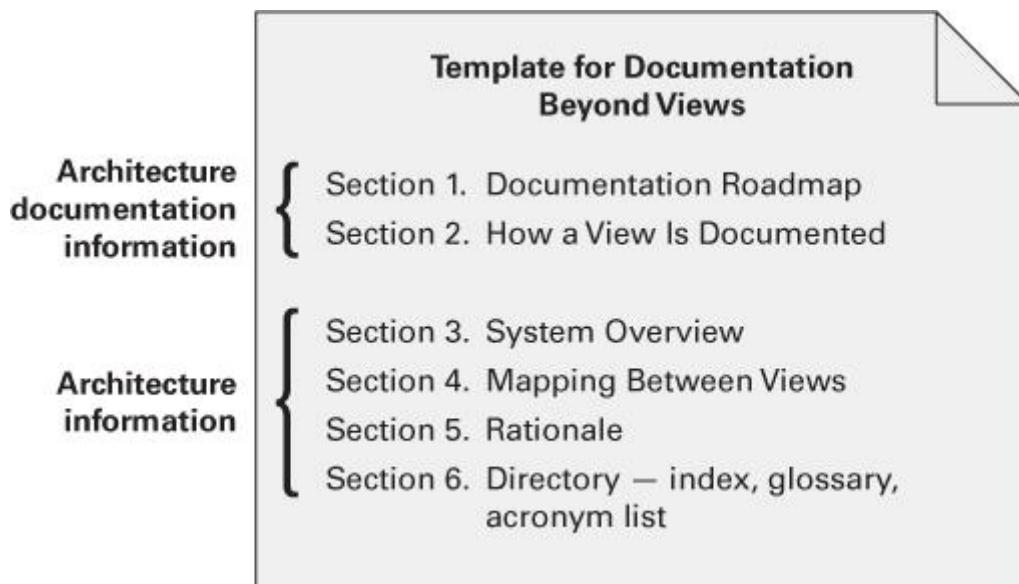


Figure 18.4. Summary of documentation beyond views

Figure 18.4 summarizes our template for documentation beyond views. Documentation beyond views consists of the following sections:

- **Document control information.** List the issuing organization, the current version number, 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.
- **Section 1: Documentation Roadmap.** The documentation roadmap tells the reader what information is in the documentation and where to find it. A documentation map consists of four sections:
 - *Scope and summary.* Explain the purpose of the document and briefly summarize what is covered and (if you think it will help) what is not covered. Explain the relation to other documents (such as downstream design documents or upstream system engineering documents).
 - *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.
 - *View overview.* The major part of the map describes the views that the architect has included in the package. For each view, the map gives the following information:
 - The name of the view and what pattern it instantiates, if any.
 - 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.
 - A description of language, modeling techniques, or analytical methods used in constructing the view.
 - *How stakeholders can use the documentation.* The map 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 view¹ to see what modules use the affected modules (and thus might also have to change)." To be compliant with ISO/IEC 42010-2007, you must consider the concerns of at least users, acquirers, developers, and maintainers.

1. The uses view is a module view. It shows the uses structure discussed in [Chapter 1](#).

- **Section 2: How a View Is Documented.** 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 given above describing our view template 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. This section provides your readers with a consistent mental model of the system and its purpose. This might be just a pointer to a concept-of-operations document.
- **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.

The associations between elements across views in an 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.

View-to-view associations can be conveniently captured as tables. 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.

- **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 requirements that led to decisions of system-wide import. The decisions about which fundamental architecture patterns to use are often described here.
- **Section 6: Directory.** The directory is a set of reference material that helps readers find more information quickly. It includes an index of terms, a glossary, and an acronym list.

Online Documentation, Hypertext, and Wikis

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 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.

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 only want architects to be able to change the architecture or at least provide architects with a "final approval" mechanism. A special kind of shared document that is ideal for this purpose is a wiki.

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. For example, the end of each iteration or sprint or incremental release could be associated with providing revised documentation to the development team.

Documenting Patterns

Architects can, and typically do, use patterns as a starting point for their design, as we have discussed in [Chapter 13](#). These patterns might be published in existing catalogs or in an organization's proprietary repository of standard designs, or created specifically for the problem at hand by the architect. In each of these cases, 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 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, this will consist essentially of showing that the problem at hand fits the problem and context of the pattern.

Using a pattern 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 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.

18.7. Documenting Behavior

Documenting an architecture requires behavior documentation that complements structural views by describing how architecture elements interact with each other. Reasoning about characteristics such as a system's potential to deadlock, a system's ability to complete a task in the desired amount of time, or 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—that is, how they behave with each other. In this section, we provide guidance as to what types of things you will want to document in order to reap these benefits. In our architecture view template, behavior has its own section in the element catalog.

There are two kinds of notations available for documenting behavior. The first kind of notation is called trace-oriented languages; the second is called comprehensive languages.

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. Below we describe four notations for documenting traces: use cases, sequence diagrams, communication diagrams, and activity diagrams. Although other notations are available (such as message sequence charts, timing diagrams, and the Business Process Execution Language), we have chosen these four as a representative sample of trace-oriented languages.

- *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 case diagrams but 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. The use case description is textual and should contain the use case name and brief description, the 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 nonsuccess cases.
- 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 18.5](#) is an example of a sequence diagram that illustrates the basic UML notation.

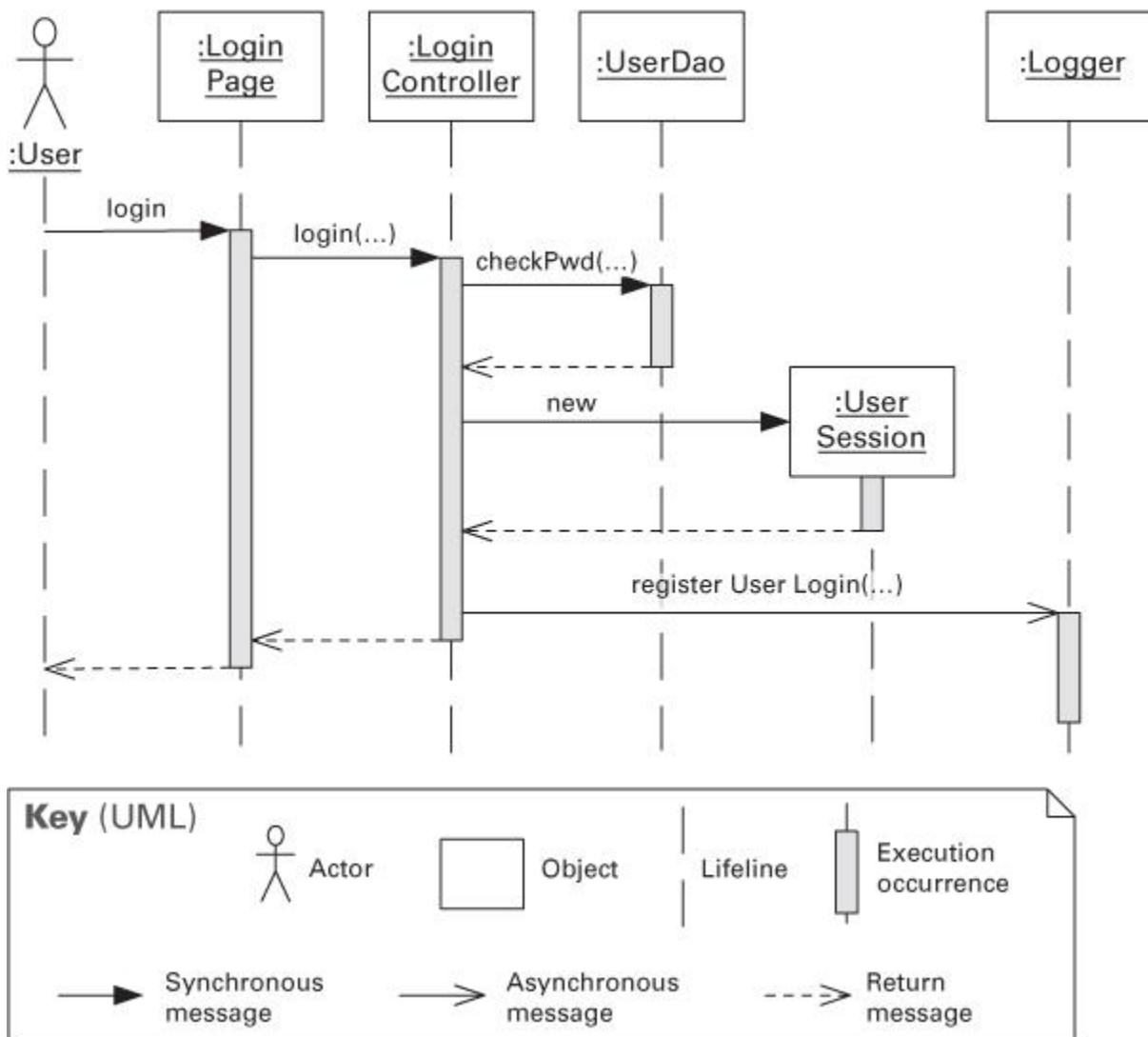


Figure 18.5. A simple example of a UML sequence diagram

Objects (i.e., 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.

- A UML *communication diagram* shows a graph of interacting elements and annotates each interaction with a number denoting order. Similarly to sequence diagrams, instances shown in a communication diagram are 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.
- 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. Activity diagrams can 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. Different from sequence and communication diagrams, activity diagrams don't show the actual operations being performed on specific objects. Activity diagrams are useful to broadly describe the steps in a specific workflow. Conditional branching (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.

In contrast to trace notations, *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 represents 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.

UML state machine diagram notation is based on the statechart graphical formalism developed by David Harel for modeling reactive systems; it allows 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 and help to illustrate their runtime interactions. [Figure 18.6](#) is a simple example showing the states of a vehicle cruise control system.

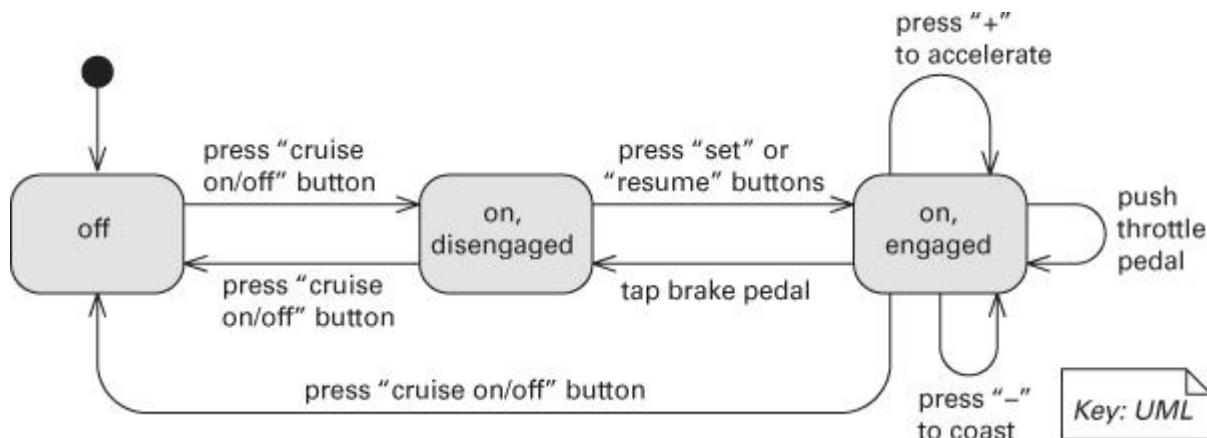


Figure 18.6. UML state machine diagram for the cruise control system of a motor vehicle

Each transition in a state machine diagram is labeled with the event causing the transition. For example, in [Figure 18.6](#), the transitions correspond to the buttons the driver can press or driving actions that affect the cruise control system. Optionally, the transition can specify a guard condition, which is enclosed in brackets. When the event corresponding to the transition occurs, the guard condition is evaluated and the transition is only enabled 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 performed when the transition occurs. The states may also specify entry and exit actions.

Other notations exist for describing comprehensive behavior. For example, Architecture Analysis and Design Language (AADL) can be used to reason about runtime behavior. Specification and Description Language (SDL) is used in telephony.

18.8. 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? Short of a full-fledged quality view (see page [340](#)), there are five major ways:

1. Any major design approach (such as an architecture pattern) 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 tradeoffs incurred. Look for the place in the documentation where such an explanation occurs. In our approach, we call that *rationale*.
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 service-level agreement. Or they may simply be recorded *as properties* that the elements exhibit.
3. Quality attributes often impart a “language” of things that you would look for. Security involves 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 it up by name or reference in your architecture document to see the places where that requirement is satisfied.
5. Every quality attribute requirement will have a constituency of stakeholders who want to know that it 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 this: “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.

18.9. Documenting 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, search for and 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) already exist. 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 websites, build and “go live” with their system many times every day.

Whether an architecture changes at runtime, or as a result of a high-frequency release-and-deploy cycle, the changes occur 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:

- *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.
- *Document the ways the architecture is allowed to change.* In the previous examples, this will usually mean adding new components and replacing components with new implementations. In the Views and Beyond approach, the place to do this is called the variability guide (captured in Section 4 of our view template).

18.10. Documenting Architecture in an Agile Development Project

“Agile” refers to an approach to software development that emphasizes rapid and flexible development and de-emphasizes project and process infrastructure for their own sake. In [Chapter 15](#) we discuss the relationships between architecture and Agile. Here we focus just on how to document architecture in an Agile environment.

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.

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.

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 earlier prescribes producing the documentation in prioritized stages to satisfy the needs of the stakeholders who need it now.

When producing Views and Beyond-based architecture documentation using Agile principles, keep the following in mind:

- Adopt a template or standard organization to capture your design decisions.
- Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.
- 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.
- 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.

- 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 templates 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).
 - Agile teams sometimes make models in brief discussions by the whiteboard. When documenting a view, the primary presentation may consist of a digital picture of the whiteboard. Further information about the elements (element catalog), rationale discussion (architecture background), variability mechanisms being used (variability guide), and all else can 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, the template will have the right place ready to receive it.
-

The Software You're Delivering Isn't the Only Software That Matters

About ninety-nine percent of the treatment of architecture in this book (and others) is concerned with the software elements that make up the operational system that is delivered to its customer. Component-and-connector views show the units of runtime behavior of *that* system. Module views show the units of implementation that have to be built in order to create *that* system.

A colleague of mine is a project manager for a Fortune 500 software company. On the day I wrote this sidebar, she found out that the development platform her project relied on had been infected with a virulent new virus, and the company's IT department was removing it from service, along with *all* the backup images, until the virus could be completely removed. That was going to take about five days. After that, all of her project's software and tooling would have to be reinstalled and brought back up to latest-version status. Her project was in user final acceptance test, racing against a delivery deadline, and the IT department's decision doomed her project to join the countless others in our industry that are delivered late. The snarling email she sent to the IT department for (a) allowing the platform to become infected and (b) not providing a backup platform (real or virtual) in a timely fashion would melt your screen.

The treatment of software architecture we describe in this book is perfectly capable of representing and usefully incorporating software other than the software that your customer is paying you to deliver. Allocation views, recall, are about mapping that software to structures in the environment. "Uses" views show which software elements rely on the correct presence of other software in order to work. Context diagrams are all about showing relations between your system and important elements of its environment. It would be the easiest thing in the world to use these constructs to represent support software including, in my friend's case, the development platform.

An avionics project I worked on years ago included in our decomposition view a module called the System Generation Module. This consisted of all of the software we needed to construct a loadable image of the product we were building. Not a single byte of code from the System Generation Module made it onto the aircraft, but it was as important as any other. Even if you don't build any of your support software but use off-the-shelf development tools from your favorite vendor, someone in your organization is responsible for the care and feeding of that software: its acquisition, installation, configuration, and upgrade. That constitutes a nontrivial work assignment, which suggests that support software also belongs in the work assignment view (a kind of allocation view). And of course you always build *some* of it yourself—test scripts, build scripts, and so forth—so it's even more deserving of a place in your architecture.

Promoting support and development software to first-class architectural status makes us ask the right questions about it, especially the most important one: What quality attributes do we require of it? Will it provide us with the right security if (for example) we want to exclude our subcontracting partners from access to some of our IP during development? Will it have the availability to be up and running at 2 a.m. Sunday morning when our project goes into its inevitable final delivery crunch? And if it crashes, will the IT folks have someone standing by to bring it back up? Will it be modifiable or configurable enough to support the way your project intends to use it?

Think about what other software and environmental resources your project depends on, and consider using the architectural tools, models, views, and concepts at your disposal to help you do what architecture always helps you do: Ask the right questions at the right time to expose risks and begin to mitigate them. These concepts include quality attribute scenarios, “uses” views, and deployment and work assignment views that include support software.

—PCC

18.11. Summary

Writing architectural documentation is much like other types of writing. You must understand the uses to which the writing is to be put and the audience for the writing. Architectural documentation serves as a means for communication among various stakeholders, not only up the management chain and down to the developers but also across to peers.

An architecture is a complicated artifact, best expressed by focusing on particular perspectives depending on the message to be communicated. These perspectives are called views, and you must choose the views to document, must choose the notation to document these views, and must choose a set of views that is both minimal and adequate. This may involve combining various views that have a large overlap. You must document not only the structure of the architecture but also the behavior.

Once you have decided on the views, you must decide how to package the documentation. The packaging will depend on the media used for expressing the documentation. Print has different characteristics for understanding and grouping than various online media. Different online media will also have different characteristics.

The context of the project will also affect the documentation. Some of the contextual factors are the important quality attributes of the system, the rate of change of the system, and the project management strategy.

18.12. For Further Reading

Documenting Software Architectures (second edition) [[Clements 10a](#)] is a comprehensive treatment of the Views and Beyond approach. It describes a multitude of different views and notations for them. It also describes how to package the documentation into a coherent whole.

ISO/IEC/IEEE 42010:2011 (“eye-so-forty-two-ten” for short) is the ISO (and IEEE) standard [[ISO 11](#)] *Systems and software engineering—Architecture description*. The first edition of that standard, IEEE Std. 1471-2000, was developed by an IEEE working group drawing on experience from industry, academia, and other standards bodies between 1995 and 2000. ISO/IEC/IEEE 42010 is centered on two key ideas: a conceptual framework for architecture description and a statement of what information must be found in any ISO/IEC/IEEE 42010-compliant architecture description, using multiple viewpoints driven by stakeholders’ concerns.

Under ISO/IEC/IEEE 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.

If you want to use the Views and Beyond approach to produce an ISO/IEC/IEEE 42010-compliant architecture document, you certainly can. The main additional obligation is to choose and document a set of viewpoints, identifying the stakeholders, their concerns, and the elements catalog for each view, and (to a lesser degree) address ISO/IEC/IEEE 42010’s other required information content.

AADL is an SAE standard. The SAE is an organization for engineering professionals in the aerospace, automotive, and commercial vehicle industries. The website for the AADL standard is at [www.aadl.info](#).

SDL is a notation used in the telecom industry. It is targeted at describing the behavior of reactive and distributed systems in general and telecom systems in particular. A real-time version of SDL can be found at [www.sdl-rt.org/standard/V2.2/pdf/SDL-RT.pdf](#).

UML 2.0 added several features specifically to allow architecture to be modeled, such as ports. It is managed by the Object Management Group and can be found at www.omg.org/spec/UML/.

18.13. Discussion Questions

1. Go to the website of your favorite open source system. On the site, look for the architectural documentation for that system. What is there? What is missing? How would this affect your ability to contribute code to this project?
2. Banks are justifiably cautious about security. Sketch the documentation you would need for an automatic teller machine (ATM) in order to reason about its security architecture.
3. Suppose your company has just purchased another company and that you have been given the task of merging a system in your company with a similar system in the other company. What views of the other system's architecture would you like to see and why? Would you ask for the same views of both systems?
4. 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?
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?
6. Antony Tang, an architect and one of the reviewers of this book, says that he has used a *development view*—a kind of quality view—that describes how the software should be developed in relation to the use of tools and development workflows, the use of standard library routines such as for exception handling, some coding conventions and standards, and some testing and deployment conventions. Sketch a definition of a development view.

19. Architecture, Implementation, and Testing

*You don't make progress by standing on the sidelines, whimpering and complaining.
You make progress by implementing ideas.*

—Shirley Hufstedler

Although this is a book about software architecture—you've noticed that by now, no doubt—we need to remind ourselves from time to time that architecture is not a goal unto itself, but only the means to an end. Building systems from the architecture is the end game, systems that have the qualities necessary to meet the concerns of their stakeholders.

This chapter covers two critical areas in system-building—implementation and testing—from the point of view of architecture. What is the relationship of architecture to implementation (and vice versa)? What is the relationship of architecture to testing (and vice versa)?

19.1. Architecture and Implementation

Architecture is intended to serve as the blueprint for implementation. The sidebar “[Potayto, Potahto . . .](#)” makes the point that architectures and implementations rely on different sets of vocabulary, which results in development tools usually serving one community or the other fairly well, but not both. Frequently the implementers are so engrossed in their immediate task at hand that they make implementation choices that degrade the modular structure of the architecture, for example.

This leads to one of the most frustrating situations for architects. It is very easy for code and its intended architecture to drift apart; this is sometimes called “architecture erosion.” This section talks about four techniques to help keep the code and the architecture consistent.

Embedding the Design in the Code

A key task for implementers is to faithfully execute the prescriptions of the architecture. George Fairbanks, in *Just Enough Architecture*, prescribes using an “architecturally-evident coding style.” Throughout the code, implementers can document the architectural concept or guidance that they’re reifying. That is, they can “embed” the architecture in their implementations. They can also try to localize the implementation of each architectural element, as opposed to scattering it across different implementation entities.

This practice is made easier if implementers (consistently across a project) adopt a set of conventions for how architectural concepts “show up” in code. For example, identifying the layer to which a code unit belongs will make it more likely that implementers and maintainers will respect (and hence not violate) the layering.

Frameworks

“Framework” is a terribly overused term, but here we mean a reusable set of libraries or classes for a software system. “Library” and “class” are implementation-like terms, but frameworks have broad architectural implications—they are a place where architecture and implementation meet. The classes (in an object-oriented framework) are appropriate to the application domain of the system that is being constructed. Frameworks can range from small and straightforward (such as ones that provide a set of standard and commonly used data types to a system) to large and sophisticated. For example, the AUTomotive Open System ARchitecture (AUTOSAR) is a framework for automotive software, jointly developed by automobile manufacturers, suppliers, and tool developers.

Frameworks that are large and sophisticated often encode architectural interaction mechanisms, by encoding how the classes (and the objects derived from them) communicate and synchronize with each other. For example, AUTOSAR is an architecture and not (just) an architecture framework.

A framework amounts to a substantial (in some cases, enormous) piece of reusable software, and it brings with it all of the advantages of reuse: saving time and cost, avoiding a costly design task, encoding domain knowledge, and decreasing the chance of errors from individual implementers coding the same thing differently and erroneously. On the other hand, frameworks are difficult to design and get correct. Adopting a framework means investing in a selection process as well as training, and the framework may not provide all the functionality that you require. The learning curve for a framework is often extremely steep. A framework that provides a complete set of functionality for implementing an application in a particular domain is called a “platform.”

Code Templates

A template provides a structure within which some architecture-specific functionality is achieved, in a consistent fashion system-wide. Many code generators, such as user interface builders, produce a template into which a developer inserts code, although templates can also be provided by the development environment.

Suppose that an architecture for a high-availability system prescribes that every component that implements a critical responsibility must use a failover technique that switches control to a backup copy of itself in case a fault is detected in its operation.

The architecture could, and no doubt would, describe the failover protocol. It might go something like this:

In the event that a failure is detected in a critical-application component, a switchover occurs as follows:

1. A secondary copy, executing in parallel in background on a different processor, is promoted to the new primary.
2. The new primary reconstitutes with the application's clients by sending them a message that means, essentially: The operational unit that was serving you has had a failure. Were you waiting for anything from us at the time? It then proceeds to service any requests received in response.
3. A new secondary is started to serve as a backup for the new primary.
4. The newly started secondary announces itself to the new primary, which starts sending it messages as appropriate to keep it up to date while it is executing in background.

If failure is detected within a secondary, a new one is started on some other processor. It coordinates with its primary and starts receiving state data.

Even though the primary and secondary copies are never doing the same thing at the same time (the primary is performing its duty and sending state updates to its backups, and the secondaries are waiting to leap into action and accepting state updates), both components come from identical copies of the same source code.

To accomplish this, the coders of each critical component would be expected to implement that protocol. However, a cleverer way is to give the coder a code template that contains the tricky failover part as boilerplate and contains fill-in-the-blank sections where coders can fill in the implementation for the functionality that is unique to each application. This template could be embedded in the development environment so that when the developer specifies that the module being developed is to support a failover protocol, the template appears as the initial code for the module.

An example of such a template, taken from an air traffic control system, is illustrated in [Figure 19.1](#). The structure is a continuous loop that services incoming events. If the event is one that causes the application to take a normal (non-fault-tolerance-related) action, it carries out the appropriate action, followed by an update of its backup counterparts' data so that the counterpart can take over if necessary. Most applications spend most of their time processing normal events. Other events that may be received involve the transfer (transmission and reception) of state and data updates. Finally, there is a set of events that involves both the announcement that this unit has become the primary and requests from clients for services that the former (now failed) primary did not complete.

```
terminate:= false
initialize application/application protocols
ask for current state (image request)
Loop
Get_event
Case Event_Type is
-- "normal" (non-fault-tolerant-related) requests to
-- perform actions; only happens if this unit is the
-- current primary address space
when X => Process X
Send state data updates to other address spaces
when Y => Process Y
Send state data updates to other address spaces
...
when Terminate Directive => clean up resources; terminate
    := true
when State_Data_Update => apply to state data
-- will only happen if this unit is a secondary address
-- space, receiving the update from the primary after it
-- has completed a "normal" action sending, receiving
-- state data
when Image_Request => send current state data to new
    address space
when State_Data_Image => Initialize state data
when Switch_Directive => notify service packages of
```

```

change in rank
-- these are requests that come in after a PAS/SAS
-- switchover; they report services that they had
-- requested from the old (failed) PAS which this unit
-- (now the PAS) must complete. A, B, etc. are the names
-- of the clients.
when Recon_from_A => reconstitute A
when Recon_from_B => reconstitute B
...
when others => log error
end case
exit when terminate
end loop

```

Figure 19.1. A code template for a failover protocol. “Process X” and “Process Y” are placeholders for application-specific code.

Using a template has architectural implications: it makes it simple to add new applications to the system with a minimum of concern for the actual workings of the fault-tolerant mechanisms designed into the approach. Coders and maintainers of applications do not need to know about message-handling mechanisms except abstractly, and they do not need to ensure that their applications are fault tolerant—that has been handled architecturally.

Code templates have implications for reliability: once the template is debugged, then entire classes of coding errors across the entire system disappear. But in the context of this discussion, templates represent a true common ground where the architecture and the implementation come together in a consistent and useful fashion.

Keeping Code and Architecture Consistent

Code can drift away from architecture in a depressingly large number of ways. First, there may be no constraints imposed on the coders to follow the architecture. This makes no apparent sense, for why would we bother to invest in an architecture if we aren’t going to use it to constrain the code? However, this happens more often than you might think. Second, some projects use the published architecture to start out, but when problems are encountered (either technical or schedule-related), the architecture is abandoned and coders scramble to field the system as best they can. Third (and perhaps most common), after the system has been fielded, changes to it are accomplished with code changes only, but these changes affect the architecture. However, the published architecture is not updated to guide the changes, nor updated afterward to keep up with them.

One simple method to remedy the lack of updating the architecture is to not treat the published architecture as an all-or-nothing affair—it’s either all correct or all useless. Parts of the architecture may become out of date, but it will help enormously if those parts are marked as “no longer applicable” or “to be revised.” Conscientiously marking sections as out of date keeps the architecture documentation a living document and (paradoxically) sends a stronger message about the remainder: it is still correct and can still be trusted.

In addition, strong management and process discipline will help prevent erosion. One way is to mandate that changes to the system, no matter when they occur, are vetted through the architecture first. The alternatives for achieving code alignment with the architecture include the following:

- *Sync at life-cycle milestone.* Developers change the code until the end of some phase, such as a release or end of an iteration. At that point, when the schedule pressure is less, the architecture is updated.
- *Sync at crisis.* This undesirable approach happens when a project has found itself in a technical quagmire and needs architectural guidance to get itself going again.
- *Sync at check-in.* Rules for the architecture are codified and used to vet any check-in. When a change to the code “breaks” the architecture rules, key project stakeholders are informed and then either the code or the architecture rules must be modified. This process is typically automated by tools.

These alternatives can work only if the implementation follows the architecture mostly, departing from it only here and there and in small ways. That is, it works when syncing the architecture involves an update and not a wholesale overhaul or do-over.

Potayto, Potahto, Tomayto, Tomahto—Let's Call the Whole Thing Off!

One of the most vexing realities about architecture-based software development is the gulf between architectural and implementation *ontologies*, the set of concepts and terms inherent in an area. Ask an architect what concepts they work with all day, and you're likely to hear things like modules, components, connectors, stakeholders, evaluation, analysis, documentation, views, modeling, quality attributes, business goals, and technology roadmaps.

Ask an implementer the same question, and you likely won't hear any of those words. Instead you'll hear about objects, methods, algorithms, data structures, variables, debugging, statements, code comments, compilers, generics, operator overloading, pointers, and build scripts.

This is a gap in language that reflects a gap in concepts. This gap is, in turn, reflected in the languages of the tools that each community uses. UML started out as a way to model object-oriented designs that could be quickly converted to code—that is, UML is conceptually “close” to code. Today it is a *de facto* architecture description language, and likely the most popular one. But it has no built-in concept for the most ubiquitous of architectural concepts, the layer. If you want to represent layers in UML, you have to adopt some convention to do it. Packages stereotyped as <<layer>>, associated with stereotyped <<allowed to use>> dependencies do the trick. But it *is* a trick, a workaround for a language deficiency. UML has “connectors,” two of them in fact. But they are a far cry from what architects think of as connectors. Architectural connectors can and do have rich functionality. For instance, an enterprise service bus (ESB) in a service-oriented architecture handles routing, data and format transformation, technology adaptation, and a host of other work. It is most natural to depict the ESB as a connector tying together services that interact with each other through it. But UML connectors are impoverished things, little more than bookkeeping mechanisms that have no functionality whatsoever. The delegation connector in UML exists merely to associate the ports of a parent component with ports of its nested children, to send inputs from the outside into a child’s input port, and outputs from a child to the output port of the parent. And the assembly connector simply ties together one component’s “requires” interface with another’s “provides” interface. These are no more than bits of string to tie two components together. To represent a true architectural connector in UML, you have to adopt a convention—another workaround—such as using simple associations tagged with explanatory annotations, or abandon the architectural concept completely and capture the functionality in another component.

Part of the concept gap between architecture and implementation is inevitable. Architectures, after all, are abstractions of systems and their implementations. Back in [Chapter 2](#), we said that was one of the valuable properties of architecture: you could build many different systems from one. And that’s what an abstraction is: a one-to-many mapping. One abstraction, many instances; one architecture, many implementations. That architecture is an abstraction of implementation is almost its whole point: architecture lets us achieve intellectual control over a system without having to capture, let alone master, all of the countless and myriad truths about its implementation.

And here comes the gap again: All of those truths about its implementation are what coders produce for a living, without which the system remains but an idea. Architects, on the other hand, dismiss all of that reality by announcing that they are not interested in implementation “details.”

Can’t we all get along?

We could. There is nothing inherently impossible about a language that embraces architectural as well as coding concepts, and several people have proposed some. But UML is beastly difficult to change, and programming language purveyors all seem to focus their attention down on the underlying machine and not up to the architecture that is directing the implementation.

Until this gap is resolved, until architects and coders (and their tools) speak the same conceptual language, we are likely to continue to deal with the most vexing result of this most vexing reality: writing code (or introducing a code change) that ignores the architecture is the easiest thing in the world.

The good news is that even though architecture and implementation speak different languages, they aren't languages from different planets. Concepts in one ontology usually correspond pretty well to concepts in another. Frameworks are an area where the languages enjoy a fair amount of overlap. So are interfaces. These constructs live on the cusp of the two domains, and provide hope that we might one day speak the same language.

—PCC

19.2. Architecture and Testing

What is the relationship between architecture and testing? One possible answer is "None," or "Not much." Testing can be seen as the process of making sure that a software system meets its requirements, that it brings the necessary functionality (endowed with the necessary quality attributes) to its user community. Testing, seen this way, is simply connected to requirements, and hardly connected to architecture at all. As long as the system works as expected, who cares what the architecture is? Yes, the architecture played the leading role in *getting* the system to work as expected, thank you very much, but once it has played that role it should make a graceful exit off the stage. Testers work with requirements: Thanks, architecture, but we'll take it from here.

Not surprisingly, we don't like that answer. This is an impoverished view of testing, and in fact an unrealistic one as well. As we'll see, architecture *cannot help* but play an important role in testing. Beyond that, though, we'll see that architecture can help make testing less costly and more effective when embraced in testing activities. We'll also see what architects can do to help testers, and what testers can do to take advantage of the architecture.

Levels of Testing and How Architecture Plays a Role in Each

There are "levels" of testing, which range from testing small, individual pieces in isolation to an entire system.

- *Unit testing* refers to tests run on specific pieces of software. Unit testing is usually a part of the job of implementing those pieces. In fact, unit tests are typically written by developers themselves. When the tests are written before developing the unit, this practice is known as test-driven development.

Certifying that a unit has passed its unit tests is a precondition for delivery of that unit to integration activities. Unit tests test the software in a standalone fashion, often relying on "stubs" to play the role of other units with which the tested unit interacts, as those other units may not yet be available. Unit tests won't usually catch errors dealing with the interaction between elements—that comes later—but unit tests provide confidence that each of the system's building blocks is exhibiting as much correctness as is possible on its own.

A unit corresponds to an architectural element in one of the architecture's module views. In object-oriented software, a unit might correspond to a class. In a layered system, a unit might correspond to a layer, or a part of a layer. Most often a unit corresponds to an element at the leaf of a module decomposition tree.

Architecture plays a strong role in unit testing. First, it defines the units: they are architectural elements in one or more of the module views. Second, it defines the responsibilities and requirements assigned to each unit.

Modifiability requirements can also be tested at unit test time. How long it will take to make specified changes can be tested, although this is seldom done in practice. If specified changes take too long for the developers to make, imagine how long they will take when a new and separate maintenance group is in charge without the intimate knowledge of the modules.

Although unit testing goes beyond architecture (tests are based on nonarchitectural information such as the unit's internal data structures, algorithms, and control flows), they cannot begin their work without the architecture.

- *Integration testing* tests what happens when separate software units start to work together. Integration testing concentrates on finding problems related to the interfaces between elements in a design. Integration testing is intimately connected to the specific increments or subsets that are planned in a system's development.

The case where only one increment is planned, meaning that integration of the entire system will occur in a single step, is called "big bang integration" and has largely been discredited in favor of integrating many incrementally larger subsets. Incremental integration makes locating errors much easier, because any new error that shows up in an integrated subset is likely to live in whatever new parts were added this time around.

At the end of integration testing, the project has confidence that the pieces of software work together correctly and provide at least some correct system-wide functionality (depending on how big a subset of the system is being integrated). Special cases of integration testing are these:

- System testing, which is a test of all elements of the system, including software and hardware in their intended environment
- Integration testing that involves third-party software

Once again, architecture cannot help but play a strong role in integration testing. First, the increments that will be subject to integration testing must be planned, and this plan will be based on the architecture. The uses view is particularly helpful for this, as it shows what elements must be present for a particular piece of functionality to be fielded. That is, if the project requires that (for example) in the next increment of a social networking system users will be able to manage photographs they've allowed other users to post in their own member spaces, the architect can report that this new functionality is part of the `user_permissions` module, which will use a new part of the `photo_sharing` module, which in turn will use a new structure in the master `user_links` database, and so forth. Project management will know, then, that all of the software must be ready for integration at the same time.

Second, the interfaces between elements are part of the architecture, and those interfaces determine the integration tests that are created and run.

Integration testing is where runtime quality attribute requirements can be tested. Performance and reliability testing can be accomplished. A sophisticated test harness is useful for performing these types of tests. How long does an end-to-end synchronization of a local database with a global database take? What happens if faults are injected into the system? What happens when a process fails? All of these conditions can be tested at integration time.

Integration testing is also the time to test what happens when the system runs for an extended period. You could monitor resource usage during the testing and look for resources that are consumed but not freed. Does your pool of free database connections decrease over time? Then maybe database connections should be managed more aggressively. Does the thread pool show signs of degradation over time? Ditto.

- *Acceptance testing* is a kind of system testing that is performed by users, often in the setting in which the system will run. Two special cases of acceptance testing are alpha and beta testing. In both of these, users are given free rein to use the system however they like, as opposed to testing that occurs under a preplanned regimen of a specific suite of tests. Alpha testing usually occurs in-house, whereas beta testing makes the system available to a select set of end users under a "User beware" proviso. Systems in beta test are generally quite reliable—after all, the developing organization is highly motivated to make a good first impression on the user community—but users are given fair warning that the system might not be bug-free or (if "bug-free" is too lofty a goal) at least not up to its planned quality level.

Architecture plays less of a role in acceptance testing than at the other levels, but still an important one. Acceptance testing involves stressing the system's quality attribute behavior by running it at extremely heavy loads, subjecting it to security attacks, depriving it of resources at critical times, and so forth. A crude analogy is that if you want to bring down a house, you can whale away at random walls with a sledgehammer, but

your task will be accomplished much more efficiently if you consult the architecture first to find which of the walls is holding up the roof. (The point of testing is, after all, to "bring down the house.")

Overlaying all of these types of testing is regression testing, which is testing that occurs after a change has been made to the system. The name comes from the desire to uncover old bugs that might resurface after a change, a sign that the software has "regressed" to a less mature state. Regression testing can occur at any of the previously mentioned levels, and often consists of rerunning the bank of tests and checking for the occurrence of old (or for that matter, new) faults.

Black-Box and White-Box Testing

Testing (at any level) can be "black box" or "white box." Black-box testing treats the software as an opaque "black box," not using any knowledge about the internal design, structure, or implementation. The tester's only source of information about the software is its requirements.

Architecture plays a role in black-box testing, because it is often the architecture document where the requirements for a piece of the system are described. An element of the architecture is unlikely to correspond one-to-one with a requirement nicely captured in a requirements document. Rather, when the architect creates an architectural element, he or she usually assigns it an amalgamation of requirements, or partial requirements, to carry out. In addition, the interface to an element also constitutes a set of "requirements" for it—the element must happily accept the specified parameters and produce the specified effect as a result. Testers performing black-box testing on an architectural element (such as a major subsystem) are unlikely to be able to do their jobs using only requirements published in a requirements document. They need the architecture as well, because the architecture will help the tester understand what portions of the requirements relate to the specified subsystem.

White-box testing makes full use of the internal structures, algorithms, and control and data flows of a unit of software. Tests that exercise all control paths of a unit of software are a primary example of white-box testing. White-box testing is most often associated with unit testing, but it has a role at higher levels as well. In integration testing, for example, white-box testing can be used to construct tests that attempt to overload the connection between two components by exploiting knowledge about how a component (for example) manages multiple simultaneous interactions.

Gray-box testing lies, as you would expect, between black and white. Testers get to avail themselves of some, but not all, of the internal structure of a system. For example, they can test the interactions between components but not employ tests based on knowledge of a component's internal data structures.

There are advantages and disadvantages with each kind of testing. Black-box testing is not biased by a design or implementation, and it concentrates on making sure that requirements are met. But it can be inefficient by (for example) running many unit tests that a simple code inspection would reveal to be unnecessary. White-box testing often keys in on critical errors more quickly, but it can suffer from a loss of perspective by concentrating tests to make the implementation break, but not concentrating on the software delivering full functionality under all points in its input space.

Risk-based Testing

Risk-based testing concentrates effort on areas where risk is perceived to be the highest, perhaps because of immature technologies, requirements uncertainty, developer experience gaps, and so forth. Architecture can inform risk-based testing by contributing categories of risks to be considered. Architects can identify areas where architectural decisions (if wrong) would have a widespread impact, where architectural requirements are uncertain, quality attributes are demanding on the architecture, technology selections risky, or third-party software sources unreliable. Architecturally significant requirements are natural candidates for risk-based test cases. If the architecturally significant requirements are not met, then the system is unacceptable, by definition.

Test Activities

Testing, depending on the project, can consume from 30 to 90 percent of a development's schedule and budget. Any activity that gobbles resources as voraciously as that doesn't just

happen, of course, but needs to be planned and carried out purposefully and as efficiently as possible. Here are some of the activities associated with testing:

- *Test planning.* Test activities have to be planned so that appropriate resources can be allocated. "Resources" includes time in the project schedule, labor to run the tests, and technology with which the testing will be carried out. Technology might include test tools, automatic regression testers, test script builders, test beds, test equipment or hardware such as network sniffers, and so forth.
- *Test development.* This is an activity in which the test procedures are written, test cases are chosen, test datasets are created, and test suites are scripted. The tests can be developed either before or after development. Developing the tests prior to development and then developing a module to satisfy the test is a characteristic of test-first development.
- *Test execution.* Here, testers apply the tests to the software and capture and record errors.
- *Test reporting and defect analysis.* Testers report the results of specific tests to developers, and they report overall metrics about the test results to the project's technical management. The analysis might include a judgment about whether the software is ready for release. Defect analysis is done by the development team usually along with the customer, to adjudicate disposition of each discovered fault: fix it now, fix it later, don't worry about it, and so on.
- *Test harness creation.* One of the architect's common responsibilities is to create, along with the architecture, a set of test harnesses through which elements of the architecture may be conveniently tested. Such test harnesses typically permit setting up the environment for the elements to be tested, along with controlling their state and the data flowing into and out of the elements.

Once again, architecture plays a role and informs each of these activities; the architect can contribute useful information and suggestions for each. For test planning, the architecture provides the list of software units and incremental subsets. The architect can also provide insight as to the complexity or, if the software does not yet exist, the expected complexity of each of the software units. The architect can also suggest useful test technologies that will be compatible with the architecture; for example, Java's ability to support assertions in the code can dramatically increase software testability, and the architect can provide arguments for or against adopting that technology. For test development, the architecture can make it easy to swap datasets in and out of the system. Finally, test reporting and defect analysis are usually reported in architectural terms: *this* element passed all of its tests, but *that* element still has critical errors showing. *This* layer passed the delivery test, but *that* layer didn't. And so forth.

The Architect's Role

Here are some of the things an architect can do to facilitate quality testing. First and foremost, the architect can design the system so that it is highly testable. That is, the system should be designed with the quality attribute of testability in mind. Applying the cardinal rule of architecture ("Know your stakeholders!"), the architect can work with the test team (and, to the extent they have a stake in testing, other stakeholders) to establish what is needed. Together, they can come up with a definition of the testability requirements using scenarios, as described in [Chapter 10](#). Testability requirements are most likely to be a concern of the developing organization and not so much of the customer or users, so don't expect to see many testing requirements in a requirements document. Using those testability requirements, the testability tactics in [Chapter 10](#) can be brought to bear to provide the testability needed.

In addition to designing for testability, the architect can also do these other things to help the test effort:

- Insure that testers have access to the source code, design documents, and the change records.
- Give testers the ability to control and reset the entire dataset that a program stores in a persistent database. Reverting the database to a known state is essential for reproducing bugs or running regression tests. Similarly, loading a test bed into the database is helpful. Even products that don't use databases can benefit from routines to automatically preload a set of test data. One way to achieve this is to design a "persistence layer" so that the

whole program is database independent. In this way, the entire database can be swapped out for testing, even using an in-memory database if desired.

- Give testers the ability to install multiple versions of a software product on a single machine. This helps testers compare versions, isolating when a bug was introduced. In distributed applications, this aids testing deployment configurations and product scalability. This capability could require configurable communication ports and provisions for avoiding collisions over resources such as the registry.

As a practical matter, the architect cannot afford to ignore the testing process because if, after delivery, something goes seriously wrong, the architect will be one of the first people brought in to diagnose the problem. In one case we heard about, this involved flying to the remote mountains of Peru to diagnose a problem with mining equipment.

19.3. Summary

Architecture plays a key role in both implementation and testing. In the implementation phase, letting future readers of the code know what architectural constructs are being used, using frameworks, and using code templates all make life easier both at implementation time and during maintenance.

During testing the architecture determines what is being tested at which stage of development. Development quality attributes can be tested during unit test and runtime quality attributes can be tested during integration testing.

Testing, as with other activities in architecture-based development, is a cost/benefit activity. Do not spend as much time testing for faults whose consequences are small and spend the most time testing for faults whose consequences are serious. Do not neglect testing for faults that show up after the system has been executing for an extended period.

19.4. For Further Reading

George Fairbanks gives an excellent treatment of architecture and implementation in [Chapter 10](#) of his book *Just Enough Software Architecture*, which is entitled "The Code Model" [[Fairbanks 10](#)].

Mary Shaw long ago recognized the conceptual gap between architecture and implementation and wrote about it eloquently in her article "Procedure Calls Are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status" [[Shaw 94](#)]. In it she pointed out the disparity between rich connectors available in architecture and the impoverished subroutine call that is the mainstay of nearly every programming language.

Details about the AUTOSAR framework can be found at www.autosar.org.

Architecture-based testing is an active field of research. [[Bertolino 96b](#)], [[Muccini 07](#)], [[Muccini 03](#)], [[Eickelman 96](#)], [[Pettichord 02](#)], and [[Binder 94](#)] specifically address designing systems so that they are more testable. In fact, the three bullets concerning the architect's role in [Section 19.2](#) are drawn from Pettichord's work.

Voas [[Voas 95](#)] defines testability, identifies its contributing factors, and describes how to measure it.

Bertolino extends Voas's work and ties testability to dependability [[Bertolino 96](#)].

Finally, Baudry et al. have written an interesting paper that examines the testability of well-known design patterns [[Baudry 03](#)].

19.5. Discussion Questions

1. In a distributed system each computer will have its own clock. It is difficult to perfectly synchronize those clocks. How will this complicate making performance measures of distributed systems? How would you go about testing that the performance of a particular system activity is adequate?

- 2.** Plan and implement a modification to a module. Ask your colleagues to do the same modification independently. Now compare your results to those of your colleagues. What is the mean and the standard deviation for the time it takes to make that modification?
- 3.** List some of the reasons why an architecture and a code base inevitably drift apart. What processes and tools might address this gap? What are their costs and benefits?
- 4.** Most user interface frameworks work by capturing events from the user and by establishing callbacks or hooks to application-specific functionality. What limitations do these architectural assumptions impose on the rest of the system?
- 5.** Consider building a test harness for a large system. What quality attributes should this harness exhibit? Create scenarios to concretize each of the quality attributes.
- 6.** Testing requires the presence of a test oracle, which determines the success (or failure) of a test. For scalability reasons, the oracle must be automatic. How can you ensure that your oracle is correct? How do you ensure that its performance will scale appropriately? What process would you use to record and fix faults in the testing infrastructure?
- 7.** In embedded systems faults often occur “in the field” and it is difficult to capture and replicate the state of the system that led to its failure. What architectural mechanisms might you use to solve this problem?
- 8.** In integration testing it is a bad idea to integrate everything all at once (big bang integration). How would you use architecture to help you plan integration increments?

20. Architecture Reconstruction and Conformance

It was six men of Indostan / To learning much inclined, Who went to see the Elephant / (Though all of them were blind), That each by observation / Might satisfy his mind.

The First approach'd the Elephant, / And happening to fall Against his broad and sturdy side, / At once began to bawl: "God bless me! but the Elephant / Is very like a wall!"

The Second, feeling of the tusk, / Cried, — "Ho! what have we here So very round and smooth and sharp? / To me 'tis mighty clear This wonder of an Elephant / Is very like a spear!"

The Third approached the animal, / And happening to take The squirming trunk within his hands, / Thus boldly up and spake: "I see," quoth he, "the Elephant / Is very like a snake!"

The Fourth reached out his eager hand, / And felt about the knee. "What most this wondrous beast is like / Is mighty plain," quoth he, "Tis clear enough the Elephant / Is very like a tree!"

The Fifth, who chanced to touch the ear, / Said: "E'en the blindest man Can tell what this resembles most; / Deny the fact who can, This marvel of an Elephant / Is very like a fan!"

The Sixth no sooner had begun / About the beast to grope, Then, seizing on the swinging tail / That fell within his scope, "I see," quoth he, "the Elephant / Is very like a rope!"

And so these men of Indostan / Disputed loud and long, Each in his own opinion / Exceeding stiff and strong, Though each was partly in the right, / And all were in the wrong!

—“The Blind Men and the Elephant,” by John Godfrey Saxe

Throughout this book we have treated architecture as something largely under your control and shown how to make architectural decisions to achieve the goals and requirements in place for a system under development. But there is another side to the picture. Suppose you have been given responsibility for a system that already exists, but you do not know its architecture. Perhaps the architecture was never recorded by the original developers, now long gone. Perhaps it was recorded but the documentation has been lost. Or perhaps it was recorded but the documentation is no longer synchronized with the system after a series of changes. How do you maintain such a system? How do you manage its evolution to maintain the quality attributes that its architecture (whatever it may be) has provided for us?

This chapter surveys techniques that allow an analyst to build, maintain, and understand a representation of an existing architecture. This is a process of reverse engineering, typically called architecture reconstruction. Architecture reconstruction is used, by the architect, for two main purposes:

- To document an architecture where the documentation never existed or where it has become hopelessly out of date
- To ensure conformance between the as-built architecture and the as-designed architecture.

In architecture reconstruction, the “as-built” architecture of an implemented system is reverse-engineered from existing system artifacts.

When a system is initially developed, its architectural elements are mapped to specific implementation elements: functions, classes, files, objects, and so forth. This is forward engineering. When we reconstruct those architectural elements, we need to apply the inverses of the original mappings. But how do we go about determining these mappings? One way is to use automated and semiautomated extraction tools; the second way is to probe the original design intent of the architect. Typically we use a combination of both techniques in reconstructing an architecture.

In practice, architecture reconstruction is a tool-intensive activity. Tools extract information about the system, typically by scouring the source code, but they may also analyze other artifacts as well, such as build scripts or traces from running systems. But architectures are abstractions—they can not be seen in the low-level implementation details, the programming constructs, of most systems. So we need tools that aid in building and aggregating the abstractions that we need, as architects, on top of the ground facts that we develop, as developers. If our tools are usable and accurate, the end result is an architectural representation that aids the architect in reasoning about the system. Of course, if the original architecture and its implementation are “spaghetti,” the reconstruction will faithfully expose this lack of organization.

Architecture reconstruction tools are not, however, a panacea. In some cases, it may not be possible to generate a useful architectural representation. Furthermore, not all aspects of architecture are easy to automatically extract. Consider this: there is no programming language construct in any major programming language for “layer” or “connector” or other architectural elements; we can’t simply pick these out of a source code file. Similarly, architectural patterns, if used, are typically not explicitly documented in code.

Architecture reconstruction is an interpretive, interactive, and iterative process involving many activities; it is not automatic. It requires the skills and attention of both the reverse-engineering expert and, in the best case, the architect (or someone who has substantial knowledge of the architecture). And whether the reconstruction is successful or not, there is a price to pay: the tools come with a learning curve that requires time to climb.

20.1. Architecture Reconstruction Process

Architecture reconstruction requires the skillful application of tools, often with a steep learning curve. No single tool does the entire job. For one reason, there is often diversity in the number of implementation languages and dialects in which a software system is implemented—a mature MRI scanner or a legacy banking application may easily comprise more than ten different programming and scripting languages. No tool speaks every language.

Instead we are inevitably led to a “tool set” approach to support architecture reconstruction activities. And so the first step in the reconstruction process is to set up the workbench.

An architecture reconstruction workbench should be open (making it easy to integrate new tools as required) and provide an integration framework whereby new tools that are added to the tool set do not impact the existing tools or data unnecessarily.

Whether or not an explicit workbench is used, the software architecture reconstruction process comprises the following phases (each elaborated in a subsequent section):

1. *Raw view extraction.* In the raw view extraction phase, raw information about the architecture is obtained from various sources, primarily source code, execution traces, and build scripts. Each of these sets of raw information is called a view.¹
1. This use of the term “view” is consistent with our definition in [Chapter 18](#): “a representation of a set of system elements and relations among them.”
2. *Database construction.* The database construction phase involves converting the raw extracted information into a standard form (because the various extraction tools may each produce their own form of output). This standardized form of the extracted views is then used to populate a reconstruction database. When the reconstruction process is complete, the database will be used to generate authoritative architecture documentation.
3. *View fusion and manipulation.* The view fusion phase combines the various views of the information stored in the database. Individual views may not contain complete or fully accurate information. View fusion can improve the overall accuracy. For example, a static view extracted from source code might miss dynamically bound information such as calling relationships. This could then be combined with a dynamic view from an execution trace, which will capture all dynamically bound calling information, but which may not provide complete coverage. The combination of these views will provide higher quality information than either could provide alone. Furthermore, view creation and fusion is typically associated with some expert interpretation and manipulation. For example, an expert might decide that a group of elements should be aggregated together to form a *layer*.
4. *Architecture analysis.* View fusion will result in a set of *hypotheses* about the architecture. These hypotheses take the form of architectural elements (such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct, and that is the function of the analysis step. Some of these hypotheses might be disproven, requiring additional view extraction, fusion, and manipulation.

The four phases of architecture reconstruction are iterative. [Figure 20.1](#) depicts the major tasks of architecture reconstruction and their relationships and outputs. Solid lines represent data flow and dashed lines represent human interaction.

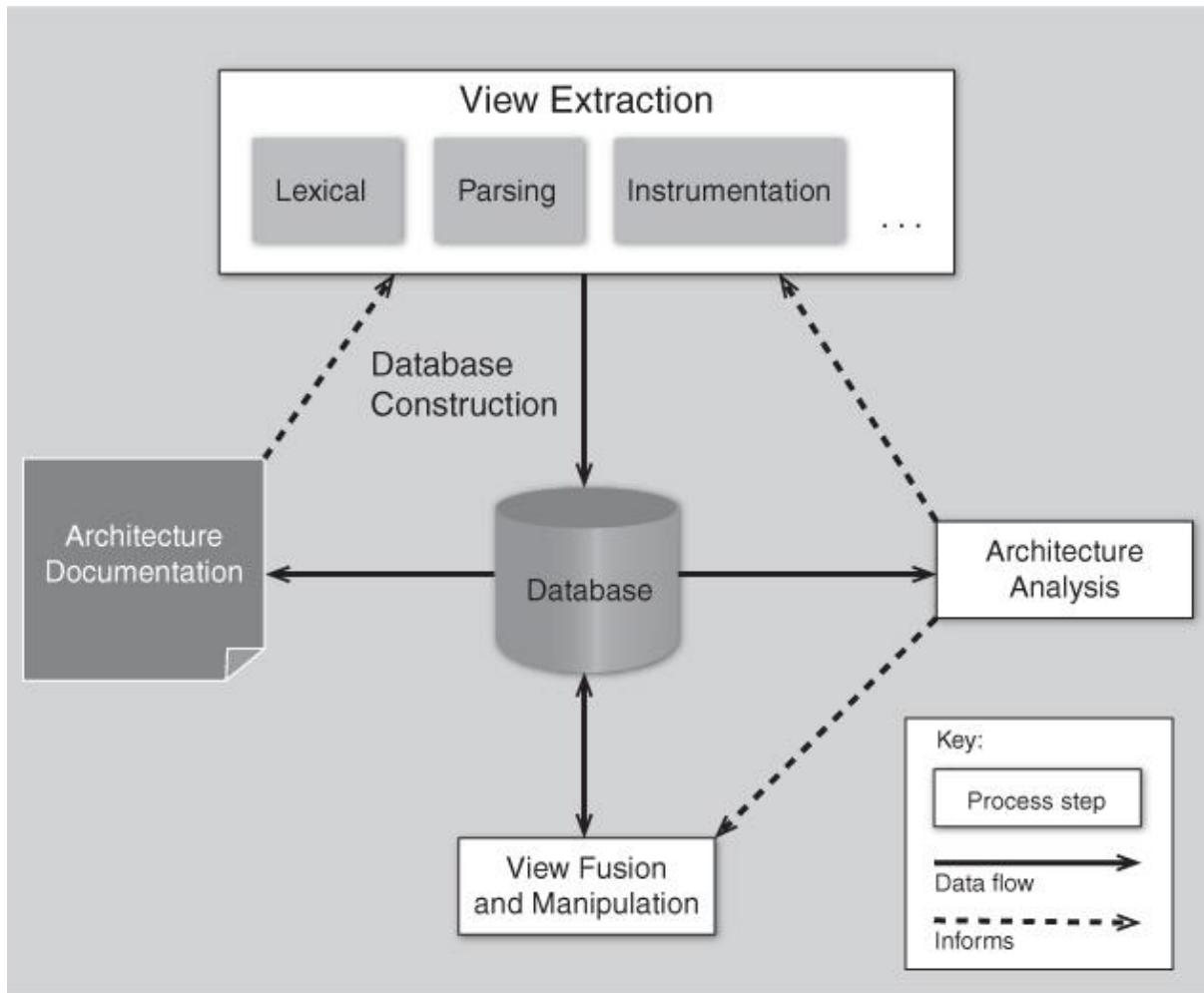


Figure 20.1. Architecture reconstruction process

All of these activities are greatly facilitated by engaging people who are familiar with the system. They can provide insights about what to look for—that is, what views are amenable to extraction—and provide a guided approach to view fusion and analysis. They can also point out or explain exceptions to the design rules (which will show up as violations of the hypotheses during the analysis phase). If the experts are long gone, reconstruction is still possible, but it may well require more backtracking from incorrect initial guesses.

20.2. Raw View Extraction

Raw view extraction involves analyzing a system's existing design and implementation artifacts to construct one or more models of it. The result is a set of information that is used in the view fusion activity to construct more-refined views of the system that directly support the goals of the reconstruction, goals such as these:

- Extracting and representing a target set of architectural views, to support the overall architecture documentation effort.
- Answering specific questions about the architecture. For example, “What components are potentially affected if I choose to rewrite component X?” or “How can I refactor my layering to remove cyclic dependencies?”

The raw view extraction process is a blend of the ideal (what information do you want to discover about the architecture that will most help you meet the goals of your reconstruction effort?) and the practical (what information can your available tools actually extract and present?).

From the source artifacts (code, header files, build files, and so on) and other artifacts (e.g., execution traces), you can identify and capture the elements of interest within the system (e.g., files, functions, variables) and their relationships to obtain several base system views. [Table 20.1](#) shows a typical list of the elements and several relationships among them that might be extracted.

Table 20.1. Examples of Extracted Elements and Relations

Source Element	Relation	Target Element	Description
File	includes	File	C preprocessor #include of one file by another
File	contains	Function	Definition of a function in a file
File	defines _ var	Variable	Definition of a variable in a file
Directory	contains	Directory	Directory contains a subdirectory
Directory	contains	File	Directory contains a file
Function	calls	Function	Static function call
Function	access _ read	Variable	Read access on a variable
Function	access _ write	Variable	Write access on a variable

Each of the relationships between the elements gives different information about the system:

- The `calls` relationship between functions helps us build a call graph.
- The `includes` relationship between the files gives us a set of dependencies between system files.
- The `access_read` and `access_write` relationships between functions and variables show us how data is used. Certain functions may write a set of data and others may read it. This information is used to determine how data is passed between various parts of the system. We can determine whether or not a global data store is used or whether most information is passed through function calls.
- Certain elements or subsystems may be stored in particular directories, and capturing relations such as `dir_contains_file` and `dir_contains_dir` is useful when trying to identify elements later.
- If the system to be reconstructed is object oriented, classes and methods are added to the list of elements to be extracted, and relationships such as `class_is_subclass_of_class` and `class_contains_method` are extracted and used.

Information obtained can be categorized as either static or dynamic. Static information is obtained by observing only the system artifacts, while dynamic information is obtained by observing how the system runs. The goal is to fuse both to create more accurate system views.

If the architecture of the system changes at runtime, that runtime configuration should be captured and used when carrying out the reconstruction. For example, in some systems a configuration file is read in by the system at startup, or a newly started system examines its operating environment, and certain elements are executed or connections are made as a result.

Another reason to capture dynamic information is that some architecturally relevant information may not exist in the source artifacts because of late binding. Examples of late binding include the following:

- Polymorphism
- Function pointers
- Runtime parameterization
- Plug-ins

- Service interactions mediated by brokers

Further, the precise topology of a system may not be determined until runtime. For example, in peer-to-peer systems, service-oriented architectures, and cloud computing, the topology of the system is established dynamically, depending on the availability, loading, and even dynamic pricing of system resources. The topology of such systems cannot be directly recovered from their source artifacts and hence cannot be reverse-engineered using static extraction tools.

Therefore, it may be necessary to use tools that can generate dynamic information about the system (e.g., profiling tools, instrumentation that generates runtime traces, or aspects in an aspect-oriented programming language that can monitor dynamic activity). Of course, this requires that such tools be available on the platforms on which the system executes. Also, it may be difficult to collect the results from code instrumentation. For example, embedded systems often have no direct way to output such information.

[Table 20.2](#) summarizes some of the common categories of tools that might be used to populate the views loaded into the reconstruction database.

Table 20.2. Tool Categories for Populating Reconstructed Architecture Views

Tool	Static or Dynamic	Description
Parsers		Parsers analyze the code and generate internal representations from it (for the purpose of generating machine code). It is possible to save this internal representation to obtain a view.
Abstract Syntax Tree (AST) Analyzers		AST analyzers do a similar job to parsers, but they build an explicit tree representation of the parsed information. We can build analysis tools that traverse the AST and output selected pieces of architecturally relevant information in an appropriate format.
Lexical Analyzers	Static	Lexical analyzers examine source artifacts purely as strings of lexical elements or tokens. The user of a lexical analyzer can specify a set of code patterns to be matched and output. Similarly, a collection of ad hoc tools such as grep and Perl can carry out pattern matching and searching within the code to output some required information. All of these tools—code-generating parsers, AST-based analyzers, lexical analyzers, and ad hoc pattern matchers—are used to output static information.

Profilers		Profiling and code coverage analysis tools can be used to output information about the code as it is being executed, and usually do not involve adding new code to the system.
Code Instrumentation Tools	Dynamic	Code instrumentation, which has wide applicability in the field of testing, involves adding code to the system to output specific information while the system is executing. Aspects, in an aspect-oriented programming language, can serve the same purpose and have the advantage of keeping the instrumentation code separate from the code being monitored.

Tools to analyze design models, build files, and executables can also be used to extract further information as required. For instance, build files include information on module or file dependencies that exist within the system, and this information may not be reflected in the source code, or anywhere else.

An additional activity that is often required prior to loading a raw view into the database is to prune irrelevant information. For example, in a C code base there may be several `main()` routines, but only one of those (and its resulting call graph) will be of concern for analysis. The others may be for test harnesses and other utility functions. Similarly if you are building or using libraries that are operating-system specific, you may only be interested in a specific OS (e.g., Linux) and thus want to discard the libraries for other platforms.

20.3. Database Construction

Some of the information extracted from the raw view extraction phase, while necessary for the process of reconstruction, may be too specific to aid in architectural understanding. Consider [Figure 20.2](#). In this figure we show a set of facts extracted from a code base consisting of classes and methods, and inclusion and calling relations. Each element is plotted on a grid and each relation is drawn as a line between the elements. This view, while accurate, provides no insight into the overarching abstractions or coarse-grained structures present in the architecture.

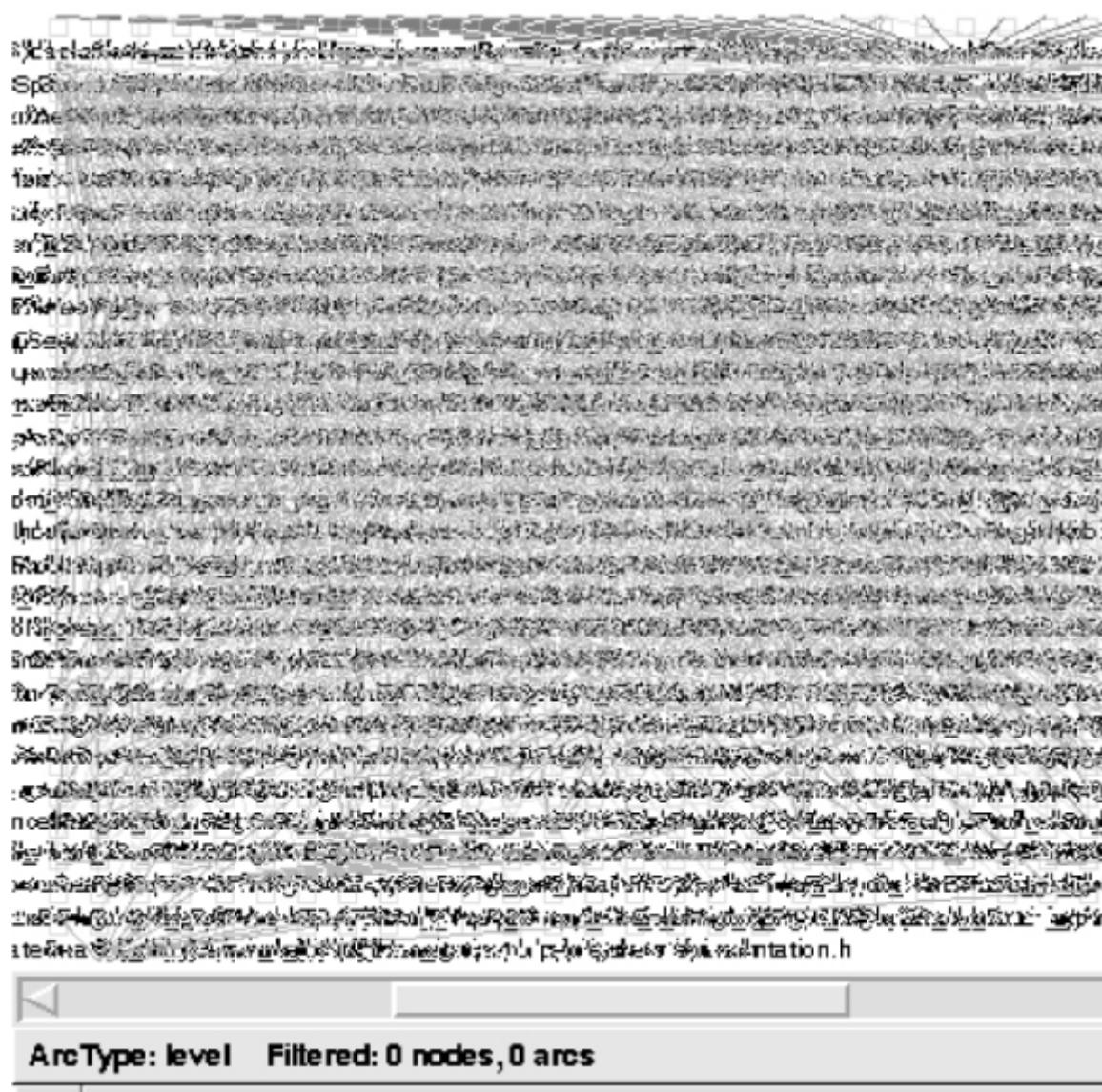


Figure 20.2. A raw extracted view: white noise

Thus we need to manipulate such raw views, to collapse information (for example, hiding methods inside class definitions), and to show abstractions (for example, showing all of the connections between business objects and user interface objects, or identifying distinct layers).

It is helpful to use a database to store the extracted information because the amount of information being stored is large, and the manipulations of the data are tedious and error-prone if done manually. Some reverse-engineering tools, such as Lattix, SonarJ, and Structure101, fully encapsulate the database, and so the user of the tool need not be concerned with its operation. However, those who are using a suite of tools together—a workbench—will need to choose a database and decide on internal representations of the views.

20.4. View Fusion

Once the raw facts have been extracted and stored in a database, the reconstructor can now perform view fusion. In this phase, the extracted views are manipulated to create *fused* views. Fused views combine information from one or more extracted views, each of which may contain specialized information. For example, a static call view might be fused with a dynamic call view.

One might want to combine these two views because a static call view will show all explicit calls (where method A calls method B) but will miss calls that are made via late binding mechanisms. A dynamically extracted call graph will never miss a call that is made during an execution, but it suffers the “testing” problem: it will only report results from those paths through the system that are traversed during its execution. So a little-used part of the system—perhaps for initialization or error recovery—might not show up in the dynamic view. Therefore we fuse these two views to produce a more complete and more accurate graph of system relationships.

The process of creating a fused view is the process of creating a hypothesis about the architecture and a visualization of it to aid in analysis. These hypotheses result in new aggregations that show various abstractions or clusterings of the elements (which may be source artifacts or previously identified abstractions). By interpreting these fused views and analyzing them, it is possible to produce hypothesized architectural views of the system. These views can be interpreted, further refined, or rejected. There are no universal completion criteria for this process; it is complete when the architectural representation is sufficient to support the analysis needs of its stakeholders.

For example, [Figure 20.3](#) shows the early results of interacting with the tool SonarJ. SonarJ first extracts facts from a set of source code files (in this case, written in Java) and lets you define a set of layers and vertical slices through those layers in a system. SonarJ will then instantiate the user-specified definitions of layers and slices and populate them with the extracted software elements.

	Common <<unrestricted...>>	Contact <<unrestricted...>>	Customer <<unrestricted...>>	Distribution... <<unrestricted...>>	Request <<unrestricted...>>	User <<unrestricted...>>	
Controller <<unrestricted...>>							
Data <<unrestricted...>>							
Domain <<unrestricted...>>							
DSI <<unrestricted...>>							
Service <<unrestricted...>>							

Figure 20.3. Hypothesized layers and vertical slices

In the figure there are five layers: Controller, Data, Domain, DSI, and Service. And there are six vertical slices defined that span these layers: Common, Contact, Customer, Distribution, Request, and User. At this point, however, there are no relationships between the layers or vertical slices shown—this is merely an enumeration of the important system abstractions.

20.5. Architecture Analysis: Finding Violations

Consider the following situation: You have designed an architecture but you have suspicions that the developers are not faithfully implementing what you developed. They may do this out of ignorance, or because they have differing agendas for the system, or simply because they were rushing to meet a deadline and ignored any concern not on their critical path. Whatever the root cause, this divergence of the architecture and the implementation spells problems for you, the architect. So how do you test and ensure conformance to the design?

There are two major possibilities for maintaining conformance between code and architecture:

- *Conformance by construction.* Ensuring consistency by construction—that is, automatically generating a substantial part of the system based on an architectural specification—is highly desirable because tools can guarantee conformance. Unfortunately,

this approach has limited applicability. It can only be applied in situations where engineers can employ specific architecture-based development tools, languages, and implementation strategies. For systems that are composed of existing parts or that require a style of architecture or implementation outside those supported by generation tools, this approach does not apply. And this is the vast majority of systems.

- *Conformance by analysis.* This technique aims to ensure conformance by analyzing (reverse-engineering) system information to flag nonconforming elements, so that they can be fixed: brought into conformance. When an implementation is sufficiently constrained so that modularization and coding patterns can be identified with architectural elements, this technique can work well. Unfortunately, however, the technique is limited in its applicability. There is an inherent mismatch between static, code-based structures such as classes and packages (which are what programmers see) and the runtime structures, such as processes, threads, clients, servers, and databases, that are the essence of most architectural descriptions. Further complicating this analysis, the actual runtime structures may not be known or established until the program executes: clients and servers may come and go dynamically, components not under direct control of the implementers may be dynamically loaded, and so forth.

We will focus on the second option: conformance by analysis.

In the previous step, view fusion gave us a set of hypotheses about the architecture. These hypotheses take the form of architectural elements (sometimes aggregated, such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct—to see if they conform with the architect's intentions. That is the function of the analysis step.

[Figure 20.4](#) shows the results of adding relationships and constraints to the architecture initially created in [Figure 20.3](#). These relationship and constraints are information added by the architect, to reflect the design intent. In this example, the architect has indicated the relationships between the layers of [Figure 20.3](#). These relationships are indicated by the directed lines drawn between the layers (and vertical slices). Using these relationships and constraints, a tool such as SonarQube is able to automatically detect and report violations of the layering in the software.

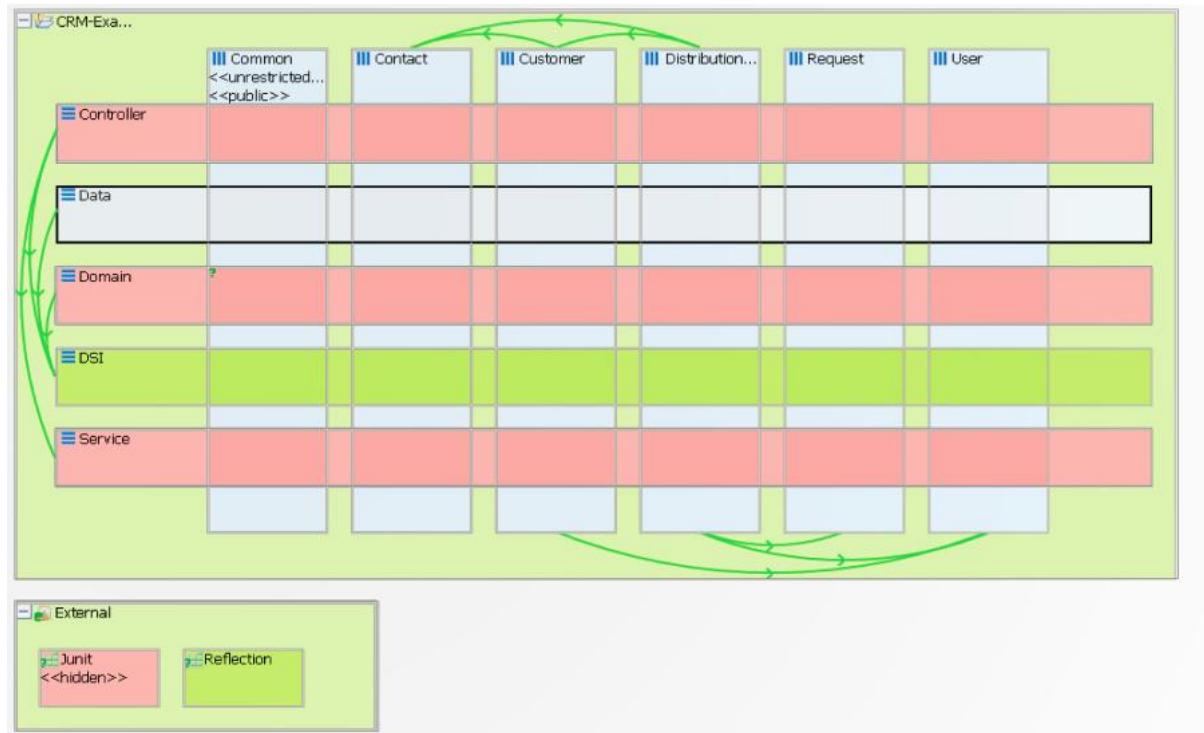


Figure 20.4. Layers, vertical slices, relationships, and constraints

We can now see that the Data layer (row 2 in [Figure 20.4](#)) can access, and hence depends on, the DSI layer. We can further see that it may not access, and has no dependencies on, Domain, Service, or Controller (rows 1, 3, and 5 in the figure).

In addition we can see that the JUnit component in the “External” component is defined to be inaccessible. This is an example of an architectural constraint that is meant to pervade the entire system: no portion of the application should depend upon JUnit, because this should only be used by test code.

[Figure 20.5](#) shows an example of an architecture violation of the previous restriction. This violation is found by SonarJ by searching through its database, applying the user-defined patterns, and finding violations of those patterns. In this figure you can see an arc between the Service layer and JUnit. This arc is highlighted to indicate that this is an illegal dependency and an architectural violation. (This figure also shows some additional dependencies, to external modules.)



Figure 20.5. Highlighting an architecture violation

Architecture reconstruction is a means of testing the conformance to such constraints. The preceding example showed how these constraints might be detected and enforced using static code analysis. But static analysis is primarily useful for understanding module structures. What if one needed to understand runtime information, as represented by C&C structures?

In the example given in [Figure 20.6](#), an architecture violation was discovered via dynamic analysis, using the research DiscoTect system. In this case an analysis of the runtime architecture of the Duke’s Bank application—a simple Enterprise JavaBeans (EJB) banking application created by Sun Microsystems as a demonstration of EJB functionality—was performed. The code was “instrumented” using AspectJ; instrumentation aspects were woven into the compiled bytecode of the EJB application. These aspects emitted events when methods entered or exited and when objects were constructed.

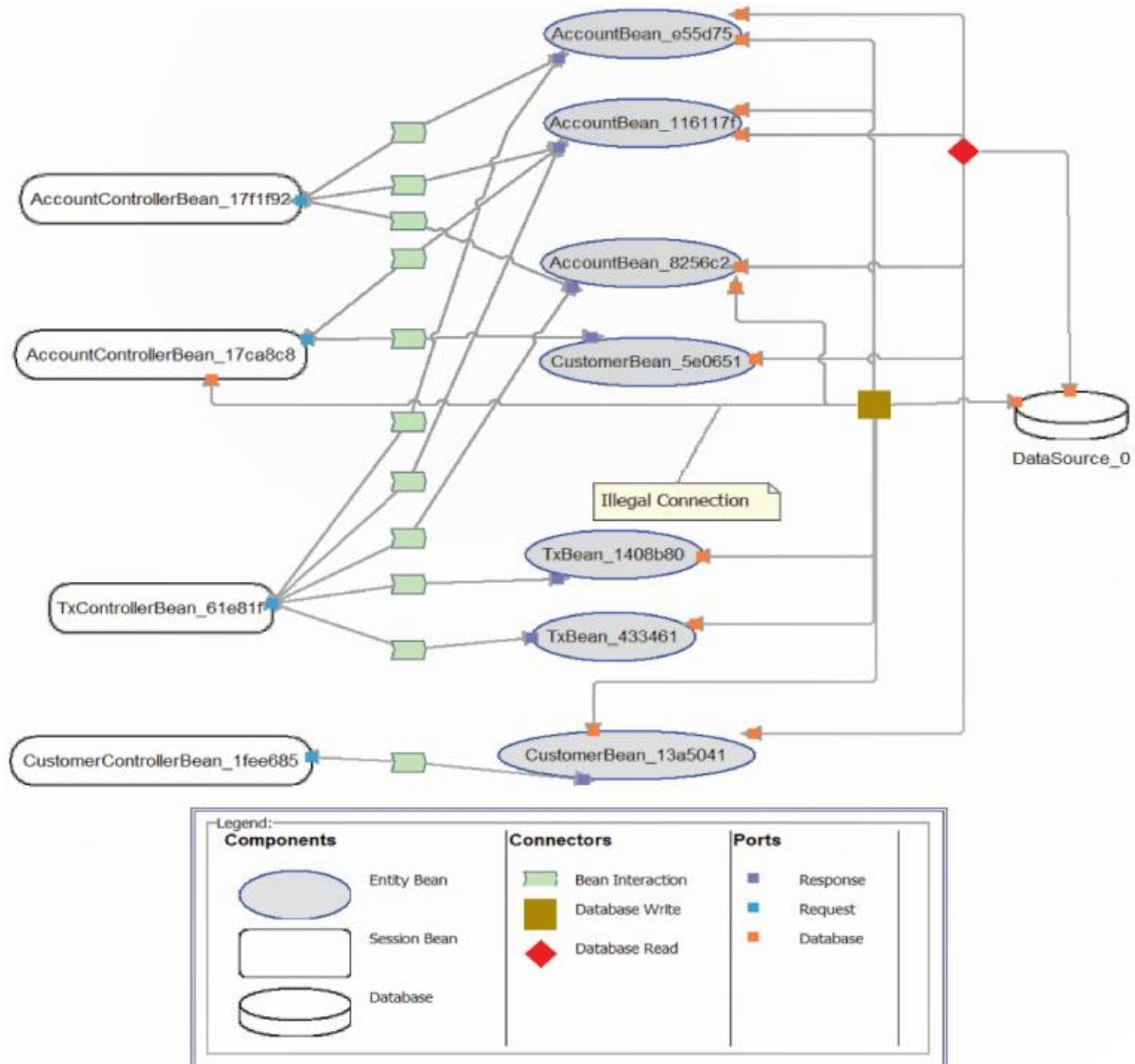


Figure 20.6. An architecture violation discovered by dynamic analysis

Figure 20.6 shows that a “database write” connector was discovered in the dynamic analysis of the architecture. Sun’s EJB specification and its documented architecture of Duke’s Bank forbid such connections. All database access is supposed to be managed by entity beans, and only by entity beans. Such architectural violations are difficult to find in the source code—often just a single line of code is involved—and yet can substantially affect the quality attributes of the resulting system.

20.6. Guidelines

The following are a set of guidelines for the reconstruction process:

- Have a goal and a set of objectives or questions in mind before undertaking an architecture reconstruction project. In the absence of these, a lot of effort could be spent on extracting information and generating architecture views that may not be helpful or serve any useful purpose.
- Obtain some representation, however coarse, of the system before beginning the detailed reconstruction process. This representation serves several purposes, including the following:

- It identifies what information needs to be extracted from the system.
- It guides the reconstructor in determining what to look for in the architecture and what views to generate.

Identifying layers is a good place to start.

- In many cases, the existing documentation for a system may not accurately reflect the system as it is implemented. Therefore it may be necessary to disregard the existing documentation and use it only to generate the high-level views of the system, because it should give an indication of the high-level concepts.
- Tools can support the reconstruction effort and shorten the reconstruction process, but they cannot do an entire reconstruction effort automatically. The work involved in the effort requires the involvement of people (architects, maintainers, and developers) who are familiar with the system. It is important to get these people involved in the effort at an early stage as it helps the reconstructor get a better understanding of the system being reconstructed.

20.7. Summary

Architecture reconstruction and architecture conformance are crucial tools in the architect's toolbox to ensure that a system is built the way it was designed, and that it evolves in a way that is consistent with its creators' intentions. All nontrivial long-lived systems evolve: the code and the architecture both evolve. This is a good thing. But if the code evolves in an ad hoc manner, the result will be the big ball of mud, and the system's quality attributes will inevitably suffer. The only defense against this erosion is consistent attention to architecture quality, which implies the need to maintain architecture conformance.

The results of architectural reconstruction can be used in several ways:

- If no documentation exists or if it is seriously out of date, the recovered architectural representation can be used as a basis for documenting the architecture, as discussed in [Chapter 18](#).
- It can be used to recover the as-built architecture, or to check conformance against an "as-designed" architecture. Conformance checking assures us that our developers and maintainers have followed the architectural edicts set forth for them and are not eroding the architecture by breaking down abstractions, bridging layers, compromising information hiding, and so forth.
- The reconstruction can be used as the basis for analyzing the architecture or as a starting point for reengineering the system to a new desired architecture.
- Finally, the representation can be used to identify elements for reuse or to establish an architecture-based software product line (see [Chapter 25](#)).

The software architecture reconstruction process comprises the following phases:

- 1. Raw view extraction.** In the raw view extraction phase, raw information about the architecture is obtained from various sources, primarily source code, execution traces, and build scripts. Each of these sets of raw information is called a view.
- 2. Database construction.** The database construction phase involves converting the extracted information into a standard form (because the various extraction tools may each produce their own form of output) and populating a reconstruction database with this information.
- 3. View fusion.** The view fusion phase combines views of the information stored in the database.
- 4. Architecture analysis.** View fusion has given us a set of hypotheses about the architecture. These hypotheses take the form of architectural elements (sometimes aggregated, such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct, and that is the function of the analysis step.

20.8. For Further Reading

The Software Engineering Institute (SEI) has developed two reconstruction workbenches: Dali and Armin. Dali was our first attempt at creating a workbench for architecture recovery and conformance [[Kazman 99](#)]. Armin, a complete rewrite and rethink of Dali, is described in [[O'Brien 03](#)].

Both Armin and Dali were primarily focused on module structures of an architecture. A later tool, called DiscoTect, was aimed at discovering C&C structures. This is described in [[Schmerl 06](#)].

Many other architecture reverse-engineering tools have been created. A few of the notable ones created in academia are [[van Deursen 04](#)], [[Murphy 01](#)], and [[Storey 97](#)].

In addition there are a number of commercial architecture extraction and reconstruction tools that have been slowly gaining market acceptance in the past decade. Among these are the following:

- SonarJ (www.hello2morrow.com)
- Lattix (www.lattix.com)
- Understand (www.scitools.com)

Cai et al. [[Cai 2011](#)] compellingly demonstrate the need for architecture conformance testing in an experimental study that they conducted, wherein they found that software engineering students, given UML designs for a variety of relatively simple systems, violate those designs over 70 percent of the time.

Finally, the set of guidelines presented in this chapter for how to go about reconstructing an architecture was excerpted from [[Kazman 02](#)].

20.9. Discussion Questions

1. Suppose that for a given system you wanted to extract the architectural structures (as discussed in [Chapter 1](#)) listed in the table rows below. For each row, fill in each column to appraise each strategy listed in the columns. "VH" (very high) means the strategy would be very effective at extracting this structure; "VL" means it would be very ineffective; "H," M," and "L" have the obvious in-between values.

		Reconstruction Strategies		
Architectural Structures Interviewing experts on the system		Analyzing structure of source code files	Static analysis of source code	Dynamic analysis of system's execution
Module structures	Decomposition			
	Uses			
	Layers			
	Class			
	Data model			
C&C structures	Service (for SOA systems)			
	Concurrency			
Allocation structures	Deployment			
	Implementation			
	Work assignment			

2. Recall that in layered systems, the relationship among layers is *allowed to use*. Also recall that it is possible for one piece of software to use another piece without actually calling it—for example, by depending on it leaving some shared resource in a usable state. Does this interpretation change your answer above for the “Uses” and “Layers” structures?
3. What inferences can you make about a system’s module structures from examining a set of behavioral traces gathered dynamically?
4. Suppose you believe that the architecture for a system follows a broker pattern. What information would you want to extract from the source code to confirm or refute this hypothesis? What behavioral or interaction pattern would you expect to observe at runtime?
5. Suppose you hypothesize that a system makes use of particular tactics to achieve a particular quality attribute. Fill in the columns of the table below to show how you would go about verifying your hypothesis. (Begin by filling in column 1 with a particular tactic for the named quality attribute.)

Tactics for...	Reconstruction Strategies			
	Interviewing experts on the system	Analyzing structure of source code files	Static analysis of source code	Dynamic analysis of system's execution
Availability				
Interoperability				
Modifiability				
Performance				
Security				
Testability				
Usability				

6. Suppose you want to confirm that developers and maintainers had remained faithful to an architecture over the lifetime of the system. Describe the reconstruction and/or auditing processes you would undertake.

21. Architecture Evaluation

Fear cannot be banished, but it can be calm and without panic; it can be mitigated by reason and evaluation.

—Vannevar Bush

We discussed analysis techniques in [Chapter 14](#). Analysis lies at the heart of architecture evaluation, which is the process of determining if an architecture is fit for the purpose for which it is intended. Architecture is such an important contributor to the success of a system and software engineering project that it makes sense to pause and make sure that the architecture you've designed will be able to provide all that's expected of it. That's the role of evaluation. Fortunately there are mature methods to evaluate architectures that use many of the concepts and techniques you've already learned in previous chapters of this book.

21.1. Evaluation Factors

Evaluation usually takes one of three forms:

- Evaluation by the designer within the design process
- Evaluation by peers within the design process
- Analysis by outsiders once the architecture has been designed

Evaluation by the Designer

Every time the designer makes a key design decision or completes a design milestone, the chosen and competing alternatives should be evaluated using the analysis techniques of [Chapter 14](#). Evaluation by the designer is the “test” part of the “generate-and-test” approach to architecture design that we discussed in [Chapter 17](#).

How much analysis? This depends on the importance of the decision. Obviously, decisions made to achieve one of the driving architectural requirements should be subject to more analysis than others, because these are the ones that will shape critical portions of the architecture. But in all cases, performing analysis is a matter of cost and benefit. Do not spend more time on a decision than it is worth, but also do not spend less time on an important decision than it needs. Some specific considerations include these:

- *The importance of the decision.* The more important the decision, the more care should be taken in making it and making sure it’s right.
- *The number of potential alternatives.* The more alternatives, the more time could be spent in evaluating them. Try to eliminate alternatives quickly so that the number of viable potential alternatives is small.
- *Good enough as opposed to perfect.* Many times, two possible alternatives do not differ dramatically in their consequences. In such a case, it is more important to make a choice and move on with the design process than it is to be absolutely certain that the best choice is being made. Again, do not spend more time on a decision than it is worth.

Peer Review

Architectural designs can be peer reviewed just as code can be peer reviewed. A peer review can be carried out at any point of the design process where a candidate architecture, or at least a coherent reviewable part of one, exists. There should be a fixed amount of time allocated for the peer review, at least several hours and possibly half a day. A peer review has several steps:

1. *The reviewers determine a number of quality attribute scenarios to drive the review.* Most of the time these scenarios will be architecturally significant requirements, but they need not be. These scenarios can be developed by the review team or by additional stakeholders.
2. *The architect presents the portion of the architecture to be evaluated.* (At this point, comprehensive documentation for it may not exist.) The reviewers individually ensure that they understand the architecture. Questions at this point are specifically for understanding. There is no debate about the decisions that were made. These come in the next step.
3. *For each scenario, the designer walks through the architecture and explains how the scenario is satisfied.* (If the architecture is already documented, then the reviews can use it to assess for themselves how it satisfies the scenario.) The reviewers ask questions to determine two different types of information. First, they want to determine that the scenario is, in fact, satisfied. Second, they want to determine whether any of the other scenarios being considered will not be satisfied because of the decisions made in the portion of the architecture being reviewed.
4. *Potential problems are captured.* The list of potential problems forms the basis for the follow-up of the review. If the potential problem is a real problem, then it either must be fixed or a decision must be explicitly made by the designers and the project manager that they are willing to accept the problem and its probability of occurrence.

If the designers are using the ADD process described in [Chapter 17](#), then a peer review can be done at the end of step 3 of each ADD iteration.

Analysis by Outsiders

Outside evaluators can cast an objective eye on an architecture. "Outside" is relative; this may mean outside the development project, outside the business unit where the project resides but within the same company; or outside the company altogether. To the degree that evaluators are "outside," they are less likely to be afraid to bring up sensitive problems, or problems that aren't apparent because of organizational culture or because "we've always done it that way."

Often, outsiders are chosen because they possess specialized knowledge or experience, such as knowledge about a quality attribute that's important to the system being examined, or long experience in successfully evaluating architectures.

Also, whether justified or not, managers tend to be more inclined to listen to problems uncovered by an outside team hired at considerable cost. (This can be understandably frustrating to project staff who may have been complaining about the same problems to no avail for months.)

In principle, an outside team may evaluate a completed architecture, an incomplete architecture, or a portion of an architecture. In practice, because engaging them is complicated and often expensive, they tend to be used to evaluate complete architectures.

Contextual Factors

For peer reviews or outside analysis, there are a number of contextual factors that must be considered when structuring an evaluation. These include the artifacts available, whether the results are public or private, the number and skill of evaluators, the number and identity of the participating stakeholders, and how the business goals are understood by the evaluators.

- *What artifacts are available?* To perform an architectural evaluation, there must be an artifact that describes the architecture. This must be located and made available. Some evaluations may take place after the system is operational. In this case, recovery tools as described in [Chapter 20](#) may be used both to assist in discovering the architecture and to test that the as-built system conforms to the as-designed system.
- *Who sees the results?* Some evaluations are performed with the full knowledge and participation of all of the stakeholders. Others are performed more privately. The private evaluations may be done for a variety of reasons, ranging from corporate culture to (in one case we know about) an executive wanting to determine which of a collection of competitive systems he should back in an internal dispute about the systems.
- *Who performs the evaluation?* Evaluations can be carried out by an individual or a team. In either case, the evaluator(s) should be highly skilled in the domain and the various quality attributes for which the system is to be evaluated. And for carrying out evaluation methods with extensive stakeholder involvement, excellent organizational and facilitation skills are a must.
- *Which stakeholders will participate?* The evaluation process should provide a method to elicit the goals and concerns that the important stakeholders have regarding the system. Identifying the individuals who are needed and assuring their participation in the evaluation is critical.
- *What are the business goals?* The evaluation should answer whether the system will satisfy the business goals. If the business goals are not explicitly captured and prioritized prior to the evaluation, then there should be a portion of the evaluation dedicated to doing so.

21.2. The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) has been used for over a decade to evaluate software architectures in domains ranging from automotive to financial to defense. The ATAM is designed so that evaluators need not be familiar with the architecture or its business goals, the system need not yet be constructed, and there may be a large number of stakeholders.

Participants in the ATAM

The ATAM requires the participation and mutual cooperation of three groups:

- *The evaluation team.* This group is external to the project whose architecture is being evaluated. It usually consists of three to five people. Each member of the team is assigned a number of specific roles to play during the evaluation. (See [Table 21.1](#) for a description of these roles, along with a set of desirable characteristics for each. A single person may adopt several roles in an ATAM.) The evaluation team may be a standing unit in which architecture evaluations are regularly performed, or its members may be chosen from a pool of architecturally savvy individuals for the occasion. They may work for the same organization as the development team whose architecture is on the table, or they may be outside consultants. In any case, they need to be recognized as competent, unbiased outsiders with no hidden agendas or axes to grind.

Table 21.1. ATAM Evaluation Team Roles

Role	Responsibilities
Team Leader	Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; forms evaluation team; sees that final report is produced and delivered (although the writing may be delegated)
Evaluation Leader	Runs evaluation; facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture; facilitates on-site analysis
Scenario Scribe	Writes scenarios on flipchart or whiteboard during scenario elicitation; captures agreed-on wording of each scenario, halting discussion until exact wording is captured
Proceedings Scribe	Captures proceedings in electronic form on laptop or workstation: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and resolution of each scenario when applied to architecture(s); also generates a printed list of adopted scenarios for handout to all participants
Questioner	Raises issues of architectural interest, usually related to the quality attributes in which he or she has expertise

- *Project decision makers.* These people are empowered to speak for the development project or have the authority to mandate changes to it. They usually include the project manager, and if there is an identifiable customer who is footing the bill for the development, he or she may be present (or represented) as well. The architect is always included—a cardinal rule of architecture evaluation is that the architect must willingly participate.
- *Architecture stakeholders.* Stakeholders have a vested interest in the architecture performing as advertised. They are the ones whose ability to do their job hinges on the architecture promoting modifiability, security, high reliability, or the like. Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and others listed in [Chapter 3](#). Their job during an evaluation is to articulate the specific quality attribute goals that the architecture should meet in order for the system to be considered a success. A rule of thumb—and that is all it is—is that you should expect to enlist 12 to 15 stakeholders for the evaluation of a large enterprise-critical architecture. Unlike the evaluation team and the project decision makers, stakeholders do not participate in the entire exercise.

Outputs of the ATAM

As in any testing process, a large benefit derives from preparing for the test. In preparation for an ATAM exercise, the project's decision makers must prepare the following:

- 1. A concise presentation of the architecture.** One of the requirements of the ATAM is that the architecture be presented in one hour, which leads to an architectural presentation that is both concise and, usually, understandable.
- 2. Articulation of the business goals.** Frequently, the business goals presented in the ATAM are being seen by some of the assembled participants for the first time, and these are captured in the outputs. This description of the business goals survives the evaluation and becomes part of the project's legacy.

The ATAM uses prioritized quality attribute scenarios as the basis for evaluating the architecture, and if those scenarios do not already exist (perhaps as a result of a prior requirements capture exercise or ADD activity), they are generated by the participants as part of the ATAM exercise. Many times, ATAM participants have told us that one of the most valuable outputs of ATAM is this next output:

- 3. Prioritized quality attribute requirements expressed as quality attribute scenarios.** These quality attribute scenarios take the form described in [Chapter 4](#). These also survive past the evaluation and can be used to guide the architecture's evolution.

The primary output of the ATAM is a set of issues of concern about the architecture. We call these risks:

- 4. A set of risks and nonrisks.** A risk is defined in the ATAM as an architectural decision that may lead to undesirable consequences in light of stated quality attribute requirements. Similarly, a nonrisk is an architectural decision that, upon analysis, is deemed safe. The identified risks form the basis for an architectural risk mitigation plan.
- 5. A set of risk themes.** When the analysis is complete, the evaluation team examines the full set of discovered risks to look for overarching themes that identify systemic weaknesses in the architecture or even in the architecture process and team. If left untreated, these risk themes will threaten the project's business goals.

Finally, along the way, other information about the architecture is discovered and captured:

- 6. Mapping of architectural decisions to quality requirements.** Architectural decisions can be interpreted in terms of the qualities that they support or hinder. For each quality attribute scenario examined during an ATAM, those architectural decisions that help to achieve it are determined and captured. This can serve as a statement of rationale for those decisions.
- 7. A set of identified sensitivity and tradeoff points.** These are architectural decisions that have a marked effect on one or more quality attributes.

The outputs of the ATAM are used to build a final written report that recaps the method, summarizes the proceedings, captures the scenarios and their analysis, and catalogs the findings.

There are intangible results of an ATAM-based evaluation. These include a palpable sense of community on the part of the stakeholders, open communication channels between the architect and the stakeholders, and a better overall understanding on the part of all participants of the architecture and its strengths and weaknesses. While these results are hard to measure, they are no less important than the others and often are the longest-lasting.

Phases of the ATAM

Activities in an ATAM-based evaluation are spread out over four phases:

- In phase 0, "Partnership and Preparation," the evaluation team leadership and the key project decision makers informally meet to work out the details of the exercise. The project representatives brief the evaluators about the project so that the team can be supplemented by people who possess the appropriate expertise. Together, the two groups agree on logistics, such as the time and place of meetings, who brings the flipcharts, and who supplies the donuts and coffee. They also agree on a preliminary list of stakeholders (by name, not just role), and they negotiate on when the final report is to be delivered and to whom. They deal with formalities such as a statement of work or nondisclosure agreements. The evaluation team examines the architecture documentation to gain an understanding of the architecture and the major design approaches that it comprises. Finally, the evaluation team leader explains what information the manager and architect will be expected to show during phase 1, and helps them construct their presentations if necessary.

- Phase 1 and phase 2 are the evaluation phases, where everyone gets down to the business of analysis. By now the evaluation team will have studied the architecture documentation and will have a good idea of what the system is about, the overall architectural approaches taken, and the quality attributes that are of paramount importance. During phase 1, the evaluation team meets with the project decision makers (for one to two days) to begin information gathering and analysis. For phase 2, the architecture's stakeholders join the proceedings and analysis continues, typically for two days. Unlike the other phases, phase 1 and phase 2 comprise a set of specific steps; these are detailed in the next section.
- Phase 3 is follow-up, in which the evaluation team produces and delivers a written final report. It is first circulated to key stakeholders to make sure that it contains no errors of understanding, and after this review is complete it is delivered to the person who commissioned the evaluation.

[Table 21.2](#) shows the four phases of the ATAM, who participates in each one, and an approximate timetable.

Table 21.2. ATAM Phases and Their Characteristics

Phase	Activity	Participants	Typical Duration
0	Partnership and preparation	Evaluation team leadership and key project decision makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation	Evaluation team and project decision makers	1–2 days followed by a hiatus of 1–3 weeks
2	Evaluation (continued)	Evaluation team, project decision makers, and stakeholders	2 days
3	Follow-up	Evaluation team and evaluation client	1 week

Source: Adapted from [\[Clements 01b\]](#).

Steps of the Evaluation Phases

The ATAM analysis phases (phase 1 and phase 2) consist of nine steps. Steps 1 through 6 are carried out in phase 1 with the evaluation team and the project's decision makers: typically, the architecture team, project manager, and project sponsor. In phase 2, with all stakeholders present, steps 1 through 6 are summarized and steps 7 through 9 are carried out.

[Table 21.3](#) shows a typical agenda for the first day of phase 1, which covers steps 1 through 5. Step 6 in phase 1 is carried out the next day.

Table 21.3. Agenda for Day 1 of the ATAM

Time	Activity
0830 – 1000	Introductions; Step 1: Present the ATAM
1000 – 1100	Step 2: Present Business Drivers
1100 – 1130	Break
1130 – 1230	Step 3: Present Architecture
1230 – 1330	Lunch
1330 – 1430	Step 4: Identify Architectural Approaches
1430 – 1530	Step 5: Generate Utility Tree
1530 – 1600	Break
1600 – 1700	Step 5: Generate Utility Tree (continued)

Step 1: Present the ATAM

The first step calls for the evaluation leader to present the ATAM to the assembled project representatives. This time is used to explain the process that everyone will be following, to answer questions, and to set the context and expectations for the remainder of the activities. Using a standard presentation, the leader describes the ATAM steps in brief and the outputs of the evaluation.

Step 2: Present the Business Drivers

Everyone involved in the evaluation—the project representatives as well as the evaluation team members—needs to understand the context for the system and the primary business drivers motivating its development. In this step, a project decision maker (ideally the project manager or the system’s customer) presents a system overview from a business perspective. The presentation should describe the following:

- The system’s most important functions
- Any relevant technical, managerial, economic, or political constraints
- The business goals and context as they relate to the project
- The major stakeholders
- The architectural drivers (that is, the architecturally significant requirements)

Step 3: Present the Architecture

Here, the lead architect (or architecture team) makes a presentation describing the architecture at an appropriate level of detail. The “appropriate level” depends on several factors: how much of the architecture has been designed and documented; how much time is available; and the nature of the behavioral and quality requirements.

In this presentation the architect covers technical constraints such as operating system, hardware, or middleware prescribed for use, and other systems with which the system must interact. Most important, the architect describes the architectural approaches (or patterns, or tactics, if the architect is fluent in that vocabulary) used to meet the requirements.

To make the most of limited time, the architect’s presentation should have a high signal-to-noise ratio. That is, it should convey the essence of the architecture and not stray into ancillary areas or delve too deeply into the details of just a few aspects. Thus, it is extremely helpful to brief the architect beforehand (in phase 0) about the information the evaluation team requires. A template such as the one in the sidebar can help the architect prepare the presentation. Depending on the architect, a dress rehearsal can be included as part of the phase 0 activities.

Architecture Presentation (Approximately 20 slides; 60 Minutes)

Driving architectural requirements, the measurable quantities you associate with these requirements, and any existing standards/models/approaches for meeting these (2–3 slides)

Important architectural information (4–8 slides):

- Context diagram—the system within the context in which it will exist. Humans or other systems with which the system will interact.
- Module or layer view—the modules (which may be subsystems or layers) that describe the system's decomposition of functionality, along with the objects, procedures, functions that populate these, and the relations among them (e.g., procedure call, method invocation, callback, containment).
- Component-and-connector view—processes, threads along with the synchronization, data flow, and events that connect them.
- Deployment view—CPUs, storage, external devices/sensors along with the networks and communication devices that connect them. Also shown are the processes that execute on the various processors.

Architectural approaches, patterns, or tactics employed, including what quality attributes they address and a description of how the approaches address those attributes (3–6 slides):

- Use of commercial off-the-shelf (COTS) products and how they are chosen/integrated (1–2 slides).
- Trace of 1 to 3 of the most important use case scenarios. If possible, include the runtime resources consumed for each scenario (1–3 slides).
- Trace of 1 to 3 of the most important change scenarios. If possible, describe the change impact (estimated size/difficulty of the change) in terms of the changed modules or interfaces (1–3 slides).
- Architectural issues/risks with respect to meeting the driving architectural requirements (2–3 slides).
- Glossary (1 slide).

Source: Adapted from [\[Clements 01b\]](#).

As may be seen in the presentation template, we expect architectural views, as described in [Chapters 1](#) and [18](#), to be the primary vehicle for the architect to convey the architecture. Context diagrams, component-and-connector views, module decomposition or layered views, and the deployment view are useful in almost every evaluation, and the architect should be prepared to show them. Other views can be presented if they contain information relevant to the architecture at hand, especially information relevant to achieving important quality attribute goals.

As a rule of thumb, the architect should present the views that he or she found most important during the creation of the architecture and the views that help to reason about the most important quality attribute concerns of the system.

During the presentation, the evaluation team asks for clarification based on their phase 0 examination of the architecture documentation and their knowledge of the business drivers from the previous step. They also listen for and write down any architectural tactics or patterns they see employed.

Step 4: Identify Architectural Approaches

The ATAM focuses on analyzing an architecture by understanding its architectural approaches. As we saw in [Chapter 13](#), architectural patterns and tactics are useful for (among other reasons) the known ways in which each one affects particular quality attributes. A layered pattern tends to bring portability and maintainability to a system, possibly at the expense of performance. A publish-subscribe pattern is scalable in the number of producers and consumers of data. The active redundancy tactic promotes high availability. And so forth.

By now, the evaluation team will have a good idea of what patterns and tactics the architect used in designing the system. They will have studied the architecture documentation, and they will

have heard the architect's presentation in step 3. During that step, the architect is asked to explicitly name the patterns and tactics used, but the team should also be adept at spotting ones not mentioned.

In this short step, the evaluation team simply catalogs the patterns and tactics that have been identified. The list is publicly captured by the scribe for all to see and will serve as the basis for later analysis.

Step 5: Generate Quality Attribute Utility Tree

In this step, the quality attribute goals are articulated in detail via a quality attribute utility tree. Utility trees, which were described in [Chapter 16](#), serve to make the requirements concrete by defining precisely the relevant quality attribute requirements that the architects were working to provide.

The important quality attribute goals for the architecture under consideration were named in step 2, when the business drivers were presented, but not to any degree of specificity that would permit analysis. Broad goals such as "modifiability" or "high throughput" or "ability to be ported to a number of platforms" establish important context and direction, and provide a backdrop against which subsequent information is presented. However, they are not specific enough to let us tell if the architecture suffices. Modifiable in what way? Throughput that is how high? Ported to what platforms and in how much time?

In this step, the evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals. These are expressed as scenarios, as described in [Chapter 4](#), which populate the leaves of the utility tree.

Step 6: Analyze Architectural Approaches

Here the evaluation team examines the highest-ranked scenarios (as identified in the utility tree) one at a time; the architect is asked to explain how the architecture supports each one. Evaluation team members—especially the questioners—probe for the architectural approaches that the architect used to carry out the scenario. Along the way, the evaluation team documents the relevant architectural decisions and identifies and catalogs their risks, nonisks, sensitivity points, and tradeoffs. For well-known approaches, the evaluation team asks how the architect overcame known weaknesses in the approach or how the architect gained assurance that the approach sufficed. The goal is for the evaluation team to be convinced that the instantiation of the approach is appropriate for meeting the attribute-specific requirements for which it is intended.

Scenario walkthrough leads to a discussion of possible risks, nonisks, sensitivity points, or tradeoff points. For example:

- The frequency of heartbeats affects the time in which the system can detect a failed component. Some assignments will result in unacceptable values of this response—these are risks.
- The number of simultaneous database clients will affect the number of transactions that a database can process per second. Thus, the assignment of clients to the server is a sensitivity point with respect to the response as measured in transactions per second.
- The frequency of heartbeats determines the time for detection of a fault. Higher frequency leads to improved availability but will also consume more processing time and communication bandwidth (potentially leading to reduced performance). This is a tradeoff.

These, in turn, may catalyze a deeper analysis, depending on how the architect responds. For example, if the architect cannot characterize the number of clients and cannot say how load balancing will be achieved by allocating processes to hardware, there is little point in a sophisticated performance analysis. If such questions can be answered, the evaluation team can perform at least a rudimentary, or back-of-the-envelope, analysis to determine if these architectural decisions are problematic vis-à-vis the quality attribute requirements they are meant to address.

The analysis is not meant to be comprehensive. The key is to elicit sufficient architectural information to establish some link between the architectural decisions that have been made and the quality attribute requirements that need to be satisfied.

[Figure 21.1](#) shows a template for capturing the analysis of an architectural approach for a scenario. As shown, based on the results of this step, the evaluation team can identify and record a set of sensitivity points and tradeoffs, risks, and nonisks.

Scenario #: A12		Scenario: Detect and recover from HW failure of main switch.		
Attribute(s)	Availability			
Environment	Normal operations			
Stimulus	One of the CPUs fails			
Response	0.999999 availability of switch			
Architectural decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14
Reasoning	<p>Ensures no common mode failure by using different hardware and operating system (see Risk 8)</p> <p>Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst</p> <p>Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog</p> <p>Watchdog is simple and has proved reliable</p> <p>Availability requirement might be at risk due to lack of backup data channel ... (see Risk 9)</p>			
Architecture diagram	<pre> graph LR PC[Primary CPU (OS1)] <--> heartbeat (1 sec.) BC[Backup CPU with Watchdog (OS2)] PC --> SC[Switch CPU (OS1)] BC --> SC SC --> Out[] </pre>			

Figure 21.1. Example of architecture approach analysis (adapted from [\[Clements 01b\]](#))

At the end of step 6, the evaluation team should have a clear picture of the most important aspects of the entire architecture, the rationale for key design decisions, and a list of risks, nonrisks, sensitivity points, and tradeoff points.

At this point, phase 1 is concluded.

Hiatus and Start of Phase 2

The evaluation team summarizes what it has learned and interacts informally (usually by phone) with the architect during a hiatus of a week or two. More scenarios might be analyzed during this period, if desired, or questions of clarification can be resolved.

Phase 2 is attended by an expanded list of participants with additional stakeholders attending. To use an analogy from programming: Phase 1 is akin to when you test your own program, using your own criteria. Phase 2 is when you give your program to an independent quality assurance group, who will likely subject your program to a wider variety of tests and environments.

In phase 2, step 1 is repeated so that the stakeholders understand the method and the roles they are to play. Then the evaluation leader recaps the results of steps 2 through 6, and shares the current list of risks, nonrisks, sensitivity points, and tradeoffs. Now the stakeholders are up to speed with the evaluation results so far, and the remaining three steps can be carried out.

Step 7: Brainstorm and Prioritize Scenarios

In this step, the evaluation team asks the stakeholders to brainstorm scenarios that are operationally meaningful with respect to the stakeholders' individual roles. A maintainer will likely propose a modifiability scenario, while a user will probably come up with a scenario that expresses useful functionality or ease of operation, and a quality assurance person will propose a scenario about testing the system or being able to replicate the state of the system leading up to a fault.

While utility tree generation (step 5) is used primarily to understand how the architect perceived and handled quality attribute architectural drivers, the purpose of scenario brainstorming is to take the pulse of the larger stakeholder community: to understand what system success means for them. Scenario brainstorming works well in larger groups, creating an atmosphere in which the ideas and thoughts of one person stimulate others' ideas.

Once the scenarios have been collected, they must be prioritized, for the same reasons that the scenarios in the utility tree needed to be prioritized: the evaluation team needs to know where to devote its limited analytical time. First, stakeholders are asked to merge scenarios they feel represent the same behavior or quality concern. Then they vote for those they feel are most important. Each stakeholder is allocated a number of votes equal to 30 percent of the number of scenarios,¹ rounded up. So, if there were 40 scenarios collected, each stakeholder would be given 12 votes. These votes can be allocated in any way that the stakeholder sees fit: all 12 votes for 1 scenario, 1 vote for each of 12 distinct scenarios, or anything in between.

¹ This is a common facilitated brainstorming technique.

The list of prioritized scenarios is compared with those from the utility tree exercise. If they agree, it indicates good alignment between what the architect had in mind and what the stakeholders actually wanted. If additional driving scenarios are discovered—and they usually are—this may itself be a risk, if the discrepancy is large. This would indicate that there was some disagreement in the system's important goals between the stakeholders and the architect.

Step 8: Analyze Architectural Approaches

After the scenarios have been collected and prioritized in step 7, the evaluation team guides the architect in the process of carrying out the highest ranked scenarios. The architect explains how relevant architectural decisions contribute to realizing each one. Ideally this activity will be dominated by the architect's explanation of scenarios in terms of previously discussed architectural approaches.

In this step the evaluation team performs the same activities as in step 6, using the highest-ranked, newly generated scenarios.

Typically, this step might cover the top five to ten scenarios, as time permits.

Step 9: Present Results

In step 9, the evaluation team groups risks into risk themes, based on some common underlying concern or systemic deficiency. For example, a group of risks about inadequate or out-of-date documentation might be grouped into a risk theme stating that documentation is given insufficient consideration. A group of risks about the system's inability to function in the face of various hardware and/or software failures might lead to a risk theme about insufficient attention to backup capability or providing high availability.

For each risk theme, the evaluation team identifies which of the business drivers listed in step 2 are affected. Identifying risk themes and then relating them to specific drivers brings the evaluation full circle by relating the final results to the initial presentation, thus providing a satisfying closure to the exercise. As important, it elevates the risks that were uncovered to the attention of management. What might otherwise have seemed to a manager like an esoteric technical issue is now identified unambiguously as a threat to something the manager is on record as caring about.

The collected information from the evaluation is summarized and presented to stakeholders. This takes the form of a verbal presentation with slides. The evaluation leader recapitulates the steps of the ATAM and all the information collected in the steps of the method, including the business context, driving requirements, constraints, and architecture. Then the following outputs are presented:

- The architectural approaches documented
 - The set of scenarios and their prioritization from the brainstorming
 - The utility tree
 - The risks discovered
 - The nonrisks documented
 - The sensitivity points and tradeoff points found
 - Risk themes and the business drivers threatened by each one
-

“. . . but it was OK.”

Years of experience have taught us that no architecture evaluation exercise ever goes completely by the book. And yet for all the ways that an exercise might go terribly wrong, for all the details that can be overlooked, for all the fragile egos that can be bruised, and for all the high stakes that are on the table, we have never had an architecture evaluation exercise spiral out of control. Every single one has been a success, as measured by the feedback we gather from clients.

While they all turned out successfully, there were a few memorable cliffhangers.

More than once, we began an architecture evaluation only to discover that the development organization had no architecture to be evaluated. Sometimes there was a stack of class diagrams or vague text descriptions masquerading as an architecture. Once we were promised that the architecture would be ready by the time the exercise began, but in spite of good intentions, it wasn't. (We weren't always so prudent about pre-exercise preparation and qualification. Our current diligence was a result of experiences like these.) But it was OK. In cases like these, the evaluation's main results included the articulated set of quality attributes, a “whiteboard” architecture sketched during the exercise, plus a set of documentation obligations on the architect. In all cases, the client felt that the detailed scenarios, the analysis we were able to perform on the elicited architecture, plus the recognition of what needed to be done, more than justified the exercise.

A couple of times we began an evaluation only to lose the architect in the middle of the exercise. In one case, the architect resigned between preparation and execution of the evaluation. This was an organization in turmoil and the architect simply got a better offer in a calmer environment elsewhere. Normally we don't proceed without the architect, but it was OK. In this case the architect's apprentice stepped in. A little additional prework to prepare him, and we were all set. The evaluation went off as planned, and the preparation that the apprentice did for the exercise helped mightily to prepare him to step into the architect's shoes.

Once we discovered halfway through an ATAM exercise that the architecture we had prepared to evaluate was being jettisoned in favor of a new one that nobody had bothered to mention. During step 6 of phase 1, the architect responded to a problem raised by a scenario by casually mentioning that “the new architecture” would not suffer from that deficiency. Everyone in the room, stakeholders and evaluators alike, looked at each other in the puzzled silence that followed. “What new architecture?” I asked blankly, and out it came. The developing organization (a contractor for the U.S. military, which had commissioned the evaluation), had prepared a new architecture for the system, to handle the more stringent

requirements they knew were coming in the future. We called a timeout, conferred with the architect and the client, and decided to continue the exercise using the new architecture as the subject instead of the old. We backed up to step 3 (the architecture presentation), but everything else on the table—business drivers, utility tree, scenarios—still were completely valid. The evaluation proceeded as before, and at the conclusion of the exercise our military client was extremely pleased at the knowledge gained.

In perhaps the most bizarre evaluation in our experience, we lost the architect midway through phase 2. The client for this exercise was the project manager in an organization undergoing a massive restructuring. The manager was a pleasant gentleman with a quick sense of humor, but there was an undercurrent about him that said he was not to be crossed. The architect was being reassigned to a different part of the organization in the near future; this was tantamount to being fired from the project, and the manager said he wanted to establish the quality of the architecture before his architect's awkward departure. (We didn't find any of this out until after the evaluation.) When we set up the ATAM exercise, the manager suggested that the junior designers attend. "They might learn something," he said. We agreed. As the exercise began, our schedule (which was very tight to begin with) kept being disrupted. The manager wanted us to meet with his company's executives. Then he wanted us to have a long lunch with someone who could, he said, give us more architectural insights. The executives, it turned out, were busy just now, and so could we come back and meet with them a bit later? By now, phase 2 was thrown off schedule by so much that the architect, to our horror, had to leave to fly back to his home in a distant city. He was none too happy that his architecture was going to be evaluated without him. The junior designers, he said, would never be able to answer our questions. Before his departure, our team huddled. The exercise seemed to be teetering on the brink of disaster. We had an unhappy departing architect, a blown schedule, and questionable expertise available. We decided to split our evaluation team. One half of the team would continue with phase 2 using the junior designers as our information resource. The second half of the team would continue with phase 2 by telephone the next day with the architect. Somehow we would make the best of a bad situation.

Surprisingly, the project manager seemed completely unperturbed by the turn of events. "It will work out, I'm sure," he said pleasantly, and then retreated to confer with various vice presidents about the reorganization.

I led the team interviewing the junior designers. We had never gotten a completely satisfactory architecture presentation from the architect. Discrepancies in the documentation were met with a breezy "Oh, well, that's not how it really works." So I decided to start over with ATAM step 3. We asked the half dozen or so designers what their view of the architecture was. "Could you draw it?" I asked them. They looked at each other nervously, but one said, "I think I can draw part of it." He took to the whiteboard and drew a very reasonable component-and-connector view. Someone else volunteered to draw a process view. A third person drew the architecture for an important offline part of the system. Others jumped in to assist.

As we looked around the room, everyone was busy transcribing the whiteboard pictures. None of the pictures corresponded to anything we had seen in the documentation so far. "Are these diagrams documented anywhere?" I asked. One of the designers looked up from his busy scribbling for a moment to grin. "They are now," he said.

As we proceeded to step 8, analyzing the architecture using the scenarios previously captured, the designers did an astonishingly good job of working together to answer our questions. Nobody knew everything, but everybody knew something. Together in a half day, they produced a clear and consistent picture of the whole architecture that was much more coherent and understandable than anything the architect had been willing to produce in two whole days of pre-exercise discussion. And by the end of phase 2, the design team was transformed. This erstwhile group of information-starved individuals with limited compartmentalized knowledge became a true architecture team. The members drew out and recognized each others' expertise. This expertise was revealed and validated in front of everyone—and most important, in front of their project manager, who had slipped back into the room to observe. There was a look of supreme satisfaction on his face. It began to dawn on me that—you guessed it—it was OK.

It turned out that this project manager knew how to manipulate events and people in ways that would have impressed Machiavelli. The architect's departure was not because of

the reorganization, but merely coincident with it. The project manager had orchestrated it. The architect had, the manager felt, become too autocratic and dictatorial, and the manager wanted the junior design staff to be given the opportunity to mature and contribute. The architect's mid-exercise departure was exactly what the project manager had wanted. And the design team's emergence under fire had been the primary purpose of the evaluation exercise all along. Although we found several important issues related to the architecture, the project manager knew about every one of them before we ever arrived. In fact, he made sure we uncovered some of them by a few discreet remarks during breaks or after a day's session.

Was this exercise a success? The client could not have been more pleased. His instincts about the architecture's strengths and weaknesses were confirmed. We were instrumental in helping his design team, which would guide the system through the stormy seas of the company's reorganization, come together as an effective and cohesive unit at exactly the right time. And the client was so pleased with our final report that he made sure the company's board of directors saw it.

These cliffhangers certainly stand out in our memory. There was no architecture documented. But it was OK. It wasn't the right architecture. But it was OK. There was no architect. But it was OK. The client really only wanted to effect a team reorganization. In every instance we reacted as reasonably as we could, and each time it was OK.

Why? Why, time after time, does it turn out OK? I think there are three reasons.

First, the people who have commissioned the architecture evaluation really want it to succeed. The architect, developers, and stakeholders assembled at the client's behest also want it to succeed. As a group, they help to keep the exercise marching toward the goal of architectural insight. Second, we are always honest. If we feel that the exercise is derailing, we call a timeout and confer among ourselves, and usually confer with the client. While a small amount of bravado can come in handy during an exercise, we never, ever try to bluff our way through an evaluation. Participants can detect that instinctively, and the evaluation team must never lose the respect of the other participants. Third, the methods are constructed to establish and maintain a steady consensus throughout the exercise. There are no surprises at the end. The participants lay down the ground rules for what constitutes a suitable architecture, and they contribute to the risks uncovered at every step of the way.

So: Do the best job you can. Be honest. Trust the methods. Trust in the goodwill and good intentions of the people you have assembled. And it will be OK. (Adapted from [\[Clements 01b\]](#))

—PCC

21.3. Lightweight Architecture Evaluation

Although we attempt to use time in an ATAM exercise as efficiently as possible, it remains a substantial undertaking. It requires some 20 to 30 person-days of effort from an evaluation team, plus even more for the architect and stakeholders. Investing this amount of time only makes sense on a large and costly project, where the risks of making a major mistake in the architecture are unacceptable.

For this reason, we have developed a Lightweight Architecture Evaluation method, based on the ATAM, for smaller, less risky projects. A Lightweight Architecture Evaluation exercise may take place in a single day, or even a half-day meeting. It may be carried out entirely by members internal to the organization. Of course this lower level of scrutiny and objectivity may not probe the architecture as deeply, but this is a cost/benefit tradeoff that is entirely appropriate for many projects.

Because the participants are all internal to the organization and fewer in number than for the ATAM, giving everyone their say and achieving a shared understanding takes much less time. Hence the steps and phases of a Lightweight Architecture Evaluation can be carried out more quickly. A suggested schedule for phases 1 and 2 is shown in [Table 21.4](#).

Table 21.4. A Typical Agenda for Lightweight Architecture Evaluation

Step	Time Allotted	Notes
1: Present the ATAM	0 hrs	The participants are familiar with the process. This step may be omitted.
2: Present Business Drivers	0.25 hrs	The participants are expected to understand the system and its business goals and their priorities. Fifteen minutes is allocated for a brief review to ensure that these are fresh in everyone's mind and that there are no surprises.
3: Present Architecture	0.5 hrs	Again, all participants are expected to be familiar with the system and so a brief overview of the architecture, using at least module and C&C views, is presented and 1 to 2 scenarios are traced through these views.
4: Identify Architectural Approaches	0.25 hrs	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5: Generate Quality Attribute Utility Tree	Variable 0.5 hrs – 1.5 hrs	Scenarios might exist: part of previous evals, part of design, part of requirements elicitation. If you've got 'em, use 'em and make them into a tree. Half hour. Otherwise, it will take longer. A utility tree should already exist; the team reviews the existing tree and updates it, if needed, with new scenarios, new response goals, or new scenario priorities and risk assessments.
6: Analyze Architectural Approaches	2–3 hrs	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.
7: Brainstorm and Prioritize Scenarios	0 hrs	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8: Analyze Architectural Approaches	0 hrs	This step is also omitted, since all analysis is done in step 6.
9: Present Results	0.5 hrs	At the end of an evaluation, the team reviews the existing and newly discovered risks, non-risks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.
TOTAL	4–6 hrs	

There is no final report, but (as in the regular ATAM) a scribe is responsible for capturing results, which can then be distributed and serve as the basis for risk remediation.

An entire Lightweight Architecture Evaluation can be prosecuted in less than a day—perhaps an afternoon. The results will depend on how well the assembled team understands the goals of the method, the techniques of the method, and the system itself. The evaluation team, being internal, is typically *not* objective, and this may compromise the value of its results—one tends to

hear fewer new ideas and fewer dissenting opinions. But this version of evaluation is inexpensive, easy to convene, and relatively low ceremony, so it can be quickly deployed whenever a project wants an architecture quality assurance sanity check.

21.4. Summary

If a system is important enough for you to explicitly design its architecture, then that architecture should be evaluated.

The number of evaluations and the extent of each evaluation may vary from project to project. A designer should perform an evaluation during the process of making an important decision. Lightweight evaluations can be performed several times during a project as a peer review exercise.

The ATAM is a comprehensive method for evaluating software architectures. It works by having project decision makers and stakeholders articulate a precise list of quality attribute requirements (in the form of scenarios) and by illuminating the architectural decisions relevant to carrying out each high-priority scenario. The decisions can then be understood in terms of risks or nonrisks to find any trouble spots in the architecture.

Lightweight Architecture Evaluation, based on the ATAM, provides an inexpensive, low-ceremony architecture evaluation that can be carried out in an afternoon.

21.5. For Further Reading

For a more comprehensive treatment of the ATAM, see [\[Clements 01b\]](#).

Multiple case studies of applying the ATAM are available. They can be found by going to [www.sei.cmu.edu/library](#) and searching for "ATAM case study."

To understand the historical roots of the ATAM, and to see a second (simpler) architecture evaluation method, you can read about the software architecture analysis method (SAAM) in [\[Kazman 94\]](#).

Several lighter weight architecture evaluation methods have been developed. They can be found in [\[Bouwers 10\]](#), [\[Kanwal 10\]](#), and [\[Bachmann 11\]](#).

Maranzano et al. have published a paper dealing with a long tradition of architecture evaluation at AT&T and its successor companies [\[Maranzano 05\]](#).

21.6. Discussion Questions

1. Think of a software system that you're working on. Prepare a 30-minute presentation on the business drivers for this system.
2. If you were going to evaluate the architecture for this system, who would you want to participate? What would be the stakeholder roles and who could you get to represent those roles?
3. Use the utility tree that you wrote for the ATM in [Chapter 16](#) and the design that you sketched for the ATM in [Chapter 17](#) to perform the scenario analysis step of the ATAM. Capture any risks and nonrisks that you discover. Better yet, perform the analysis on the design carried out by a colleague.
4. It is not uncommon for an organization to evaluate two competing architectures. How would you modify the ATAM to produce a quantitative output that facilitates this comparison?
5. Suppose you've been asked to evaluate the architecture for a system in confidence. The architect isn't available. You aren't allowed to discuss the evaluation with any of the system's stakeholders. How would you proceed?
6. Under what circumstances would you want to employ a full-strength ATAM and under what circumstances would you want to employ a Lightweight Architecture Evaluation?

22. Management and Governance

How does a project get to be a year behind schedule? One day at a time.

—Fred Brooks

In this chapter we deal with those aspects of project management and governance that are important for an architect to know. The project manager is the person with whom you, as the architect, must work most closely, from an organizational perspective, and consequently it is important for you to have an understanding of the project manager's problems and the techniques available to solve those problems. We will deal with project management from the perspectives of planning, organizing, implementing, and measuring. We will also discuss various governance issues associated with architecture.

In this chapter, we advocate a middleweight approach to architecture. It has the following aspects:

- Design the software architecture
- Use the architecture to develop realistic schedules
- Use incremental development to get to market quickly

Architecture is most useful in medium- to large-scale projects—projects that typically have multiple teams, too much complexity for any individual to fully comprehend, substantial investment, and multiyear duration. For such projects the teams need to coordinate, quality attribute problems are not easily corrected, and management demands both short time to market and adequate oversight. Lightweight management methods do not provide for a framework to guide team coordination and frequently require extensive restructuring to repair quality attribute problems. Heavyweight management is usually associated with heavy oversight and a great emphasis on contractual commitments. In some contexts, this is unavoidable, but it has an inherent overhead that slows down development.

22.1. Planning

The planning for a project proceeds over time. There is an initial plan that is necessarily top-down to convince upper management to build this system and give them some idea of the cost and schedule. This top-down schedule is inherently going to be incorrect, possibly by large amounts. It does, however, enable the project manager to educate upper managers as to different elements necessary in software development. According to Dan Paulish, based on his experience at Siemens Corporation, some rules of thumb that can be used to estimate the top-down schedule for medium-sized (~150 KSLOC) projects are these:

- Number of components to be estimated: ~150
- Paper design time per component: ~4 hours
- Time between engineering releases: ~8 weeks
- Overall project development allocation:
 - 40 percent design: 5 percent architectural, 35 percent detailed
 - 20 percent coding
 - 40 percent testing

Once the system has been given a go-ahead and a budget, the architecture team is formed and produces an initial architecture design. The budget item deserves some further mention. One case is that the budget is for the whole project and includes the schedule as well. We will call this case top-down planning. The second case is that the budget is just for the architecture design phase. In this case, the overall project budget emerges from the architecture design phase. This provides a gate that the team has to pass through and gives the holders of the purse strings a chance to consider whether the value of the project is worth the cost.

We now describe a merged process that includes both the top-down budget and schedule as well as a bottom-up budget and schedule that emerges from the architecture design phase.

The architecture team produces the initial architecture design and the release plans for the system: what features will be released and when the releases will occur. Once an initial architecture design has been produced, then leads for the various pieces of the project can be assigned and they can build their teams. The definition of the various pieces of the project and their assignment to teams is sometimes called the work breakdown structure. At this point, cost and schedule estimates from the team leads and an overall project schedule can be produced. This bottom-up schedule is usually much more accurate than the top-down schedule, but there may also be significant differences due to differing assumptions. These differences between the top-down and bottom-up schedules need to be discussed and sometimes negotiated. Finally, a software development plan can be written that shows the initial (internal) release as the architectural skeleton with feature-oriented incremental development after that. The features to be in each release are developed in consultation with marketing.

[Figure 22.1](#) shows a process that includes both a top-down schedule and a bottom-up schedule.

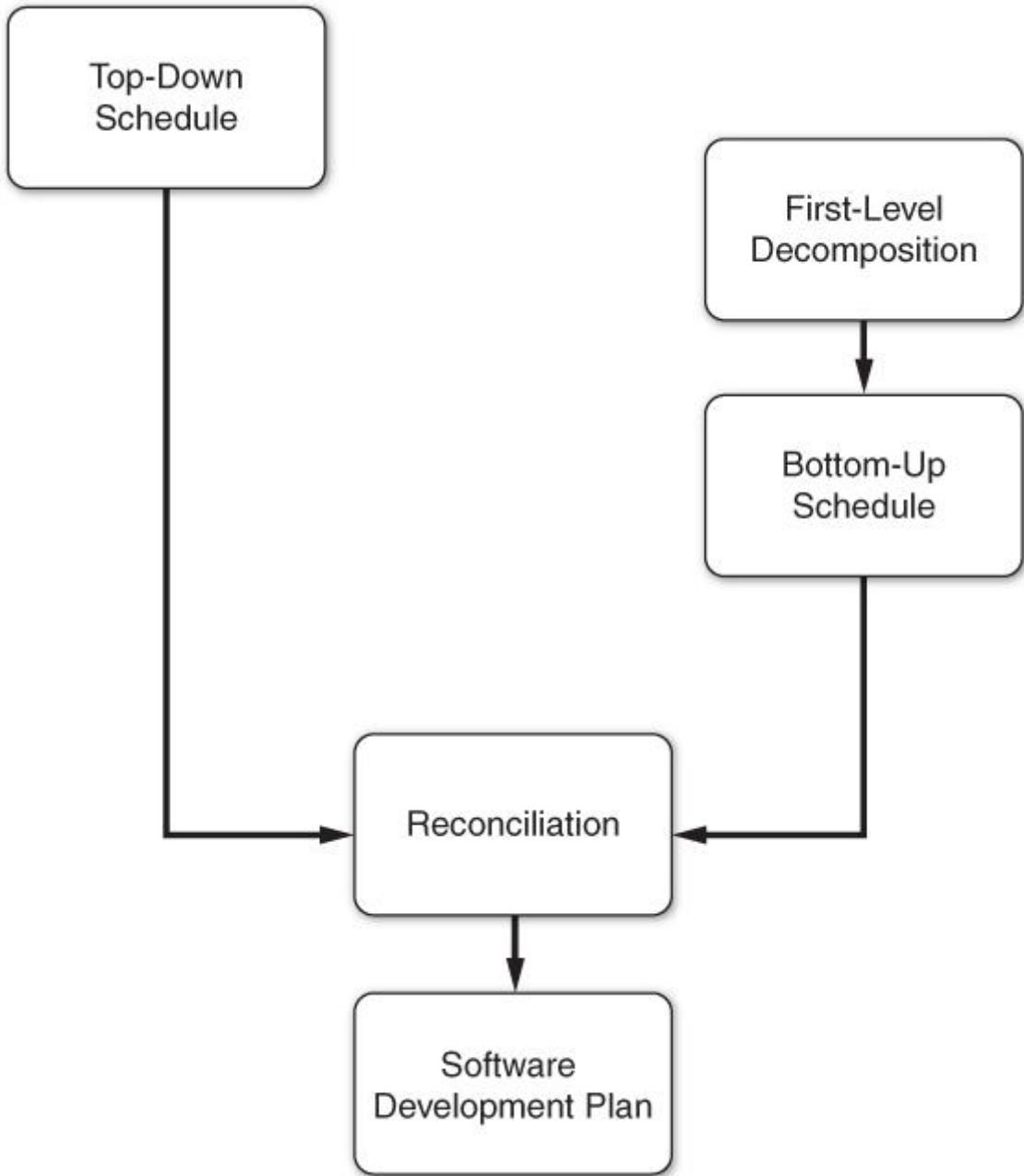


Figure 22.1. Overview of planning process

Once the software development plan has been written, the teams can determine times and groups for integration and can define the coordination needs for the various projects. As we will see in the subsection on global development in [Section 22.2](#), the coordination needs for distributed teams can be significant.

22.2. Organizing

Some of the elements of organizing a project are team organization, division of responsibilities between project manager and software architect, and planning for global or distributed development.

Software Development Team Organization

Once the architecture design is in place, it can be used to define the project organization. Each member of the team that designed the software architecture becomes the lead for a team whose responsibility is to implement a portion of the architecture. Thus, responsibility for fleshing out and implementing the design is distributed to those who had a role in its definition.

In addition, many support functions such as writing user documentation, system testing, training, integration, quality assurance, and configuration management are done in a “matrix” form. That is, individuals who are “matrixed” report to one person—a functional manager—for their tasking and professional advancement and to another individual (or to several individuals)—project managers—for their project responsibilities. Matrix organizations have the advantage of being able to allocate and balance resources as needed rather than assign them permanently to a project that may have sporadic needs for individuals with particular skills. They have the disadvantage that the people in them tend to work on several projects simultaneously. This can cause problems such as divided loyalties or competition among projects for resources.

Typical roles within a software development team are the following:

- *Team leader*—manages tasks within the team.
- *Developer*—designs and implements subsystem code.
- *Configuration manager*—performs regular builds and integration tests. This role can frequently be shared among multiple software development teams.
- *System test manager*—system test and acceptance testing.
- *Product manager*—represents marketing; defines feature sets and how system being developed integrates with other systems in a product suite.

Division of Responsibilities between Project Manager and Software Architect

One of the important relations within a team is between the software architect and the project manager. You can view the project manager as responsible for the external-facing aspects of the project and the software architect as responsible for the internal aspects of the project. This division will only work if the external view accurately reflects the internal situation and the internal activities accurately reflect the expectations of the external stakeholders. That is, the project manager should know, and reflect to management, the progress and the risks within the project, and the software architect should know, and reflect to developers, stakeholder concerns. The relationship between the project manager and the software architect will have a large impact on the success of a project. They need to have a good working relation and be mindful of the roles they are filling and the boundaries of those roles.

We use the knowledge areas for project management taken from the Project Management Body of Knowledge (PMBOK) to show the duties of these two roles in a variety of categories. [Table 22.1](#) (on the next page) gives the knowledge area in the language of the PMBOK, defines the knowledge area in English, what that means in software terms, and how the project manager and the software architect collaborate to satisfy that category.

Table 22.1. Division of Responsibilities between Project Manager and Architect

PMBOK Knowledge Area	Description	Task	Project Manager	Software Architect
Project Integration Management	Ensuring that the various elements of the project are properly coordinated	Developing, overseeing, and updating the project plan. Managing change control process.	Organize project, manage resources, budgets and schedules. Define metrics and metric collection strategy. Oversee change control process.	Create, design, and organize team around design. Manage dependencies. Implement the capture of the metrics. Orchestrate requests for changes. Ensure that appropriate IT infrastructure exists.
Project Scope Management	Ensuring that the project includes all of the work required and only the work required	Requirements	Negotiate project scope with marketing and software architect.	Elicit, negotiate, and review run-time requirements and generate development requirements. Estimate cost, schedule, and risk of meeting requirements.
Project Time Management	Ensuring that the project completes in a timely fashion	Work breakdown structure and completion tracking. Project network diagram with dates.	Oversee progress against schedule. Help define work breakdown structure. Schedule coarse activities to meet deadlines.	Help define work breakdown structure. Define tracking measures. Recommend assignment of resources to software development teams.
Project Cost Management	Ensuring that the project is completed within the required budget	Resource planning, cost estimation, cost budgeting	Calculate cost to completion at various stages. Make decisions regarding build/buy and allocation of resources.	Gather costs from individual teams. Make recommendations regarding build/buy and resource allocations.
Project Quality Management	Ensuring that the project will satisfy the needs for which it was undertaken	Quality and metrics	Define productivity, size, and project-level quality measures.	Design for quality and track system against design. Define code-level quality metrics.
Project Human Resource Management	Ensuring that the project makes the most effective use of the people involved with the project	Managing people and their careers	Map skill sets of people against required skill sets. Ensure that appropriate training is provided. Monitor and mentor career paths of individuals. Authorize recruitment.	Define required technical skill sets. Mentor developers about career paths. Recommend training. Interview candidates.
Project Communications Management	Ensuring timely and appropriate generation, collection, dissemination, storage, and disposition of project information	Communicating	Manage communication between team and external entities. Report to upper management.	Ensure communication and coordination among developers. Solicit feedback as to progress, problems, and risks.
Project Risk Management	Identifying, analyzing, and responding to project risk	Risk management	Prioritize risks. Report risks to management. Take steps to mitigate risks.	Identify and quantify risks. Adjust architecture and processes to mitigate risk.
Project Procurement Management	Acquiring goods and services from outside organization	Technology	Procure necessary resources. Introduce new technology.	Determine technology requirements. Recommend technology, training, and tools.

Observe in this table that the project manager is responsible for the business side of the project—providing resources, creating and managing the budget and schedule, negotiating with marketing, ensuring quality—and the software architect is responsible for the technical side of the project—achieving quality, determining measures to be used, reviewing requirements for feasibility, generating develop-time requirements, and leading the development team.

Global Development

Most substantial projects today are developed by distributed teams. In many organizations these teams are globally distributed. Some reasons for this trend are the following:

- **Cost.** Labor costs vary depending on location, and there is a perception that moving some development to a low-cost venue will decrease the overall cost of the project. Experience has shown that, at least for software development, savings may only be reaped in the long term. Until the developers in the low-cost venue have a sufficient level of domain expertise and until the management practices are adapted to compensate for the difficulties of distributed development, a large amount of rework must be done, thereby cutting into and perhaps overwhelming any savings from wages.

- *Skill sets and labor availability.* Organizations may not be able to hire developers at a single location: relocation costs are high, the size of the developer pool may be small, or the skill sets needed are specialized and unavailable in a single location. Developing a system in a distributed fashion allows for the work to move to where the workers are rather than forcing the workers to move to the work location.
- *Local knowledge of markets.* Developers who are developing variants of a system to be sold in their market have more knowledge about the types of features that are assumed and the types of cultural issues that may arise.

A consequence of performing global or distributed development is that architecture and allocation of responsibilities to teams is more important than in co-located development, where all of the developers are in a single office, or at least in close proximity. For example, assume Module A uses an interface from Module B. In time, as circumstances change, this interface may need to be modified. This means the team responsible for Module B must coordinate with the team responsible for Module A, as indicated in [Figure 22.2](#).

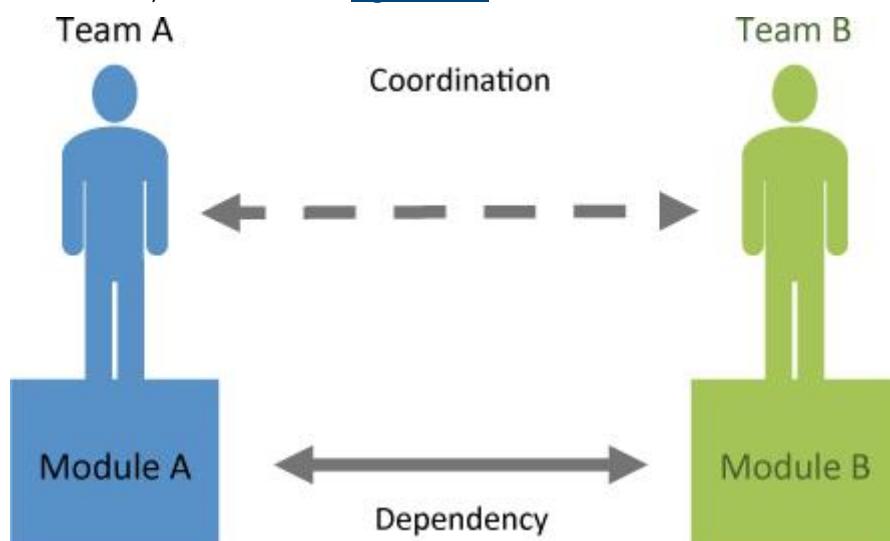


Figure 22.2. If there is a dependency between Module A and Module B, then the teams must coordinate to develop and modify the interface.

Methods for coordination include the following:

- *Informal contacts.* Informal contacts, such as meeting at the coffee room or in the hallway, are only possible if the teams are co-located.
- *Documentation.* Documentation, if it is well written, well organized, and properly disseminated, can be used as a means to coordinate the teams, whether co-located or at a distance.
- *Meetings.* Teams can hold meetings, either scheduled or ad hoc and either face to face or remote, to help bring the team together and raise awareness of issues.
- *Electronic communication.* Various forms of electronic communication can be used as a coordination mechanism, such as email, news groups, blogs, and wikis.

The choice of method depends on many factors, including infrastructure available, corporate culture, language skills, time zones involved, and the number of teams dependent on a particular module. Until an organization has established a working method for coordinating among distributed teams, misunderstandings among the teams are going to cause delays and, in some cases, serious defects in a project.

22.3. Implementing

During the implementation phase of a project, the project manager and architect have a series of decisions to make. In this section we discuss those involving tradeoffs, incremental development, and managing risk.

Tradeoffs

From the project manager's perspective, tradeoffs are between quality, schedule, functionality, and cost. These are the aspects of the project that are important to the external stakeholders, and the external stakeholders are the project manager's constituency. Which of these aspects is most important depends on the project context, and one of the project manager's major responsibilities is to make this determination.

Over time, there is always new functionality that someone wants to have added to the project. Frequently these requests come from the marketing department. It is important that the consequences of these new requirements, in terms of cost and schedule, be communicated to all concerned stakeholders. This is an area where the project manager and the architect must cooperate. What appear to be small requirements changes from an outsider's perspective can, at times, require major modifications to the architecture and consequently delay a project significantly.

The project manager's first response to creeping functionality is to resist it. Acting as a gatekeeper for the project and shielding it from distractions is a portion of the job description. One technique that is frequently used to manage change is a change control board. Bureaucracy can, at times, be your friend. Change control boards are committees set up for the purpose of managing change within a project. The original architecture team members are good candidates to sit on the change control board. Before changing an interface, for example, the impact on those modules that depend on the interface needs to be considered.

Any change to the architecture will incur costs, and it is the architect's responsibility to be the gatekeeper for such changes. A change in the architecture implies changes in code, changes in the architecture documentation, and perhaps changes in build-time tools that enforce architectural conformance.

Documentation is especially important in distributed development. Co-located teams have a variety of informal coordination possibilities, such as going to the next office or meeting in the coffee room or the hall. Remote teams do not have these informal mechanisms and so must rely on more formal mechanisms such as documentation; team members must have the initiative to talk to each other when doubts arise. One company mounted a webcam on each developer's desktop to facilitate personal communication.

Incremental Development

Recall that the software development plan lays out the overall schedule. Every six to eight weeks a new release should be available and the specifics of the next release are decided. Forty percent of a typical project's effort is devoted to testing. This means that testing should begin as soon as possible. Testing for a release can begin once the forward development has begun on the next release. The schedule also has to accommodate repairing the faults uncovered by the testing. This leads to a release being in one of three states:

1. *Planning.* This occurs toward the end of the prior development release. Enough of the prior release must be completed to understand what will be unfinished in that release and must be carried forward to the next one. At this stage, the software development plan is updated.
2. *Development.* The planned release is coded. We will discuss below how the project manager and architect track progress on the release. Daily builds and automated testing can give some insight into problems during development.
3. *Test and repair.* The release is tested through exercise of the test plan. In [Chapter 19](#) we described how the architecture can inform the test plan and even obviate the need for certain types of testing. The problems found during test are repaired or are carried forward to the next release.

Tracking Progress

The project manager can track progress through personal contact with developers (this tends to not scale up well), formal status meetings, metrics, and risk management. Metrics will be discussed in the next section. Personal contact involves checking with key personnel individually to determine progress. These are one-on-one meetings, either scheduled or unscheduled.

Meetings, in general, are either status or working meetings. The two types of meetings should not be mixed. In a status meeting, various teams report on progress. This allows for communication among the teams. Issues raised at status meetings should be resolved outside of these meetings—either by individuals or by separately scheduled working meetings. When an issue is raised at a status meeting, a person should be assigned to be responsible for the resolution of that issue.

Meetings are expensive. Holding effective meetings is an important skill for a manager, whether the project manager or the architect. Meetings should have written agendas that are circulated before the meetings begin, attendees should be expected to have done some prework for the meeting (such as read-ahead), and only essential individuals should attend.

One of the outputs of status meetings is a set of risks. A risk is a potential problem together with the consequences if it occurs. The likelihood of the risk occurring should also be recorded. Risks are also raised at reviews. We have discussed architecture evaluation, and it is important from a project management perspective that reviews are included in the schedule. These can be code reviews, architecture reviews, or requirements reviews. Risks are also raised by developers. They are the ones who have the best perspective on potential problems at the implementation level. Architecture evaluation is also important because it is a source of discovering risks.

The project manager must prioritize the risks, frequently with the assistance of the architect, and, for the most serious risks, develop a mitigation strategy. Mitigating risks is also a cost, and so implementing the strategy to reduce a risk depends on its priority, likelihood of occurrence, and cost if it does occur.

22.4. Measuring

Metrics are an important tool for project managers. They enable the manager to have an objective basis both for their own decision making and for reporting to upper management on the progress of the project.

Metrics can be global—pertaining to the whole project—or they may depend on a particular phase of the project. Another important class of metrics is “cost to complete.” We discuss each of these below.

Global Metrics

Global metrics aid the project manager in obtaining an overall sense of the project and tracking its progress over time. All global metrics should be updated from time to time as the project proceeds.

First, the project manager needs a measure of project size. The three most common measures of project size are lines of code, function points, and size of the test suite. None of these is completely satisfactory as a predictor of cost or effort, but these are the most commonly used size metrics in practice.

Schedule deviation is another global metric. Schedule deviation is measured by taking the difference between the planned work time and the actual work time. Once time has passed, it can never be recovered. If a project falls behind in its schedule, a tradeoff must be made between the aspects we mentioned before: schedule, quality, functionality, and cost. As the project proceeds, schedule deviation indicates a failure of estimation or an unforeseen occurrence. By drilling down in this metric and discovering which teams are slipping, the project manager can decide to reallocate resources, if necessary.

Developer productivity is another metric that the project manager can track. The project manager should look for anomalies in the productivity of the developers. An anomaly indicates a potential problem that should be investigated: perhaps a developer is inadequately trained for a task, or perhaps the task was improperly estimated. Earned value management is one technique for measuring the productivity of developers.

Finally, defects should be tracked. Again, anomalies in the number of defects discovered over time indicate a potential problem that should be investigated. Not only defects but technical debt should be tracked as well (see [Chapter 15](#)).

All of these measures have both a historical basis and a project basis. The historical basis is used to make the initial estimates and then the project basis is used for ongoing management activities.

Phase Metrics and Costs to Complete

Open issues should be kept for each phase. For example, until a design is complete, there are always open issues. These should be tracked and additional resources allocated if they are not resolved in a timely fashion. Risks represent open issues that should be tracked, as does the project backlog.

Unmitigated risks from reviews are treated in a similar fashion. We have already discussed risk management, but one item a project manager can report to upper management is the number of high-priority risks and the status of their mitigation.

Costs to complete is a bottom-up measure that derives from the bottom-up schedule. Once the various pieces of the architecture have been assigned to teams, then the teams take ownership of their schedule and the cost to complete their pieces.

22.5. Governance

Up to this point, we have maintained a focus in this chapter. We have focused on the project manager as the embodiment of the project and have not discussed the external forces that act on the project manager. The topic of governance deals directly with these other forces. The Open Group defines architecture governance as “the practice and orientation by which enterprise architectures and other architectures are managed and controlled.” Implicit in this definition is the idea that the project—which is the focus of this book—exists in an organizational context. This context will mediate the interactions of the system being constructed with the other systems in the organization.

The Open Group goes on to identify four items as responsibilities of a governance board:

- Implementing a system of controls over the creation and monitoring of all architectural components and activities, to ensure the effective introduction, implementation, and evolution of architectures within the organization
- Implementing a system to ensure compliance with internal and external standards and regulatory obligations
- Establishing processes that support effective management of the above processes within agreed parameters
- Developing practices that ensure accountability to a clearly identified stakeholder community, both inside and outside the organization

Note the emphasis on processes and practices. Maintaining an effective governance process without excessive overhead is a line that is difficult to maintain for an organization.

The problem comes about because each system that exists in an enterprise has its own stakeholders and its own internal governance processes. Creating a system that utilizes a collection of other systems raises the issue of who is in control.

Consider the following example. Company A has a collection of products that cover different portions of a manufacturing facility. One collection of systems manages the manufacturing process, another manages the processes by which various portions of the end product are integrated, and a third collection manages the enterprise. Each of these collections of systems has its own set of customers.

Now suppose that the board of directors wishes to market the collection as an end-to-end solution for a manufacturing facility. It further turns out that the systems that manage the manufacturing process have a 6-month release cycle because the technology is changing quickly in this area. The systems that manage the integration process have a 9-month release cycle because they are based on a widely used commercial product with a 9-month release cycle. The enterprise systems have a 12-month release cycle because they are based on an organization’s fiscal year and reflect tax and regulatory changes that are likely to occur on fiscal year boundaries. [Table 22.2](#) shows these schedules beginning at date 0.

Table 22.2. Release Schedules of Different Types of Products

Manufacturing Process Control	Integration Process Control	Enterprise Management
Version 1—date 0	Version 1—date 0	Version 1—date 0
Version 2—date 6 months	Version 2—date 9 months	Version 2—date 12 months
Version 3—date 12 months	Version 3—date 18 months	Version 3—date 24 months

What should the release schedule be for the combined end-to-end solution? Recall that each of these sets of products has reasons for their release schedule and each has its own set of customers who will not be receptive to changes in the release schedule. This is typical of the sort of problem that a governance committee deals with.

22.6. Summary

A project must be planned, organized, implemented, tracked, and governed.

The plan for a project is initially developed as a top-down schedule with an acknowledgement that it is only an estimate. Once the decomposition of the system has been done, a bottom-up schedule can be developed. The two must be reconciled, and this becomes the basis for the software development plan.

Teams are created based on the software development plan. The software architect and the project manager must coordinate to oversee the implementation. Global development creates a need for an explicit coordination strategy that is based on more formal methods than needed for co-located development.

The implementation itself causes tradeoffs between schedule, function, and cost. Releases are done in an incremental fashion and progress is tracked by both formal metrics and informal communication.

Larger systems require formal governance mechanisms. The issue of who has control over a particular portion of the system may prevent some business goals from being realized.

22.7. For Further Reading

Dan Paulish has written an excellent book on managing in an architecture-centric environment—[\[Paulish 02\]](#)—and much of the material in this chapter is adapted from his book.

The Agile community has independently arrived at a medium-weight management process that advocates up-front architecture. See[\[Coplein 10\]](#) for an example of the Agile community’s description of architecture.

The responsibilities of a governance board can be found in The Open Group Architecture Framework (TOGAF), produced by the Open Group. www.opengroup.org/architecture/togaf8-doc/arch/chap26.html

Basic concepts of project management are covered in [\[IEEE 11\]](#).

Using concepts of lean manufacturing, Kanban is a method for scheduling the production of a system [\[Ladas 09\]](#).

Earned value management is discussed in en.wikipedia.org/wiki/Earned_value_management

22.8. Discussion Questions

1. What are the reasons that top-down and bottom-up schedule estimates differ and how would you resolve this conflict in practice?

- 2.** Generic project management practices advocate creating a work breakdown structure as the first artifact produced by a project. What is wrong with this practice from an architectural perspective?
- 3.** If you were managing a globally distributed team, what architectural documentation artifacts would you want to create first?
- 4.** If you were managing a globally distributed team, what aspects of project management would have to change to account for cultural differences?
- 5.** Enumerate three different global metrics and discuss the advantages and disadvantages of each?
- 6.** How could you use architectural evaluation as a portion of a governance plan?
- 7.** In [Chapter 1](#) we described a work assignment structure for software architecture, which can be documented as a work assignment view. (Because it maps software elements—modules—to a nonsoftware environmental structure—the organizational units that will be responsible for implementing the modules—it is a kind of allocation view.) Discuss how documenting a work assignment view for your architecture provides a vehicle for software architects and managers to work together to staff a project. Where is the dividing line between the part of the work assignment view that the architect should provide and the part that the manager should provide?

Part Four. Architecture and Business

Perhaps the most important job of an architect is to be a fulcrum where business and technical decisions meet and interact. The architect is responsible for many aspects of the business, and must be continually translating business needs and goals into technical realizations, and translating technical opportunities and limitations into business consequences.

In this section of the book, we explore some of the most important business consequences of architecture. This includes treating software architecture decisions as business investments, dealing with the organizational aspects of architecture (such as organizational learning and knowledge management), and looking at architecture as the key enabler of software product lines.

In [Chapter 23](#) we discuss the economic implications of architectural decisions and provide a method—called the CBAM, or Cost Benefit Analysis Method—for making rational, business-driven architectural choices. The CBAM builds upon other architecture methods and principles that you have already seen in this book—scenarios, quality attributes, active stakeholder involvement—but adds a new twist to the evaluation of architectural improvements: an explicit consideration of the utility that an architectural improvement brings.

In [Chapter 24](#) we consider the fact that architectures are created by actual human beings, called *architects* working in actual *organizations*. This chapter considers the questions of how to foster individual competence—that is, how to create competent architects—and how to create architecturally competent organizations.

Finally, in [Chapter 25](#) we look at the concept of software product lines. Not surprisingly, we find that software architectures are the most important component of software-intensive product lines.

23. Economic Analysis of Architectures

Arthur Dent: "I think we have different value systems." Ford Prefect: "Well mine's better."

—Douglas Adams, *Mostly Harmless*

Thus far, we have been primarily investigating the relationships between architectural decisions and the quality attributes that the architecture's stakeholders have deemed important: If I make this architectural decision, what effect will it have on the quality attributes? If I have to achieve that quality attribute requirement, what architectural decisions will do the trick?

As important as this effort is, this perspective is missing a crucial consideration: What are the economic implications of an architectural decision?

Usually an economic discussion is focused on costs, primarily the costs of building the system in the first place. Other costs, often but not always downplayed, include the long-term costs incurred through cycles of maintenance and upgrade. However, as we argue in this chapter, as important as costs are the benefits that an architectural decision may bring to an organization.

Given that the resources for building and maintaining a system are finite, there must be a rational process that helps us choose among architectural options, during both an initial design phase and subsequent upgrade periods. These options will have different costs, will consume differing amounts of resources, will implement different features (each of which brings some benefit to the organization), and will have some inherent risk or uncertainty. To capture these aspects, we need economic models of software that take into account costs, benefits, risks, and schedule implications.

23.1. Decision-Making Context

As we saw in [Chapter 16](#), business goals play a key role in requirements for architectures. Because major architectural decisions have technical and economic implications, the business goals behind a software system should be used to directly guide those decisions. The most immediate economic implication of a business goal decision on an architecture is how it affects the cost of implementing the system. The quality attributes achieved by the architecture decisions have additional economic implications because of the benefits (which we call *utility*) that can be derived from those decisions; for example, making the system faster or more secure or easier to maintain and update. It is this interplay between the costs and the benefits of architectural decisions that guides (and torments) the architect. [Figure 23.1](#) shows this interplay.

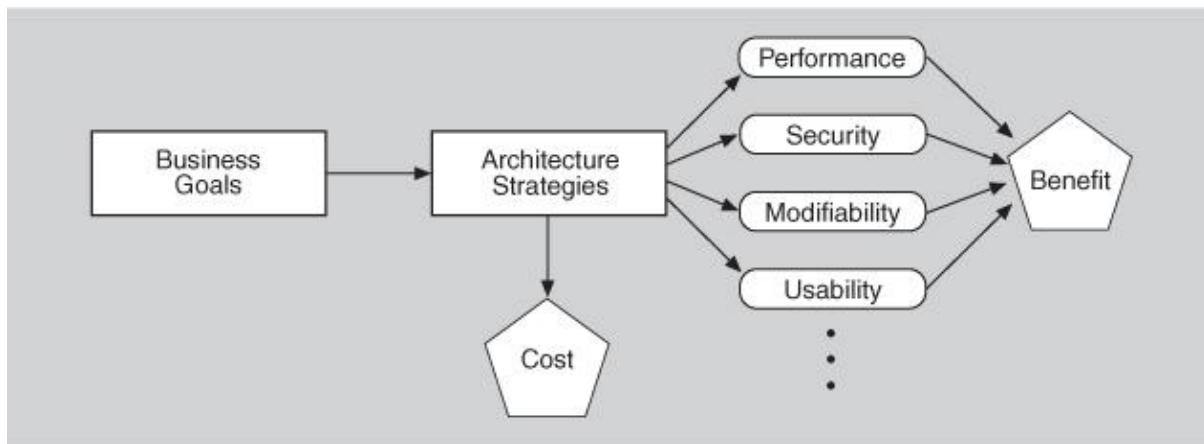


Figure 23.1. Business goals, architectural decisions, costs, and benefits

For example, using redundant hardware to achieve a desired level of availability has a cost; checkpointing to a disk file has a different cost. Furthermore, both of these architectural decisions will result in (presumably different) measurable levels of availability that will have some value to the organization developing the system. Perhaps the organization believes that its stakeholders will pay more for a highly available system (a telephone switch or medical monitoring software, for example) or that it will be sued if the system fails (for example, the software that controls antilock brakes in an automobile).

Knowing the costs and benefits associated with particular decisions enables reasoned selection from among competing alternatives. The economic analysis does not make decisions for the stakeholders, just as a financial advisor does not tell you how to invest your money. It simply aids in the elicitation and documentation of *value for cost* (VFC): a function of the costs, benefits, and uncertainty of a “portfolio” of architectural investments. It gives the stakeholders a framework within which they can apply a rational decision-making process that suits their needs and their risk aversion.

Economic analysis isn’t something to apply to every architectural decision, but rather to the most basic ones that put an overarching *architectural strategy* in place. It can help you assess the viability of that strategy. It can also be the key to objective selection among competing strategies, each of which might have advocates pushing their own self-interests.

23.2. The Basis for the Economic Analyses

We now describe the key ideas that form the basis for the economic analyses. The practical realization of these ideas can be packaged in a variety of ways, as we describe in [Section 23.3](#). Our goal here is to develop the theory underpinning a measure of VFC for various architectural strategies in light of scenarios chosen by the stakeholders.

We begin by considering a collection of scenarios generated as a portion of requirements elicitation, an architectural evaluation, or specifically for the economic analysis. We examine how these scenarios differ in the values of their projected responses and we then assign utility to those

values. The utility is based on the importance of each scenario being considered with respect to its anticipated response value.

Armed with our scenarios, we next consider the architectural strategies that lead to the various projected responses. Each strategy has a cost, and each impacts multiple quality attributes. That is, an architectural strategy could be implemented to achieve some projected response, but while achieving that response, it also affects some other quality attributes. The utility of these “side effects” must be taken into account when considering a strategy’s overall utility. It is this overall utility that we combine with the project cost of an architectural strategy to calculate a final VFC measure.

Utility-Response Curves

Our economic analysis uses quality attribute scenarios (from [Chapter 4](#)) as the way to concretely express and represent specific quality attributes. We vary the values of the responses, and ask what the utility is of each response. This leads to the concept of a utility-response curve.

Each scenario’s stimulus-response pair provides some utility (value) to the stakeholders, and the utility of different possible values for the response can be compared. This concept of utility has roots that go back to the eighteenth century, and it is a technique to make comparable very different concepts. To help us make major architectural decisions, we might wish to compare the value of high performance against the value of high modifiability against the value of high usability, and so forth. The concept of utility lets us do that.

Although sometimes it takes a little prodding to get them to do it, stakeholders can express their needs using concrete response measures, such as “99.999 percent available.” But that leaves open the question of how much they would value slightly less demanding quality attributes, such as “99.99 percent available.” Would that be almost as good? If so, then the lower cost of achieving that lower value might make that the preferred option, especially if achieving the higher value was going to play havoc with another quality attribute like performance. Capturing the utility of alternative responses of a scenario better enables the architect to make tradeoffs involving that quality attribute.

We can portray each relationship between a set of utility measures and a corresponding set of response measures as a graph—a utility-response curve. Some examples of utility-response curves are shown in [Figure 23.2](#). In each, points labeled a, b, or c represent different response values. The utility-response curve thus shows utility as a function of the response value.

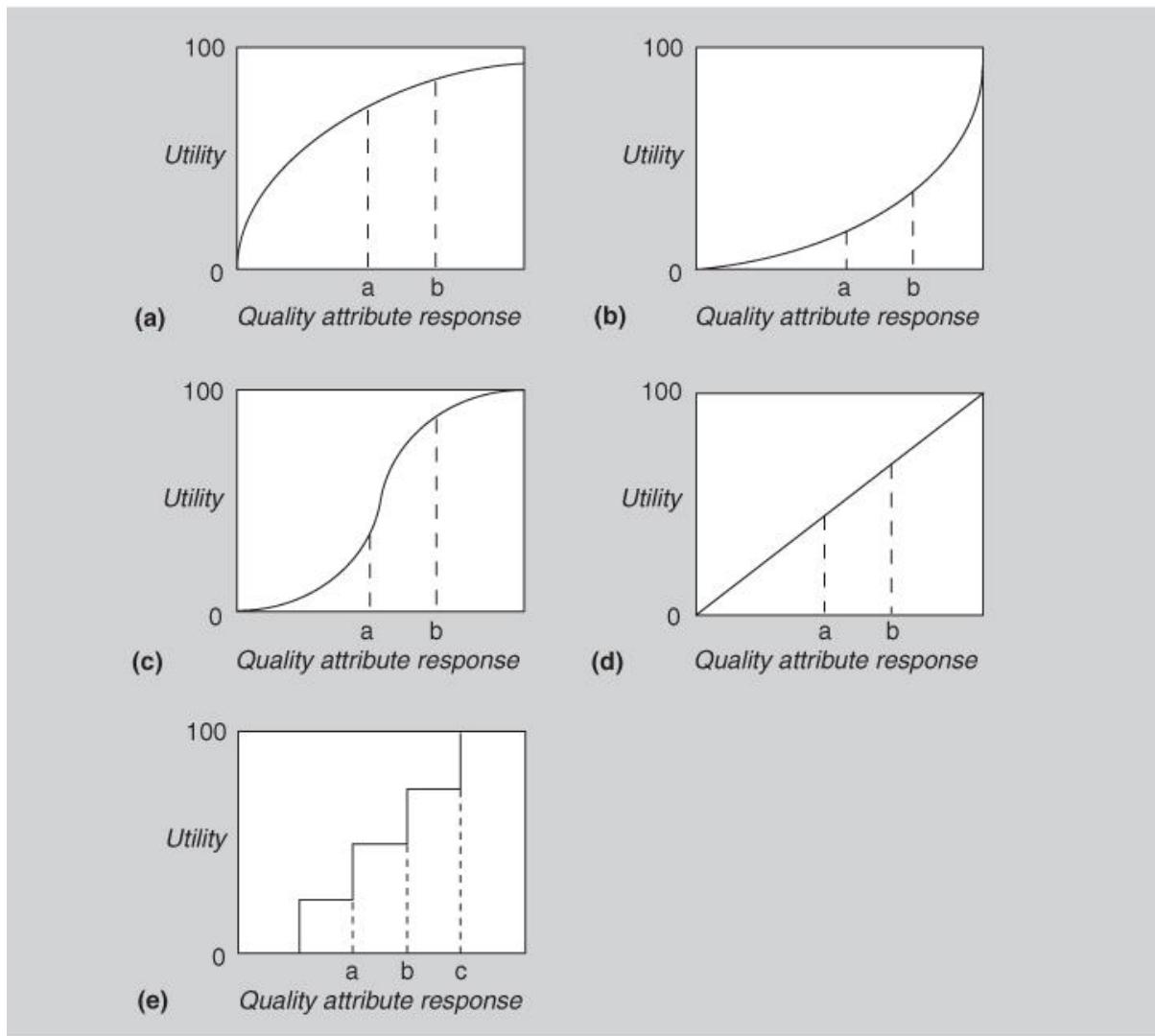


Figure 23.2. Some sample utility-response curves

The utility-response curve depicts how the utility derived from a particular response varies as the response varies. As seen in [Figure 23.2](#), the utility could vary nonlinearly, linearly, or even as a step function. For example, graph (c) portrays a steep rise in utility over a narrow change in a quality attribute response level. In graph (a), a modest change in the response level results in only a very small change in utility to the user.

In [Section 23.3](#) we illustrate some ways to engage stakeholders to get them to construct utility curves.

Weighting the Scenarios

Different scenarios will have different importance to the stakeholders; in order to make a choice of architectural strategies that is best suited to the stakeholders' desires, we must weight the scenarios. It does no good to spend a great deal of effort optimizing a particular scenario in which the stakeholders actually have very little interest. [Section 23.3](#) presents a technique for applying weights to scenarios.

Side Effects

Every architectural strategy affects not only the quality attributes it was selected to achieve, but also other quality attributes as well. As you know by now, these side effects on other quality attributes are often negative. If those effects are too negative, we must make sure there is a

scenario for the side effect attribute and determine its utility-response curve so that we can add its utility to the decision-making mix. We calculate the benefit of applying an architectural strategy by summing its benefits to all relevant quality attributes; for some quality attributes the benefit of a strategy might be negative.

Determining Benefit and Normalization

The overall benefit of an architectural strategy across quality attribute scenarios is the sum of the utility associated with each one, weighted by the importance of the scenario. For each architectural strategy i , its benefit B_i over j scenarios (each with weight W_j) is

$$B_i = \sum_j (b_{i,j} \times W_j)$$

Referring to [Figure 23.2](#), each $b_{i,j}$ is calculated as the change in utility (over whatever architectural strategy is currently in place, or is in competition with the one being considered) brought about by the architectural strategy with respect to this scenario:

$$b_{i,j} = U_{expected} - U_{current}$$

That is, the utility of the expected value of the architectural strategy minus the utility of the current system relative to this scenario.

Calculating Value for Cost

The VFC for each architectural strategy is the ratio of the total benefit, B_i , to the cost, C_i , of implementing it:

$$VFC = B_i / C_i$$

The cost C_i is estimated using a model appropriate for the system and the environment being developed, such as a cost model that estimates implementation cost by measuring an architecture's interaction complexity. You can use this VFC score to rank-order the architectural strategies under consideration.

Consider curves (a) and (b) in [Figure 23.2](#). Curve (a) flattens out as the quality attribute response improves. In this case, it is likely that a point is reached past which VFC decreases as the quality attribute response improves; spending more money will not yield a significant increase in utility. On the other hand, curve (b) shows that a small improvement in quality attribute response can yield a very significant increase in utility. In that situation, an architectural strategy whose VFC is low might rank significantly higher with a modest improvement in its quality attribute response.

23.3. Putting Theory into Practice: The CBAM

With the concepts in place we can now describe techniques for putting them into practice, in the form of a method we call the Cost Benefit Analysis Method (CBAM). As we describe the method, remember that, like all of our stakeholder-based methods, it could take any of the forms for stakeholder interaction that we discussed in the introduction to [Part III](#).

Practicalities of Utility Curve Determination

To build the utility-response curve, we first determine the quality attribute levels for the best-case and worst-case situations. The best-case quality attribute level is that above which the stakeholders foresee no further utility. For example, a system response to the user of 0.1 second is perceived as instantaneous, so improving it further so that it responds in 0.03 second has no additional utility. Similarly, the worst-case quality attribute level is a minimum threshold above which a system *must* perform; otherwise it is of no use to the stakeholders. These levels—best-case and worst-case—are assigned utility values of 100 and 0, respectively. We then determine the current and desired utility levels for the scenario. The respective utility values (between 0 and 100) for various alternative strategies are elicited from the stakeholders, using the best-case and worst-case values as reference points. For example, our current design provides utility about half

as good as we would like, but an alternative strategy being considered would give us 90 percent of the maximum utility. Hence, the current utility level is set to 50 and the desired utility level is set to 90.

In this manner the utility curves are generated for all of the scenarios.

Show Business or Accounting?

As software architects, what kind of business are we in? One of Irving Berlin's most famous songs is entitled "There's No Business Like Show Business." David Letterman, riffing off this song title, once quipped, "There's no business like show business, but there are several businesses like accounting."

How should we think of ourselves, as architects? Consider two more quotations from famous business leaders:

I never get the accountants in before I start up a business. It's done on gut feeling.

—Richard Branson

It has been my experience that competency in mathematics, both in numerical manipulations and in understanding its conceptual foundations, enhances a person's ability to handle the more ambiguous and qualitative relationships that dominate our day-to-day financial decision-making.

—Alan Greenspan

Architectures are at the fulcrum of a set of business, social, and technical decisions. A poor decision in any dimension can be disastrous for an organization. A decision in any one dimension is influenced by the other dimensions. So in our roles as architects, which are we, Alan Greenspan or Richard Branson?

My claim is that, as an industry, we in software are more like Richard Branson. We make decisions that have enormous economic consequences on a gut feeling, without ever examining their financial consequences in a disciplined way. This might be OK if you are an intuitive genius, but it doesn't work well for most of us. Engineering is about making good decisions in a rational, predictable way. For this we need methods.

—RK

Practicalities of Weighting Determination

One method of weighting the scenarios is to prioritize them and use their priority ranking as the weight. So for N scenarios, the highest priority one is given a weight of 1, the next highest is given a weight of $(N-1)/N$, and so on. This turns the problem of weighting the scenarios into one of assigning priorities.

The stakeholders can determine the priorities through a variety of voting schemes. One simple method is to have each stakeholder prioritize the scenarios (from 1 to N) and the total priority of the scenario is the sum of the priorities it receives from all of the stakeholders. This voting can be public or secret.

Other schemes are possible. Regardless of the scheme used, it must make sense to the stakeholders and it must suit their culture. For example, in some corporate environments, everything is done by consensus. In others there is a strict hierarchy, and in still others decisions are made in a democratic fashion. In the end it is up to the stakeholders to make sure that the scenario weights agree with their intuition.

Practicalities of Cost Determination

One of the shortcomings of the field of software architecture is that there are very few cost models for various architectural strategies. There are many software cost models, but they are based on overall system characteristics such as size or function points. These are inadequate to answer the

question of how much does it cost to, for example, use a publish-subscribe pattern in a particular portion of the architecture. There are cost models that are based on complexity of modules (by function point analysis according to the requirements assigned to each module) and the complexity of module interaction, but these are not widely used in practice. More widely used in practice are corporate cost models based on previous experience with the same or similar architectures, or the experience and intuition of senior architects.

Lacking cost models whose accuracy can be assured, architects often turn to estimation techniques. To proceed, remember that an absolute number for cost isn't necessary to rank candidate architecture strategies. You can often say something like "Suppose strategy A costs \$x. It looks like strategy B will cost \$2x, and strategy C will cost \$0.5x." That's enormously helpful. A second approach is to use very coarse estimates. Or if you lack confidence for that degree of certainty, you can say something like "Strategy A will cost a lot, strategy B shouldn't cost very much, and strategy C is probably somewhere in the middle."

CBAM

Now we describe the method we use for economic analysis: the Cost Benefit Analysis Method. CBAM has for the most part been applied when an organization was considering a major upgrade to an existing system and they wanted to understand the utility and value for cost of making the upgrade, or they wanted to choose between competing architectural strategies for the upgrade. CBAM is also applicable for new systems as well, especially for helping to choose among competing strategies. Its key concepts (quality attribute response curves, cost, and utility) do not depend on the setting.

Steps

A process flow diagram for the CBAM is given in [Figure 23.3](#). The first four steps are annotated with the relative number of scenarios they consider. That number steadily decreases, ensuring that the method concentrates the stakeholders' time on the scenarios believed to be of the greatest potential in terms of VFC.

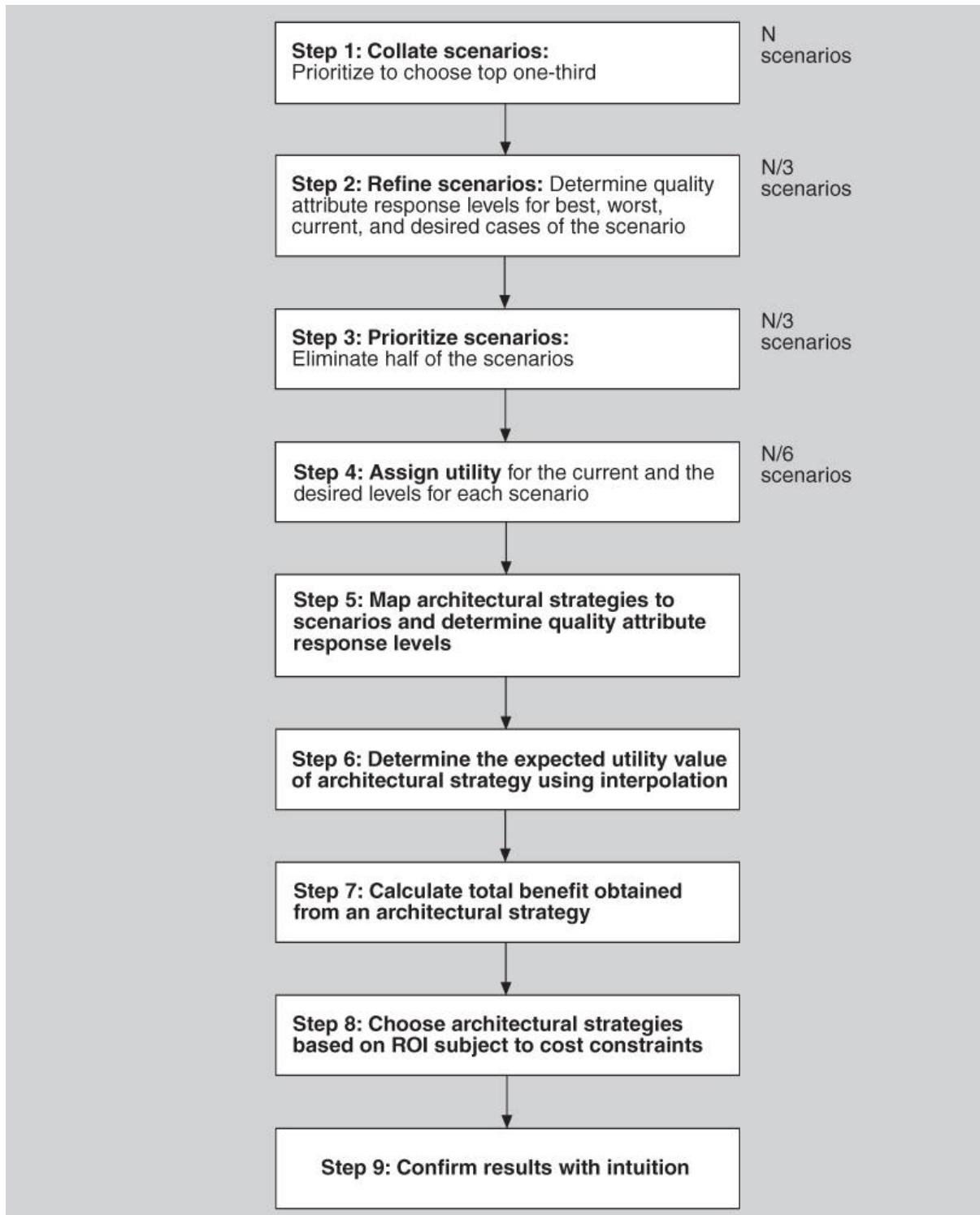


Figure 23.3. Process flow diagram for the CBAM

This description of CBAM assumes that a collection of quality attribute scenarios already exists. This collection might have come from a previous elicitation exercise such as an ATAM exercise (see [Chapter 21](#)) or quality attribute utility tree construction (see [Chapter 16](#)).

The stakeholders in a CBAM exercise include people who can authoritatively speak to the utility of various quality attribute responses, and probably include the same people who were the source of the quality attribute scenarios being used as input. The steps are as follows:

- 1. Collate scenarios.** Give the stakeholders the chance to contribute new scenarios. Ask the stakeholders to prioritize the scenarios based on satisfying the business goals of the system. This can be an informal prioritization using a simple scheme such as "high, medium, low" to rank the scenarios. Choose the top one-third for further study.
- 2. Refine scenarios.** Refine the scenarios chosen in step 1, focusing on their stimulus-response measures. Elicit the worst-case, current, desired, and best-case quality attribute response level for each scenario. For example, a refined performance scenario might tell us that worst-case performance for our system's response to user input is 12 seconds, the best case is 0.1 seconds, and our desired response is 0.5 seconds. Our current architecture provides a response of 1.5 seconds:

Scenario	Worst Case	Current	Desired	Best Case
Scenario #17: Response to user input	12 seconds	1.5 seconds	0.5 seconds	0.1 seconds
...				

- 3. Prioritize scenarios.** Prioritize the refined scenarios, based on stakeholder votes. You give 100 votes to each stakeholder and have them distribute the votes among the scenarios, where their voting is based on the *desired* response value for each scenario. Total the votes and choose the top 50 percent of the scenarios for further analysis. Assign a weight of 1.0 to the highest-rated scenario; assign the other scenarios a weight relative to the highest rated. This becomes the weighting used in the calculation of a strategy's overall benefit. Make a list of the quality attributes that concern the stakeholders.
- 4. Assign utility.** Determine the utility for each quality attribute response level (worst-case, current, desired, best-case) for the scenarios from step 3. You can conveniently capture these utility curves in a table (one row for each scenario, one column for each of the four quality attribute response levels). Continuing our example from step 2, this step would assign utility values from 1 to 100 for each of the latency values elicited for this scenario in step 2:

Scenario	Worst Case	Current	Desired	Best Case
Scenario #17: Response to user input	12 seconds	1.5 seconds	0.5 seconds	0.1 seconds
	Utility 5	Utility 50	Utility 80	Utility 85

- 5. Map architectural strategies to scenarios and determine their expected quality attribute response levels.** For each architectural strategy under consideration, determine the expected quality attribute response levels that will result for each scenario.
- 6. Determine the utility of the expected quality attribute response levels by interpolation.** Using the elicited utility values (that form a utility curve), determine the utility of the expected quality attribute response level for the architectural strategy. Do this for each relevant quality attribute enumerated in step 3. For example, if we are considering a new architectural strategy that would result in a response time of 0.7 seconds, we would assign this a utility proportionately between 50 (which it exceeds) and 80 (which it doesn't exceed).

The formula for interpolation between two data points (x_a, y_a) and (x_b, y_b) is given by:

$$y = y_a + (y_b - y_a) \frac{(x - x_a)}{(x_b - x_a)}$$

For us, the x values are the quality attribute response levels and the y values are the utility values. So, employing this formula, the utility value of a 0.7-second response time is 74.

7. *Calculate the total benefit obtained from an architectural strategy.* Subtract the utility value of the “current” level from the expected level and normalize it using the votes elicited in step 3. Sum the benefit due to a particular architectural strategy across all scenarios and across all relevant quality attributes.
 8. *Choose architectural strategies based on VFC subject to cost and schedule constraints.* Determine the cost and schedule implications of each architectural strategy. Calculate the VFC value for each as a ratio of benefit to cost. Rank-order the architectural strategies according to the VFC value and choose the top ones until the budget or schedule is exhausted.
 9. *Confirm results with intuition.* For the chosen architectural strategies, consider whether these seem to align with the organization’s business goals. If not, consider issues that may have been overlooked while doing this analysis. If there are significant issues, perform another iteration of these steps.
-

Computing Benefit for Architectural Variation Points

This chapter is about calculating the cost and benefit of competing architectural options. While there are plenty of metrics and methods to measure cost, usually as some function of complexity, benefit is more slippery. CBAM uses an assemblage of stakeholders to work out jointly what *utility* each architectural option will bring with it. At the end of the day, CBAM’s measures of utility are subjective, intuitive, and imprecise. That’s all right; stakeholders seldom are able to express benefit any better than that, and so CBAM takes what stakeholders know and formulates it into a justified choice. That is, CBAM elicits inputs that are imprecise, because nothing better is available, and does the best that can be done with them.

As a counterpoint to CBAM, there is one area of architecture in which the architectural options are of a specific variety: product-line architectures. In [Chapter 25](#), you’ll be introduced to software architectures that serve for an entire family of systems. The architect introduces variation points into the architecture, which are places where it can be quickly tailored in preplanned ways so that it can serve each of a variety of different but related products. In the product-line context, the major architectural option is whether to build a particular variation point in the architecture. Doing so isn’t free; otherwise, every product-line architecture would have an infinitude of variation points. So the question becomes: When will adding a variation point pay off?

CBAM would work for this case as well; you could ask the product line’s stakeholders what the utility of a new variation point would be, vis-à-vis other options such as including a different variation point instead, or none at all. But in this case, there is a quantitative formula to measure the benefit. John McGregor of Clemson University has long been interested in the economics of software product lines, and along with others (including me) invented SIMPLE, a cost-modeling language for software product lines. SIMPLE is great at estimating the cost of product-line options, but not so great at estimating its benefits. Recently, McGregor took a big step toward remedying that.

Here is his formula for modeling the marginal value of building an additional variation point to the architecture:

$$v_i(t, T) = \max(0, -E \left[\sum_{\tau=t}^T c_i(\tau) e^{-r(\tau-t)} \right] + P_{i,T} E \left[\sum_k \max(0, \sum_{\tau=T}^{T^*} X_{i,k}(\tau) e^{-r(\tau-t)}) \right])$$

Got that? No? Right, me neither. But we can understand it if we build it up from pieces.

The equation says (as all value equations do) that value is benefit minus cost, and so those are the two terms.

The first term measures the expected cost of building variation point i over a time period from now until time T (some far-off horizon of interest such as fielding your fifth system or getting your next round of venture capital funding). The function $c_i(t)$ measures the cost of building variation point i at time t ; it's evaluated using conventional cost-estimating techniques for software. The e factor is a standard economic term that accounts for the net present value of money; r is the assumed current interest rate. This is summed up over all time between now and T , and made negative to reflect that cost is negative value. So here's the first term decomposed:

SYMBOLS FOR TIME

τ = time variable

t = time now

T = target date

T^* = modeling limit ($t=$ forever)

i = index over variation points

r = assumed interest rate

$$E \left[\sum_{\tau=t}^T c_i(\tau) e^{-r(\tau-t)} \right]$$

The second term evaluates the benefit. The function $X_{i,k}(t)$ is the key; it measures the value of the variation point in the k th product of the product line. This is equal to the marginal value of the variation point to the k th product, minus the cost of using (exercising) the variation point in that k th product. That first part is the hardest part to come by, but your marketing department should be able to help by expressing the marginal value in terms of the additional products that the variation point would enable you to build and how much revenue each would bring in. (That's what marketing departments are paid to figure out.)

$$X_{i,k}(\tau) e^{-r(\tau-t)}$$

... adjusted by a factor to account for net present value of money

value of variation point i in product k at time $\tau = VMP_{i,k}(\tau) - MC_{i,k}(\tau)$

marginal value of the i^{th} variation point in the k^{th} product at time τ

marginal cost of tailoring variation point i for use in product k

To the function X we apply the same factor as before to account for the net present value of money, sum it up over all time periods between now and T , and make it nonnegative:

SYMBOLS FOR TIME

t = time variable

t = time now

T = target date

T^* = modeling limit ($t=\text{forever}$)

i = index over variation points

r = assumed interest rate

k = index over products

... adjusted by a factor to account for net present value of money

Value cannot be negative

$$\max(0, \sum_{\tau=T}^{T^*} X_{i,k}(\tau) e^{-r(\tau-t)})$$

summed over all time

value of variation point i in product k at time $\tau = VMP_{i,k}(\tau) - MC_{i,k}(\tau)$

marginal value of the i^{th} variation point in the k^{th} product at time T

marginal cost of tailoring variation point i for use in product k

Then we sum that up over all products k in the product line and multiply it by the probability that the variation point will in fact be ready by the time it's needed:

$$P_{i,T} E \left[\sum_k \max \left(0, \sum_{\tau=T}^{T^*} X_{i,k}(\tau) e^{-r(\tau-T)} \right) \right]$$

probability that variation point i will be ready for use by time T

value of variation point i in product k over all time . . .
... and over all products

Add the two terms together and there you have it. It's easy to put this in a spreadsheet that calculates the result given the relevant inputs, and it provides a quantitative measurement to help guide selection of architectural options—in this case, introduction of variation points to support a product line.

—PCC

23.4. Case Study: The NASA ECS Project

We will now apply the CBAM to a real-world system as an example of the method in action.

The Earth Observing System is a constellation of NASA satellites that gathers data for the U.S. Global Change Research Program and other scientific communities worldwide. The Earth Observing System Data Information System (EOSDIS) Core System (ECS) collects data from various satellite downlink stations for further processing. ECS's mission is to process the data into higher-form information and make it available to scientists in searchable form. The goal is to provide both a common way to store (and hence process) data and a public mechanism to introduce new data formats and processing algorithms, thus making the information widely available.

The ECS processes an input stream of hundreds of gigabytes of raw environment-related data per day. The computation of 250 standard "products" results in thousands of gigabytes of information that is archived at eight data centers in the United States. The system has important performance and availability requirements. The long-term nature of the project also makes modifiability important.

The ECS project manager had a limited annual budget to maintain and enhance his current system. From a prior analysis—in this case an ATAM exercise—a large set of desirable changes to the system was elicited from the system stakeholders, resulting in a large set of architectural strategies. The problem was to choose a (much) smaller subset for implementation, as only 10 to 20 percent of what was being proposed could actually be funded. The manager used the CBAM to make a rational decision based on the economic criterion of return on investment.

In the execution of the CBAM described next, we concentrated on analyzing the Data Access Working Group (DAWG) portion of the ECS.

Step 1: Collate Scenarios

A subset of the raw scenarios put forward by the DAWG team were as shown in [Table 23.1](#). Note that they are not yet well formed and that some of them do not have defined responses. These issues are resolved in step 2, when the number of scenarios is reduced.¹

¹ In the presentation of the DAWG case study, we only show the reduced set of scenarios.

Table 23.1. Collected Scenarios in Priority Order

Scenario	Scenario Description
1	Reduce data distribution failures that result in hung distribution requests requiring manual intervention.
2	Reduce data distribution failures that result in lost distribution requests.
3	Reduce the number of orders that fail on the order submission process.
4	Reduce order failures that result in hung orders that require manual intervention.
5	Reduce order failures that result in lost orders.
6	There is no good method of tracking ECSGuest failed/canceled orders without much manual intervention (e.g., spreadsheets).
7	Users need more information on why their orders for data failed.
8	Because of limitations, there is a need to artificially limit the size and number of orders.
9	Small orders result in too many notifications to users.
10	The system should process a 50-GB user request in one day, and a 1-TB user request in one week.

Step 2: Refine Scenarios

The scenarios were refined, paying particular attention to precisely specifying their stimulus-response measures. The worst-case, current-case, desired-case, and best-case response goals for each scenario were elicited and recorded, as shown in [Table 23.2](#).

Table 23.2. Response Goals for Refined Scenarios

Scenario	Response Goals			
	Worst	Current	Desired	Best
1	10% hung	5% hung	1% hung	0% hung
2	> 5% lost	< 1% lost	0% lost	0% lost
3	10% fail	5% fail	1% fail	0% fail
4	10% hung	5% hung	1% hung	0% hung
5	10% lost	< 1% lost	0% lost	0% lost
6	50% need help	25% need help	0% need help	0% need help
7	10% get information	50% get information	100% get information	100% get information
8	50% limited	30% limited	0% limited	0% limited
9	1/granule	1/granule	1/100 granules	1/1,000 granules
10	< 50% meet goal	60% meet goal	80% meet goal	> 90% meet goal

Step 3: Prioritize Scenarios

In voting on the refined representation of the scenarios, the close-knit team deviated slightly from the method. Rather than vote individually, they chose to discuss each scenario and arrived at a

determination of its weight via consensus. The votes allocated to the entire set of scenarios were constrained to 100, as shown in [Table 23.3](#). Although the stakeholders were not required to make the votes multiples of 5, they felt that this was a reasonable resolution and that more precision was neither needed nor justified.

Table 23.3. Refined Scenarios with Votes

Scenario	Votes	Response Goals			
		Worst	Current	Desired	Best
1	10	10% hung	5% hung	1% hung	0% hung
2	15	> 5% lost	< 1% lost	0% lost	0% lost
3	15	10% fail	5% fail	1% fail	0% fail
4	10	10% hung	5% hung	1% hung	0% hung
5	15	10% lost	< 1% lost	0% lost	0% lost
6	10	50% need help	25% need help	0% need help	0% need help
7	5	10% get information	50% get information	100% get information	100% get information
8	5	50% limited	30% limited	0% limited	0% limited
9	10	1/granule	1/granule	1/100 granules	1/1,000 granules
10	5	< 50% meet goal	60% meet goal	80% meet goal	> 90% meet goal

Step 4: Assign Utility

In this step the utility for each scenario was determined by the stakeholders, again by consensus. A utility score of 0 represented no utility; a score of 100 represented the most utility possible. The results of this process are given in [Table 23.4](#).

Table 23.4. Scenarios with Votes and Utility Scores

Scenario	Votes	Utility Scores			
		Worst	Current	Desired	Best
1	10	10	80	95	100
2	15	0	70	100	100
3	15	25	70	100	100
4	10	10	80	95	100
5	15	0	70	100	100
6	10	0	80	100	100
7	5	10	70	100	100
8	5	0	20	100	100
9	10	50	50	80	90
10	5	50	50	80	90

Step 5: Develop Architectural Strategies for Scenarios and Determine Their Expected Quality Attribute

Response Levels

Based on the requirements implied by the preceding scenarios, a set of 10 architectural strategies was developed by the ECS architects. Recall that an architectural strategy may affect more than one scenario. To account for these complex relationships, the expected quality attribute response level that each strategy is predicted to achieve had to be determined with respect to each relevant scenario. The set of architectural strategies, along with the determination of the scenarios they address, is shown in [Table 23.5](#). For each architectural strategy/scenario pair, the response levels expected to be achieved with respect to that scenario are shown (along with the current response, for comparison purposes).

Table 23.5. Architectural Strategies and Scenarios Addressed

Strategy Name	Description	Scenarios Affected	Current Response	Expected Response
1 Order persistence on submission	Store an order as soon as it arrives in the system.	3	5% fail	2% Fail
		5	<1% lost	0% lost
		6	25% need help	0% need help
2 Order chunking	Allow operators to partition large orders into multiple small orders.	8	30% limited	15% limited
3 Order bundling	Combine multiple small orders into one large order.	9	1 per granule	1 per 100
		10	60% meet goal	55% meet goal
4 Order segmentation	Allow an operator to skip items that cannot be retrieved due to data quality or availability issues.	4	5% hung	2% hung
5 Order reassignment	Allow an operator to reassign the media type for items in an order.	1	5% hung	2% hung
6 Order retry	Allow an operator to retry an order or items in an order that may have failed due to temporary system or data problems.	4	5% hung	3% hung
7 Forced order completion	Allow an operator to override an item's unavailability due to data quality constraints.	1	5% hung	3% hung
8 Failed order notification	Ensure that users are notified only when part of their order has truly failed and provide detailed status of each item; user notification occurs only if operator okays notification; the operator may edit notification.	6	25% need help	20% need help
		7	50% get information	90% get information
9 Granule-level order tracking	An operator and user can determine the status for each item in their order.	6	25% need help	10% need help
		7	50% get information	95% get information
10 Links to user information	An operator can quickly locate a user's contact information. Server will access SDSRV information to determine any data restrictions that might apply and will route orders/order segments to appropriate distribution capabilities, including DDIST, PDS, external subsetters and data processing tools, etc.	7	50% get information	60% get information

Step 6: Determine the Utility of the “Expected” Quality Attribute Response Levels by Interpolation

Once the expected response level of every architectural strategy has been characterized with respect to a set of scenarios, their utility can be calculated by consulting the utility scores for each scenario’s current and desired responses for all of the affected attributes. Using these scores, we may calculate, via interpolation, the utility of the expected quality attribute response levels for the architectural strategy/scenario pair applied to the DAWG of ECS.

Step 7: Calculate the Total Benefit Obtained from an Architectural Strategy

Based on the information collected, as represented in [Table 23.6](#), the total benefit of each architectural strategy can now be calculated, following the equation from [Section 23.2](#), repeated here:

$$B_i = S_j (b_{i,j} \times W_j)$$

Table 23.6. Architectural Strategies and Their Expected Utility

Strategy	Name	Scenarios Affected	Current Utility	Expected Utility
1	Order persistence on submission	3	70	90
		5	70	100
		6	80	100
2	Order chunking	8	20	60
3	Order bundling	9	50	80
		10	70	65
4	Order segmentation	4	80	90
5	Order reassignment	1	80	92
6	Order retry	4	80	85
7	Forced order completion	1	80	87
8	Failed order notification	6	80	85
		7	70	90
9	Granule-level order tracking	6	80	90
		7	70	95
10	Links to user information	7	70	75

This equation calculates total benefit as the sum of the benefit that accrues to each scenario, normalized by the scenario's relative weight. Using this formula, the total benefit scores for each architectural strategy are now calculated, and the results are given in [Table 23.7](#).

Table 23.7. Total Benefit of Architectural Strategies

Strategy	Scenario Affected	Scenario Weight	Raw Architectural Strategy Benefit	Normalized Architectural Strategy Benefit	Total Architectural Strategy Benefit
1	3	15	20	300	
1	5	15	30	450	
1	6	10	20	200	950
2	8	5	40	200	200
3	9	10	30	300	
3	10	5	-5	-25	275
4	4	10	10	100	100
5	1	10	12	120	120
6	4	10	5	50	50
7	1	10	7	70	70
8	6	10	5	50	
8	7	5	20	100	150
9	6	10	10	100	
9	7	5	25	125	225
10	7	5	5	25	25

Step 8: Choose Architectural Strategies Based on VFC Subject to Cost Constraints

To complete the analysis, the team estimated cost for each architectural strategy. The estimates were based on experience with the system, and a return on investment for each architectural strategy was calculated. Using the VFC, we were able to rank each strategy. This is shown in [Table 23.8](#). Not surprisingly, the ranks roughly follow the ordering in which the strategies were proposed: strategy 1 has the highest rank; strategy 3 the second highest. Strategy 9 has the lowest rank; strategy 8, the second lowest. This simply validates stakeholders' intuition about which architectural strategies were going to be of the greatest benefit. For the ECS these were the ones proposed first.

Table 23.8. VFC of Architectural Strategies

Strategy	Cost	Total Strategy Benefit	Strategy VFC	Strategy Rank
1	1200	950	0.79	1
2	400	200	0.5	3
3	400	275	0.69	2
4	200	100	0.5	3
5	400	120	0.3	7
6	200	50	0.25	8
7	200	70	0.35	6
8	300	150	0.5	3
9	1000	225	0.22	10
10	100	25	0.25	8

Results of the CBAM Exercise

The most obvious results of the CBAM are shown in [Table 23.8](#): an ordering of architectural strategies based on their predicted VFC. However, just as for the ATAM method, the benefits of the CBAM extend beyond the qualitative outcomes. There are social and cultural benefits as well.

Just as important as the ranking of architectural strategies in CBAM is the discussion that accompanies the information-collecting and decision-making processes. The CBAM process provides a great deal of structure to what is always largely unstructured discussions, where requirements and architectural strategies are freely mixed and where stimuli and response goals are not clearly articulated. The CBAM process forces the stakeholders to make their scenarios clear in advance, to assign utility levels of specific response goals, and to prioritize these scenarios based on the resulting determination of utility. Finally, this process results in clarification of both scenarios and requirements, which by itself is a significant benefit.

23.5. Summary

Architecture-based economic analysis is grounded on understanding the utility-response curve of various scenarios and casting them into a form that makes them comparable. Once they are in this common form—based on the common coin of utility—the VFC for each architecture improvement, with respect to each relevant scenario, can be calculated and compared.

Applying the theory in practice has a number of practical difficulties, but in spite of those difficulties, we believe that the application of economic techniques is inherently better than the ad hoc decision-making approaches that projects (even quite sophisticated ones) employ today. Our experience with the CBAM tells us that giving people the appropriate tools to frame and structure their discussions and decision making is an enormous benefit to the disciplined development of a complex software system.

23.6. For Further Reading

The origins of the CBAM can be found in two papers: [\[Moore 03\]](#) and [\[Kazman 01\]](#).

A more general background in economic approaches to software engineering may be found in the now-classic book by Barry Boehm [\[Boehm 81\]](#).

And a more recent, and somewhat broader, perspective on the field can be found in [\[Biffi 10\]](#).

The product-line analysis we used in the sidebar on the value of variation points came from a paper in the 2011 International Software Product Line Conference by John McGregor and his colleagues [\[McGregor 11\]](#).

23.7. Discussion Questions

1. This chapter is about choosing an architectural strategy using rational, economic criteria. See how many other ways you can think of to make a choice like this. *Hint:* Your candidates need not be "rational."
2. Have two or more different people generate the utility curve for a quality attribute scenario for an ATM. What are the difficulties in generating the curve? What are the differences between the two curves? How would you reconcile the differences?
3. Discuss the advantages and disadvantages of the method for generating scenario priorities used in the CBAM. Can you think of a different way to prioritize the scenarios? What are the pluses and minuses of your method?
4. Using the results of your design exercise for the ATM from [Chapter 17](#) as a starting point, develop an architectural strategy for achieving a quality attribute scenario that your design does not cover.
5. Generate the utility curves for two different systems in the same domain. What are the differences? Do you believe that there are standard curves depending on the domain? Defend your answer.

24. Architecture Competence

The ideal architect should be a man of letters, a skillful draftsman, a mathematician, familiar with historical studies, a diligent student of philosophy, acquainted with music, not ignorant of medicine, learned in the responses of jurisconsults, familiar with astronomy and astronomical calculations.

—Vitruvius, *De Architectura* (25 B.C.)

The lyf so short, the craft so long to lerne.

—Geoffrey Chaucer

If software architecture is worth “doing,” surely it’s worth doing well. Most of the literature about architecture concentrates on the technical aspects. This is not surprising; it is a deeply technical discipline. There is little information that speaks to the fact that architectures are created by architects working in organizations, full of actual human beings. Dealing with these humans is decidedly nontechnical. What can be done to help architects, especially architects-in-training, be better at this important dimension of their job? And what can be done to help architecture organizations do a better job in fostering their architects to produce their best work?

An organization’s ability to routinely produce high-quality architectures that are aligned with its business goals well cannot be understood simply through examination of past architectures and measurement of their deficiencies. The organizational and human causes of those deficiencies also need to be understood.

This chapter is about the competence of individual architects and the organizations that wish to produce high-quality architects. We define the architecture competence of an organization as follows:

The architecture competence of an organization is the ability of that organization to grow, use, and sustain the skills and knowledge necessary to effectively carry out architecture-centric practices at the individual, team, and organizational levels to produce architectures with acceptable cost that lead to systems aligned with the organization’s business goals.

Because the architecture competence of an organization depends, in part, on the competence of architects, we begin by asking what it is that architects are expected to do, know, and be skilled at. Then we’ll look at what organizations can and should do to help their architects produce better architectures. Individual and organizational competencies are intertwined. Studying only one or the other won’t do.

24.1. Competence of Individuals: Duties, Skills, and Knowledge of Architects

Architects perform many activities beyond directly producing an architecture. These activities, which we call *duties*, form the backbone of an individual's architecture competence. We surveyed a broad body of information aimed at architects (such as websites, courses, books, and position descriptions for architects). We also surveyed practicing architects. These surveys tell us that duties are but one aspect of individual competence. Writers about architects also speak of *skills* and *knowledge*. For example, the ability to communicate ideas clearly and to negotiate effectively are skills often ascribed to competent architects. In addition, architects need to have up-to-date knowledge about patterns, database platforms, web services standards, quality attributes, and a host of other topics.

Duties, skills, and knowledge¹ form a triad upon which architecture competence for individuals rests. The relationship among these three is shown in [Figure 24.1](#)—namely, skills and knowledge support the ability to perform the required duties. Infinitely talented architects are of no use if they cannot (for whatever reason) perform the duties required of the position; we would not say they were competent.

1. Some writers speak of the importance of experience. We count experience as a form of knowledge.

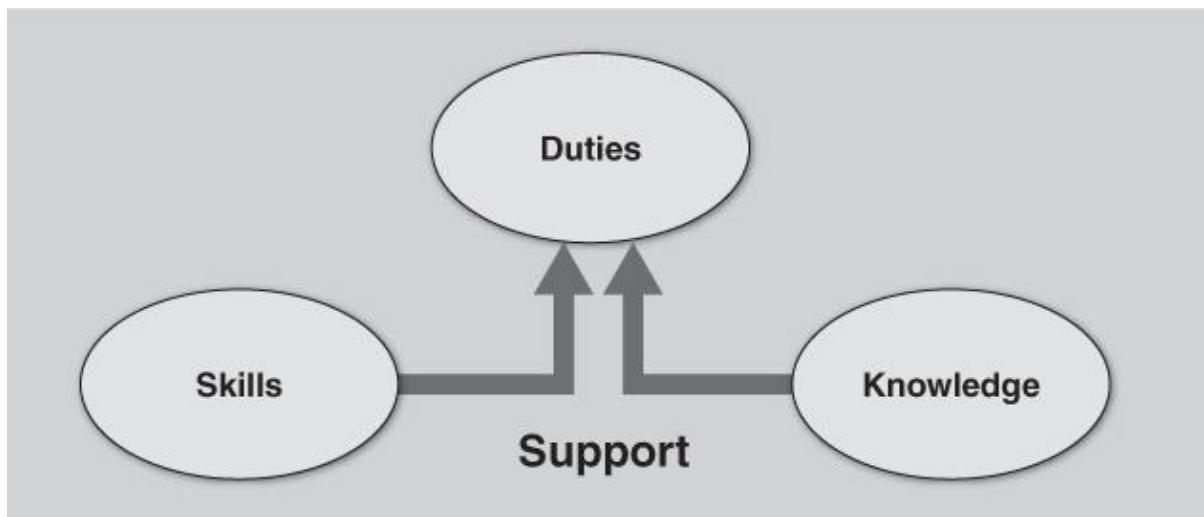


Figure 24.1. Skills and knowledge support the execution of duties.

To give examples of these concepts:

- “Design the architecture” is a duty.
- “Ability to think abstractly” is a skill.
- “Patterns and tactics” is a part of the body of knowledge.

This example purposely illustrates that skills and knowledge are important (only) for supporting the ability to carry out duties effectively. As another example, “documenting the architecture” is a duty, “ability to write clearly” is a skill, and “ISO Standard 42010” is part of the related body of knowledge. Of course, a skill or knowledge area can support more than one duty.

Knowing the duties, skills, and knowledge of architects (or, more precisely, the duties, skills, and knowledge that are needed of architects in a particular organizational setting) can help establish measurement and improvement strategies for individual architects. If you want to improve your individual architectural competence, you should do the following:

1. *Gain experience carrying out the duties.* Apprenticeship is a productive path to achieving experience. Education alone is not enough, because education without on-the-job application merely enhances knowledge.
2. *Improve your nontechnical skills.* This dimension of improvement involves taking professional development courses, for example, in leadership or time management.

Some people will never become truly great leaders or communicators, but we can all improve on these skills.

3. *Master the body of knowledge.* One of the most important things a competent architect must do is master the body of knowledge and remain up to date on it. To emphasize the importance of remaining up to date, consider the advances in knowledge required for architects that have emerged in just the last few years. For example, the cloud and edge computing that we discuss in [Chapters 26](#) and [27](#) were not important topics several years ago. Taking courses, becoming certified, reading books and journals, visiting websites and portals, reading blogs, attending architecture-oriented conferences, joining professional societies, and meeting with other architects are all useful ways to improve knowledge.

Duties

This section summarizes a wide variety of an architect's duties. Not every architect in every organization will perform every one of these duties on every project. But a competent architect should not be surprised to find himself or herself engaged in any of the activities listed here. We divide the duties into technical duties ([Table 24.1](#)) and nontechnical duties ([Table 24.2](#)). One immediate observation you should make is how many nontechnical duties there are. An obvious implication, for those of you who wish to be architects, is that you must pay adequate attention to the nontechnical aspects of your education and your professional activities.

Table 24.1. The Technical Duties of a Software Architect

General Duty Area	Specific Duty Area	Example Duties
Architecting	Creating an architecture	Design or select an architecture. Create software architecture design plan. Build product line or product architecture. Make design decisions. Expand detail and refine design to converge on final design. Identify the patterns and tactics and articulate the principles and key mechanisms of the architecture. Partition the system. Define how the components fit together and interact.
	Evaluating and analyzing an architecture	Evaluate an architecture (for your current system or for other systems) to determine satisfaction of use cases and quality attribute scenarios. Create prototypes. Participate in design reviews. Review construction-level designs. Review the designs of the components designed by junior engineers. Review designs for compliance with the architecture. Compare software architecture evaluation techniques. Apply value-based architecting techniques to evaluate architectural decisions. Model alternatives. Perform tradeoff analysis.
	Documenting an architecture	Prepare architectural documents and presentations useful to stakeholders. Document software interfaces. Produce documentation standards. Document variability and dynamic behavior.
	Working with and transforming existing system(s)	Maintain and evolve existing system and its architecture. Redesign existing architecture(s) for migration to new technology and platforms.
	Performing other architecting duties	Sell the vision, keep the vision alive. Participate in product design meetings. Give technical advice on architecture, design, and development. Provide architectural guidelines for software design activities. Lead architecture improvement activities. Participate in software process definition and improvement. Define philosophy and principles for global architecture. Oversee or manage the architecture definition process. Provide architecture oversight of software development projects.

Duties concerned with life-cycle activities other than architecting	Managing the requirements	Analyze functional and quality attribute software requirements. Understand business and customer needs and ensure that the requirements meet these needs. Capture customer, organizational, and business requirements on the architecture. Create software specifications from business requirements. Articulate and refine architectural requirements. Listen to and understand the scope of the project. Understand the client's key design needs and expectations.
	Implementing the product	Produce code. Conduct code reviews. Develop reusable software components. Analyze, select, and integrate software components. Set and ensure adherence to coding guidelines. Recommend development methodologies and coding standards. Monitor, mentor, and review the work of outside consultants and vendors.
	Testing the product	Establish architecture-based testing procedures. Support system testers. Support field testing. Support bug fixing and maintenance.
	Evaluating future technologies	Evaluate and recommend enterprise's software solutions. Manage the introduction of new software solutions. Analyze current IT environment and recommend solutions for deficiencies. Work with vendors to represent organization's requirements and influence future products. Develop and present technical white papers.
	Selecting tools and technology	Perform technical feasibility studies of new technologies and architectures. Evaluate commercial tools and software components from an architectural perspective. Develop internal technical standards and contribute to the development of external technical standards.

Table 24.2. The Nontechnical Duties of a Software Architect

General Duty Area	Specific Duty Area	Example Duties
Management	Managing the project	Help with budgeting and planning. Follow budgetary constraints. Manage resources. Perform sizing and estimation. Perform migration planning and risk assessment. Take care of or oversee configuration control. Create development schedules. Measure results using quantitative metrics and improve both personal results and teams' productivity. Identify and schedule architectural releases.
	Managing the people	Build "trusted advisor" relationships. Coordinate. Motivate. Advocate. Delegate. Act as a supervisor.
	Supporting the management	Provide feedback on appropriateness and difficulty of project. Advise the project manager on the tradeoffs between software design choices and requirements choices. Provide input to software project manager in the software project planning and estimation process. Serve as a "bridge" between the technical team and the project manager.
Organization and business-related duties	Supporting the organization	Grow an architecture evaluation capability in the organization. Review and contribute to research and development efforts. Participate in the hiring process for the team. Help with product marketing. Institute and oversee cost-effective software architecture design reviews. Help develop intellectual property.

	Supporting the business	Translate business strategy into technical vision and strategy. Influence the business strategy. Understand and evaluate business processes. Understand and communicate the business value of software architecture. Help the organization meet its business goals. Understand customer and market trends. Identify, understand, and resolve business issues. Align architecture with the business goals and objectives.
Leadership and team building	Providing technical leadership	Mentor other architects. Produce technology trend analysis or roadmaps.
	Building a team	Set team context (vision). Build the architecture team and align them with the vision. Mentor junior architects. Educate the team on the use of the architecture. Maintain morale, both within and outside the architecture group. Foster the professional development of team members. Coach teams of software design engineers for planning, tracking, and completion of work within the agreed plan. Mentor and coach staff in the use of software technologies. Work both as a leader and as an individual contributor.

Skills

Given the wide range of duties enumerated in the previous section, what skills (beyond mastery of the technical body of knowledge) does an architect need to possess? Much has been written about the architect's special role of leadership in a project; the ideal architect is an effective communicator, manager, team builder, visionary, and mentor. Some certificate or certification programs emphasize nontechnical skills. Common to these certification programs are nontechnical assessment areas of leadership, organization dynamics, and communication.

[Table 24.3](#) enumerates the set of nontechnical skills most useful to an architect.

Table 24.3. The Nontechnical Skills of a Software Architect

General Skill Area	Specific Skill Area	Example Skills
Communication skills	Outward	Ability to make oral and written communications and presentations. Ability to present and explain technical information to diverse audiences. Ability to transfer knowledge. Ability to persuade. Ability to see from and sell to multiple viewpoints.
	Inward	Ability to listen, interview, consult, and negotiate. Ability to understand and express complex topics.
Interpersonal skills	Within team	Ability to be a team player. Ability to work effectively with superiors, colleagues, and customers. Ability to maintain constructive working relationships. Ability to work in a diverse team environment. Ability to inspire creative collaboration. Ability to build consensus.
	With other people	Ability to demonstrate interpersonal skills. Ability to be diplomatic and respect others. Ability to mentor others. Ability to handle and resolve conflict.
Work skills	Leadership	Ability to make decisions. Ability to take initiative and be innovative. Ability to demonstrate independent judgment, be influential, and command respect.
	Workload management	Ability to work well under pressure, plan, manage time, estimate. Ability to support a wide range of issues and work on multiple complex projects concurrently. Ability to effectively prioritize and execute tasks in a high-pressure environment.
	Skills to excel in corporate environment	Ability to think strategically. Ability to work under general supervision and under given constraints. Ability to organize workflow. Ability to sense where the power is and how it flows in an organization. Ability to do what it takes to get the job done. Ability to be entrepreneurial, be assertive without being aggressive, and receive constructive criticism.
	Skills for handling information	Ability to be detail oriented while maintaining overall vision and focus. Ability to see the big picture. Ability to deal with abstraction.
Skills for handling the unexpected		Ability to tolerate ambiguity. Ability to take and manage risks. Ability to solve problems. Ability to be adaptable, flexible, open minded, and resilient. Ability to do the juggling necessary to deploy successful software projects.

Knowledge

A competent architect has an intimate familiarity with the architectural body of knowledge. The software architect should

- Be comfortable with all branches of software engineering from requirements definition to implementation, development, verification and validation, and deployment
- Be familiar with supporting disciplines such as configuration management and project management
- Understand current design and implementation tools and technologies

Knowledge and experience in one or more application domains is also necessary.

[Table 24.4](#) is the set of knowledge areas for an architect.

Table 24.4. The Knowledge Areas of a Software Architect

General Knowledge Area	Specific Knowledge Area	Specific Knowledge Examples
Computer science knowledge	Knowledge of architecture concepts	Knowledge of architecture frameworks, architectural patterns, tactics, viewpoints, standard architectures, relationship to system and enterprise architecture, architecture description languages, emerging technologies, architecture evaluation models and methods, and quality attributes.
	Knowledge of software engineering	Knowledge of systems engineering. Knowledge of software development life cycle, software process management, and improvement techniques. Knowledge of requirements analysis, mathematics, development methods and modeling techniques, elicitation techniques. Knowledge of component-based software development, reuse methods and techniques, software product-line techniques, documentation, testing and debugging tools.
Design knowledge		Knowledge of different tools and design techniques. Knowledge of how to design complex multi-product systems. Knowledge of object-oriented analysis and design, UML diagrams, and UML analysis modeling.
	Programming knowledge	Knowledge of programming languages and programming language models. Knowledge of specialized programming techniques for security, real time, etc.

Knowledge of technologies and platforms	Knowledge of specific technologies and platforms	Knowledge of hardware/software interfaces, web-based applications, and Internet technologies. Knowledge of specific software/operating systems, such as RDBMS concepts, cloud platforms, and SOA implementations.
	General knowledge of technologies and platforms	Knowledge of IT industry future directions and the ways in which infrastructure impacts an application.
Knowledge about the organization's context and management	Domain knowledge	Knowledge of the most relevant domain(s) and domain-specific technologies.
	Industry knowledge	Knowledge of the industry's best practices and industry standards. Knowledge of how to work in onshore/offshore team environment.
	Enterprise knowledge	Knowledge of the company's business practices, and your competition's products, strategies, and processes. Knowledge of business and technical strategy, and business reengineering principles and processes. Knowledge of strategic planning, financial models, and budgeting.
Leadership and management techniques	Leadership and management techniques	Knowledge of coaching, mentoring, and training software developers. Knowledge of project management. Knowledge of project engineering.

24.2. Competence of a Software Architecture Organization

Organizations by their practices and structure can help or hinder architects in performing their duties. For example, if an organization has a career path for architects, that will motivate employees to become architects. If an organization has a standing architecture review board, then the project architect will know how and with whom to schedule a review. Lack of these items will mean that an architect has to fight battles with the organization or determine how to carry out a review without internal guidance. It makes sense, therefore, to ask whether a particular organization is architecturally competent and to develop instruments whose goal is measuring the architectural competence of an organization. The architectural competence of organizations is the topic of this section.

Activities Carried Out by a Competent Organization

Organizations have duties, skills, and knowledge for architecture as well. For example, adequately funding the architecture effort is an organizational duty, as is effectively using the available architecture workforce (by appropriate teaming, and so on). These are organizational duties because they are outside the control of individual architects. An organization-level skill might be effective knowledge management or human resource management as applied to architects. An example of organizational knowledge is the composition of an architecture-based life-cycle model that software projects may employ.

Here are some things—duties—that an organization can perform to help improve the success of its architecture efforts:

- Hire talented architects.
- Establish a career track for architects.
- Make the position of architect highly regarded through visibility, reward, and prestige.
- Establish a clear statement of responsibilities and authority for architects.
- Establish a mentoring program for architects.
- Establish an architecture training and education program.
- Establish an architect certification program.
- Have architects receive external architect certifications.
- Measure architects' performance.
- Establish a forum for architects to communicate and share information and experience.
- Establish a repository of reusable architectures and architecture-based artifacts.
- Develop reusable reference architectures.
- Establish organization-wide architecture practices.
- Establish an architecture review board.
- Measure the quality of architectures produced.
- Provide a centralized resource to analyze and help with architecture tools.
- Hold an organization-wide architecture conference.
- Have architects join professional organizations.
- Bring in outside expert consultants on architecture.
- Include architecture milestones in project plans.
- Have architects provide input into product definition.
- Have architects advise on the development team structure.
- Give architects influence throughout the entire project life cycle.
- Reward or penalize architects based on project success or failure.

Assessment Goals

The activities enumerated above can be assessed. What are the potential goals from an assessment of an organization's architecture competence? There are at least four sets of reasons for assessing organizational architectural competence:

- 1.** There are goals relevant to any business that wishes to improve their architectural competence. Businesses regularly assess their own performance in a variety of means—technical, fiscal, operational (for example, consider the widespread use of multi-criteria techniques such as the Balanced Scorecard or Business/IT Alignment in industry)—for a variety of reasons. These include determining whether they are meeting industry norms and gauging their progress over time in meeting organizational goals.
- 2.** There are goals relevant to an acquisition organization. For example, an organization can use an assessment of architecture competence to assess a contractor in much the same way that contractors are scrutinized with respect to their CMMI level. Or an organization might use an assessment of architecture competence to aid in deciding among competing bids from contractors. All other things being equal, an acquiring organization would prefer a contractor with a higher level of architectural competence because this typically means fewer downstream problems and rework. An acquisition organization might assess the contractors directly, or hire a third party to do the assessment.
- 3.** There are goals relevant to service organizations: such organizations might be motivated to maintain, measure, and advertise their architectural competence as a means of attracting and retaining customers. In such a case they would typically rely on outside organizations to assess their level of competence in an objective fashion.

4. Finally, there are goals that are relevant to product builders: these organizations would be motivated to assess, monitor (and, over time, increase) their level of architectural competence as it would (1) aid in advertising the quality of their products and (2) aid in their internal productivity and predictability. In fact, in these ways their motivations are aligned with those of service organizations.

Assessing an Organization's Competence

In addition to duties, skills, and knowledge, there are other models of individual and organizational competence that are helpful in building an instrument for assessing an organization's competence. They are the following:

- The *Human Performance Technology model*, which measures the value of an individual or department's output and the cost of producing that output. It holds that competent people produce the most value for every organizational dollar spent.
- The *Organizational Coordination model*, which measures how teams in multiple sites developing a single product or related set of products cooperate to produce a functioning product. An organization that is architecturally competent will have more effective and efficient coordination mechanisms than an organization that is not architecturally competent.
- The *Organizational Learning model*, which measures how well an organization's learning processes transform experience into knowledge, moderated by context.

We have created a framework for organizational architecture competence that forms the foundation for a competence assessment procedure. A small team of trained assessors can use the framework to conduct interviews (with architects, developers, and technical and organizational managers), examine current practices, read documents and evidentiary artifacts (such as organizational standards), and investigate architecture-based successes and failures in the recent past. They can use their findings to identify systemic trouble spots and recommend improvement strategies.

[Table 24.5](#) shows the framework. For convenience, it is divided into practice areas that relate to *software and system engineering*, *technical management* (which is by and large the management of single projects or small numbers of related projects), and *organizational management* (which is management at a scope more broad than that of projects).

Table 24.5. Framework for Organizational Architecture Competence

Software Engineering Practice Areas	Quality Attribute Elicitation Practices
	Tools and Technology Selection
	Modeling and Prototyping Practices
	Architecture Design Practices
	Architecture Description Practices
	Architecture Evaluation Practices
	System Implementation Practices
	<ul style="list-style-type: none">▪ Software design practices (design conforms to architecture)▪ Software coding practices (code conforms to design and architecture)
	Software Verification Practices
	<ul style="list-style-type: none">▪ Proving properties of the software▪ Software testing
	Architecture Reconstruction Practices
	Business or Mission Goals Practices
	<ul style="list-style-type: none">▪ Setting goals▪ Measuring achievement of organization's goals▪ Performance-based compensation
	Product or System Definition Practices
	<ul style="list-style-type: none">▪ Setting functional requirements

Technical Management Practice Areas

	<p>Allocating Resources</p> <ul style="list-style-type: none">▪ Setting architect's workload and schedule▪ Funding stakeholder involvement
	<p>Project Management Practices</p> <ul style="list-style-type: none">▪ Project plan structure aligned with architecture structure▪ Adequate time planned for architecture evaluation
	<p>Process Discipline Practices</p> <ul style="list-style-type: none">▪ Establish organization-wide architecture practices▪ Process monitoring and improvement practices▪ Reuse practices
	<p>Collaboration with Manager Practices</p> <ul style="list-style-type: none">▪ Architects advise managers▪ Architects support managers
	<p>Hire Talented Architects</p>
	<p>Establish a Career Track for Architects</p> <ul style="list-style-type: none">▪ Leadership roles for architects▪ Succession planning
Organizational Management Practice Areas	<p>Professional Development Practices</p> <ul style="list-style-type: none">▪ Ongoing training▪ Creating and sustaining an internal community of architects▪ Supporting participation in external communities
	<p>Organizational Planning Practices</p>
	<p>Technology Planning and Forecasting Practices</p>

The framework is populated by questions inspired by the four models of competence that we previously described. Each question has a set of answers that we might expect to see in a competent organization. For example, a question associated with the practice area "Hire talented architects" deals with how a candidate architect's experience and capabilities are assessed. Expected answers might include having the architect take a test, requiring that candidates possess an architecture certification, or examining previous architectures designed by the candidate. (Our expected answers grow as we visit more and more organizations. It's a pleasant surprise when we find an organization carrying out a practice area in a clever way that we hadn't thought of.)

Questions Based on the Duties, Skills, and Knowledge Model

Our assessment framework contains dozens of questions related to duties, skills, and knowledge. The questions are posed in terms of the organization: The questions ask how the organization ensures that the architectural duties are carried out in a competent manner, and how the organization measures and nurtures its architects' skills and knowledge. Here is a small set of example questions based on the Duties, Skills, and Knowledge model (chosen from among the dozens that populate our assessment framework) that we use in an architecture competence assessment exercise with an organization.

Duty: Creating an Architecture

Question: How do you create an architecture?

- How do you ensure that the architecture is aligned with the business goals?
- What is the input into the architecture creation process? What inputs are provided to the architect?
- How does the architect validate the information provided? What does the architect do in case the input is insufficient or inadequate?

Duty: Architecture Evaluation and Analysis

Question: How do you evaluate and analyze an architecture?

- Are evaluations part of the normal software development life cycle or are they done when problems are encountered?
- Is the evaluation incremental or “big bang”? How is the timing determined?
- Does the evaluation include an explicit activity relating architecture to business goals?
- What are the inputs to the evaluation? How are they validated?
- What are the outputs from an evaluation? How are the outputs of the evaluation utilized? Are the outputs differentiated according to impact or importance? How are the outputs validated? Who is communicated what outputs?

Knowledge: Architecture Concepts

Question: How does your organization ensure that its architects have adequate architectural knowledge?

- How are architects trained in general knowledge of architecture?
- How do architects learn about architectural frameworks, patterns, tactics, standards, documentation notations, and architecture description languages?
- How do architects learn about new or emerging architectural technologies (e.g., multi-core processors)?
- How do architects learn about analysis and evaluation techniques and methods?
- How do architects learn quality attribute-specific knowledge, such as techniques for analyzing and managing availability, performance, modifiability, and security?
- How are architects tested to ensure that their level of knowledge is adequate, and remains adequate, for the tasks that they face?

Questions Based on the Organizational Coordination Model

Questions based on the Organizational Coordination model focus on how the organization establishes its teams and what support it provides for those teams to coordinate effectively. Here are a couple of example questions:

Question: How is the architecture designed with distribution of work to teams in mind?

- How available or broadly shared is the architecture to various teams?
- How do you manage the evolution of architecture during development?
- Is the work assigned to the teams before or after the architecture is defined, and with due consideration of the architectural structure?

Question: Are the aspects of the architecture that will require a lot of interteam coordination supported by the organization’s coordination/communication infrastructure?

- Do you co-locate teams with high coordination? Or at least put them in the same time zone?
- Must all coordination among teams go through the architecture team?

Questions Based on the Human Performance Technology Model

The Human Performance Technology questions deal with the value and cost of the organization's architectural activities. Here are examples of questions based on the Human Performance Technology model:

Question: Do you track how much the architecture effort costs, and how it impacts overall project cost and schedule?

- How do you track the end of architecture activities?
- How do you track the impact of architecture activities?

Question: Do you track the value or benefits of the architecture?

- How do you measure stakeholder satisfaction?
- How do you measure quality?

Questions Based on the Organizational Learning Model

Finally, a set of example questions, based on the Organizational Learning model, which deal with how the organization systematically internalizes knowledge to its advantage:

Question: How do you capture and share experiences, lessons learned, technological decisions, techniques and methods, and knowledge about available tooling?

- Do you use any knowledge management tools?
- Is capture and use of architectural knowledge embedded in your processes?
- Where is the information about "who knows what" captured and how is this information maintained?
- How complete and up to date is your architecture documentation? How widely disseminated is it?

Performing an Assessment

Our organizational competence assessment is carried out using a team of three to four assessors. The exercise is set up by establishing the scope of the review: Are we assessing the entire company? One of its divisions? Or perhaps a single important project?

After we establish the scope, we identify the groups we wish to interview. Of course, we'll want to interview the architecture team(s) within the scope. From there we identify groups both upstream and downstream of the architects. Upstream are groups that manage the architects or provide organization-wide architecture training. Downstream, we interview the "consumers" of the architectures, such as developers, integrators, testers, and maintainers. We interview small groups, making sure that no members of an interview group have reporting relationships with each other.

We try hard to establish an informal atmosphere in the group interviews, to avoid inhibiting the participants. The tone is conversational, not inquisitional. We begin each interview by reminding the participants of the purpose of the exercise, and to assure them that nothing they say will be quoted to anyone outside the group in any way that could identify them.

For each group, we have planned which parts of the framework we wish to discuss with that group. We won't ask testers, for example, questions intended for managers, and vice versa. We use the questions as a guide for the conversation, but not a rigid script. Whenever we pose a question in the assessment instrument, there are a number of meta-questions that automatically accompany it. For example:

- What evidence could you show us to support your answer? Supporting evidence might include a software development plan that lays out the role of architecture in a project, an organization's architecture-based training curriculum, or many other kinds of documentation.
- How sustainable is your answer over time, over different systems, and across different architects? For example, we might ask how an answer might change if a different architect came on board the project.

The outcome of an assessment is organized by the practice areas of the framework. For each practice, we assign one of three values that correspond to "you're doing this well," "you could be doing this better (and experiencing more benefit)," and "this is an area of high risk." Graphically,

we show this as green light, yellow light, red light. We have found that this simple metric provides organizations with enough granularity to turn their attention to problem areas, which is the whole point of the assessment. We do not give an overall rating. Thus, we closely mirror the “continuous representation” option of maturity models such as CMMI, in which the result is a vector rather than a scalar.

We present the findings in a written report and a slide presentation. In both, we describe and justify each finding, based on what we were told in the interviews and/or read in provided documents.

24.3. Summary

The vast majority of work on software and systems architecture (including our own) has focused on the technical aspects. But an architecture is much more than a technical “blueprint” for a system. This has led us to try to understand, in a more holistic way, what an architect and an architecture-centric organization must do to succeed. To this end, we have developed a framework that aids us in assessing an organization for competence.

We use the framework to ask questions about an organization’s practices. We can also ask about recent architecture successes and failures, and investigate the causes of each. The output of this exercise is a formal report that assesses competence at organization, team, and individual levels. Along with this report we make improvement recommendations based on assessment results; these, too, are tied to the underlying competence models.

You can do the same sort of evaluation on your own organization. The key to the process is in understanding the various models and in creating questions based on these models that aid you in assessing how well you are doing in those areas that you care about. Given this knowledge, you can create your own improvement plan, as an individual architect or for an entire organization.

24.4. For Further Reading

The four models that underlie the assessment framework presented here are described in more detail in the Technical Note “Models for Evaluating and Improving Architecture Competence” [\[Bass 08\]](#). These models are the following:

- *Duties, Skills, and Knowledge (DSK) model of competence.* This model is predicated on the belief that architects and architecture-producing organizations are useful sources for understanding the tasks necessary to the job of architecting. To assemble a comprehensive set of duties, skills, and knowledge for architects, we surveyed approximately 200 sources of information targeted to professional architects—books, websites, blogs, position descriptions, and more. The results of this survey can be found in [\[Clements 07\]](#).
- *Human Performance model of competence.* This model is based on the human performance engineering work of Thomas Gilbert [\[Gilbert 07\]](#). This model is predicated on the belief that competent individuals in any profession are the ones who produce the most valuable results at a reasonable cost. Using this model will involve figuring out how to measure the value and cost of the outputs of architecture efforts, finding areas where that ratio can be improved, and crafting improvement strategies based on environmental and behavioral factors.
- *Organizational Coordination model of competence.* The focus of this model is on creating an interteam coordination model for teams developing a single product or a closely related set of products. The architecture for the product induces a requirement for teams to coordinate during the realization or refinement of architectural decisions. The organizational structure, practices, and tool environment of the teams allow for particular types of coordination with a particular interteam communication bandwidth. The coordination model of competence compares the requirements for coordination that the architecture induces with the bandwidth for coordination supported by the organizational structure, practices, and tool environment [\[Cataldo 07\]](#).
- *Organizational Learning model of competence.* This model is based on the concept that organizations, and not just individuals, can learn. Organizational learning is a change in

the organization that occurs as a function of experience. This change can occur in the organization's cognitions or knowledge (e.g., as presented by Fiol and Lyles [[Fiol 85](#)]), its routines or practices (e.g., as demonstrated by Levitt and March [[Levitt 88](#)]), or its performance (e.g., as presented by Dutton and Thomas [[Dutton 84](#)]). Although individuals are the medium through which organizational learning generally occurs, learning by individuals within the organization does not necessarily imply that organizational learning has occurred. For learning to be organizational, it has to have a supra-individual component [[Levitt 88](#)]. There are three approaches to measure organizational learning: (1) measure knowledge directly through questionnaires, interviews, and verbal protocols; (2) treat changes in routines and practices as indicators of changes in knowledge; or (3) view changes in organizational performance indicators associated with experience as reflecting changes in knowledge [[Argote 07](#)].

The Open Group offers a certification program for qualifying the skills, knowledge, and experience of IT, business, and enterprise architects, which is related to measuring and certifying an individual architect's competence. Visit [opengroup.org](#) for details. The International Association of Software Architects (IASA) offers a similar certification; see [iasahome.org](#).

Dana Bredemeyer and Ruth Malan have written many articles on the role of the software architect ([www.bredemeyer.com/who.htm](#)), including their duties and skills [[Bredemeyer 11](#)] ([www.bredemeyer.com/Architect/ArchitectSkillsLinks.htm](#)). They have their own competence framework as well as a skills development program.

The U.K. Chapter of the International Council on Systems Engineering (INCOSE) maintains a "Core Competencies Framework" for systems engineers that includes a "Basic Skills and Behaviours" section listing "the usual common attributes required by any professional engineer" [[INCOSE 05](#)]. The list includes coaching, communication, negotiation and influencing, and "team working."

The classic work on the Balanced Scorecard was created by Kaplan and Norton [[Kaplan 92](#)] and the classic work on Business/IT Alignment was originally created by Luftman [[Luftman 00](#)], although this has been updated to explicitly consider the role of architecture in alignment [[Chen 10](#)].

Boehm, Valerdi, and Honour [[Boehm 07](#)] provide one of the few empirical studies of systems engineering and how an investment in "better" engineering pays off (or doesn't) in the future.

24.5. Discussion Questions

1. In which skills and knowledge discussed in this chapter do you think you might be deficient? How would you reduce these deficiencies?
2. Which duties, skills, or knowledge do you think are the most important or cost-effective to improve in an individual architect? Justify your answer.
3. How would you measure the specific value of architecture in a project? How would you distinguish the value added by architecture from the value added by other activities such as quality assurance or configuration management?
4. How do you measure items such as "customer satisfaction" or "negotiation skills"? How would you validate such measurements?
5. How would you distinguish benefits caused by systematic organizational learning from the benefits due to heroic efforts by individuals within the organization?
6. [Section 24.2](#) listed a number of practices of an architecturally competent organization. Prioritize that list based on expected benefit over expected cost.
7. Suppose you are in charge of hiring an architect for an important system in your company. How would you go about it? What would you ask the candidates in an interview? Would you ask them to produce anything? If so, what? Would you have them take a test of some kind? If so, what? Who in your company would you have interview them? Why?

25. Architecture and Software Product Lines

Coming together is a beginning. Keeping together is progress. Working together is success.

—Henry Ford

A software architecture represents a significant investment of time and effort, usually by senior talent. So it is natural to want to maximize the return on this investment by reusing an architecture across multiple systems.

There are many ways this happens in practice. The patterns we discussed in [Chapter 13](#) are a big step in this direction; using a pattern is reusing a package of architectural decisions (albeit not a complete architecture). And strictly speaking, every time you make a change to a system, you are reusing its architecture (or whatever portion of its architecture you don't have to change).

This chapter shows yet another way to reuse a software architecture (and many other assets as well) across a family of related systems, and the benefits that doing so can bring. Many software-producing organizations tend to produce systems or products that resemble each other more than they differ. This is an opportunity for reusing the architecture across these similar products. These software product lines simplify the creation of new members of a family of similar systems.

This kind of reuse has been shown to bring substantial benefits that include reduced cost of construction, higher quality, and greatly reduced time to market. This is the lure of the software product line approach to system building.

The Software Engineering Institute defines a software product line as “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 core assets in a prescribed way.”

The vision is of a set of reusable assets (called core assets) based on a common architecture and the software elements that populate that architecture. The core assets also include designs and their documentation, user manuals, project management artifacts such as budgets and schedules, software test plans and test cases, and more.

The product line approach works because the core assets were built specifically to support multiple members of the same family of products. Hence, reusing them is faster and less expensive than reinventing those software assets for each new product or system in the organization’s portfolio. Core assets, including the architecture, are usually designed with built-in variation points—places where they can be quickly tailored in preplanned ways.

Once the core assets are in place, system building becomes a matter of

- Accessing the appropriate assets in the core asset base
- Exercising the variation points to configure them as required for the system being built
- Assembling that system

In the ideal case, this can be done automatically. Additional software developed for an individual product, if needed at all, tends to account for a small fraction of the total software. Integration and testing replace design and coding as the predominant activities.

Product lines are nothing new in manufacturing. Many historians trace the concept to Eli Whitney’s use of interchangeable parts to build rifles in the early 1800s, but earlier examples also exist. Today, there are hundreds of examples in manufacturing: think of the products of companies like General Motors, Toyota, Boeing, Airbus, Dell, even McDonald’s, and the portfolio of similar products that each one produces. Each company exploits commonality in different ways. Boeing, for example, developed the 757 and 767 in tandem, and the parts lists of these two very different aircraft overlap by about 60 percent.

The improvements in cost, time to market, and productivity that come with a successful software product line can be breathtaking. Consider:

- Nokia credits the software product line approach with giving it flexibility to bring over a dozen phones to market each year, as opposed to the three or so it could manage before, all with an unprecedented variety of features.

- Cummins, Inc., was able to reduce the time it takes to produce the software for a diesel engine from about a year to about a week.
- Hewlett-Packard builds products using one-quarter of the staff, in one-third of the time, and with one twenty-fifth the number of defects, compared with software built before the advent of software product line engineering.
- Deutsche Bank estimates \$4 million in savings per year realized from building its global transaction and settlement software as a product line.
- Philips reports reduced faults during integration in its high-end television portfolio by adopting the product line approach. Product diversity used to be one of the top three concerns of their architects. Now it doesn't even make the list of concerns at all; the product line approach has taken software development off the critical path—the software no longer determines the delivery date of the product.
- With a product line of satellite ground control systems it commissioned, the U.S. National Reconnaissance Office reported the first product requiring 10 percent the expected number of developers and having one-tenth the expected number of defects.
- In Philips's medical systems product line, the software product line approach has cut both software defects and time to market by more than half.

Creating a successful product line depends on a coordinated strategy involving software engineering, technical management, and organization management. Because this is a book on software architecture, we focus on the architectural aspects of software product lines, but all aspects must work together in order for an organization to successfully create a product line.

That Silver Lining Might Have a Cloud

The software product line paradigm is a powerful way to leverage an investment in architecture (and other core assets) into a family of related systems and thus see order-of-magnitude improvements in time to market, quality, and productivity. These results are possible and have been demonstrated by companies large and small in many different domains. The effects are real. Further, data from many sources and companies confirms with astonishing consistency that, to make the investment pay off, an organization needs to build only three products. This is the minimum number we would expect to have in a product line.

But other results are possible as well, and a spectacular crash-and-burn is not out of the question when trying to adopt this approach. Product line practice, like any technology, needs careful thought given to its adoption, and a company's history, situation, and culture must be taken into account. Factors that can contribute to product line failure include these:

- Lack of a champion in a position of sufficient control and visibility
- Failure of management to provide sustained and unwavering support
- Reluctance of middle managers to relinquish autocratic control of projects
- Failure to clearly identify business goals for adopting the product line approach
- Abandoning the approach at the first sign of difficulty
- Failure to adequately train staff in the approach and failure to explain or justify the change adequately
- Lack of discipline in managing the architecture's variation points
- Scoping the product line too broadly or too narrowly
- Lack of product line tooling to help manage and exercise the variation points

Fortunately, there are strategies for overcoming most of these factors. One good strategy is to launch a small but visible pilot project to demonstrate the quantitative benefits of software product lines. The pilot can be staffed by those most willing to try something new while the skeptics go about their business. It can work out process issues, clarify roles and responsibilities, and in general work out the bugs before the approach is transitioned to a wider setting.

—PCC

25.1. An Example of Product Line Variability

The following example will help us illustrate the concept of product line variability. In a product line of software to support U.S. bank loan offices, suppose we have a software module that calculates what a customer owes in the current month. For 18 of the 21 products in our product line, this module is completely adequate. However, our company is about to enter the market in the state of Delaware, which has certain laws that affect what a customer can owe. For the three products we plan to sell in Delaware, we need a module that differs from the “standard” module. Analysis shows that the difference will affect about 250 lines of source code in our 8,000-line module.

To build one of the Delaware products, what do we do? An obvious option is to copy the module, change the 250 or so lines, and use the new version in the three products. This practice is called “clone-and-own”—the new projects “clone” the module, change it, and then “own” the new version. Most companies, when faced with this situation, resort to clone-and-own. It’s expedient in that it provides a quick start to a new product, but it comes with a substantial cost down the road.

The problem with clone-and-own is that it doesn’t scale. Suppose each of our 21 products comprises roughly 100 modules. If each module is allowed to diverge for each product, that’s potentially 2,100 modules that the maintenance staff has to deal with, each one spiraling off on its own separate maintenance trajectory based on the needs of the lone project each version is used in. Many companies’ growth in a market is limited—brought to a halt, in fact—by their inability to staff the maintenance of so many separate versions of so many different assets composing the products in their portfolio. An organization fielding several versions of several products finds itself dealing with a staggeringly complex code base. The strain begins to show when a systematic change needs to be made to all of the products—for example, to add a new feature, or migrate to a new platform, or make the user interface work in a different language. Because each version of each component used in each product has been allowed to evolve separately, now suddenly making a systematic change becomes prohibitively expensive (and only gets worse each time a new product is added—the labor involved grows as the *square* of the number of products). It only takes a few such portfolio-wide changes before organizations feel that they’ve hit a wall of complexity and expense.

So much for clone-and-own. What else can we do? Instead of allowing up to 21 versions of each module, we would much rather find a way to take advantage of the fact that these nearly identical modules vary only in small, well-defined ways. To take advantage of their similarities, we introduce a *variation mechanism* into the module. (Variation mechanisms are often realized as tactics, such as the “defer binding” set of tactics described in [Chapter 7](#).) This variation mechanism will let us maintain a single module that can adapt to the range of variations in the applications (in our example, the 21 banking products) that it has to support. If we plan to market our products in states that, like Delaware, have their own laws affecting what a customer owes, we may need to support additional variations of the module. So our variation mechanism should be able to accommodate those possibilities as well.

The payoff for this up-front planning is that an asset used in any of the products exists as a *single* version that (through the exercising of built-in variation mechanisms) works for all of the products in the product line. And now, making a portfolio-wide change merely consists of changing the core assets that are affected. Because all future versions of all products use the same core assets, changing the core asset base has the effect of changing all of the products in the organization’s portfolio.

25.2. What Makes a Software Product Line Work?

What makes product lines succeed is that the commonalities shared by the products can be exploited through reuse to achieve production economies. The potential for reuse is broad and far-ranging, including the following:

- *Requirements.* Most of the requirements are common with those of earlier systems and so can be reused. In fact, many organizations simply maintain a single set of requirements that apply across the entire family as a core asset; the requirements for a

particular system are then written as “delta” documents off the full set. In any case, most of the effort consumed by requirements analysis is saved from system to system.

- *Architectural design.* An architecture for a software system represents a large investment of time from the organization’s most talented engineers. As we have seen, the quality goals for a system—performance, reliability, modifiability, and so forth—are largely promoted or inhibited once the architecture is in place. If the architecture is wrong, the system cannot be saved. For a new product, however, this most important design step is already done and need not be repeated.
- *Software elements.* Software elements are applicable across individual products. Element reuse includes the (often difficult) initial design work. Design successes are captured and reused; design dead ends are avoided, not repeated. This includes design of each element’s interface, its documentation, its test plans and procedures, and any models (such as performance models) used to predict or measure its behavior. One reusable set of elements is the system’s user interface, which represents an enormous and vital set of design decisions. And as a result of this interface reuse, products in a product line usually enjoy the same look and feel as each other, an advantage in the marketplace.
- *Modeling and analysis.* Performance models, schedulability analysis, distributed system issues (such as proving the absence of deadlock), allocation of processes to processors, fault tolerance schemes, and network load policies all carry over from product to product. Companies that build real-time distributed systems report that one of the major headaches associated with production has all but vanished. When they field a new product in their product line, they have high confidence that the timing problems have been worked out and that the bugs associated with distributed computing—synchronization, network loading, and absence of deadlock—have been eliminated.
- *Testing.* Test plans, test processes, test cases, test data, test harnesses, and the communication paths required to report and fix problems are already in place.
- *Project planning artifacts.* Budgeting and scheduling are more predictable because experience is a high-fidelity indicator of future performance. Work breakdown structures need not be invented each time. Teams, team size, and team composition are all easily determined.

All of these represent valuable core assets, each of which can be imbued with its own variation points that can be exercised to build a product. We’ll look at architectural variation points later in this chapter, but for now imagine that any artifact represented by text can consist of text blocks that are exposed or hidden for a particular product. Thus, the artifact that is maintained in the core asset base represents a superset of any version that will be produced for a product.

Artifact reuse in turn enables reuse of knowledge:

- *Processes, methods, and tools.* Configuration control procedures and facilities, documentation plans and approval processes, tool environments, system generation and distribution procedures, coding standards, and many other day-to-day engineering support activities can all be carried over from product to product. The software development process is in place and has been used before.

Giving Software Reuse a New Lease on Life

Software product lines rely on reuse, but reuse has a long but less than stellar history in software engineering, with the promise almost always exceeding the payoff. One reason for this failure is that until now reuse has been predicated on the idea of “If you build it, they will come.” A reuse library is stocked with snippets from previous projects, and developers are expected to check it first before coding new elements. Almost everything conspires against this model. If the library is too sparse, the developer will not find anything of use and will stop looking. If the library is too rich, it will be hard to understand and search. If the elements are too small, it is easier to rewrite them than to find them and carry out whatever modifications they might need. If the elements are too large, it is difficult to determine exactly what they do in detail, which in any case is not likely to be exactly right for the new application. In most reuse libraries, pedigree is hazy at best. The developer cannot be sure exactly what the element does, how reliable it is, or under what conditions it was tested. And there is almost never a match between the quality attributes needed for the new application and those provided by the elements in the library.

In any case, it is common that the elements were written for a different architectural model than the one the developer of the new system is using. Even if you find something that does the right thing with the right quality attributes, it is doubtful that it will be the right kind of architectural element (if you need an object, you might find a process), that it will have the right interaction protocol, that it will comply with the new application's error-handling or failover policies, and so on.

This has led to so many reuse failures that many project managers have given up on the idea. "Bah!" they exclaim. "We tried reuse before, and it doesn't work!"

Software product lines make reuse work by establishing a strict context for it. The architecture is defined; the functionality is set; the quality attributes are known. Nothing is placed in the reuse library—or "core asset base" in product line terms—that was not built to be reused in that product line. Product lines work by relying on strategic or planned, not opportunistic, reuse.

—PCC

- *People*. Because of the commonality of applications, personnel can be transferred among projects as required. Their expertise is applicable across the entire line.
- *Exemplar systems*. Deployed products serve as high-quality demonstration prototypes or engineering models of performance, security, safety, and reliability.
- *Defect elimination*. Product lines enhance quality because each new system takes advantage of the defect elimination in its forebears. Developer and customer confidence both rise with each new instantiation. The more complicated the system, the higher the payoff for solving vexing performance, distribution, reliability, and other engineering issues once for the entire family.

All of this reuse helps products launch more quickly, with higher quality, lower cost, and more predictable budget and schedule. This is critical for getting a product to market in a timely fashion. However, these benefits do not come for free. A product line may require a substantial up-front investment of time and effort to set up and manage, as well as to keep the core assets responsive to changing market needs.

25.3. Product Line Scope

One of the most important inputs to an architect building an architecture for a software product line is the product line's scope. A product line's scope is a statement about what systems an organization is willing to build as part of its line and what systems it is not willing to build. Defining a product line's scope is like drawing a doughnut in the space of all possible systems, as shown in [Figure 25.1](#). The doughnut's center represents the systems that the organization could easily build using its base of core assets; these are within its production capability. Systems outside the doughnut are out of scope because they are ones the product line's core assets are not well equipped to handle; this would be like asking Toyota to build, say, apple pies on one of its automotive assembly lines.

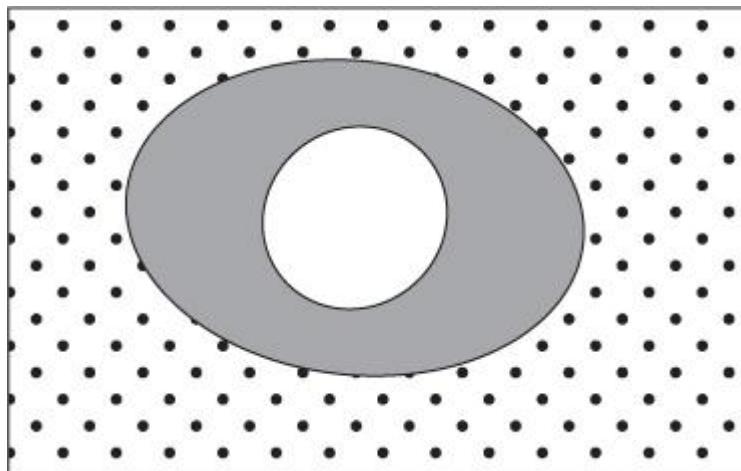


Figure 25.1. The space of all possible systems is divided into areas within scope (white), areas outside of scope (speckled), and areas that require case-by-case disposition (gray).

Systems on the doughnut itself could be handled, but with some effort. These often represent invitations from the marketplace asking the organization to extend its product line. To take advantage of such an opportunity, the organization would have to broaden its production capability—that is, make its core asset base able to handle the new product. These opportunities require case-by-case disposition as they arise, to see if the potential payoff (such as entry into a slightly different area of the market) would outweigh the cost to modify the core assets. This would be like asking Toyota to build a riding lawnmower.

The scope represents the organization's best prediction about what products it will be asked to build in the foreseeable future. Input to the scoping process comes from the organization's strategic planners, marketing staff, domain analysts who can catalog similar systems (both existing and on the drawing board), and technology experts.

A product line scope is a critical factor in the success of the product line. Scope too narrowly (the products only vary in a small number of features) and an insufficient number of products will be derived to justify the investment in development. Scope too broadly (the products vary in kind as well as in features) and the effort required to develop individual products from the core assets is too great to lead to significant savings. Scope can be refined as a portion of the initial establishment of the product line or opportunistically depending on the product line adoption strategy (see the section on adoption strategies in [Section 25.8](#)).

The problem in defining the scope is not in finding commonality—a creative architect can find points of commonality between *any* two systems—but in finding commonality that can be exploited to substantially reduce the cost of constructing the systems that an organization intends to build. When considering scope, more than just the systems being built should be considered. Market segmentation and types of customer interactions assumed will help determine the scope of any particular product line. For example, Philips, the Dutch manufacturer of consumer electronics, has distinct product lines for home video electronic systems and digital video communication. Video is the common thread, but one is a mass market, where the customer is assumed to have very little video sophistication, and the other is a much smaller market consisting purely of video professionals. The products being developed reflect these assumptions about the sophistication of customers and the amount of care each customer will receive. These differences were sufficient to keep Philips from attempting to develop a single product line for both markets.

Narrowly scoped product lines offer opportunities to build specialized tools to support the specification of new products. For example, General Motors' Powertrain division builds a software product line of automotive software. It makes an individual product from its product line core assets based on contracts stored in a database. Each element has well-defined interfaces and possible variation points. A tool searches the database based on desired features and assembles the product.

The scope definition is vital to the product line architect because the scope defines what is common across all members of the product line, and the specific ways in which the products differ

from each other. The fixed part of a product line architecture reflects what is constant, and the architecture's variation points accommodate the variations among products.

25.4. The Quality Attribute of Variability

Scoping decisions, which tell the product line architect what kinds of systems are "in" and what kinds of systems are "out" of the product line, lead to the introduction of variability in the core assets. In fact, the quality attribute of variability is most closely associated with product lines. Some may feature high-performance products, or high-security products, or high-availability products, but *all* product lines feature variability aimed at satisfying the commonalities and variations identified by the product line's scope.

We introduced variability in [Chapter 12](#). There we said that variability is a special form of modifiability, pertaining to the ability of a core asset to adapt to usages in the different product contexts that are within the product line scope. The goal of variability in a software product line is to make it easy to build and maintain products in the product line over time.

[Table 25.1](#) gives the general scenario for variability. The source is some actor in the product line organization who identifies a need for variation; this actor is probably someone involved in setting the product line's scope, such as a marketer.

Table 25.1. The General Scenario for Variability

Portion of Scenario	Possible Values
Source	Actor requesting variability
Stimulus	Requests to support variations in the following: <ul style="list-style-type: none">▪ Hardware▪ Feature sets▪ Technologies▪ User interfaces▪ Quality attributes▪ ... and more for the range of products affected, such as: <ul style="list-style-type: none">▪ All▪ A specified subset▪ Those that include feature set x▪ New products
Environment	Variants are to be created at: <ul style="list-style-type: none">▪ Runtime▪ Build time▪ Development time
Artifact	Asset(s) affected, such as: <ul style="list-style-type: none">▪ Requirements▪ Architecture▪ Component x▪ Test suite y▪ Project plan z▪ ... and more
Response	The requested variants can be created.
Response measure	A specified cost and/or time to create the core assets and to create the variants using these core assets

Identifying variation is a constant, iterative process in the life of a software product line. Because of the many different ways a product can vary, particular variants can be identified at virtually any time during the development process. Some variations are identified during product line requirement elicitations; others, during architecture design; and still others, during implementation. Variations may also be identified during implementation of the second (and subsequent) products as well.

Product line architectures feature *variability* as an important quality attribute. They achieve this by incorporation of *variation mechanisms*, which we will discuss in more detail shortly.

25.5. The Role of a Product Line Architecture

Of all of the assets in a core asset repository, the software architecture plays the most central role. There is both a tactical and a strategic reason for this.

The tactical reason is the importance the architecture plays in building products in a product line. The essence of building a successful software product line is discriminating between what is expected to remain constant across all family members and what is expected to vary. Software architecture is ideal for handling this variation, because all architectures are abstractions that admit multiple instances. By its very nature every architecture is a statement about what we expect to remain constant and what we admit may vary. For example, interfaces to components are designed to remain stable, with anticipated changes hidden behind those interfaces.

In a software product line, the architecture has to encompass both the varying and the nonvarying aspects. A product line architecture must be designed to accommodate a set of explicitly allowed variations. Thus, identifying the allowable variations is part of the architect's responsibility, as is providing built-in mechanisms for achieving them. Those variations may be substantial. Products in a software product line exist simultaneously and may vary in terms of their behavior, quality attributes, platform, network, physical configuration, middleware, scale factors, and so forth.

The strategic reason has to do with the capability it imparts to an organization outside the realm of an existing product line. As we saw in [Chapters 2](#) and [3](#), an architecture can serve as a technical platform for launching new applications and even new business models, and it can serve as a springboard for an organization diving into a new business area. This seems to be especially true for product line architectures. There are many cases where an organization has taken advantage of its production capability—that is, its core asset base crowned by a product line architecture—by using that capability to enter new markets. For example, Cummins took its product line of automotive diesel engines to enter and quickly dominate the neighboring market for industrial diesel engines. Industrial diesel engines power things like rock crushers and ski lifts, markets of low volume and high specialization. Systems in that market built uniquely for each application are expensive and don't yield a high return. But a product line that includes industrial diesel engines in its scope, and whose production capability supports industrial diesel engines, is a recipe for rapid market capture.

A product line architect needs to consider three things that are unique to product line architectures:

- *Identifying variation points.* This is done by using the scope definition and product line requirements as input. The product line architect determines where in the architecture variation points should be made available to support the rapid building of products.
- *Supporting variation points.* This is done by introducing variation mechanisms, which will be discussed in the next section.
- *Evaluating the architecture for product line suitability,* which will be discussed later in this chapter.

25.6. Variation Mechanisms

In a conventional architecture, the mechanism for achieving different instances often comes down to modifying the code. But in a software product line, modifying code is undesirable, because this leads to a large number of separately maintained implementations that quickly outstrip an organization's ability to keep them up to date and consistent.

Three primary *architectural* variation mechanisms are these:

- *Inclusion or omission of elements.* This decision can be reflected in the build procedures for different products, or the implementation of an element can be conditionally compiled based on some parameter indicating its presence or absence.
- *Inclusion of a different number of replicated elements.* For instance, high-capacity variants might be produced by adding more servers—the actual number should be unspecified, as a point of variation, and may be done dynamically.
- *Selection of different versions of elements that have the same interface but different behavioral or quality attribute characteristics.* Selection can occur at compile time, build time, or runtime. Selection mechanisms include static libraries, which contain external functions linked after compilation time; dynamic link libraries, which have the flexibility of static libraries but defer the decision until runtime based on context and execution conditions; and add-ons (e.g., plug-ins, extensions, and themes), which add or modify

application functionality at runtime. By changing the libraries, we can change the implementation of functions whose names and signatures are known.

Some variation mechanisms can be introduced that change aspects of a particular software element. Modifying the source code each time the element is used in a new product—that is, clone-and-own—falls into this category, although it is undesirable. More sophisticated techniques include the following:

- *Extension points*. These are identified places in the architecture where additional behavior or functionality can be safely added.
- *Reflection*. This is the ability of a program to manipulate data on itself or its execution environment or state. Reflective programs can adjust their behavior based on their context.
- *Overloading*. This is a means of reusing a named functionality to operate on different types. Overloading promotes code reuse, but at the cost of understandability and code complexity.

Other commonly used variation mechanisms include those in [Table 25.2](#).

Table 25.2. Common Variation Mechanisms

Variation Mechanism	Properties Relevant to Building the Core Assets	Properties Relevant to Exercising the Variation Mechanism When Building Products
Inheritance; specializing or generalizing a particular class	Cost: Medium Skills: Object-oriented languages	Stakeholder: Product developers Tools: Compiler Cost: Medium
Component substitution	Cost: Medium Skills: Interface definitions	Stakeholder: Product developer, system administrator Tools: Compiler Cost: Low
Add-ons, plugins	Cost: High Skills: Framework programming	Stakeholder: End user Tools: None Cost: Low
Templates	Cost: Medium Skills: Abstractions	Stakeholder: Product developer, system administrator Tools: None Cost: Medium
Parameters (including text preprocessors)	Cost: Medium Skills: No special skills required	Stakeholder: Product developer, system administrator, end user Tools: None Cost: Low
Generator	Cost: High Skills: Generative programming	Stakeholder: System administrator, end user Tools: Generator Cost: Low
Aspects	Cost: Medium Skills: Aspect-oriented programming	Stakeholder: Product developer Tools: Aspect-oriented language compiler Cost: Medium
Runtime conditionals	Cost: Medium Skills: No special skills required	Stakeholder: None Tools: None Cost: No development cost; some performance cost
Configurator	Cost: Medium Skills: No special skills required	Stakeholder: Product developer Tools: Configurator Cost: Low to medium

Choosing the right variation mechanism affects numerous costs:

- The skill set required to implement, or learn and use, the specific variation mechanism, such as server or framework programming
- The one-time costs of building or acquiring the tools (such as compilers or generators) required to create the variation mechanism
- The recurring cost and time to exercise the variation mechanism

The choice of variation mechanism also affects downstream users and developers:

- The targeted group of users that use the mechanism for product-specific adaptation, such as product developer, integrator, system administrator, and end user

Finally, the choice of variation mechanism affects product quality:

- The impact of the variation mechanism on quality, such as possible performance penalties or memory consumption
- The impact on the mechanism's maintainability

The architect should document the choice of variation mechanisms. In fact, the documentation of variation mechanisms is the primary way in which the documentation for a product line architecture differs from that of a conventional architecture. In the documentation template we presented in [Chapter 18](#), the section called the *variability guide* is reserved for exactly this purpose. The variability guide should describe each variation mechanism, how and when to exercise it, and what allowed variations it supports. The architecture documentation should also describe the architecture's instantiation process—that is, how its variation points are exercised. Also, if certain combinations of variations are disallowed, then the documentation needs to explain valid and invalid variation choices.

25.7. Evaluating a Product Line Architecture

Like any other, the architecture for a software product line should be evaluated for fitness of purpose. The architecture should be evaluated for its robustness and generality, to make sure it can serve as the basis for products in the product line's envisioned scope. It should also be evaluated to make sure it meets the specific behavioral and quality requirements of the product at hand. We begin by focusing on the what and how of the evaluation and then turn to when it should take place.

What and How to Evaluate. The evaluation will have to focus on the variation points to make sure they are appropriate, that they offer sufficient flexibility to cover the product line's intended scope, that they allow products to be built quickly, and that they do not impose unacceptable runtime performance costs. If your evaluation is scenario based, expect to elicit scenarios that involve instantiating the architecture to support different products in the family. Also, different products in the product line may have different quality attribute requirements, and the architecture will have to be evaluated for its ability to provide all required combinations. Here again, try to elicit scenarios that capture the quality attributes required of family members.

Often, some of the hardware and other performance-affecting factors for a product line architecture are unknown to begin with. In this case, evaluation can establish bounds on the performance that the architecture is able to achieve, assuming bounds on hardware and other variables. The evaluation can identify potential contention so that you can put in place the policies and strategies to resolve it.

When to Evaluate. An evaluation should be performed on an instance or variation of the architecture that will be used to build one or more products in the product line. The extent to which this is a separate, dedicated evaluation depends on the extent to which the product's requirements differ from the product line architecture envelope. If it does not differ, the product architecture evaluation can be abbreviated, because many of the issues normally raised in a single product evaluation will have been dealt with in the product line evaluation. In fact, just as the product architecture is a variation of the product line architecture, the product architecture evaluation is a variation of the product line architecture evaluation. Therefore, depending on the evaluation method used, the evaluation artifacts (scenarios, checklists, and so on) will have reuse potential, and you should create them with that in mind. The results of evaluation of product architectures often provide useful feedback to the product line architects and fuel architectural improvements.

When a new product is proposed that falls outside the scope of the original product line (for which the architecture was presumably evaluated), the product line architecture can be reevaluated to see if it will suffice for it. If it does, the product line's scope can be expanded to include the new product, or to spawn a new product line. If it does not, the evaluation can determine how the architecture will have to be modified to accommodate the new product. The product line and product instance architectures can be evaluated not only to determine architectural risks but also to understand economic consequences (see [Chapter 23](#)), to determine which products will yield the most return.

25.8. Key Software Product Line Issues

It takes considerable maturity in the developing organization to successfully field a product line. Technology is not the only barrier to this; organization, process, and business issues are equally vital to master to fully reap the benefits of the software product line approach.

Architecture definition is an important activity for any project, but as we saw in the previous section, it needs to emphasize variation points in a software product line. Configuration management is also an important activity for any project, but it is more complex for a software product line because each product is the result of binding a large number of variations. The configuration management problem for product lines is to reproduce any version of any product delivered to any customer, where “product” means code and supporting artifacts ranging from requirement specs and test cases to user manuals and installation guides. This involves knowing what version of each core asset was used in a product’s construction, how every asset was tailored, and what special-purpose code or documentation was added.

Examining every facet of launching a product line and institutionalizing a product line culture is outside the scope of this book, but the next sections will examine a few of the key areas that must be addressed. These are issues that an organization will have to face when considering whether to adopt a product line approach for software development and, if so, how to go about it.

Adoption Strategies

An organization’s culture and context will dramatically affect how it goes about adopting a product line approach. Here are some of the important organizational and process factors that we have seen in practice.

Top-Down vs. Bottom-Up

Top-down adoption arises when a (typically high level) manager decrees that the organization will use the approach. The problem is to get employees in the trenches to change the way they work. Bottom-up adoption happens when designers and developers working at the product level realize that they are needlessly duplicating each other’s work and begin to share resources and develop generic core assets. The problem is finding a manager willing to sponsor the work and spread the technique to other parts of the organization. Both approaches work; both are helped enormously by the presence of a strong *champion*—someone who has thoroughly internalized the product line vision and can share that compelling vision with others. (It works better if the champion is in a position of some authority.)

Proactive vs. Reactive

There are two primary models for how an organization may grow a product line:

- In a proactive product line, an organization defines the family using a comprehensive definition of scope. They do this not with a crystal ball but by taking advantage of their experience in the application area, their knowledge about the market and technology trends, and their good business sense. The proactive model allows the organization to make the most far-reaching strategic decisions. Explicitly scoping the product line allows you to look at areas that are underrepresented by products already in the marketplace, make small extensions to the product line, and move quickly to fill the gap. In short, proactive product line scope allows an organization to take charge of its own fate. Sometimes an organization does not have the ability to forecast the needs of the market with the certainty suggested by the proactive model. The proactive model also takes some time to define and implement, and in that time the organization needs to continue to construct products.
- In a reactive product line, an organization builds the next member or members of the product family from earlier products. This is best used when there is uncertainty of requirements. Perhaps the domain is a new one. Perhaps the market is in flux. Or perhaps the organization cannot afford to build a core asset base that will cover the entire scope all at once. In the reactive model, with each new product the architecture is extended as needed and the core asset base is built up from what has turned out to be common. The reactive model puts much less emphasis on up-front planning and strategic direction

setting. Rather, the organization lets itself be taken where the market dictates. This is an example of agile architecting, as described in [Chapter 15](#).

Incremental vs. Big Bang

If you are proactively building a product line, you still need to choose how to populate it: all at once or incrementally over time. Populating the core asset base all at once is a strategy that has worked successfully for some organizations. However, it tends to require all or nearly all of the organization's resources be focused on that task, at the expense of new product production. A different approach is to populate the core asset base incrementally, as circumstances and resources permit. Each product that goes out the door is built with whatever core assets are available at the time. That means that early products will include software not derived from core assets. But those products will still be better off (that is, faster to market, of higher quality, and easier to maintain) than products built entirely from unique code. And it's entirely possible that some of the software unique to those early products can be extracted, adapted, and generalized to become core assets themselves, thus helping populate the core asset base in a reactive fashion.

Knowing the various adoption models can help an organization choose the one that is right for it. For example, the proactive model requires a heavier initial investment but less rework than the reactive model. The reactive model relies exclusively on rework with little initial investment. Which model should act as a guide for a particular organization depends on the business situation.

Creating Products and Evolving a Product Line

An organization that has a product line will have an architecture and a collection of elements associated with it. From time to time, the organization will create a new member of the product line that will have features both in common with and different from those of other members.

One problem associated with a product line is managing its evolution. As time passes, the line—or, more precisely, the set of core assets from which products are built—must evolve. That evolution will be driven by both external and internal sources:

External sources

- New versions of existing elements within the line will be released by their vendors, and future products will need to be constructed from them.
- New externally created elements may be added to the line. Thus, for example, functions that were previously performed by internally developed elements may now be performed by elements acquired externally, or vice versa. Or future products will need to take advantage of new technology, as embodied in externally developed elements.
- New features may be added to the product line to keep it responsive to user needs or competitive pressures.

Internal sources

- Some entity within the organization must determine if new functions added to a product are within the product line's scope. If so, they can simply be built from the asset base. If not, a decision must be made: either the enhanced product spins off from the product line, following its own evolutionary path, or the asset base must be expanded to include it. Updating the line may be the wisest choice if the new functionality is likely to be used in future products, but this capability comes at the cost of the time necessary to update the core assets.
- An organization may wish to replace old products with ones built from the most up-to-date version of the asset base. Keeping products compatible with the product line takes time and effort. But not doing so may make future upgrades more time consuming, because either the product will need to be brought into compliance with the latest product line elements or it will not be able to take advantage of improvements in the line.

Organizational Structure

An asset base on which products depend, but which has its own evolutionary path (perhaps driven by technology change), requires an organization to decide how to manage both it and product development. There are two main organizational strategies from which to choose, plus a number of minor variations. The two main structures reflect different answers to the question "Shall we have a dedicated group whose sole job is to build and maintain our core asset base?"

- 1. We're all in this together.** In this scheme, there is no separate core asset group. The product-building development teams coordinate closely, and divide up the core asset responsibilities among themselves. That is, Product Team 1 might be assigned responsibility for the development and maintenance of Core Assets 3, 6, 9, 12, and 15; Product Team 2 might take Core Assets 1, 4, and 8; and so forth. This works well enough for small organizations, but as size grows the communication channels become untenable. Also, each team has to resist the temptation to build core assets that are especially appropriate to its needs, but less so to other teams' needs.
- 2. Separate core asset unit.** In this scheme, a special unit is given responsibility for the development and maintenance of the core asset base. Separate development teams in the organization's business units build the products. In this scheme, the core asset unit (sometimes called a domain engineering unit) assumes the responsibility for the overall strategic direction of the product line. To the product teams, they appear almost like an external supplier. The product teams coordinate among themselves to set the core asset team's development and test priorities, based on product delivery obligations.

25.9. Summary

This chapter presented an architecture-based development paradigm known as software product lines. The product line approach is steadily climbing in popularity as more organizations see true order-of-magnitude improvements in cost, schedule, and quality from using it.

Like all technologies, however, this one holds some surprises for the unaware. Architecturally, the key is identifying and managing commonalities and variations, but nontechnical issues must be addressed as well, including how the organization adopts the model, structures itself, and maintains its external interfaces.

25.10. For Further Reading

[\[Clements 01a\]](#) is a comprehensive treatment of software product lines. It includes a number of case studies as well as a thorough discussion of product line "practice areas," which are areas of expertise a product line organization should have (or should develop) to help bring about product line success.

[\[van der Linden 07\]](#) contains a rich set of product line case studies.

[\[Anastasopoulos 00\]](#) presents a good list of variation mechanisms, as do [\[Jacobson 97\]](#) and [\[Svahnberg 00\]](#). [\[Bachmann 05\]](#) provides a list of their own, as well as a treatment of each in terms of cost (it was the source for [Table 25.2](#)). Organizational models for software product lines are treated in [\[Bosch 00\]](#).

There is an active software product line community of research and practice. The Software Product Line Conference (SPLC) is the mainstream forum for new software product line research and success stories. You can find it at [www.splc.net](#). SPLC maintains a "Software Product Line Hall of Fame," which showcases successful software product lines that can serve as engineering models (and inspiration) to aspiring product line organizations. Each year, new members of the Hall of Fame are nominated, and in most years a new candidate is inducted. You can see the winners at [www.splc.net/fame.html](#).

The SEI's website contains a wealth of material about software product lines, including a collection of "getting started" material: [www.sei.cmu.edu/productlines](#).

25.11. Discussion Questions

1. Variability is achieved by adding variation mechanisms to a system. Variation mechanisms include inheritance, component substitution, plug-ins, templates, parameters (including text preprocessors), generators, aspects, runtime conditionals, and a configurator tool. Because variability can be seen as a kind of modifiability, see if you can map each of these variation mechanisms to one or more modifiability tactics given in [Chapter 7](#).

- 2.** Suppose a company builds two similar systems using a large set of common assets, including an architecture. Which of the following would you say constitutes a product line?
- Sharing only an architecture but no elements.
 - Sharing only a single element.
 - Sharing the same operating system and programming language runtime libraries.
 - Sharing the same team of developers.

Defend your answer.

- 3.** Pick a type of system you're familiar with—for example, an automobile or a smartphone. Think of three instances of that kind of system. Make a list of all of the things the three instances have in common. Now make a list of all of the things that distinguish the three instances from each other (that is, their variation points). If automobiles turn out to be too complex, start with a simpler kind of "system," such as an electric light.
- 4.** Write some concrete scenarios to express the variability you identified in the previous question.
- 5.** Do the list of variation mechanisms in this chapter constitute tactics for variability? Discuss.
- 6.** In many software product lines, products differ by the quality attributes they exhibit. For instance, a company might sell a cheap, low-security version of its product alongside a more expensive, high-security version of the same product. Which variation mechanisms might you choose to achieve this kind of variability?

Part Five. The Brave New World

[Parts I](#) through [IV](#) of this book have dealt with the technical, organizational, and business perspectives on software architecture. In this part, we turn our attention to emerging technologies. We have often been asked whether principles or technology is more important, and the answer, of course, is "Yes, they are both important." Principles have a long lifetime; technology that affects architects tends to change every decade or so. In this part, we provide brief introductions to two technologies that we believe will last and have a significant impact on architects—the cloud and the edge. We also discuss one of the continuing problems of many architects: How do I get my organization to embrace architectural principles?

The cloud provides you with the option of outsourcing your data center. The vision is that computing resources are available to an application as electricity is available to a consumer. That is, one plugs in an appliance and electricity is available. In data center terms, you hook your web browser up to an application and computation power is available. All of the capacity, management, and operational issues of a data center are taken care of by a third party, and all you, as an architect, need to do is to utilize the resources you need. This trend has been accompanied by a vast expansion of the amount of data that organizations manage. Google, Yahoo!, Facebook, and the other web giants all must manage petabytes of data. In [Chapter 26](#), we provide a brief introduction to the technologies associated with the cloud and with managing these vast amounts of data.

Cloud computing is associated with the world of social networks and open source. The term "edge-periphery" is used to describe both the crowdsourcing and the open source movements. The term refers simultaneously to crowdsourced systems such as Facebook and Wikipedia and open source systems such as the Apache Web Server and Hadoop. In [Chapter 27](#), we describe this phenomenon and explore some of the architectural implications of this aspect of the brave new world.

We end by discussing "adoption." It describes an approach to dealing with the following problem: "OK, you guys have convinced me. Now I need to convince my organization of the importance of architectural principles. How do I do that?"

26. Architecture in the Cloud

There was a time when every household, town, farm or village had its own water well. Today, shared public utilities give us access to clean water by simply turning on the tap; cloud computing works in a similar fashion.

—Vivek Kundra

If you have read anything about the history of computing, you will have read about time-sharing. This was the era, in the late 1960s and the 1970s, sandwiched between eras when individuals had sole, although limited, access to multimillion-dollar computers and when individuals had access to their own personal computers. Time-sharing involved multiple users (maybe as many as several hundred) simultaneously accessing a powerful mainframe computer through a terminal, potentially remote from the mainframe. The operating system on the mainframe made it appear as if each user had sole access to that computer except, possibly, for performance considerations. The driving force behind the development of time-sharing was economic; it was infeasible to provide every user with a multimillion-dollar computer, but efficiently sharing this expensive but powerful resource was the solution.

In some ways, cloud computing is a re-creation of that era. In fact, some of the basic techniques—such as virtualization—that are used in the cloud today date from that period. Any user of an application in the cloud does not need to know that the application and the data it uses are situated several time zones away, and that thousands of other users are sharing it. Of course, with the advent of the Internet, the availability of much more powerful computers today, and the requirement for controlled sharing, designing the architecture for a cloud-based application is much different from designing the architecture for a time-sharing-based application. The driving forces, however, remain much the same. The economics of using the cloud as a deployment platform are so compelling that few organizations today can afford to ignore this set of technologies.

In this chapter we introduce cloud concepts, and we discuss various service models and deployment options for the cloud, the economic justification for the cloud, the base architectures and mechanisms that make the cloud work, and some sample technologies. We will conclude by discussing how an architect should approach building a system in the cloud.

26.1. Basic Cloud Definitions

The essential characteristics of cloud computing (based, in part, on definitions provided by the U.S. National Institute of Standards and Technology, or NIST) are the following:

1. *On-demand self-service.* A resource consumer can unilaterally provision computing services, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider. This is sometimes called empowerment of end users of computing resources. Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.
2. *Ubiquitous network access.* Cloud services and resources are available over the network and accessed through standard networking mechanisms that promote use by a heterogeneous collection of clients. For example, you can effectively run large applications on small platforms such as smart phones, laptops, and tablets by running the resource-intensive portion of those applications on the cloud. This capability is independent of location and device; all you need is a client and the Internet.
3. *Resource pooling.* The cloud provider's computing resources are pooled. In this way they can efficiently serve multiple consumers. The provider can dynamically assign physical and virtual resources to consumers, according to their instantaneous demands.
4. *Location independence.* The location independence provided by ubiquitous network access is generally a good thing. It does, however, have one potential drawback. The consumer generally has less control over, or knowledge of, the location of the provided resources than in a traditional implementation. This can have drawbacks for data latency. The consumer may be able to ameliorate this drawback by specifying abstract location information (e.g., country, state, or data center).
5. *Rapid elasticity.* Due to resource pooling, it is easy for capabilities to be rapidly and elastically provisioned, in some cases automatically, to quickly scale out or in. To the consumer, the capabilities available for provisioning often appear to be virtually unlimited.
6. *Measured service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability for the chosen service (e.g., storage, processing, bandwidth, and user accounts). Resource usage can be monitored, controlled, and reported so that consumers of the services are billed only for what they use.
7. *Multi-tenancy.* Multi-tenancy is the use of a single application that is responsible for supporting distinct classes or users. Each class or user has its own set of data and access rights, and different users or classes of users are kept distinct by the application.

26.2. Service Models and Deployment Options

In this section we discuss more terminology and basic concepts. First we discuss the most important models for a consumer using the cloud.

Cloud Service Models

Software as a Service (SaaS)

The consumer in this case is an end user. The consumer uses applications that happen to be running on a cloud. The applications can be as varied as email, calendars, video streaming, and real-time collaboration. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

Platform as a Service (PaaS)

The consumer in this case is a developer or system administrator. The platform provides a variety of services that the consumer may choose to use. These services can include various database options, load-balancing options, availability options, and development environments. The

consumer deploys applications onto the cloud infrastructure using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations. Some levels of quality attributes (e.g., uptime, response time, security, fault correction time) may be specified by service-level agreements (SLAs).

Infrastructure as a Service (IaaS)

The consumer in this case is a developer or system administrator. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer can, for example, choose to create an instance of a virtual computer and provision it with some specific version of Linux. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls). Again, SLAs are often used to specify key quality attributes.

Deployment Models

The various deployment models for the cloud are differentiated by who owns and operates the cloud. It is possible that a cloud is owned by one party and operated by a different party, but we will ignore that distinction and assume that the owner of the cloud also operates the cloud.

There are two basic models and then two additional variants of these. The two basic models are private cloud and public cloud:

- *Private cloud*. The cloud infrastructure is owned solely by a single organization and operated solely for applications owned by that organization. The primary purpose of the organization is not the selling of cloud services.
- *Public cloud*. The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

The two variants are community cloud and hybrid cloud:

- *Community cloud*. The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations).
- *Hybrid cloud*. The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities. The consumer will deploy applications onto some combination of the constituent cloud. An example is an organization that utilizes a private cloud except for periods when spikes in load lead to servicing some requests from a public cloud. Such a technique is called "cloud bursting."

26.3. Economic Justification

In this section we discuss three economic distinctions between (cloud) data centers based on their size and the technology that they use:

1. [Economies of scale](#)
2. [Utilization of equipment](#)
3. [Multi-tenancy](#)

The aggregated savings of the three items we discuss may be as large as 80 percent for a 100,000-server data center compared to a 10,000-server data center. Economic considerations have made almost all startups deploy into the cloud. Many larger enterprises deploy a portion of their applications into the cloud, and almost every enterprise with substantial computation needs at least considers the cloud as a deployment platform.

Economies of Scale

Large data centers are inherently less expensive to operate per unit measure, such as cost per gigabyte, than smaller data centers. Large data centers may have hundreds of thousands of

servers. Smaller data centers have servers numbered in the thousands or maybe even the hundreds. The cost of maintaining a data center depends on four factors:

1. *Cost of power.* The cost of electricity to operate a data center currently is 15 to 20 percent of the total cost of operation. The per-server power usage tends to be significantly lower in large data centers than in smaller ones because of the ability to share items such as racks and switches. In addition, large power users can negotiate significant discounts (as much as 50 percent) compared to the retail rates that operators of small data centers must pay. Some areas of the United States provide power at significantly lower rates than the national average, and large data centers can be located in those areas. Finally, organizations such as Google are buying or building innovative and cheaper power sources, such as on- and offshore wind farms and rooftop solar energy.
2. *Infrastructure labor costs.* Large data centers can afford to automate many of the repetitive management tasks that are performed manually in smaller data centers. In a traditional data center, an administrator can service approximately 140 servers, whereas in a cloud data center, the same administrator can service thousands of servers.
3. *Security and reliability.* Maintaining a given level of security, redundancy, and disaster recovery essentially requires a fixed level of investment. Larger data centers can amortize that investment over their larger number of servers and, consequently, the cost per server will be lower.
4. *Hardware costs.* Operators of large data centers can get discounts on hardware purchases of up to 30 percent over smaller buyers.

These economies of scale depend only on the size of the data center and do not depend on the deployment model being used. Operators of public clouds have priced their offerings so that many of the cost savings are passed on to their consumers.

Utilization of Equipment

Common practice in nonvirtualized data centers is to run one application per server. This is caused by the dependency of many enterprise applications on particular operating systems or even particular versions of these operating systems. One result of the restriction of one application per server is extremely low utilization of the servers. Figures of 10 to 15 percent utilization for servers are quoted by several different vendors.

Use of virtualization technology, described in [Section 26.4](#), allows for easy co-location of distinct applications and their associated operating systems on the same server hardware. The effect of this co-location is to increase the utilization of servers. Furthermore, variations in workload can be managed to further increase the utilization. We look at five different sources of variation and discuss how they might affect the utilization of servers:

1. *Random access.* End users may access applications randomly. For example, the checking of email is for some people continuous and for others time-boxed into a particular time period. The more users that can be supported on a single server, the more likely that the randomness of their accesses will end up imposing a uniform load on the server.
2. *Time of day.* Those services that are workplace related, unsurprisingly, tend to be more heavily used during the work day. Those that are consumer related tend to be heavily used during evening hours. Co-locating different services with different time-of-day usage patterns will increase the overall utilization of a server. Furthermore, time differences among geographically distinct locations will also affect utilization patterns and can be considered when planning deployment schedules.
3. *Time of year.* Some applications respond to dates as well as time of day. Consumer sites will see increases during the Christmas shopping season, and floral sites will see increases around Valentine's Day and Mother's Day. Tax preparation software will see increases around the tax return submission due date. Again, these variations in utilization are predictable and can be considered when planning deployment schedules.
4. *Resource usage patterns.* Not all applications use resources in the same fashion. Search, for example, is heavier in its usage of CPU than email but lighter in its use of storage.

Co-locating applications with complementary resource usage patterns will increase the overall utilization of resources.

5. *Uncertainty*. Organizations must maintain sufficient capacity to support spikes in usage. Such spikes can be caused by news events if your site is a news provider, by marketing events if your site is consumer-facing, or even sporting events because viewers of sporting events may turn to their computers during breaks in the action. Startups can face surges in demand if their product catches on more quickly than they can build capacity.

The first four sources of variation are supported by virtualization without reference to the cloud or the cloud deployment model. The last source of variation (uncertainty) depends on having a deployment model that can accommodate spikes in demand. This is the rationale behind cloud bursting, or keeping applications in a private data center and offloading spikes in demand to the public cloud. Presumably, a public cloud provider can deploy sufficient capacity to accommodate any single organization's spikes in demand.

Multi-tenancy

Multi-tenancy applications such as Salesforce.com or Microsoft Office 365 are architected explicitly to have a single application that supports distinct sets of users. The economic benefit of multi-tenancy is based on the reduction in costs for application update and management. Consider what is involved in updating an application for which each user has an individual copy on their own desktop. New versions must be tested by the IT department and then pushed to the individual desktops. Different users may be updated at different times because of disconnected operation, user resistance to updates, or scheduling difficulties. Incidents result because the new version may have some incompatibilities with older versions, the new version may have a different user interface, or users with old versions are unable to share information with users of the new version.

With a multi-tenant application, all of these problems are pushed from IT to the vendor, and some of them even disappear. Any update is available at the same instant to all of the users, so there are no problems with sharing. Any user interface changes are referred to the vendor's hotline rather than the IT hotline, and the vendor is responsible for avoiding incompatibilities for older versions.

The problems of upgrading do not disappear, but they are amortized over all of the users of the application rather than being absorbed by the IT department of every organization that uses the application. This amortization over more users results in a net reduction in the costs associated with installing an upgraded version of an application.

26.4. Base Mechanisms

In this section we discuss the base mechanisms that clouds use to provide their low-level services. In an IaaS instance, the cloud provides to the consumer a virtual machine loaded with a machine image. Virtualization is not a new concept; it has been around since the 1960s. But today virtualization is economically enticing. Modern hardware is designed to support virtualization, and the overhead it adds has been measured to be just 1 percent per instance running on the bare hardware.

We will discuss the architecture of an IaaS platform in [Section 26.5](#). In this section, we describe the concepts behind a virtual machine: the hypervisor and how it manages virtual machines, a storage system, and the network.

Hypervisor

A hypervisor is the operating system used to create and manage virtual machines. Because each virtual machine has its own operating system, a consumer application is actually managed by two layers of operating system: the hypervisor and the virtual machine operating system. The hypervisor manages the virtual machine operating system and the virtual machine operating system manages the consumer application. The key services used by the hypervisor to support the virtual machines it manages are a virtual page mapper and a scheduler. A hypervisor, of course, provides additional services and has a much richer structure than we present here, but these key services are the two that we will discuss.

Page Mapper

We begin by describing how virtual memory works on a bare (nonvirtualized) machine. All modern servers utilize virtual memory. Virtual memory allows an application to assume it has a large amount of memory in which to execute. The assumed memory is mapped into a much smaller physical memory through the use of page tables. The consumer application is divided into pages that are either in physical memory or temporarily residing on a disk. The page table contains the mapping of logical address (consumer application address) to physical address (actual machine address) or disk location. [Figure 26.1](#) shows the consumer application executing its next instruction. This causes the CPU to generate a target address from which to fetch the next instruction or data item. The target address is used to address into a page table. The page table provides a physical address within the computer where the actual instruction or data item can be found if it is currently in main memory. If the physical address is not currently resident in the main memory of the computer, an interrupt is generated that causes a page that contains the target address to be loaded. This is the mechanism that allows a large (virtual) address space to be supported on much smaller physical memory.

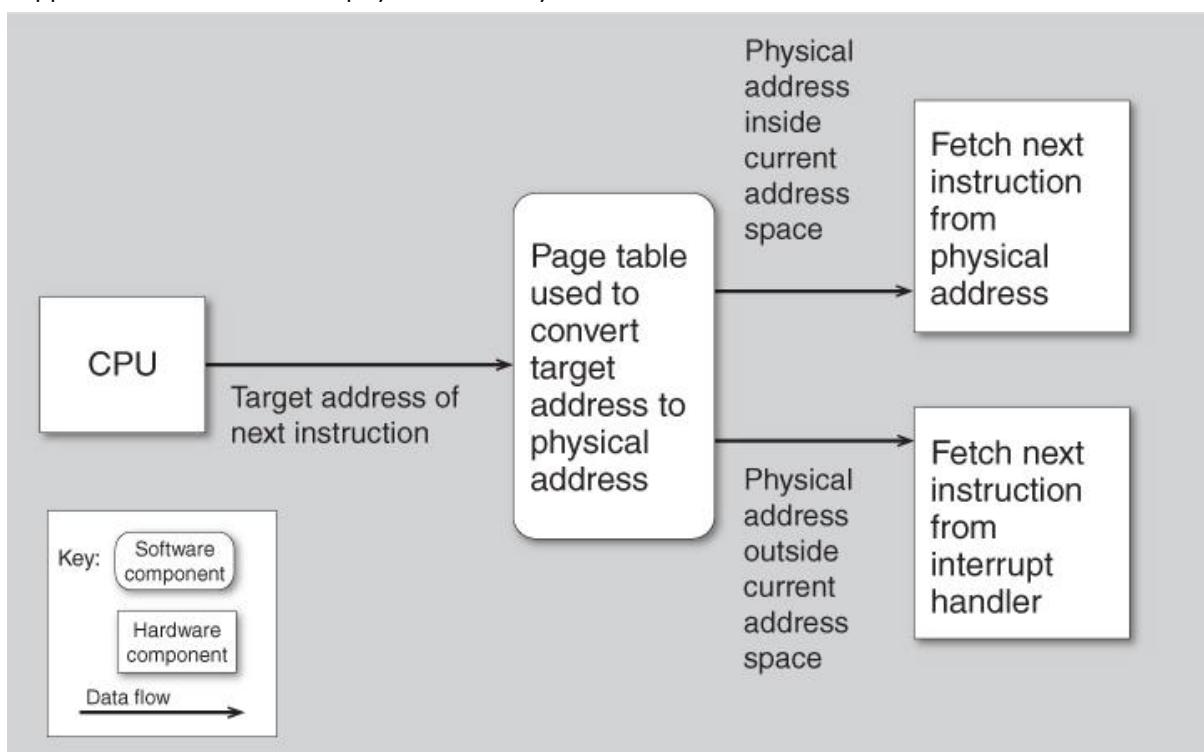


Figure 26.1. Virtual memory page table

Turning the virtual memory mechanism into a virtualization mechanism involves adding another level of indirection. [Figure 26.2](#) shows a logical sequence that maps from the consumer application to a physical machine address. Modern processors contain many optimizations to make this process more efficient. A consumer application generates the next instruction with its target address. This target address is within the virtual machine in which the consumer application is executing. The virtual machine page table maps this target address to an address within the virtual machine based on the target address as before (or indicates that the page is not currently in memory). The address within the virtual machine is converted to a physical address by use of a page table within the hypervisor that manages the current virtual machines.

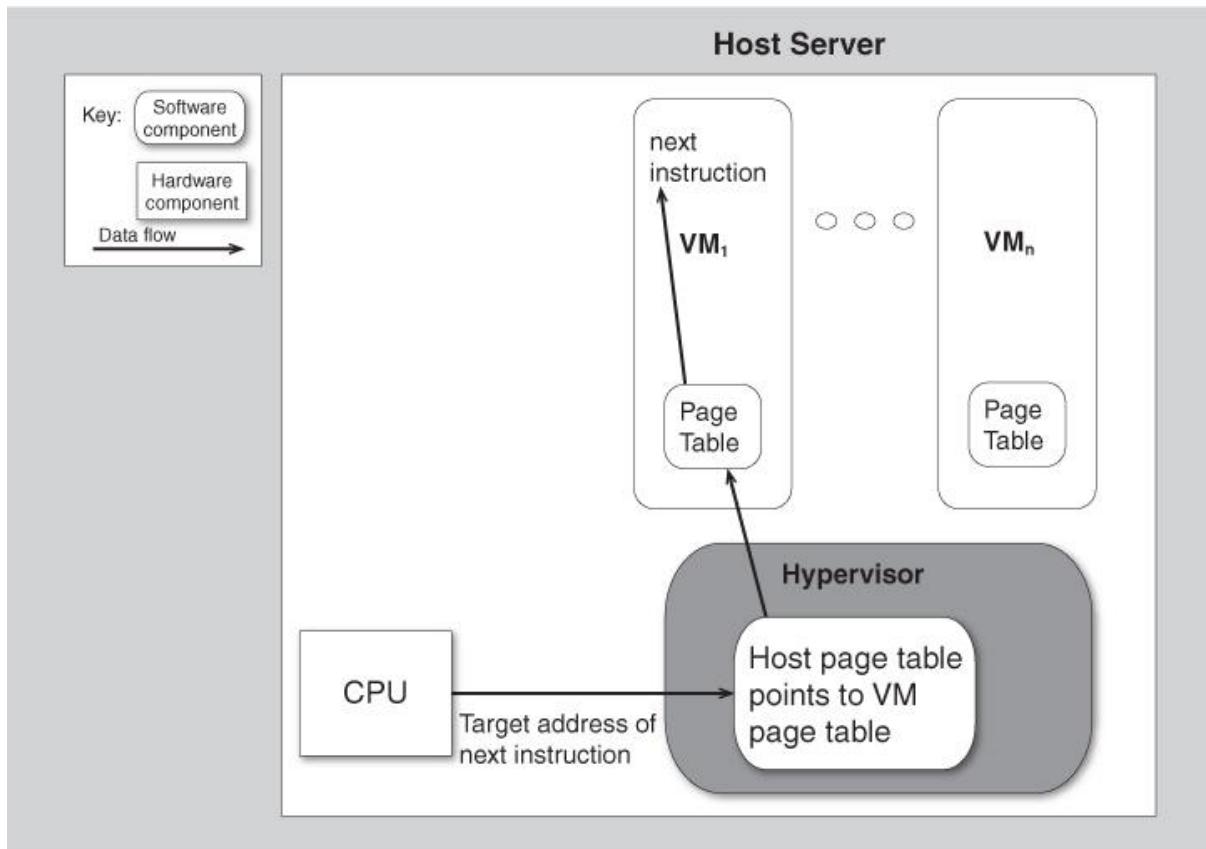


Figure 26.2. Adding a second level of indirection to determine which virtual machine the address references

Scheduler

The hypervisor scheduler operates like any operating system scheduler. Whenever the hypervisor gets control, it decides on the virtual machine to which it will pass control. A simple round-robin scheduling algorithm assigns the processor to each virtual machine in turn, but many other possible scheduling algorithms exist. Choosing the correct scheduling algorithm requires you to make assumptions about the demand characteristics of the different virtual machines hosted within a single server. One area of research is the application of real-time scheduling algorithms to hypervisors. Real-time schedulers would be appropriate for the use of virtualization within embedded systems, but not necessarily within the cloud.

Storage

A virtual machine has access to a storage system for persistent data. The storage system is managed across multiple physical servers and, potentially, across clusters of servers. In this section we describe one such storage system: the Hadoop Distributed File System (HDFS).

We describe the redundancy mechanism used in HDFS as an example of the types of mechanisms used in cloud virtual file systems. HDFS is engineered for scalability, high performance, and high availability.

A component-and-connector view of HDFS within a cluster is shown in [Figure 26.3](#). There is one NameNode process for the whole cluster, multiple DataNodes, and potentially multiple client applications. To explain the function of HDFS, we trace through a use case. We describe the successful use case for “write.” HDFS also has facilities to handle failure, but we do not describe these. See the “[For Further Reading](#)” section for a reference to the HDFS failure-handling mechanisms.

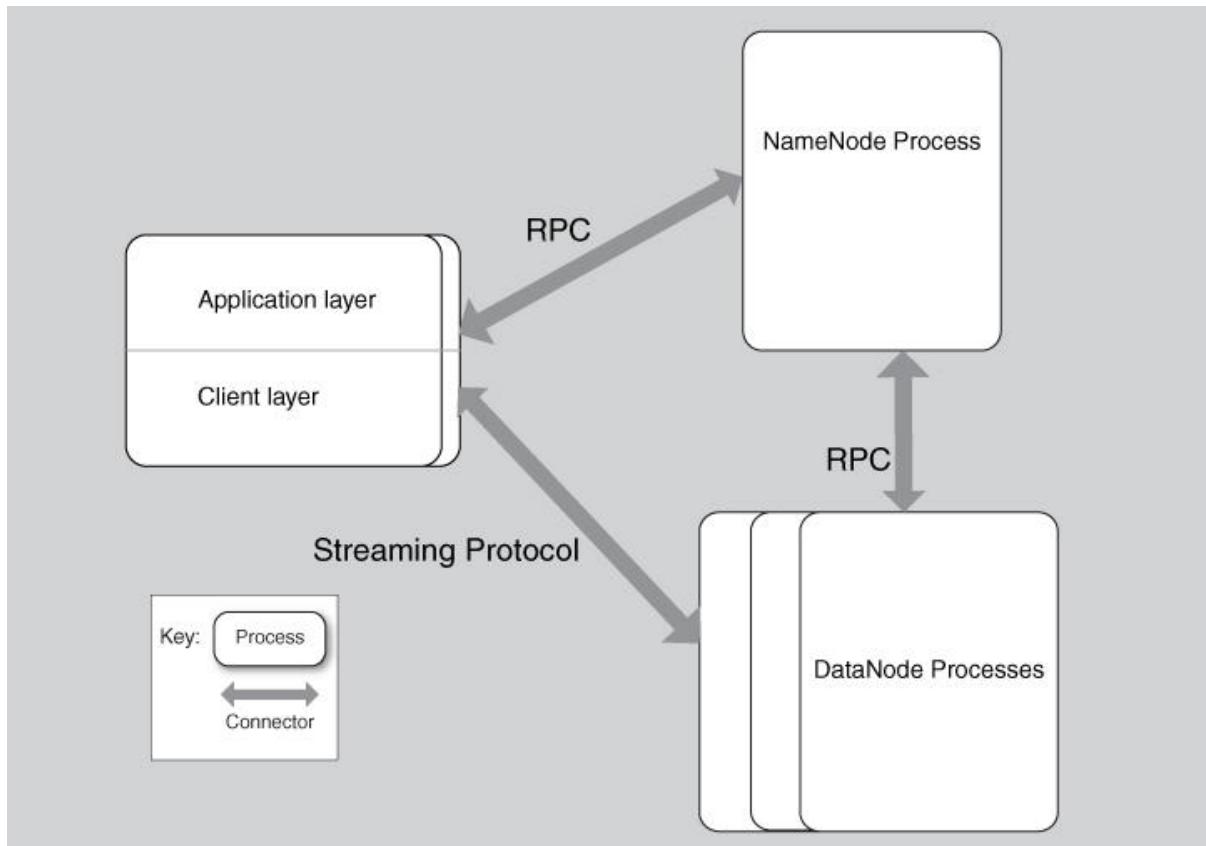


Figure 26.3. A component-and-connector view of an HDFS deployment. Each process exists on a distinct computer.

For the “write” use case, we will assume that the file has already been opened. HDFS does not use locking to allow for simultaneous writing by different processes. Instead, it assumes a single writer that writes until the file is complete, after which multiple readers can read the file simultaneously. The application process has two portions: the application code and a client library specific to HDFS. The application code can write to the client using a standard (but overloaded) Java I/O call. The client buffers the information until a block of 64 MB has been collected. Two of the techniques used by HDFS for enhancing performance are the avoidance of locks and the use of 64-MB blocks as the only block size supported. No substructure of the blocks is supported by HDFS. The blocks are undifferentiated byte strings. Any substructure and typing of the information is managed solely by the application. This is one example of a phenomenon that we will notice in portions of the cloud: moving application-specific functionality *up* the stack as opposed to moving it down the stack to the infrastructure.

For reliability purposes each block is replicated a parameterizable number of times, with a default of three. For each block to be written, the NameNode allocates DataNodes to write the replicas. The DataNodes are chosen based on two criteria: (1) their location—replicas are spread across racks to protect against the possibility that a rack fails; and (2) the dynamic load on the DataNode. Lightly loaded DataNodes are given preference over heavily loaded DataNodes to reduce the possibility of contention for the DataNodes among different files being simultaneously accessed.

Once the client has collected a buffer of 64 MB, it asks the NameNode for the identities of the DataNodes that will contain the actual replicas. The NameNode manages only metadata; it is not involved in the actual transfer or recording of data. These DataNode identities are sent from the NameNode to the client, which then treats them as a pipeline. At this point the client streams the block to the first DataNode in the pipeline. The first DataNode then streams the data to the second DataNode in the pipeline, and so forth until the pipeline (of three DataNodes, unless the client has specified a different replication value) is completed. Each DataNode reports back to the client when it has successfully written the block, and also reports to the NameNode that it has successfully written the block.

Network

In this section we describe the basic concepts behind Internet Protocol (IP) addressing and how a message arrives at your computer. In [Section 26.5](#) we discuss how an IaaS system manages IP addresses.

An IP address is assigned to every “device” on a network whether this device is a computer, a printer, or a virtual machine. The IP address is used both to identify the device and provide instructions on how to find it with a message. An IPv4 address is a constrained 32-bit number that is, typically, represented as four groups for human readability. For example, 192.0.2.235 is a valid IP address. The familiar names that we use for URLs, such as “<http://www.pearsonhighered.com/>”, go through a translation process, typically through a domain name server (DNS), that results in a numeric IP address. A message destined for that IP address goes through a routing process to arrive at the appropriate location.

Every IP message consists of a header plus a payload. The header contains the source IP address and the destination IP address. IPv6 replaces the 32-bit number with a 128-bit number, but the header of an IP message still includes the source and destination IP addresses.

It is possible to replace the header of an IP message for various reasons. One reason is that an organization uses a gateway to manage traffic between external computers and computers within the organization. An IP address is either “public,” meaning that it is unique within the Internet, or “private,” meaning that multiple copies of the IP address are used, with each copy owned by a different organization. Private IP addresses must be accessed through a gateway into the organization that owns it. For outgoing messages, the gateway records the address of the internal machine and its target and replaces the source address in the TCP header with its own public IP address. On receipt of a return message, the gateway would determine the internal address for the message and overwrite the destination address in the header and then send the message onto the internal network. Network address translation (NAT) is the name of this process of translation.

26.5. Sample Technologies

Building on the base mechanisms, we now discuss some of the technologies that exist in the cloud. We begin by discussing the design of a generic IaaS platform, then we move up the stack to a PaaS, and finally we discuss database technology in the cloud.

Infrastructure as a Service

Fundamentally, an IaaS installation provides three services: virtualized computation, virtualized networking, and a virtualized file system. In the previous section on base mechanisms, we described how the operating system for an individual server manages memory to isolate each virtual machine and how TCP/IP messages could be manipulated. An IaaS provides a management structure around these base concepts. That is, virtual machines must be allocated and deallocated, messages must be routed to the correct instance, and persistence of storage must be ensured.

We now discuss the architecture of a generic IaaS platform. Various providers will offer somewhat different services within different architectures. Open-Stack is an open source movement to standardize IaaS services and interfaces, but as of this writing, it is still immature.

[Figure 26.4](#) shows an allocation view of a generic cloud platform. Each server shown provides a different function to the platform, as we discuss next.

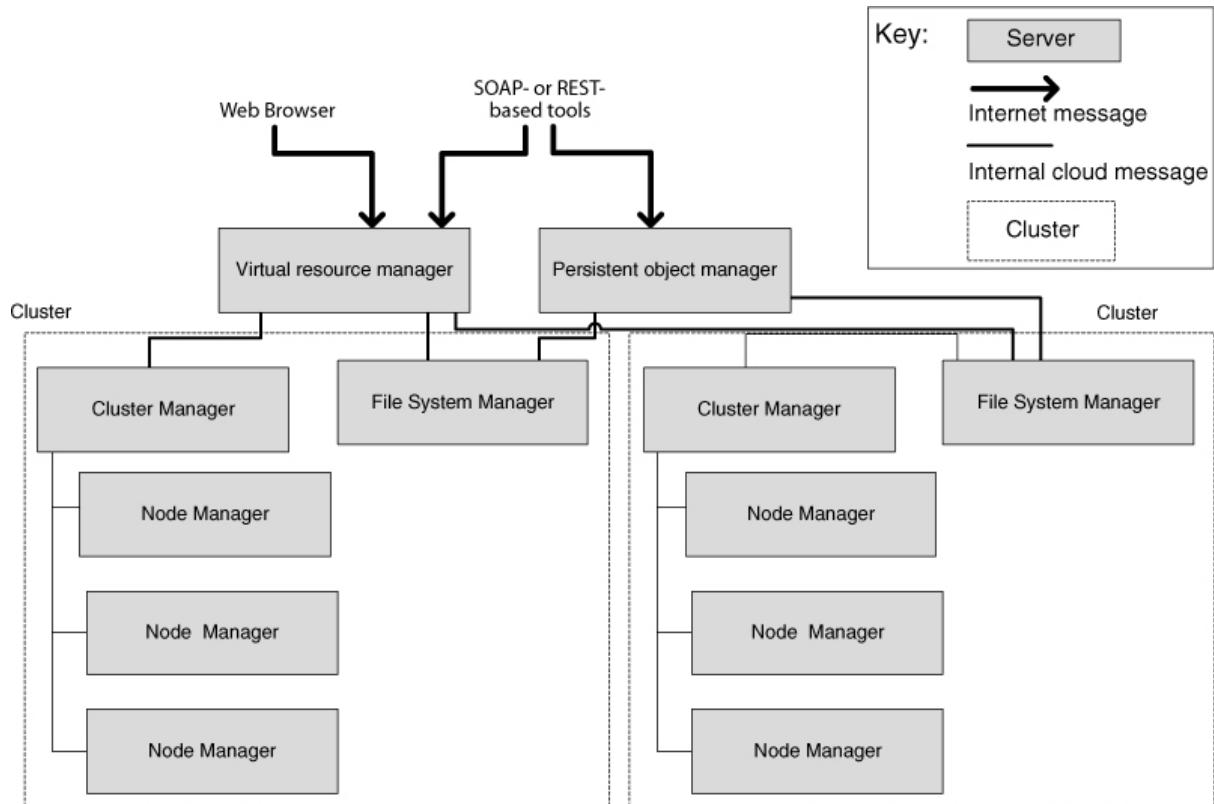


Figure 26.4. A generic cloud allocation view

An IaaS installation has a variety of clusters. Each cluster may have thousands of physical servers. Each cluster has a *cluster manager* responsible for that cluster's resources. The *persistent object manager* supports the manipulation of persistent objects, and the *virtual resource managers* are in charge of the other virtualized resources. For requests for new resources, the virtual resource manager is in charge of determining which cluster manager will service the request. For requests sent to existing resources, the virtual resource manager is responsible for seeing that the requests get forwarded to the correct server. The virtual resource manager, in this case, acts as a gateway, as described in [Section 26.4](#).

Some of the services that IaaS providers offer to support applications are these:

- *Automatic reallocation of IP addresses in the case of a failure of the underlying virtual machine instance.* This service is useful in case the instance has a public IP address. Unless the provider offers this service, the client must register the IP address of a replacement instance with a domain name server to ensure that messages are sent to the correct location.
- *Automatic scaling.* One of the virtues of the cloud is that new instances can be created or deleted relatively quickly in the event of a variation in demand. Detecting the variation in demand, allocating (or deleting) an instance in the event of a variation, and ensuring that the remaining instances are allocated their fair share of messages is another service that could be provided by the IaaS.

The persistent object manager is responsible for maintaining files that are intended to persist past the deletion of a virtual machine instance. It may maintain these files across multiple clusters in a variety of different geographic locations.

Failure of the underlying hardware is a common occurrence in a large data center, consequently the virtual resource manager has mechanisms to manage requests in the event of failure. These mechanisms are typically designed to maintain the availability of the IaaS infrastructure and do not extend to the applications deployed with the virtual machines. What this means in practice is that if you make a request for a new resource, it will be honored. If you make a request to an existing virtual machine instance, the infrastructure will guarantee that, if your virtual machine instance is active, your request is delivered. If, however, the host on which your

virtual machine instance has been allocated has failed, then your virtual machine instance is no longer active and it is your responsibility as an application architect to install mechanisms to recognize a failure of your virtual machine instances and recover from them.

The *file system manager* manages the file system for each cluster. It is similar to the Hadoop Distributed File System that we discussed in [Section 26.4](#). It also assumes that failure is a common occurrence and has mechanisms to replicate the blocks and to manage handoffs in the event of failures.

The cluster manager controls the execution of virtual machines running on the nodes within its clusters and manages the virtual networking between virtual machines and between virtual machines and external users.

The final piece of [Figure 26.4](#) is the *node manager*; it (through the functionality of a hypervisor) controls virtual machine activities, including the execution, inspection, and termination of virtual machine instances.

A client initially requests a virtual machine instance and the virtual resource manager decides on which cluster the virtual machine instance should reside. It passes the instance request to the cluster manager, which in turn decides which node should host the virtual machine instance.

Subsequent requests are routed through the pieces of the generic infrastructure to the correct instance. The instance can create files using the file system manager. These files will either be deleted when the virtual machine instance is finished or will be persisted past the existence of the virtual machine instance. The choice is the client's as to how long storage is persisted. If the storage is persisted, it can be accessed independently of the creating instance through the persistence manager.

Platform as a Service

A Platform as a Service provides a developer with an integrated stack within the cloud to develop and deploy applications. IaaS provides virtual machines, and it is the responsibility of the developer using IaaS to provision the virtual machines with the software they desire. PaaS is preprovisioned with a collection of integrated software.

Consider a virtual machine provisioned with the LAMP (Linux, Apache, MySQL, PHP/Perl/Python) stack. The developer writes code in Python, for example, and has available the services provided by the other elements of the stack. Take this example and add automatic scaling across virtual machines based on customer load, automatic failure detection and recovery, backup/restore, security, operating system patch installation, and built-in persistence mechanisms. This yields a simple example of a PaaS.

The vendors offering PaaS and the substance of their offerings are rapidly evolving. Google and Microsoft are two of the current vendors.

1. The Google App Engine provides the developer with a development environment for Python or Java. Google manages deploying and executing developed code. Google provides a database service that is automatically replicated across data centers.
2. Microsoft Azure provides an operating system and development platform to access/develop applications on Microsoft data centers. Azure provides a development environment for applications running on Windows using .NET. It also provides for the automatic scaling and replication of instances. For example, if an application instance fails, then the Azure infrastructure will detect the failure and deploy another instance automatically. Azure also has a database facility that automatically keeps replicas of your databases.

Databases

A number of different forces have converged in the past decade, resulting in the creation of database systems that are substantially different from the relational database management systems (RDBMSs) that were prevalent during the 1980s and '90s.

- Massive amounts of data began to be collected from web systems. A search engine must index billions of pages. Facebook, today, has over 800 million users. Much of this data is processed sequentially and, consequently, the sophisticated indexing and query optimizations of RDBMSs are not necessary.

- Large databases are continually being created during various types of processing of web data. The creation and maintenance of databases using a traditional RDBMS requires a sophisticated data administrator.
- A theoretical result (the so-called CAP theorem) shows that it is not possible to simultaneously achieve consistency, availability, and partitioning. One of these properties must be sacrificed. For many applications, the choice is to sacrifice consistency and provide immediate availability and “eventual consistency.” What this means, in practice, is that occasionally a user will access stale data, but updates will be subsequently available. The alternative approach, taken by RDBMSs, is to lock values and not allow access until they become consistent.
- The relational model is not the best model for some applications. The relational model assumes there is one data item for each row-value/column-name pair. One method for handling web searches, for example, is to store different versions of a single web page indexed by the same row-value/column-name pair so that the different versions of the web page can be quickly accessed and differences easily determined. Using the relational model requires that the system perform joins to retrieve all of the attributes associated with a particular row value. Joins are expensive from a performance perspective, and consequently, newly emerging database systems tend to not support joins and require storing data in a denormalized form.

These forces resulted in the creation of new types of databases with different data models and different access mechanisms. These new types of databases go under the name of NoSQL—although as Michael Stonebraker has pointed out, the existence or nonexistence of SQL within the database system is irrelevant to the rationale for their existence.

We discuss two open source NoSQL database systems: a key-value one (HBase) and a document-centric one (MongoDB).

HBase

HBase is a key-value database system based on the BigTable database system developed by Google. Google uses BigTable to store data for many of their applications. The number of data items in a HBase database can be in the billions or trillions.

HBase supports tables, although there is no schema used. One column is designated as the key. The other columns are treated as field names. A data value is indexed by a row value, a column name, and a time stamp. Each row value/column name can contain multiple versions of the same data differentiated by time stamps.

One use of HBase is for web crawling. In this application, the row value is the URL for the web page. Each column name refers to an attribute of a web page that will support the analysis of the web page. For example, “contents” might be one column name. In the relational model, each row value/column name would retrieve the contents of the web page. Web pages change over time, however, and so in the relational model, there would need to be a separate column with the time stamp, and the primary key for the table would be the URL/time stamp. In HBase, the versions of the web page are stored together and retrieved by the URL value/“contents”. All of the versions of the web page are retrieved, and it is the responsibility of the application to separate the versions of the web page and determine which one is desired based on the time stamp.

MongoDB

MongoDB uses a document-centric data model. You can think of it as storing objects rather than tables. An object contains all of the information associated with a particular concept without regard to whether relations among data items are stored in multiple different objects. Two distinct objects may have no field names in common, some field names in common, or all of the field names in common.

You may store links rather than data items. Links support the concept of joining different objects without requiring the maintenance of indices and query optimization. It is the responsibility of the application to follow the link.

Documents are stored in binary JavaScript Object Notation (JSON) form. Indices can be created on fields and can be used for retrieval, but there is no concept of primary versus secondary keys. A field is either indexed or it is not. Because the same field can occur in multiple different documents, a field is indexed wherever it occurs.

What Is Left Out of NoSQL Databases

One motivation for NoSQL databases is performance when accessing millions or billions of data items. To this end, several standard RDBMS facilities are omitted in NoSQL databases. If an application wishes to have these features, it must implement them itself. Mainly, the features are omitted for performance reasons.

- *Schemas.* NoSQL databases typically do not require schemas for their data model and, consequently, there is no checking of field names for consistency.
- *Transactions.* NoSQL typically does not support transactions. Transactions lock data items, which hinders performance. Applications use techniques such as time stamps to determine whether fields have been modified through simultaneous access.
- *Consistency.* NoSQL databases are “eventually consistent.” This means that after some time has passed, different replicas of a data item will have the same value, but in the interim, it is possible to run two successive queries that access the same data item and retrieve two different values.
- *Normalization.* NoSQL databases do not support joins. Joins are a requirement if you are to normalize your database.

26.6. Architecting in a Cloud Environment

Now we take the point of view of an architect who is designing a system to execute in the cloud. In some ways, the cloud is a platform, and architecting a system to execute in the cloud, especially using IaaS, is no different than architecting for any other distributed platform. That is, the architect needs to pay attention to usability, modifiability, interoperability, and testability, just as he or she would for any other platform. The quality attributes that have some significant differences are security, performance, and availability.

Security

Security, as always, has both technical and nontechnical aspects. The nontechnical aspects of security are items such as what trust is placed in the cloud provider, what physical security does the cloud provider utilize, how are employees of the cloud provider screened, and so forth. We will focus on the technical aspects of security.

Applications in the cloud are accessed over the Internet using standard Internet protocols. The security and privacy issues deriving from the use of the Internet are substantial but no different from the security issues faced by applications not hosted in the cloud. The one significant security element introduced by the cloud is multi-tenancy. Multi-tenancy means that your application is utilizing a virtual machine on a physical computer that is hosting multiple virtual machines. If one of the other tenants on your machine is malicious, what damage can they do to you?

There are four possible forms of attack utilizing multi-tenancy:

1. *Inadvertent information sharing.* Each tenant is given a set of virtual resources. Each virtual resource is mapped to some physical resource. It is possible that information remaining on a physical resource from one tenant may “leak” to another tenant.
2. *A virtual machine “escape.”* A virtual machine is isolated from other virtual machines through the use of a distinct address space. It is possible, however, that an attacker can exploit software errors in the hypervisor to access information they are not entitled to. Thus far, such attacks are extremely rare.
3. *Side-channel attacks.* It is possible for a malicious attacker to deduce information about keys and other sensitive information by monitoring the timing activity of the cache. Again, so far, this is primarily an academic exercise.
4. *Denial-of-service attacks.* Other tenants may use sufficient resources on the host computer so that your application is not able to provide service.

Some providers allow customers to reserve entire machines for their exclusive use. Although this defeats some of the economic benefits of using the cloud, it is a mechanism to prevent multi-tenancy attacks. An organization should consider possible attacks when deciding which applications to host in the cloud, just as they should when considering any hosting option.

Performance

The instantaneous computational capacity of any virtual machine will vary depending on what else is executing on that machine. Any application will need to monitor itself to determine what resources it is receiving versus what it will need.

One virtue of the cloud is that it provides an elastic host. Elasticity means that additional resources can be acquired as needed. An additional virtual machine, for example, will provide additional computational capacity. Some cloud providers will automatically allocate additional resources as needed, whereas other providers view requesting additional resources as the customer's responsibility.

Regardless of whether the provider automatically allocates additional resources, the application should be self-aware of both its current resource usage and its projected resource usage. The best the provider can do is to use general algorithms to determine whether there is a need to allocate or free resources. An application should have a better model of its own behavior and be better equipped to do its own allocation or freeing of resources. In the worst case, the application can compare its predictions to those of the provider to gain insight into what will happen. It takes time for the additional resources to be allocated and freed. The freeing of resources may not be instantaneously reflected in the charging algorithm used by the provider, and that charging algorithm also needs to be considered when allocating or freeing resources.

Availability

The cloud is assumed to be always available. But everything can fail. A virtual machine, for example, is hosted on a physical machine that can fail. The virtual network is less likely to fail, but it too is fallible. It behooves the architect of a system to plan for failure.

The service-level agreement that Amazon provides for its EC2 cloud service provides a 99.95 percent guarantee of service. There are two ways of looking at that number: (1) That is a high number. You as an architect do not need to worry about failure. (2) That number indicates that the service may be unavailable for .05 percent of the time. You as an architect need to plan for that .05 percent.

Netflix is a company that streams videos to home television sets, and its reliability is an important business asset. Netflix also hosts much of its operation on Amazon EC2. On April 21, 2011, Amazon EC2 suffered a four-day sporadic outage. Netflix customers, however, were unaware of any problem.

Some of the things that Netflix did to promote availability that served them well during that period were reported in their tech blog. We discussed their Simian Army in [Chapter 10](#). Some of the other things they did were applications of availability tactics that we discussed in [Chapter 5](#).

- *Stateless services.* Netflix services are designed such that any service instance can serve any request in a timely fashion, so if a server fails, requests can be routed to another service instance. This is an application of the spare tactic, because the other service instance acts as a spare.
- *Data stored across zones.* Amazon provides what they call "availability zones," which are distinct data centers. Netflix ensured that there were multiple redundant hot copies of the data spread across zones. Failures were retried in another zone, or a hot standby was invoked. This is an example of the active redundancy tactic.
- *Graceful degradation.* The general principles for dealing with failure are applications of the degradation or the removal from service tactic:
 - Fail fast: Set aggressive timeouts such that failing components don't make the entire system crawl to a halt.
 - Fallbacks: Each feature is designed to degrade or fall back to a lower quality representation.
 - Feature removal: If a feature is noncritical, then if it is slow it may be removed from any given page.

The CAP Theorem

The CAP theorem—created by Eric Brewer at UC Berkeley—emerged over a decade ago. Unlike most theories postulated by academics, this one did not sink into obscurity but rather has grown in renown and influence since then. The theory states that there are three important properties of a distributed system managing shared data. These are the following:

- Consistency (C): the data will be consistent throughout the distributed system.
- Availability (A): the data will be highly available.
- Partitioning (P): the system will tolerate network partitioning.

And the theory further states that no system can achieve all of these properties simultaneously; the best we can hope for is to satisfy two out of three while sacrificing (to some extent) the third property. Brewer explains it thus:

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P.

In fact, there is really another important facet to the CAP theorem that has come to dominate the engineering challenge: latency. It wasn't part of the original acronym (although CLAP is certainly catchy), but a concern for latency now infuses much of the discussion of the tradeoffs in implementing NoSQL databases.

Creators of large-scale distributed NoSQL databases are constantly faced with tradeoffs. These days no one believes that you simply choose two of the three properties of CAP; the decisions are far richer and more subtle than that. For example, designers of these systems like to speak of "eventual consistency"—partitions are allowed to become inconsistent on a regular basis, but with bounds that are carefully engineered and monitored. They might want to specify that no more than x percent of the data should be stale at any given time, and it should not take more than y seconds to restore consistency (on average, or in the worst case). Another common tradeoff seen in practice is that availability and latency are typically favored over consistency. That is, a Facebook user should get quick response from the system, even if their newsfeed is slightly stale.

All of this adds complexity to the system. The designer has to choose between faster/less consistent and slower/more consistent (as well as a host of other quality issues). And the mechanisms for achieving eventual consistency—caching, replication, message retries, timeouts, and so forth—are themselves nontrivial. Consistency, partitioning, latency, and availability are four qualities that can be traded off with NoSQL databases. In addition, other quality attributes—interoperability, security, and so forth—also add complexity, and so the tradeoffs involved can get more and more complicated.

Alas, this is, increasingly, the world that we live in. Systems with global reach and enormous bases of distributed data are not going away anytime soon. So as architects we need to be prepared to deal with tradeoffs and complexities for the foreseeable future.

—RK

26.7. Summary

The cloud has become a viable alternative for the hosting of data centers primarily for economic reasons. It provides an elastic set of resources through the use of virtual machines, virtual networks, and virtual file systems.

The cloud can be used to provide infrastructure, platforms, or services. Each of these has its own characteristics.

NoSQL database systems arose in reaction to the overhead introduced by large relational database management systems. NoSQL database systems frequently use a data model based on key-value or documents and do not provide support for common database services such as transactions.

Architecting in the cloud means that the architect should pay attention to specific aspects of quality attributes that are substantially different in cloud environments, namely: performance, availability, and security.

26.8. For Further Reading

Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age [[Hiltzik 00](#)] has a discussion of time-sharing and covers the technologies and the personalities involved in the development of the modern personal workstation.

The economics of the cloud are described in [[Harms 10](#)].

The Computer Measurement Group (CMG) is a not-for-profit worldwide organization that provides measurement and forecasting of various quantitative aspects of computer usage. Their measurements of the overhead due to virtualization can be found at www.cmg.org/measureit/issues/mit39/m_39_1.html.

If you want to learn more about TCP/IP and NAT, you can find a discussion at www.ipcortex.co.uk/wp/fw.rhtm.

The BigTable system is described in [[Chang 06](#)].

Netflix maintains a tech blog that is almost entirely focused on cloud issues. It can be found at techblog.netflix.com.

The home page for MongoDB is www.mongodb.org/display/DOCS/Home and for HBase is hbase.apache.org.

Michael Stonebraker is a database expert who has written extensively comparing NoSQL systems with RDBMSs. Some of his writings are [[Stonebraker 09](#)], [[Stonebraker 10a](#)], [[Stonebraker 11](#)], and [[Stonebraker 10b](#)].

Eric Brewer has provided a nice overview of the issues surrounding the CAP theorem for today's cloud-based systems: [[Brewer 12](#)].

26.9. Discussion Questions

1. "Service-oriented or cloud-based systems cannot meet hard-real-time requirements because it's impossible to guarantee how long a service will take to complete." Do you think this statement is true or false? In either case, identify the one or two categories of design decisions that are most responsible for the correctness (or incorrectness) of the statement.
2. "Using the cloud assumes your application is service oriented." Do you think this is true or false? Find some examples that would support that statement and, if it is not universally true, find some that would falsify it.
3. Netflix discussed their movement from Oracle to SimpleDB on their tech blog. They also discussed moving from SimpleDB to Cassandra. Describe their rationale for these two moves.
4. Netflix also describes their Simian Army in their tech blog. Which elements of the Simian Army could be offered as a SaaS? What would the design of such a SaaS look like?
5. Develop the "Hello World" application on an IaaS and on a PaaS.

27. Architectures for the Edge

With Hong-Mei Chen

Human nature is not a machine to be built after a model, and set to do exactly the work prescribed for it, but a tree, which requires to grow and develop itself on all sides, according to the tendency of the inward forces which make it a living thing.

—John Stuart Mill

In this chapter we discuss the place of architecture in edge-dominant systems and discuss how an architect should approach building such systems. An edge-dominant system is one that depends crucially on the inputs of users for its success.

What would Wikipedia be without the encyclopedic entries contributed by users? What would YouTube be without the contributed videos? What would Facebook and Twitter be without their user communities? YouTube serves up approximately 1 billion videos a day. Twitter boasts that its users tweet 50 million times per day. Facebook reports that it serves up about 30 billion pieces of content each month. Flickr recently announced that users had uploaded more than 6 billion photos. Strong, almost maniacal, user participation has elevated each of these sites from fairly routine repositories to forces that shape society.

In each case, the value of these systems comes almost entirely from its users—who happily contribute their opinions and knowledge, their artistic content, their software, and their innovations—and not from some centralized organization. This phenomenon is a cornerstone of the so-called “Web 2.0” movement. Darcy DiNucci, credited with coining the term, wrote in 1999, “The [new] Web will be understood not as screenfuls of text and graphics but as a transport mechanism, the ether through which interactivity happens.” The “old” web was about going to web pages for static information; the “new” web is about participating in the information creation (“crowdsourcing”) and even becoming part of its organization (“folksonomy”).

Many have written about the social, political, and economic consequences of this change, and some see it as nothing short of a revolution along the lines of the industrial revolution. Yochai Benkler’s book *The Wealth of Networks*—a play on the title of Adam Smith’s classic book *The Wealth of Nations*, which heralded the start of the industrial revolution—argues that the “radical transformation” of how we create our information environment is restructuring society, particularly our models of production and consumption. Benkler calls this new economic model commons-based peer production. And it’s big: crowdsourced websites that are built on this model have become some of the dominant forces on the web and in society in the past few years. Populist revolutions are catalyzed by Twitter and Facebook. As of the time this book went to press, five of the top ten websites by traffic are peer produced: Facebook, YouTube, Blogger, Wikipedia, and Twitter. And the other five are portals or search engines that pore through the content created by billions of worldwide users. Websites that actually sell something from a centralized organization are rare in the world’s top sites; [Amazon.com](#) is the only example, and it’s no accident that it’s the bookseller that derives much of its popularity from the value created by customers.

Along with this paradigm shift, much of the world’s software is now open source. The two most popular web browsers in the world are open source (Mozilla Firefox and Google Chrome). Apache is the most popular web server, currently powering almost two out of every three websites. Open source databases, IDEs, content management systems, and operating systems are all heavy hitters in their respective market spaces.

Why study this in a book about architecture? First, commons-based peer-produced systems are an excellent example of the architecture influence cycle. Second, the architecture of such systems has some important differences from the architectures that you would build for traditional systems. We start by examining how the forces of commons-based peer production change the very nature of the system’s development life cycle.

27.1. The Ecosystem of Edge-Dominant Systems

All successful edge-dominant systems, and the organizations that develop and use these systems, share a common ecosystem structure, as shown in [Figure 27.1](#). This is called a “Metropolis” structure, by analogy with a city.

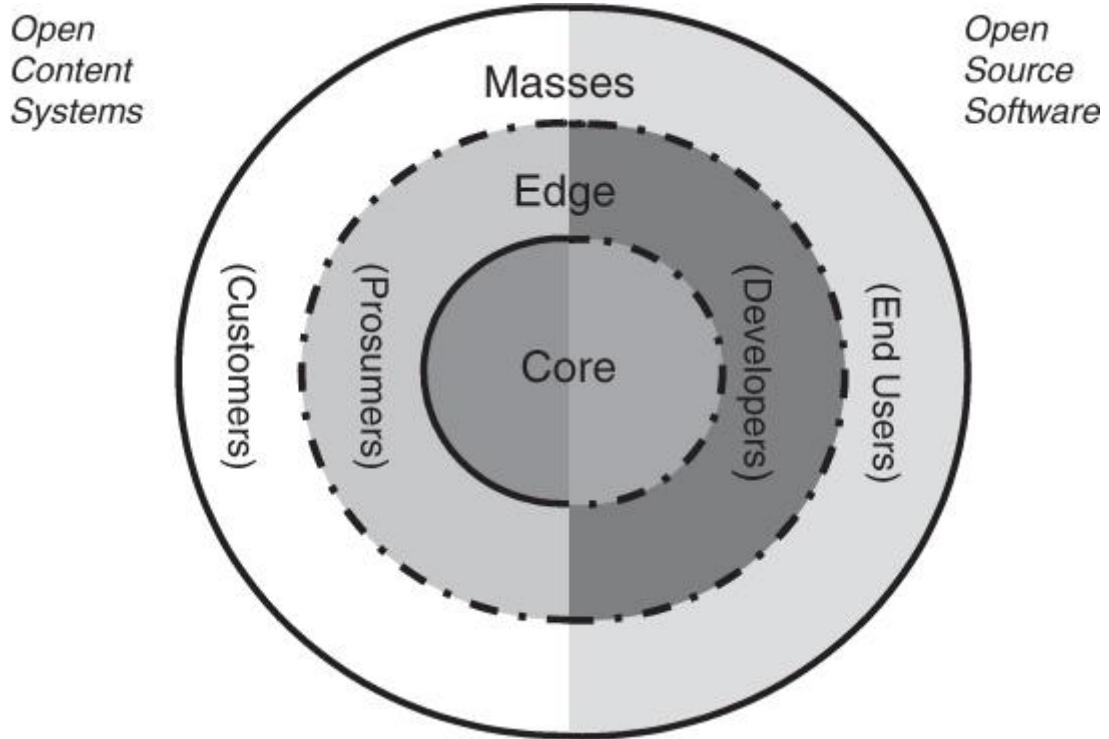


Figure 27.1. The Metropolis structure of an edge-dominant system

The Metropolis structure is not an architecture diagram: it is a representation of three communities of stakeholders:

- *Customers and end users*, who consume the value produced by the Metropolis
- *Developers*, who write software and key content for the Metropolis
- *Prossumers*, who consume content but also produce it

The Metropolis structure also presents three realms of roles and infrastructure:

- In the outermost ring reside the masses of end users of such systems. They contribute requirements but not content.
- The middle ring contains developers and prossumers. These are the stakeholders at the edge whose actions and whose value creation the organization would like to facilitate.
- All of this is held together by the core. The core is software; it provides its services through a set of APIs upon which the periphery can build. Linux’s kernel, Apache’s core, Wikipedia’s wiki, Facebook’s application platform, or the application platforms of the iPhone or Android.

In the Metropolis structure, the realms have different “permeability,” which the figure indicates by the dashed and solid lines.

In open source systems (such as Linux, MySQL, Apache, Eclipse, and Firefox), it is possible to move from the role of an end user to a developer to a core architect, by consistently contributing and moving up in the meritocracy. However, in open *content* systems (such as Wikipedia, Twitter, YouTube, Slashdot, and Facebook), nobody will invite you to become a software developer for the core, no matter how many cool movies or blog entries you contribute.

Thus, a key question for an organization wishing to foster edge-dominant systems: How should we architect the core and what development principles should we embrace for the periphery/edge?

27.2. Changes to the Software Development Life Cycle

All our familiar software development life cycles—waterfall, Agile, iterative, or prototyping-based—are broken in an edge-dominant, crowdsourced world. These models all assume that requirements can be known; software is developed, tested, and released in planned increments; projects have dedicated finite resources; and management can “manage” these resources. *None* of these conditions is true in the Metropolis. Let us consider each aspect in turn:

- *Requirements can be known.* In edge-dominant systems, requirements emerge from its individuals, operating independently. Requirements are never knowable in any global sense and they will inevitably conflict, just as the requirements of a city’s inhabitants often conflict (some want better highways, some want more park land).
- *Software is developed, tested, and released in planned increments.* All existing software development models assume that systems evolve in an orderly way, through planned releases. An edge-dominant system, on the other hand, is constantly changing. It doesn’t make sense, for example, to talk about the “latest release” of Wikipedia. Resources are noncentralized and so such a system is never “stable.” One cannot conceive of its functionality in terms of “releases” any more than a city has a release; parts are being created, modified, and torn down at all times.
- *Projects have dedicated finite resources.* Edge-dominant projects are “staffed” by members who are not employed by the project. Such projects are subject to the whims of the members who are not *required* and cannot be compelled to contribute anything. However, successful projects tend to attract large numbers of contributors. Unlike traditional projects, which have finite resources, typically limited by budgetary constraints, there is no natural limit to the resources available to an edge-dominant project. And these large numbers tend to ameliorate the (unreliable) actions of any individual contributor.
- *Management can “manage” these resources.* In edge-dominant systems, the developers are often volunteers. They participate in decentralized production processes with no identifiable managers. Linus Torvalds, the creator of the Linux operating system, has noted that he has no authority to order anyone to do anything; he can only attempt to lead and persuade, in the hope that others will follow. Teams in this world are often diverse with differing, sometimes irreconcilable, views.

For edge-dominant systems, the old rules and tools of software development won’t work. Such projects are, to varying degrees, community driven and decentralized with little overall control, as is the case with the major social networking communities (e.g., Twitter, Google+, Facebook), and open content systems (e.g., Wikipedia, YouTube), especially coupled with open source software development.

27.3. Implications for Architecture

The Metropolis structure presented in the previous section, while not an architecture, has important implications for architecture. The key architectural choice for an edge-dominant system is the distinction between core and edge. That is, the architecture of *successful* edge-dominant systems is, without fail, bifurcated into

- a core (or kernel) infrastructure and
- a set of peripheral functions or services that are built on the core.

This constitutes an architectural pattern very reminiscent of layering. Linux, Firefox, and Apache—to name just a few—are based upon this architectural pattern. Linux applies this pattern twice: at the outermost level the core is the entire Linux kernel, and individual applications, libraries, resources, and auxiliaries act as extensions to the kernel’s functionality—the periphery. Digging into the Linux kernel, we can once again discern a core/periphery pattern. Inside the Linux kernel, modules are defined to enable parallel development of different subsystems. The different functions that one expects to find in an operating system kernel are all present, but they are designed to be separate modules. For instance, there are modules for processor/cache control, memory management, resource management, file system interfacing, networking stacks, device

I/O, security, and so forth. All of these modules interact, but they are clearly identifiable, separate modules within the kernel.

We have said many times in this book that architectures come from business goals, as interpreted through the lens of architecturally significant requirements. But what are the business goals for an edge-dominant system? We said earlier that you cannot “know” the requirements for such a system, in any complete sense. Well, this was perhaps a bit hasty.

Requirements for such systems are typically bifurcated into core requirements and periphery requirements:

- Core requirements deliver little or no end-user value and focus on quality attributes and tradeoffs—defining the system’s performance, modularity, security, and so forth. These requirements are generally slow to change as they define the major capabilities and qualities of the system.
- Periphery requirements, on the other hand, are unknowable because they are contributed by the peer network. These requirements deliver the majority of the function and end-user value and change relatively rapidly.

Given this structure, the majority of implementation (the periphery) is crowdsourced to the world using their own tools, to their own standards, at their own pace. The implementers of the core, on the other hand, are generally close knit and highly motivated.

This has at least three implications for the core, which the architect will need to address:

- *The core needs to be highly modular, and it provides the foundation for the achievement of quality attributes.* The core in a successful core/periphery pattern is designed by a small, coherent team. In open source projects, these people are referred to as the “committers.” In Linux, for example, a strong emphasis on modularity has been postulated to account for its enormous growth. This allows for successful contributions of independent enhancements by scores of distributed and unknown-to-each-other programmers. The peripheral services are enabled by and constrained by the kernel, but are otherwise unspecified.
- *The core must be highly reliable.* Most cores are heavily tested, which means that testability is important. Heavy testing for the core is tractable because the core is typically small—often orders of magnitude smaller than the periphery—highly controlled, and relatively slow to change. If the core cannot be made small, then its components can be made to be as independent of each other as possible, which eases the testing burden.
- *The core must be highly robust with respect to errors in its environment.* The reliability of the periphery software is entirely in the hands of the periphery community and the masses (end users and customers). The masses are typically recruited as testers (Mozilla claims to have three million), although this testing is often no more than clicking a button that signals a user’s agreement to have bugs and quality information reported back to the project. Given that the core will undoubtedly be supporting flawed periphery components, robustness of the core is a key requirement; quite simply, failures in the periphery must not cause failures of the core. This means that a system employing the core/periphery pattern should create monitoring mechanisms to determine the current state of the system, and control mechanisms so that bugs in the periphery cannot undermine the core.

The core (often called a *platform*) is usually implemented as a set of services; complex platforms have hundreds. The Amazon EC2 cloud, for example, has over 110 different APIs documented, and EC2 is only a portion of the Amazon platform. To make these services available to peripheral developers, a number of conditions must hold:

- *Documentation must be available for each API, it must be well written, it must be well organized, and it must be up to date.* Because a peripheral developer is frequently a volunteer, incomplete, out of date, or unclear documentation presents a barrier to entry. Even if there is a financial motivation (such as from developing an iPhone or Android application), the documentation still must not present a barrier.
- *There must be a discovery service.* Having hundreds of services means that some of them are going to be redundant and others are going to be unavailable. A discovery service becomes a necessity to enable navigation and flexibility in such a world. A discovery service, in turn, implies a registration service. Services must proactively register upon initialization and be removed if they are no longer active.

- *Error detection becomes extremely complicated.* If you as a peripheral developer encounter a bug in a service, it may be a bug in the service you are invoking, it may be a bug in a service invoked by the service you are invoking, or anywhere in the chain of services. Reporting a problem and getting it resolved may end up being extremely time-consuming. Quality assurance of services requires constant testing of their availability and correctness. The Netflix Simian Army we discussed in [Chapter 10](#) is an example of how quality assurance on a platform might be structured.
- *All of the peer services might be potential denial-of-service attackers.* Throttling, monitoring, and quotas must be employed to ensure that service requesters receive adequate responses to their requests.

Building a platform to be a core to support peripheral developers is a nontrivial undertaking. Yet having such a core has paid dramatic dividends for companies comprising the Who's Who of today's web.

27.4. Implications of the Metropolis Model

The Metropolis model, as we've seen, is paired with the core/periphery pattern for architecture for edge-dominant systems. Adopting this duo brings with it changes to the way that software is developed; in effect, it implies a software development model, with its implications on tools, processes, activities, roles, and expectations. Many such models have evolved over the years, each with its own characteristics, strengths, and weaknesses. Clearly, no one model is best for all projects. For instance, Agile methods are best in projects with rapidly evolving requirements and short time-to-market constraints, whereas a waterfall model is best for large projects with well-understood and stable requirements and complex organizational structures.

The Metropolis model requires a new perspective on system development, resulting in several important changes to how we must create systems:

1. *Indifference to phases.* The Metropolis model uses the metaphor of a bull's eye, as opposed to a waterfall, a spiral, a "V," or other representations that previous models have adopted. The contrast to these previous models is salient: the "phases" of development disappear in the bull's eye. Instead, we must focus managerial attention on the explicit inclusion of customers (the periphery and the masses) for system development.
2. *Crowd management.* Policies for crowd management must be aligned with the organization's strategic goals and must be established early. Crowds are good for certain tasks, but not for all. This implies that business models must be examined to consider policies and associated system development tasks for crowd engagement, performance management monitoring, community protection, and so on. As crowdsourcing is rooted in the "gift" culture, for-profit organizations must carefully align tasks with the volunteers' values and intentions.
3. *Core versus periphery.* The Metropolis model differentiates the core and periphery communities, with different tools, processes, activities, roles, and expectations for each. The core must be small and tightly controlled by a group who focus on modularity, core services, and core quality attributes; this enables the unbridled and uncoordinated growth at the periphery.
4. *Requirements process.* The requirements for a Metropolis system are primarily asserted by the periphery, not elicited from the masses; they emerge from the collective experiences of the community of the periphery, typically through their emails, wikis, and discussion forums. So such forums must be made available—typically provided by members of the core—and the periphery should be encouraged to participate in discussions about the requirements, in effect, to create a community. This changes the fundamental nature of requirements engineering, which has traditionally focused on collecting requirements, making them complete and consistent, and removing redundancies wherever possible.
5. *Focus on architecture.* The core architecture is the fabric that holds together a Metropolis system. As such, it must be consciously designed to accommodate the specific characteristics of open content and open source systems. For this reason, the architecture cannot "emerge," as it often does in traditional life-cycle models, and in

Agile models. It must be designed up front, built by a small, experienced, motivated team who focus on (1) modularity, to enable the parallel activities of the periphery, and (2) the core quality attributes (security, performance, availability, and so on). There should be a lead architect, or a small team of leads, who can manage project coordination and who have the final say in matters affecting the core. Linus Torvalds, for example, still exerts “veto” rights on matters affecting Linux’s kernel. Virtually every open source project distinguishes between the roles of contributor (who can contribute patches) and committer (who chooses which patches make it into any given release).

6. *Distributed testing.* The core/periphery distinction also provides guidance for testing activities. The core must be heavily tested and validated, because it is the fabric that holds the system together. Thus, when planning a Metropolis project, it is important to focus on validation of the core and to put tools, guidelines, and processes in place to facilitate this. For example, the core should be kept small; the project should have frequent (perhaps nightly) builds and frequent releases; bug reporting should be built in to the system and require little effort on the part of the periphery. The project must explicitly take advantage of the “many eyes” provided by the periphery.
7. *Automated delivery.* Delivery mechanisms must be created that work in a distributed, asynchronous manner. These mechanisms must be flexible enough to accept incompleteness of the installed base as the norm. Thus, any delivery mechanism must be tolerant of older versions, multiple coexisting versions, or even incomplete versions. A Metropolis system should also, as far as possible, be tolerant of incompatibilities both within the system and between systems. For example, modern web browsers will still parse old versions of HTML or interact with old versions of web servers; browser add-ons and plug-ins coexist at different version levels and no combination of them will “break” the browser.
8. *Management of the periphery.* One important aspect of the core/periphery model is that the core exercises very little control over the periphery. Yet this does not mean that the periphery is totally unmanaged. If we examine the extant platforms that are either crowdsourced or peripheral developer sourced, we see that there is always a governance policy set by a managing organization. The Internet and the World Wide Web have a collection of governing boards, large open source projects and Wikipedia are managed by foundations and meritocracies, and private companies such as Facebook or Apple have their own management structures. The governance policies created by the management organizations are enforced in either a proactive or reactive fashion. Some policies are enforced by a combination of both:
 - *Proactive enforcement.* Proactive enforcement inhibits contributions by the prosumers or the peripheral developers unless they meet certain criteria. Within the Internet, for example, IP addresses are assigned. One cannot make up one’s own IP address. Communication protocols and web standards are defined by groups chartered by one of the Internet or web governing organizations. Apple, as another example, screens applications before they are eligible for inclusion in the App Store. And every platform has a collection of APIs that also constrain and govern how a peripheral application interacts with it.
 - *Reactive enforcement.* Reactive enforcement dictates the response in case there is a violation of the organization’s policy. Wikipedia has a collection of editors who are responsible for ensuring the quality of contributions after they have been made. Facebook, YouTube, Flickr and most other crowdsourced sites have procedures to report violations. And if a peripheral developer does not adhere to a protocol or a set of APIs, then their product is flawed in some fashion and the market will likely punish them.

The analogy of a city to explain some of the facets of the core/periphery model can be extended. Zoning is a policy that describes permissible land use for a city or other governmental organization. It specifies, for example, that certain pieces of land are for residential use and other pieces are for industrial use. Zoning policies have both proactive and reactive enforcement. [Figure 27.2](#) shows some of the actors associated with a zoning board. The zoning board is the governance organization; it produces a building code that prescribes legitimate uses and restrictions on various buildings. The building inspector is a reactive enforcer who is responsible for verifying that the buildings conform to permissible standards and usage. As with any analogy, the zoning board is not an exact description of the core/periphery, but it does identify many of the elements that go into controlling contributions.

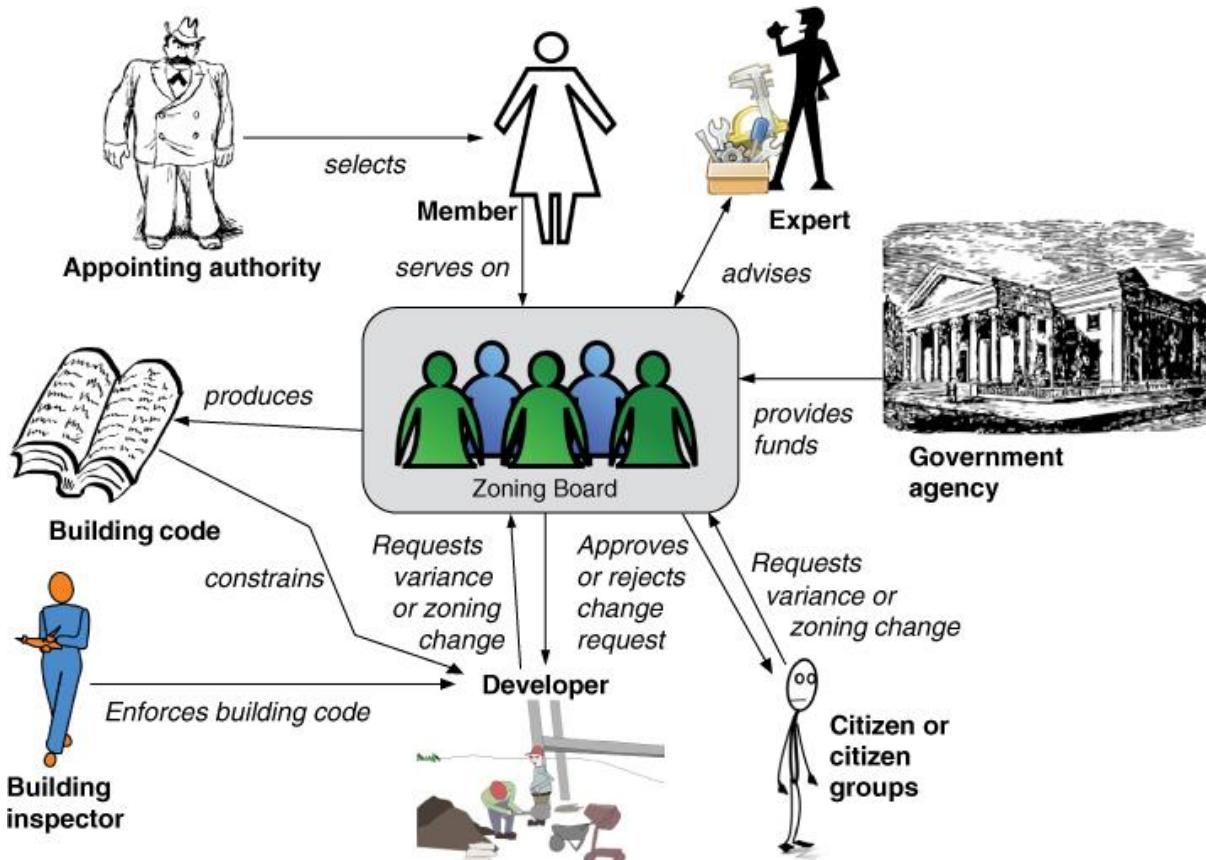


Figure 27.2. Zoning board stakeholders

Life-cycle models are never revolutionary; they arise in reaction to ambient conditions in the software development world. The Waterfall model was created to focus more attention on removing flaws early in a project's life cycle, in reaction to the delays, bugs, and failures of projects of ever-increasing complexity. The spiral model and, later, the Rational Unified Process were created because projects needed to produce working versions of software more quickly and to mitigate risks earlier in the software development life cycle. Agile methods grew out of the desire for less bureaucracy, more responsiveness to customer needs, and shorter times to market.

Similarly, the Metropolis model is formally capturing a market response that is already occurring: the rise of commons-based peer production and service-dominant logic. Prior life-cycle models are simply inadequate—mostly mute—on the concerns of edge-dominant systems: crowdsourcing, emergent requirements, and change as a constant. This model offers new ways to think about how a new breed of systems can be developed; its principles help management shift to new project management styles and architecture models that take advantage of the “wisdom of crowds.”

Metropolis model concepts are not appropriate for all forms of development. Smaller systems with limited scope will continue to benefit from the conceptual integrity that accompanies a small, cohesive team. High-security and safety-critical systems, and systems that are built around protected intellectual property, will continue to be built in traditional ways for the foreseeable future. But more and more crowdsourcing, mashups, open source, and other forms of edge-dominant development are being harnessed for value cocreation, and the Metropolis model does speak to this.

27.5. Summary

An edge-dominant system is one that depends crucially on the inputs of users for its success. Users participate in information creation (“crowdsourcing”) and even its organization

("folksonomy"). These systems, part of the "Web 2.0" movement, are having profound social, political, and economic consequences.

All successful edge-dominant systems, and the organizations that develop and use these systems, share a common ecosystem structure known as the "Metropolis" structure. This structure shows how customers, end users, developers, and "prosumers" are related.

Edge systems bring a new life-cycle model to the fore, in which requirements are not completely known, software developed in planned increments is replaced by software that is constantly changing, and projects are staffed by members outside the purview of the central developing organization.

The dominant pattern for edge systems is the core/periphery pattern. This pattern divides the world into a closely controlled core and a loosely controlled periphery. To work, the core needs to be highly modular, highly reliable, and highly robust with respect to external faults. Cores are often designed as a set of services with well-documented APIs, discovery and registration, and sophisticated error detection and reporting.

27.6. For Further Reading

An interesting interview of Linus Torvalds, showing his management style—what he calls "shepherding"—appeared in *BusinessWeek* magazine several years ago [[Hamm 04](#)].

Yochai Benkler's intriguing book *The Wealth of Networks* [[Benkler 07](#)] puts forth a powerful premise: that the networked information economy is transforming society. It shows how the modern networked economy transforms methods of production and consumption, and creates new forms of value that do not depend on market strategies.

For more information and background on the Metropolis model, you can read the original paper describing it [[Kazman 09](#)].

Much of the inspiration for the Metropolis model comes from the Ultra-Large-Scale Systems report [[Feiler 06](#)].

MacCormack and colleagues have written extensively on the architecture and properties of what they call "core/periphery" systems. See, for example, [[MacCormack 10](#)].

27.7. Discussion Questions

1. Draw the architecture influence cycle for Web 2.0 software systems in general, and for one of its flagship examples (such as Twitter or Facebook).
2. Create a complete pattern description for the core/periphery pattern, modeled on those in [Chapter 13](#).
3. How might the role of architecture documentation be different for an edge-dominant system?
4. Which architectural views would you expect to be the most important to document for a system built under the Metropolis model?
5. How would you establish the architecturally significant requirements for the core? Would you use a Quality Attribute Workshop, or would you use something less structured and more open? Why?
6. Metropolis systems are frequently open source. Some organizations that might want to contribute to or build on top of such a system may balk at releasing all of their code to the public. What architectural means might you employ to address this situation?
7. Choose your favorite crowdsourced system. Write a testability scenario for this system, and choose a set of testability tactics that you would use for it.
8. Constructing and releasing an application on a platform such as the iPhone or the Android requires the developer to adhere to certain specifications and to pass through certain hoops. Redraw [Figure 27.2](#) to reflect the Apple iPhone ecosystem and the Android ecosystem.

- 9.** Find a study that discusses the motivation of Wikipedia contributors. Find another study that discusses the motivation of open source developers. Compare the results of these two studies.

28. Epilogue

Don't let it end like this. Tell them I said something.

—Pancho Villa

You've arrived at the end of the journey. Nice work.

We hope you can find some valuable takeaways from this book. We suggest the list should include the following:

1. What architecture is, why it's important, what influences it, and what it influences.
2. The role that architecture plays in a business, technical, project, and professional context.
3. The critically important symbiosis between architecture and quality attributes, how to specify quality attribute requirements, and the quick immersion you've had into the dozen or so of the most important QAs.
4. How to capture the architecturally significant requirements for an architecture.
5. How to design an architecture, document it, guide an implementation with it, evaluate it to see if it's a good one for your needs, reverse-engineer it, and manage a project around it.
6. How to evaluate an architecture's cost and benefit, what it means to be architecturally competent, and how to use architecture as the basis for an entire software product line.
7. Architectural concepts and patterns for systems on the current technological frontier: edge applications and the cloud.

Fine. Now what?

It's very tempting to end the book with a cheery imperative to go forth and architect great systems. But the truth is, life isn't that simple.

The authors of this book have, collectively, an embarrassingly large number of years of experience in teaching software architecture. We teach it through books like this one, in the classroom to students, and in industrial short courses to practicing software architects of all stripes, from "aspiring" to "old hand." And often, much too often, we know that after our students conscientiously learn what we have to offer, they walk into an organization that is not as architecturally savvy as the students now are, and our students have no practical way to put what they have learned into practice.

Most of you will not be able to dictate an architecture-based philosophy to the projects to which you'll be assigned, if it's not already present. You won't be able to say, "This Agile project needs a lead software architect!" and make it stick if the team leaders think that Agile methods won't permit any overarching design. You won't be able to say, "We're going to include an explicit architecture evaluation in our project schedule" and have everyone comply. You won't be able to say, "We're going to use the Views and Beyond approach and templates for our architecture documentation" and have your directive obeyed.

Lest you feel that all your time absorbing the material in this book was time spent for a lost cause, we want to close the book with some advice for carrying what you've learned into your professional setting:

1. **Speak the right language.** You know by now that architecture is the means to an end, and not an end in itself. The decision makers in your organization typically care about the ends, not the means. They care about products, not the architectures of those products. They care about ensuring that the products are competitive in the marketplace. And they care about executing the organization's marketing and business strategy. They may not express their concerns in architecturally significant terms, but rather in market terms that you'll have to translate.
2. **Speak the right language, part 2.** Project managers care about reduction of technical risk, reliable and realistic scheduling and budgeting, and planning the production of those products. To the extent that you can justify a focus on architecture in those terms, you'll more likely be successful in gaining the freedom to carry out some of the practices espoused in this book. And you really can justify this focus: A

sound architecture is an unparalleled risk reduction strategy, a reliable work estimator, and a good predictor of production methods.

- 3. Get involved.** One of the best ways to insert architectural concerns into a project is to show its value to stakeholders who don't often get a chance to see it. Requirements engineers are a case in point. Most often, they meet with customers and users, capture their concerns, write them up, and toss the requirements over the fence (usually a very tall fence) to the designers. Challenge this segregation! Try to get involved in the requirements capture activity. Invite yourself to meetings. Say that, as an architect, you want to understand the problem by hearing the concerns straight from the horse's mouth. This will give you critical exposure to the very stakeholders you'll need to help with your design, evaluation, and documentation chores. Furthermore it will give you a chance to add value to the requirements capture process. Because you may have a design approach in mind, you may be able to offer better quality attribute responses than the customer has in mind, and that might make our marketers very happy. Or you may be able to spot troublesome requirements early on, and help nudge the customer to a perfectly adequate but more architecturally palatable QA response. Also, you can take it upon yourself to contact your organization's marketers. They are often the ones who come up with product concepts. You would do well to learn how they do that, and eventually you could help them by pointing out useful variations on existing products that could use the same architecture.
- 4. It's the economy, stupid.** Think in, and couch your arguments in, economic terms. If you think an architectural trade study, or an architecture document, or an architecture evaluation, or ensuring code compliance with the architecture is a good idea for your project, make a business case for it. Pointy-haired bosses in comic strips notwithstanding, managers are really—trust us here—rational people. But their goals are broad and almost always have to do with economics. You should be able to argue, using back-of-the-envelope arithmetic, that (for example) producing an updated version of the architecture document is a worthwhile activity when the system undergoes a major change. You should be able to argue that activities undertaken with the new architecture documentation will be much less error-prone (and therefore less expensive) than those same activities undertaken without a guiding architecture. And the effort to keep the documentation up to date is much less than the expected savings. You can plug some numbers in a spreadsheet to make this argument. The numbers don't have to be accurate, just reasonable, and they'll still make your case.
- 5. Start a guerrilla movement.** Find like-minded people in your organization and nurture their interest in architecture. Start a brown-bag lunchtime reading and discussion group that covers books or book chapters or papers or even blogs about architecture. For example, your group could read the chapter in this book about architecture competence, and discuss what practices you'd like to see your organization adopt, and what it would take to adopt them. Or the group could agree on a joint documentation template for architecture, or come up with a set of quality attribute scenarios that apply across your collective projects. Especially appealing is to come up with a set of patterns that apply to the systems you're building. Or bring a vexing design problem from your individual project, and let the group work on a written solution to it. Or have the group offer its services as a roving architecture evaluation team to other projects. Your group should meet regularly and often, and adopt a specific set of tangible goals. The importance of an enthusiastic and dedicated leader—you?—who is keen to mature the organization's architecture practices cannot be overstated. Advertise your meetings, advertise your results, and keep asking more members to join your group.
- 6. Relish small victories.** You don't have to change the world overnight. Every improvement you make will put you and your organization in a better place than it would have been otherwise.

Getting Architecture Reviews into an Organization through the Back Door

If you search the web for "code review computer science," you'll turn up millions of hits that describe code reviews and the steps that are taken to perform them. If you search for "design review computer science," you'll turn up little that is useful.

Other disciplines routinely practice and teach design critiques. Search for “design critique” and you will find many hits together with instructions. A design is a set of decisions of whatever type that attempts to solve a particular problem, whether an art problem, a user interface design problem, or a software problem. Solutions to important design problems should be subject to peer review, just as code should be subject to peer review.

There is a wealth of data that points out that the earlier in the life cycle a problem is discovered and fixed, the less the cost of finding and fixing the problem. Design precedes code and so having appropriate design reviews seems both intuitively and empirically justified. In addition, the documents around the review, both the original design document and also the critiques, are valuable learning tools for new developers. In many organizations developers switch systems frequently, and so they are constantly learning.

This view is not universally shared. A software engineer working in a major software house tells me that even though the organization aspires to writing and reviewing design documents, it rarely happens. Senior developers tend to limit their review to a cursory glance. Code reviews, on the other hand, are taken quite seriously by the senior developers.

My software engineer friend offers two possible explanations for this state of affairs:

- 1.** The code review is the last opportunity to affect what is built: “review this or live with it.” This explanation assumes that senior developers do not believe that the output of design reviews are actionable and thus wait to engage until later in the process.
- 2.** The code is more concrete than the design, and is therefore easier to assess. This explanation assumes that senior developers are incapable of understanding designs.

I do not find either of these explanations compelling, but I am unable to come up with a better one.

What to do?

What this software engineer did is to look for a surrogate process where a design review could be surreptitiously performed. This individual noticed that when the organization did code reviews, questions such as “Why did you do that?” were frequently asked. The result of such questions was a discussion of rationale. So the individual would code up a solution to a problem, submit it to a code review, and wait for the question that would lead to the rationale discussion.

A design review is a review where design decisions are presented together with their rationale. Frequently, design alternatives are explored. Whether this is done under the name of code review or design review is not nearly as important as getting it done.

Of course, my friend’s surreptitious approach has drawbacks. It is inefficient to code a solution that may have to be thrown away. Also, embedding design reviews into code reviews means that the designs and reviews end up being embedded in the code review tool, making it difficult to search this tool for design and design rationale. But these inefficiencies are dwarfed by the inefficiency of pursuing an incorrect solution to a particular problem.

—LB

The practice and discipline of architecture for software systems has come of age. You can be proud of joining a profession that has always strived, and is still striving, to be more disciplined, more reliable, more productive, and more efficient, to produce systems that improve the lives of their stakeholders.

With that thought, *now* it’s time for the cheery book-ending imperative: Go forth and architect great systems. Your predecessors have designed systems that have changed the world. It’s your turn.

References

- [Abrahamsson 10] P. Abrahamsson, M.A. Babar, and P. Kruchten. "Agility and Architecture: Can They Coexist?" *IEEE Software*, Vol. 27, No. 2, (March-April 2010), pp. 16-22.
- [AdvBuilder 10] Java Adventure Builder Reference Application. <https://adventurebuilder.dev.java.net>
- [Anastasopoulos 00] M. Anastasopoulos and C. Gacek. "Implementing Product Line Variabilities" (IESE-Report No. 089.00/E, V1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2000.
- [Anderson 08] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Second Edition*. Wiley, 2008.
- [Argote 07]. L. Argote and G. Todorova. *International Review of Industrial and Organizational Psychology*. John Wiley & Sons, Ltd., 2007.
- [Avižienis 04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1 (January 2004), pp. 11-33.
- [Bachmann 05] F. Bachmann and P. Clements. "Variability in Software Product Lines," CMU/SEI-2005-TR-012, 2005.
- [Bachmann 07] Felix Bachmann, Len Bass, and Robert Nord. "Modifiability Tactics," CMU/SEI-2007-TR-002, September 2007.
- [Bachmann 11] F. Bachmann. "Give the Stakeholders What They Want: Design Peer Reviews the ATAM Style," *Crosstalk*, November/December 2011, pp. 8-10, <http://www.crosstalkonline.org/storage/issue-archives/2011/201111/201111-Bachmann.pdf>
- [Barbacci 03] M. Barbacci, R. Ellison, A. Lattanze, J. Stafford, C. Weinstock, and W. Wood. "Quality Attribute Workshops (QAWs), Third Edition," CMU/SEI-2003-TR-016, <http://www.sei.cmu.edu/reports/03tr016.pdf>
- [Bass 03] L. Bass and B.E. John. "Linking Usability to Software Architecture Patterns through General Scenarios," *Journal of Systems and Software* 66(3), pp. 187-197.
- [Bass 08] Len Bass, Paul Clements, Rick Kazman, and Mark Klein. "Models for Evaluating and Improving Architecture Competence," CMU/SEI-2008-TR-006, March 2008, <http://www.sei.cmu.edu/library/abstracts/reports/08tr006.cfm>
- [Baudry 03] B. Baudry, Yves Le Traon, Gerson Sunyé, and Jean-Marc Jézéquel. "Measuring and Improving Design Patterns Testability," *Proceedings of the Ninth International Software Metrics Symposium (METRICS '03)*, 2003.
- [Baudry 05] B. Baudry and Y. Le Traon. "Measuring Design Testability of a UML Class Diagram," *Information & Software Technology* 47(13)(October 2005), pp. 859-879.
- [Beck 04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley, 2004.
- [Beizer 90] B. Beizer. *Software Testing Techniques, Second Edition*. International Thomson Computer Press, 1990.
- [Bellcore 98] Bell Communications Research. GR-1230-CORE, SONET Bidirectional Line-Switched Ring Equipment Generic Criteria. 1998.
- [Bellcore 99] Bell Communications Research. GR-1400-CORE, SONET Dual-Fed Unidirectional Path Switched Ring (UPSR) Equipment Generic Criteria. 1999.

[Benkler 07] Y. Benkler. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, 2007.

[Bertolino 96a] Antonia Bertolino and Lorenzo Strigini. "On the Use of Testability Measures for Dependability Assessment," *IEEE Transactions on Software Engineering*, Vol. 22, No. 2 (February 1996), pp. 97-108.

[Bertolino 96b] A. Bertolino and P. Inverardi. "Architecture-Based Software Testing," in *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, L. Vidal, A. Finkelstain, G. Spanoudakis, and A.L. Wolf, eds. Joint Proceedings of the SIGSOFT '96 Workshops, San Francisco, October 1996, ACM Press.

[Biffl 10] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grunbacher, eds. *Value-Based Software Engineering*. Springer, 2010.

[Binder 94] R.V. Binder. "Design for Testability in Object-Oriented Systems," *CACM* 37(9), pp. 87-101, 1994.

[Boehm 78] B.W. Boehm, J.R. Brown, J.R. Kaspar, M.L. Lipow, and G. MacCleod. *Characteristics of Software Quality*. American Elsevier, 1978.

[Boehm 81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[Boehm 91] Barry Boehm. "Software Risk Management: Principles and Practices," *IEEE Software*, Vol. 8, No. 1, pp. 32-41, January 1991.

[Boehm 04] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

[Boehm 07] B. Boehm, R. Valerdi, and E. Honour. "The ROI of Systems Engineering: Some Quantitative Results for Software Intensive Systems," *Systems Engineering*, Vol. 11, No. 3, pp. 221-234.

[Boehm 10] B. Boehm, J. Lane, S. Koolmanojwong, and R. Turner. "Architected Agile Solutions for Software-Reliant Systems," Technical Report USCCSSE-2010-516, 2010.

[Booch 11] Grady Booch. "An Architectural Oxymoron," podcast available at <http://www.computer.org/portal/web/computingnow/onarchitecture>. Retrieved January 21, 2011.

[Bosch 00] J. Bosch. "Organizing for Software Product Lines," *Proceedings of the 3rd International Workshop on Software Architectures for Product Families (IWSAPF-3)*, pp. 117-134. Las Palmas de Gran Canaria, Spain, March 15-17, 2000. Springer, 2000.

[Bouwers 10] E. Bouwers and A. van Deursen. "A Lightweight Sanity Check for Implemented Architectures," *IEEE Software* 27(4), July/August 2010, pp. 44-50.

[Bredemeyer 11] D. Bredemeyer and R. Malan. "Architect Competencies: What You Know, What You Do and What You Are," <http://www.bredemeyer.com/Architect/ArchitectSkillsLinks.htm>

[Brewer 12] E. Brewer. "CAP Twelve Years Later: How the 'Rules' Have Changed," *IEEE Computer*, February 2012, pp. 23-29.

[Brown 10] N. Brown, R. Nord, and I. Ozkaya. "Enabling Agility Through Architecture," *Crosstalk*, November/December 2010, pp. 12-17.

[Brownsword 96] Lisa Brownsword and Paul Clements. "A Case Study in Successful Product Line Development," Technical Report CMU/SEI-96-TR-016, October 1996.

[Brownsword 04] Lisa Brownsword, David Carney, David Fisher, Grace Lewis, Craig Meterys, Edwin Morris, Patrick Place, James Smith, and Lutz Wrage. "Current Perspectives on Interoperability," CMU/SEI-2004-TR-009, <http://www.sei.cmu.edu/reports/04tr009.pdf>

- [Bruntink 06] Magiel Bruntink and Arie van Deursen. "An Empirical Study into Class Testability," *Journal of Systems and Software* 79(9)(2006), pp. 1219-1232.
- [Buschmann 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.
- [Cai 11] Yuanfang Cai, Daniel Iannuzzi, and Sunny Wong. "Leveraging Design Structure Matrices in Software Design Education," *Conference on Software Engineering Education and Training 2011*, pp. 179-188.
- [Cappelli 12] Dawn M. Cappelli, Andrew P. Moore, and Randall F. Trzeciak. *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. Addison-Wesley, 2012.
- [Carriere 10] J. Carriere, R. Kazman, and I. Ozkaya. "A Cost-Benefit Framework for Making Architectural Decisions in a Business Context," *Proceedings of 32nd International Conference on Software Engineering (ICSE 32)*, Capetown, South Africa, May 2010.
- [Cataldo 07] M. Cataldo, M. Bass, J. Herbsleb, and L. Bass. "On Coordination Mechanisms in Global Software Development," *Proceedings Second IEEE International Conference on Global Software Development*, 2007.
- [Chandran 10] S. Chandran, A. Dimov, and S. Punnekat. "Modeling Uncertainties in the Estimation of Software Reliability—A Pragmatic Approach," *Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2010.
- [Chang 06] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, et al. "Bigtable: A Distributed Storage System for Structured Data," *Proceedings Operating Systems Design and Implementation*, 2006, <http://research.google.com/archive/bigtable.html>
- [Chen 10] H.-M. Chen, R. Kazman, and O. Perry. "From Software Architecture Analysis to Service Engineering: An Empirical Study of Enterprise SOA Implementation," *IEEE Transactions on Services Computing* 3(2)(April-June 2010), pp. 145-160.
- [Chidamber 94] S. Chidamber and C. Kemerer. "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6 (June 1994).
- [Clements 01a] P. Clements and L. Northrop. *Software Product Lines*. Addison-Wesley, 2001.
- [Clements 01b] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures*. Addison-Wesley, 2001.
- [Clements 07] P. Clements, R. Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma. "The Duties, Skills, and Knowledge of Software Architects," *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, 2007.
- [Clements 10a] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley, 2010.
- [Clements 10b] Paul Clements and Len Bass. "Relating Business Goals to Architecturally Significant Requirements for Software Systems," CMU/SEI-2010-TN-018, May 2010.
- [Clements 10c] P. Clements and L. Bass. "The Business Goals Viewpoint," *IEEE Software* 27(6)(November-December 2010), pp. 38-45.
- [Cockburn 04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.
- [Conway 68] Melvin E. Conway. "How Do Committees Invent?" *Datamation*, Vol. 14, No. 4 (1968), pp. 28-31.

[Coplein 10] J. Coplein and G. Bjornvig. *Lean Architecture for Agile Software Development*. Wiley, 2010.

[Cunningham 92] W. Cunningham. "The Wycash Portfolio Management System," in Addendum to the *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 29-30, ACM Press, 1992.

[CWE 12] The Common Weakness Enumeration. <http://cwe.mitre.org/>

[Dean 04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings Operating System Design and Implementation*, 1994, <http://research.google.com/archive/mapreduce.html>

[Dijkstra 68] E.W. Dijkstra. "The Structure of the 'THE'-Multiprogramming System," *Communications of the ACM* 11(5), pp. 341-346.

[Dix 04] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction, Third Edition*. Prentice Hall, 2004.

[Douglass 99] Bruce Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 1999.

[Dutton 84] J.M. Dutton and A. Thomas. "Treating Progress Functions as a Managerial Opportunity," *Academy of Management Review* 9 (1984), pp. 235-247.

[Eickelman 96] N. Eickelman and D. Richardson. "What Makes One Software Architecture More Testable Than Another?" in *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, eds., Joint Proceedings of the SIGSOFT '96 Workshops, San Francisco, October 1996, ACM Press.

[EOSAN 07] "WP 8.1.4—Define Methodology for Validation within OATA: Architecture Tactics Assessment Process," <http://www.eurocontrol.int/valfor/gallery/content/public/OATA-P2-D8.1.4-01%20DMVO%20Architecture%20Tactics%20Assessment%20Process.pdf>

[FAA 00] "System Safety Handbook," http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/

[Fairbanks 10] G. Fairbanks. *Just Enough Software Architecture*. Marshall & Brainerd, 2010.

[Feiler 06] P. Feiler, R.P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*, http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

[Fiol 85] C.M. Fiol and M.A. Lyles. "Organizational Learning," *Academy of Management Review* 10(4)(1985), p. 803.

[Freeman 09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, 2009.

[Gacek 95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm. "On the Definition of Software System Architecture," USC/CSE-95-TR-500, April 1995.

[Gagliardi 09] M. Gagliardi, W. Wood, J. Klein, and J. Morley. "A Uniform Approach for System of Systems Architecture Evaluation," *Crosstalk*, Vol. 22, No. 3 (March/April 2009), pp. 12-15.

[Gamma 94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Garlan 93] D. Garlan and M. Shaw. "An Introduction to Software Architecture," in Ambriola and Tortola, eds., *Advances in Software Engineering & Knowledge Engineering, Vol. II*. World Scientific Pub. Co., 1993, pp. 1-39.

[Garlan 95] David Garlan, Robert Allen, and John Ockerbloom. "Architectural Mismatch or Why it's hard to build systems out of existing parts," ICSE 1995. 17th International Conference on Software Engineering, April 1995.

[Gilbert 07] T. Gilbert. *Human Competence: Engineering Worthy Performance*. Pfeiffer, Tribute Edition, 2007.

[Gokhale 05] S. Gokhale, J. Crigler, W. Farr, and D. Wallace. "System Availability Analysis Considering Hardware/Software Failure Severities," *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW '05)*, Greenbelt, MD, April 2005, IEEE 2005.

[Gorton 10] Ian Gorton. *Essential Software Architecture, Second Edition*. Springer, 2010.

[Graham 07] T.C.N. Graham, R. Kazman, and C. Walmsley. "Agility and Experimentation: Practical Techniques for Resolving Architectural Tradeoffs," *Proceedings of the 29th International Conference on Software Engineering (ICSE 29)*, Minneapolis, MN, May 2007.

[Gray 93] Jim Gray and Andreas Reuter. *Distributed Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Grinter 99] Rebecca E. Grinter. "Systems Architecture: Product Designing and Social Engineering," in *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC '99)*, Dimitrios Georgakopoulos, Wolfgang Prinz, and Alexander L. Wolf, eds. ACM, 1999, pp. 11-18.

[Hamm 04] "Linus Torvalds' Benevolent Dictatorship," *BusinessWeek*, August 18, 2004, http://www.businessweek.com/technology/content/aug2004/tc20040818_1593.htm

[Hamming 80] R.W. Hamming. *Coding and Information Theory*. Prentice Hall, 1980.

[Hanmer 07] Robert Hanmer. *Patterns for Fault Tolerant Software*, Wiley, 2007.

[Harms 10] R. Harms and M. Yamartino. "The Economics of the Cloud," http://economics.uchicago.edu/pdf/Harms_110111.pdf

[Hartman 10] Gregory Hartman. "Attentiveness: Reactivity at Scale," CMU-ISR-10-111, 2010.

[Hiltzik 00] M. Hiltzik. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. Harper Business, 2000.

[Hoffman 00] Daniel M. Hoffman and David M. Weiss. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2000.

[Hofmeister 00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[Hofmeister 07] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. "A General Model of Software Architecture Design Derived from Five Industrial Approaches," *Journal of Systems and Software*, Vol. 80, No. 1 (January 2007), pp. 106-126.

[Howard 04] Michael Howard. "Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users," *MSDN Magazine*, <http://msdn.microsoft.com/en-us/magazine/cc163882.aspx>

[IEEE 94] "IEEE Standard for Software Safety Plans," STD-1228-1994, <http://standards.ieee.org/findstds/standard/1228-1994.html>

[IEEE 11] "IEEE Guide—Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide), Fourth Edition," <http://www.projectsmart.co.uk/pmbok.html>

[IETF 04] Internet Engineering Task Force. "RFC 3746, Forwarding and Control Element Separation (ForCES) Framework," 2004.

[IETF 05] Internet Engineering Task Force. "RFC 4090, Fast Reroute Extensions to RSVP-TE for LSP Tunnels," 2005.

[IETF 06a] Internet Engineering Task Force. "RFC 4443, Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," 2006.

[IETF 06b] Internet Engineering Task Force. "RFC 4379, Detecting Multi-Protocol Label Switched (MPLS) Data Plane Failures," 2006.

[INCOSE 05] International Council on Systems Engineering. "System Engineering Competency Framework 2010-0205," <http://www.incose.org/ProductsPubs/products/competenciesframework.aspx>

[ISO 11] International Organization for Standardization. "ISO/IEC 25010: 2011 Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models."

[Jacobson 97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1997.

[Kanwal 10] F. Kanwal, K. Junaid, and M.A. Fahiem. "A Hybrid Software Architecture Evaluation Method for FDD—An Agile Process Mode," *2010 International Conference on Computational Intelligence and Software Engineering (CiSE)*, December 2010, pp. 1-5.

[Kaplan 92] R. Kaplan and D. Norton. "The Balanced Scorecard: Measures That Drive Performance," *Harvard Business Review*, January/February 1992, pp. 71-79.

[Karat 94] Claire Marie Karat. "A Business Case Approach to Usability Cost Justification," in *Cost-Justifying Usability*, R. Bias and D. Mayhew, eds. Academic Press, 1994.

[Kazman 94] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. "SAAM: A Method for Analyzing the Properties of Software Architectures," in *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. Los Alamitos, CA. IEEE Computer Society Press, pp. 81-90.

[Kazman 99] R. Kazman and S.J. Carriere. "Playing Detective: Reconstructing Software Architecture from Available Evidence," *Automated Software Engineering* 6(2)(April 1999), pp. 107-138.

[Kazman 01] R. Kazman, J. Asundi, and M. Klein. "Quantifying the Costs and Benefits of Architectural Decisions," *Proceedings of the 23rd International Conference on Software Engineering (ICSE 23)*, Toronto, Canada, May 2001, pp. 297-306.

[Kazman 02] R. Kazman, L. O'Brien, and C. Verhoef. "Architecture Reconstruction Guidelines, Third Edition," CMU/SEI Technical Report, CMU/SEI-2002-TR-034, 2002.

[Kazman 04] R. Kazman, P. Kruchten, R. Nord, and J. Tomayko. "Integrating Software-Architecture-Centric Methods into the Rational Unified Process," Technical Report CMU/SEI-2004-TR-011, July 2004, <http://www.sei.cmu.edu/library/abstracts/reports/04tr011.cfm>

[Kazman 05] Rick Kazman and Len Bass. "Categorizing Business Goals for Software Architectures," CMU/SEI-2005-TR-021, December 2005.

[Kazman 09] R. Kazman and H.-M. Chen. "The Metropolis Model: A New Logic for the Development of Crowdsourced Systems," *Communications of the ACM*, July 2009, pp. 76-84.

[Kircher 03] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. Wiley, 2003.

[Klein 10] J. Klein and M. Gagliardi. "A Workshop on Analysis and Evaluation of Enterprise Architectures," CMU/SEI-2010-TN-023, <http://www.sei.cmu.edu/reports/10tn023.pdf>

[Klein 93] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour. *A Practitioner's Handbook for Real-Time Systems Analysis*. Kluwer Academic, 1993.

[Koziol 10] H. Koziolek. "Performance Evaluation of Component-Based Software Systems: A Survey," *Performance Evaluation* 67(8)(August 2010).

[Kruchten 95] P.B. Kruchten. "The 4+1 View Model of Architecture," *IEEE Software*, Vol. 12, No. 6 (November 1995), pp. 42-50.

[Kruchten 03] Philippe Kruchten. *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley, 2003.

[Kruchten 04] Philippe Kruchten. "An Ontology of Architectural Design Decisions," in Jan Bosch, ed., *Proceedings of the 2nd Workshop on Software Variability Management*, Groningen, NL, Dec. 3-4, 2004.

[Kumar 10a] K. Kumar and TV Prabhakar. "Pattern-Oriented Knowledge Model for Architecture Design," in *Pattern Languages of Programs Conference 2010*, October 15-18, 2010, Reno/Tahoe, Nevada.

[Kumar 10b] Kiran Kumar and TV Prabhakar. "Design Decision Topology Model for Pattern Relationship Analysis," *Asian Conference on Pattern Languages of Programs 2010*, March 15-17, 2010, Tokyo, Japan.

[Ladas 09] Corey Ladas. *Scrumban: Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, 2009.

[Lattanzie 08] Tony Lattanzie. *Architecting Software Intensive Systems: A Practitioner's Guide*. Auerbach Publications, 2008.

[Le Traon 97] Y. Le Traon and C. Robach. "Testability Measurements for Data Flow Designs," *Proceedings of the 4th International Symposium on Software Metrics (METRICS '97)*, pp. 91-98. November 1997, Washington, D.C.

[Leveson 04] Nancy G. Leveson. "The Role of Software in Spacecraft Accidents," *Journal of Spacecraft and Rockets* 41(4)(July 2004), pp. 564-575.

[Leveson 11] Nancy G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2011.

[Levitt 88] B. Levitt and J. March. "Organizational Learning," *Annual Review of Sociology* 14 (1988), pp. 319-340.

[Liu 00] Jane Liu. *Real-Time Systems*. Prentice Hall, 2000.

[Liu 09] Henry Liu. *Software Performance and Scalability: A Quantitative Approach*. Wiley, 2009.

[Luftman 00] J. Luftman. "Assessing Business Alignment Maturity," *Communications of AIS*, Vol. 4, No. 14, 2000.

[Lyons 62] R. E. Lyons and W. Vanderkulk. "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM J. Res. Dev.* 6(2)(April 1962), pp. 200-209.

[MacCormack 06] A. MacCormack, J. Rusnak, and C. Baldwin. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science* 52(7)(July 2006), pp. 1015-1030.

[MacCormack 10] A. MacCormack, C. Baldwin, and J. Rusnak. "The Architecture of Complex Systems: Do Core-Periphery Structures Dominate?" MIT Sloan Research Paper No. 4770-10, <http://www.hbs.edu/research/pdf/10-059.pdf>

[Malan 00] Ruth Malan and Dana Bredemeyer. "Creating an Architectural Vision: Collecting Input," http://www.bredemeyer.com/pdf_files/vision_input.pdf, July 25, 2000.

[Maranzano 05] Joseph F. Maranzano, Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss. "Architecture Reviews: Practice and Experience," *IEEE Software*, March/April 2005, pp. 34-43.

[Mavis 02] D.G. Mavis. "Soft Error Rate Mitigation Techniques for Modern Microcircuits," *40th Annual Reliability Physics Symposium Proceedings*, April 2002, Dallas, TX. IEEE, 2002.

[McCall 77] J.A. McCall, P.K. Richards, and G.F. Walters. *Factors in Software Quality*. Griffiths Air Force Base, N.Y. : Rome Air Development Center Air Force Systems Command.

[McGregor 11] John D. McGregor, J. Yates Monteith, and Jie Zhang. "Quantifying Value in Software Product Line Design," in *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*, Ina Schaefer, Isabel John, and Klaus Schmid, eds.

[Mettler 91] R. Mettler. "Frederick C. Lindvall," in *Memorial Tributes: National Academy of Engineering, Volume 4*, pp. 213-216. National Academy of Engineering, 1991.

[Moore 03] M. Moore, R. Kazman, M. Klein, and J. Asundi. "Quantifying the Value of Architecture Design Decisions: Lessons from the Field," *Proceedings of the 25th International Conference on Software Engineering (ICSE 25)*, Portland, OR, May 2003, pp. 557-562.

[Morelos-Zaragoza 06] R.H. Morelos-Zaragoza. *The Art of Error Correcting Coding, Second Edition*. Wiley, 2006.

[Muccini 03] H. Muccini, A. Bertolino, and P. Inverardi. "Using Software Architecture for Code Testing," *IEEE Transactions on Software Engineering* 30(3), pp. 160-171.

[Muccini 07] H. Muccini. "What Makes Software Architecture-Based Testing Distinguishable," in *Proc. Sixth Working IEEE/IFIP Conference on Software Architecture, WICSA 2007*, Mumbai, India, January 2007.

[Murphy 01] G. Murphy, D. Notkin, and K. Sullivan. "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Transactions on Software Engineering*, Vol. 27, pp. 364-380, 2001.

[Nielsen 08] Jakob Nielsen. "Usability ROI Declining, But Still Strong," <http://www.useit.com/alertbox/roi.html>

[NIST 02] National Institute of Standards and Technology. "Security Requirements For Cryptographic Modules," FIPS Pub. 140-2,<http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf>

[NIST 04] National Institute of Standards and Technology. "Standards for Security Categorization of Federal Information Systems," FIPS Pub. 199, <http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf>

[NIST 06] National Institute of Standards and Technology. "Minimum Security Requirements for Federal Information and Information Systems," FIPS Pub. 200, <http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf>

[NIST 09] National Institute of Standards and Technology. "800-53 v3 Recommended Security Controls for Federal Information Systems and Organizations," August 2009, <http://csrc.nist.gov/publications/nistpubs/800-53-Rev3/sp800-53-rev3-final.pdf>

[Nord 04] R. Nord, J. Tomayko, and R. Wojcik. "Integrating Software Architecture-Centric Methods into Extreme Programming (XP)," CMU/SEI-2004-TN-036. Software Engineering Institute, Carnegie Mellon University, 2004.

[Nygard 07] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Programmers, 2007.

[Obbink 02] H. Obbink, P. Kruchten, W. Kozaczynski, H. Postema, A. Ran, L. Dominic, R. Kazman, R. Hilliard, W. Tracz, and E. Kahane. "Software Architecture Review and Assessment (SARA) Report, Version 1.0," 2002,<http://pkruchten.wordpress.com/architecture/SARAv1.pdf/>

[O'Brien 03] L. O'Brien and C. Stoermer. "Architecture Reconstruction Case Study," CMU/SEI Technical Note, CMU/SEI-2003-TN-008, 2003.

[ODUSD 08] Office of the Deputy Under Secretary of Defense for Acquisition and Technology. "Systems Engineering Guide for Systems of Systems, Version 1.0," 2008, <http://www.acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf>

[Palmer 02] Stephen Palmer and John Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.

[Parnas 72] D.L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* 15(12)(December 1972).

[Parnas 74] D. Parnas. "On a 'Buzzword': Hierarchical Structure," *Proceedings IFIP Congress 74*, pp. 336-339. North Holland Publishing Company, 1974.

[Parnas 76] D.L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, SE-2, 1 (March 1976), pp. 1-9.

[Parnas 79] D. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, SE-5, 2 (1979), pp. 128-137.

[Parnas 95] David Parnas and Jan Madey. "Functional Documents for Computer Systems," chapter in *Science of Computer Programming*. Elsevier, 1995.

[Paulish 02] Daniel J. Paulish. *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley, 2002.

[Pena 87] William Pena. *Problem Seeking: An Architectural Programming Primer*. AIA Press, 1987.

[Perry 92] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture," *SIGSOFT Softw. Eng. Notes* 17(4)(October 1992), pp. 40-52.

[Pettichord 02] B. Pettichord. "Design for Testability," Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002.

[Powel Douglass 99] B. Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[Sangwan 08] Raghvinder Sangwan, Colin Neill, Matthew Bass, and Zakaria El Houda. "Integrating a Software Architecture-Centric Method into Object-Oriented Analysis and Design," *Journal of Systems and Software*, Vol. 81, No. 5 (May 2008), pp. 727-746.

[Schmerl 06] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. "Discovering Architectures from Running Systems," *IEEE Transactions on Software Engineering* 32(7)(July 2006), pp. 454-466.

[Schmidt 00] Douglas Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Schmidt 10] Klaus Schmidt. *High Availability and Disaster Recovery: Concepts, Design, Implementation*. Springer, 2010.

[Schneier 96] B. Schneier. *Applied Cryptography*. Wiley, 1996.

[Schneier 08] Bruce Schneier. *Schneier on Security*. Wiley, 2008.

[Schwaber 04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[Scott 09] James Scott and Rick Kazman. "Realizing and Refining Architectural Tactics: Availability," Technical Report CMU/SEI-2009-TR-006, August 2009.

[Seacord 05] Robert Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005.

[SEI 12] Software Engineering Institute. "A Framework for Software Product Line Practice, Version 5.0," http://www.sei.cmu.edu/productlines/frame_report/PL.essential.act.htm

[Shaw 94] Mary Shaw. "Procedure Calls Are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status," Carnegie Mellon University Technical Report, 1994, <http://repository.cmu.edu/cgi/viewcontent.cgi?article=1234&context=sei>

[Shaw 95] Mary Shaw. "Beyond Objects: A Software Design Paradigm Based on Process Control," *ACM Software Engineering Notes*, Vol. 20, No. 1 (January 1995), pp. 27-38.

[Smith 01] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2001.

[Soni 95] Dilip Soni, Robert L. Nord, and Christine Hofmeister. "Software Architecture in Industrial Applications," *International Conference on Software Engineering 1995*, April 1995, pp. 196-207.

[Stonebraker 09] M. Stonebraker. "The 'NoSQL' Discussion Has Nothing to Do with SQL," <http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext>

[Stonebraker 10a] M. Stonebraker. "SQL Databases v. NoSQL Databases," *Communications of the ACM* 53(4), p. 10.

[Stonebraker 10b] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. "MapReduce and Parallel DBMSs," *Communications of the ACM* 53, p. 6.

[Stonebraker 11] M. Stonebraker. "Stonebraker on NoSQL and Enterprises," *Communications of the ACM* 54(8), p. 10.

[Storey 97] M.-A. Storey, K. Wong, and H. Müller. "Rigi—A Visualization Environment for Reverse Engineering (Research Demonstration Summary)," *19th International Conference on Software Engineering (ICSE 97)*, May 1997, pp. 606-607. IEEE Computer Society Press.

[Svahnberg 00] M. Svahnberg and J. Bosch. "Issues Concerning Variability in Software Product Lines," in *Proceedings of the Third International Workshop on Software Architectures for Product Families*, Las Palmas de Gran Canaria, Spain, March 15-17, 2000, pp. 50-60. Springer, 2000.

[Taylor 09] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[Telcordia 00] Telcordia. "GR-253-CORE, Synchronous Optical Network (SONET) Transport Systems: Common Generic Criteria." 2000.

[Urdangarin 08] R. Urdangarin, P. Fernandes, A. Avritzer, and D. Paulish. "Experiences with Agile Practices in the Global Studio Project," *Proceedings of the IEEE International Conference on Global Software Engineering*, 2008.

[Utas 05] G. Utas. *Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems*. Wiley, 2005.

[van der Linden 07] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action*. Springer, 2007.

[van Deursen 04] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. "Symphony: View-Driven Software Architecture Reconstruction," *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, June 2004, Oslo, Norway. IEEE Computer Society.

[van Vliet 05] H. van Vliet. "The GRIFFIN project, A GRId For inFormatIoN about architectural knowledge," <http://griffin.cs.vu.nl/>, Vrije Universiteit, Amsterdam, April 16, 2005.

[Verizon 12] "Verizon 2012 Data Breach Investigations Report," http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf

[Vesely 81] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. "Fault Tree Handbook," <http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/sr0492.pdf>

[Vesely 02] William Vesely, Michael Stamatelatos, Joanne Dugan, Joseph Fragola, Joseph Minarick III, and Jan Railsback. "Fault Tree Handbook with Aerospace Applications," <http://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf>

[Viega 01] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[Voas 95] Jeffrey M. Voas and Keith W. Miller. "Software Testability: the New Verification," *IEEE Software* 12(3)(May 1995), pp. 17-28.

[Von Neumann 56] J. Von Neumann. "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, C.E. Shannon and J. McCarthy, eds. Princeton University Press, 1956.

[Wojcik 06] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and W. Wood. "Attribute-Driven Design (ADD), Version 2.0," Technical Report CMU/SEI-2006-TR-023, November 2006, <http://www.sei.cmu.edu/library/abstracts/reports/06tr023.cfm>

[Wood 07] W. Wood. "A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0," Technical Report CMU/SEI-2007-TR-005, February 2007, <http://www.sei.cmu.edu/library/abstracts/reports/07tr005.cfm>

[Woods 11] E. Woods and N. Rozanski. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*. Addison-Wesley, 2011.

[Wozniak 07] J. Wozniak, V. Baggio, D. Garcia Quintas, and J. Wenninger. "Software Interlocks System," *Proceedings ICALPCS07*, <http://ics-web4.sns.ornl.gov/icalpcs07/WPPB03/WPPB03.PDF>

[Wu 06] W. Wu and T. Kelly. "Deriving Safety Requirements as Part of System Architecture Definition," in *Proceedings of 24th International System Safety Conference*, published by the System Safety Society, August 2006, Albuquerque, NM.

[Yacoub 02] S. Yacoub and H. Ammar. "A Methodology for Architecture-Level Reliability Risk Analysis," *IEEE Transactions on Software Engineering*, Vol. 28, No. 6 (June 2002).

[Yin 94] James Bieman and Hwei Yin. "Designing for Software Testability Using Automated Oracles," *Proceedings International Test Conference*, September 1992, pp. 900-907.

About the Authors

Len Bass is a Senior Principal Researcher at National ICT Australia Ltd. (NICTA). He joined NICTA in 2011 after 25 years at the Software Engineering Institute (SEI) at Carnegie Mellon University. He is the coauthor of two award-winning books in software architecture, including *Documenting Software Architectures: Views and Beyond, Second Edition* (Addison-Wesley, 2011), as well as several other books and numerous papers in computer science and software engineering on a wide range of topics. Len has almost 50 years' experience in software development and research in multiple domains, such as scientific analysis systems, embedded systems, and information systems.

Paul Clements is the Vice President of Customer Success at BigLever Software, Inc., where he works to spread the adoption of systems and software product line engineering. Prior to this position, he was Senior Member of the Technical Staff at the SEI, where for 17 years he was leader or co-leader of projects in software product line engineering and software architecture documentation and analysis. Other books Paul has coauthored include *Documenting Software Architectures: Views and Beyond, Second Edition* (Addison-Wesley, 2011), *Evaluating Software Architectures: Methods and Case Studies* (Addison-Wesley, 2002), and *Software Product Lines: Practices and Patterns* (Addison-Wesley, 2002). In addition, he has also published dozens of papers in software engineering reflecting his long-standing interest in the design and specification of challenging software systems. Paul was a founding member of the IFIP WG2.10 Working Group on Software Architecture.

Rick Kazman is a Professor at the University of Hawaii and a Visiting Scientist (and former Senior Member of the Technical Staff) at the SEI. He is a coauthor of *Evaluating Software Architectures: Methods and Case Studies* (Addison-Wesley, 2002) and author of more than 100 technical papers. Rick's primary research interests focus on software architecture, design and analysis, software visualization, and software engineering economics. Rick has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method), and the Dali architecture reverse-engineering tool.

Index

- AADL (Architecture Analysis and Design Language), [354](#)
- Abstract common services tactic, [124](#)
- Abstract data sources for testability, [165](#)
- Abstract Syntax Tree (AST) analyzers, [386](#)
- Abstraction, architecture as, [5–6](#)
- Acceptance testing, [372](#)
- Access
 - basis sets, [261](#)
 - network, [504](#)
- access_read relationship, [384](#)
- access_write relationship, [384](#)
- ACID (atomic, consistent, isolated, and durable) properties, [95](#)
- Acknowledged system of systems, [106](#)
- Active redundancy, [91](#), [256–259](#)
- ActiveMQ product, [224](#)
- Activities
 - competence, [468](#)
 - test, [374–375](#)
- Activity diagrams for traces, [353](#)
- Actors tactic, [152–153](#)
- Adams, Douglas, [437](#)
- ADD method. See [Attribute-Driven Design \(ADD\) method](#)
- Add-ons, [491–492](#)
- ADLs (architecture description languages), [330](#)
- Adolphus, Gustavus, [42](#)
- Adoption strategies, [494–496](#)
- Adventure Builder system, [224](#), [226](#), [237](#)
- Aggregation for usability, [180](#)
- Agile projects, [533](#)
 - architecture example, [283–285](#)
 - architecture methods, [281–283](#)
 - architecture overview, [277–281](#)
 - description, [44–45](#)
 - documenting, [356–357](#)
 - guidelines, [286–287](#)
 - introduction, [275–277](#)
 - patterns, [238](#)
 - requirements, [56](#)
 - summary, [287–288](#)
- AIC (Architecture Influence Cycle)
 - description, [58](#)
 - Vasa ship, [43](#)
- Air France flight [447](#), [192](#)
- Air traffic control systems, [366–367](#)
- Allen, Woody, [79](#)
- Allocated to relation
 - allocation views, [339–340](#)
 - deployment structure, [14](#)
 - multi-tier pattern, [237](#)

Allocation of responsibilities category

- ASRs, [293](#)
- availability, [96](#)
- interoperability, [114](#)
- modifiability, [126](#)
- performance, [143](#)
- quality design decisions, [73](#)
- security, [154](#)
- testability, [169](#)
- usability, [181](#)

Allocation patterns

- map-reduce, [232–235](#)
- miscellaneous, [238](#)
- multi-tier, [235–237](#)

Allocation structures, [5](#), [11](#), [14](#)

Allocation views, [339–340](#)

Allowed-to-use relationship, [206–207](#)

Alpha testing, [372](#)

Alternatives, evaluating, [398](#)

Amazon service-level agreements, [81](#), [522](#)

Analysis

- architecture, [47–48](#)
- ATAM, [408–409](#), [411](#)
- availability, [255–259](#)
- back-of-the-envelope, [262–264](#)
- conformance by, [389–392](#)
- economic. See [Economic analysis](#)
- outsider, [399](#)
- performance, [252–255](#)

Analysts, [54](#)

Analytic model space, [259–260](#)

Analytic perspective on up-front work vs. agility, [279–281](#)

Analytic redundancy tactic, [90](#)

AND gate symbol, [84](#)

Anonymizing test data, [171](#)

Antimissile system, [104](#)

Apache web server, [528](#), [531](#)

Approaches

- ATAM, [407–409](#), [411](#)
- CIA, [147–148](#)
- Lightweight Architecture Evaluation, [416](#)

Architects

- background and experience, [51–52](#)
- cloud environments, [520–523](#)
- communication with, [29](#)
- competence, [459–467](#)
- description and interests, [54](#)
- duties, [460–464](#)
- knowledge, [466–467](#)
- responsibilities, [422–423](#)
- skills, [463](#), [465](#)
- test role, [375–376](#)

Architectural structures

allocation, [14](#)
component-and-connector, [13–14](#)
documentation, [17–18](#)
insight from, [11–12](#)
kinds, [10–11](#)
limiting, [17](#)
module, [12–13](#)
relating to each other, [14, 16–17](#)
selecting, [17](#)
table of, [15](#)
views, [9–10](#)

Architecturally significant requirements (ASRs), [46–47, 291–292](#)
ADD method, [320–321](#)
from business goals, [296–304](#)
designing to, [311–312](#)
interviewing stakeholders, [294–296](#)
from requirements documents, [292–293](#)
utility trees for, [304–307](#)

Architecture
Agile projects. See [Agile projects](#)
analyzing, [47–48](#)
availability. See [Availability](#)
business context, [49–51](#)
changes, [27–28](#)
cloud. See [Cloud environments](#)
competence. See [Competence](#)
conceptual integrity of, [189](#)
design. See [Design and design strategy](#)
documenting. See [Documentation](#)
drivers in PALM, [305](#)
economics. See [Economic analysis](#)
evaluation. See [Evaluation](#)
implementation. See [Implementation](#)
influences, [56–58](#)
in life cycle, [271–274](#)
management. See [Management and governance](#)
modifiability. See [Modifiability](#)
patterns. See [Patterns](#)
performance. See [Performance](#)
product lines. See [Software product lines](#)
product reuse, [483–484](#)
QAW drivers, [295](#)
QAW plan presentation, [295](#)
quality attributes. See [Quality attributes](#)
reconstruction and conformance. See [Reconstruction and conformance](#)
requirements. See [Architecturally significant requirements \(ASRs\)](#); [Requirements](#)
security. See [Security](#)
structures. See [Architectural structures](#)
tactics. See [Tactics](#)
testability. See [Testability](#)
usability. See [Usability](#)

Architecture Analysis and Design Language (AADL), [354](#)
Architecture-centric projects, [279](#)

Architecture description languages (ADLs), [330](#)
Architecture Influence Cycle (AIC)
 description, [58](#)
 Vasa ship, [43](#)
Architecture Tradeoff Analysis Method (ATAM), [48](#), [283](#), [400](#)
 approaches, [407–409](#), [411](#)
 business drivers, [404–405](#)
 example exercise, [411–414](#)
 outputs, [402–403](#)
 participants, [400–401](#)
 phases, [403–404](#)
 presentation, [403–406](#)
 results, [411](#)
 scenarios, [408](#), [410](#)
 steps, [404–411](#)
Ariane 5 explosion, [192](#)
Aristotle, [185](#)
Arrival pattern for events, [133](#)
Artifacts
 availability, [85–86](#)
 in evaluation, [399](#)
 interoperability, [107–108](#)
 modifiability, [119–120](#)
 performance, [134](#)
 product reuse, [484](#)
 quality attributes expressions, [69–70](#)
 security, [148](#), [150](#)
 testability, [162–163](#)
 usability, [176](#)
 variability, [489](#)
ASP.NET framework, [215](#)
Aspects
 for testability, [167](#)
 variation mechanism, [492](#)
ASRs. See [Architecturally significant requirements \(ASRs\)](#)
Assembly connectors in UML, [369](#)
Assertions for system state, [166](#)
Assessment goals, [469](#)
Assessment of competence, [469–472](#), [474–475](#)
Assign utility
 CBAM, [446](#)
 NASA ECS project, [452](#)
AST (Abstract Syntax Tree) analyzers, [386](#)
Asymmetric flow in client-server pattern, [218](#)
Asynchronous messaging, [223](#), [225](#)
ATAM. See [Architecture Tradeoff Analysis Method \(ATAM\)](#)
ATM (automatic teller machine) banking system, [219](#)
Atomic, consistent, isolated, and durable (ACID) properties, [95](#)
Attachment relation
 broker pattern, [211](#)
 client-server pattern, [218](#)
 component-and-connector structures, [13](#)
 pipe-and-filter pattern, [216](#)

publish-subscribe pattern, [227](#)
shared-data pattern, [231](#)

Attachments in component-and-connector views, [336–337](#)

Attribute-Driven Design (ADD) method, [316](#)

- ASRs, [320–321](#)
- element choice, [318–319](#)
- element design solution, [321](#)
- inputs, [316](#)
- output, [317–318](#)
- repeating steps, [324](#)
- verify and refine requirements step, [321–323](#)

Attributes. See [Quality attributes](#)

Audiences for documentation, [328–329](#)

Auditor checklists, [260](#)

Audits, [153](#)

- Authenticate actors tactic, [152](#)
- Authentication in CIA approach, [148](#)
- Authorization in CIA approach, [148](#)
- Authorize actors tactic, [152](#)

Automated delivery in Metropolis model, [535](#)

Automatic reallocation of IP addresses, [516](#)

Automatic scaling, [516](#)

Automatic teller machine (ATM) banking system, [219](#)

Automation for testability, [171–172](#)

AUTOSAR framework, [364](#)

Availability

- analytic model space, [259](#)
- analyzing, [255–259](#)
- broker pattern, [240](#)
- calculations, [259](#)
- CAP theorem, [523](#)
- CIA approach, [147](#)
- cloud, [521](#)
- design checklist, [96–98](#)
- detect faults tactic, [87–91](#)
- general scenario, [85–86](#)
- introduction, [79–81](#)
- planning for failure, [82–85](#)
- prevent faults tactic, [94–95](#)
- recover-from-faults tactics, [91–94](#)
- summary, [98–99](#)
- tactics overview, [87](#)

Availability of resources tactic, [136](#)

Availability quality attribute, [307](#)

Availability zones, [522](#)

Avižienis, Algirdas, [79](#)

Back door reviews, [544–545](#)

Back-of-the-envelope analysis, [262–264](#)

Background of architects, [51–52](#)

Bank application, [391–392](#)

Base mechanisms in cloud, [509–514](#)

Basis sets for quality attributes, [261](#)
BDUF (Big Design Up Front) process, [278](#)
Behavior
 documenting, [351–354](#)
 element, [347](#)
 in software architecture, [6–7](#)
Benefit in economic analysis, [441–442](#)
Benkler, Yochai, [528](#)
Beta testing, [372](#)
Big bang integration, [371](#)
Big bang models, [495–496](#)
Big Design Up Front (BDUF) process, [278](#)
BigTable database system, [518](#)
Binder, Robert, [167](#)
Binding
 late, [385, 388](#)
 modifiability, [124–125](#)
 user interface, [178](#)
Binding time category
 ASRs, [293](#)
 availability, [98](#)
 interoperability, [115](#)
 modifiability, [122, 127](#)
 performance, [144](#)
 quality design, [75–76](#)
 security, [156](#)
 testability, [170](#)
 usability, [182](#)
BitTorrent networks, [221](#)
Black-box testing, [372–373](#)
“Blind Men and the Elephant” (Saxe), [379](#)
Blocked time in performance, [136](#)
Blogger website, [528](#)
Boehm, Barry, [279, 281, 286, 288](#)
Booch, Grady, [286](#)
Boolean logic diagrams, [83](#)
Bottom-up adoption, [495](#)
Bottom-up analysis mode, [284](#)
Bottom-up schedules, [420–421](#)
Bound execution times tactic, [138](#)
Bound queue sizes tactic, [139](#)
Boundaries in ADD method, [317](#)
Box-and-line drawings
 as architectures, [6](#)
 component-and-connector views, [338](#)
BPEL (Business Process Execution Language), [108](#)
Brainstorming
 ATAM, [410](#)
 Lightweight Architecture Evaluation, [416](#)
 QAW, [295](#)
Branson, Richard, [443](#)
Breadth first ADD strategy, [319](#)
Brewer, Eric, [522](#)

Broadcast-based publish-subscribe pattern, [229](#)

Broker pattern

- availability, [255–259](#)
- description, [210–212](#)
- weaknesses, [240–242](#)

Brooks, Fred, [47](#), [419](#)

Buley, Taylor, [147](#)

Bureaucracy in implementation, [427](#)

Bush, Vannevar, [397](#)

Business cases in project life-cycle context, [46](#)

Business context

- architecture influence on, [58](#)
- architectures and business goals, [49–50](#)

Business drivers

- ATAM, [404–405](#)
- Lightweight Architecture Evaluation, [416](#)
- PALM method, [305](#)

Business goals

- ASRs from, [296–304](#)
- assessment, [469](#)
- ATAM, [402](#)
- business context, [49–50](#)
- capturing, [304](#)
- categorization, [297–299](#)
- evaluation process, [400](#)
- expressing, [299–301](#)
- general scenario, [301–303](#)
- PALM method, [305](#)
- views for, [332](#)

Business managers, [54](#)

Business/mission presentation in QAW, [295](#)

Business Process Execution Language (BPEL), [108](#)

Business process improvements as business goal, [299](#)

Business-related architect skills, [465](#)

C&C structures. See [Component-and-connector \(C&C\) patterns and structures](#)

Caching tactic, [139](#)

Callbacks in Model-View-Controller pattern, [214](#)

Calls relationship in view extraction, [384](#)

Cancel command, [179](#)

CAP theorem, [518](#), [522–523](#)

Capture scenarios for quality attributes, [196–197](#)

Capturing

- ASRs in utility trees, [304–307](#)
- business goals, [304–307](#)

Catastrophic failures, [82](#)

Categorization of business goals, [297–299](#)

CBAM. See [Cost Benefit Analysis Method \(CBAM\)](#)

Change

- documenting, [355–356](#)
- modifiability. See [Modifiability](#)
- reasoning and managing, [27–28](#)

Change control boards, [427](#)
Change default settings tactic, [153](#)
Chaos Monkey, [160–161](#)
Chaucer, Geoffrey, [459](#)
Check-in, syncing at, [368](#)
Choice of technology category
 ASRs, [293](#)
 availability, [98](#)
 interoperability, [115](#)
 modifiability, [127](#)
 performance, [144](#)
 security, [156](#)
 testability, [170](#)
 usability, [182](#)
CIA (confidentiality, integrity, and availability) approach, [147–148](#)
City analogy in Metropolis model, [536](#)
class_contains_method relationship, [384](#)
class_is_subclass_of_class relationship, [384](#)
Class structure, [13](#)
Classes in testability, [167](#)
Clements, Paul, [66](#)
Client-server patterns, [19, 217–219](#)
Client-side proxies, [211](#)
Clients
 broker pattern, [211](#)
 simulators, [265](#)
Clone-and-own practice, [482–483](#)
Cloud environments
 architecting in, [520–523](#)
 availability, [521](#)
 base mechanisms, [509–514](#)
 database systems, [517–520](#)
 definitions, [504–505](#)
 deployment models, [506](#)
 economic justification, [506–509](#)
 equipment utilization, [508–509](#)
 IaaS model, [515–517](#)
 introduction, [503–504](#)
 multi-tenancy applications, [509](#)
 PaaS model, [517](#)
 performance, [521](#)
 security, [520–521](#)
 service models, [505–506](#)
 summary, [524](#)
Cluster managers, [515](#)
CMG (Computer Measurement Group), [524](#)
Co-located teams
 Agile, [277](#)
 coordination, [427](#)
Cockburn, Alistair, [287](#)
COCOMO II (COnstructive COst MOdel II) scale factor, [279](#)
Code
 architecture consistency, [366–368](#)

design in, [364](#)
KSLOC, [279–281](#)
mapping to, [334](#)
security, [157](#)
templates, [365–367](#)

Cohesion
in modifiability, [121–123](#)
in testability, [167](#)

Cold spares, [92, 256–259](#)

Collaborative system of systems, [106](#)

Collating scenarios
CBAM, [445](#)
NASA ECS project, [451](#)

COMBINATION gate symbol, [84](#)

Combining views, [343–345](#)

Commercial implementations of map-reduce patterns, [234](#)

Common Object Request Broker Architecture (CORBA), [212](#)

Communicates with relation, [237](#)

Communication
Agile software development, [277](#)
architect skills, [465](#)
architecture, [47](#)
documentation for, [329](#)
global development, [425](#)
stakeholder, [29–31](#)

Communication diagrams for traces, [353](#)

Communications views, [341](#)

Community clouds, [506](#)

Compatibility in component-and-connector views, [336](#)

Compatibility quality attribute, [193](#)

Competence
activities, [468](#)
architects, [459–467](#)
assessment, [469–472, 474–475](#)
assessment goals, [469](#)
introduction, [459–460](#)
models, [476](#)
questions, [470, 472–474](#)
software architecture organizations, [467–475](#)
summary, [475](#)

Competence center patterns, [19, 238](#)

Competence set tactic, [95](#)

Complexity
broker pattern, [211](#)
quality attributes, [71](#)
in testability, [167–168](#)

Component-and-connector (C&C) patterns and structures, [5, 10–11](#)
broker, [210–212](#)
client-server, [217–219](#)
Model-View-Controller, [212–215](#)
peer-to-peer, [220–222](#)
pipe-and-filter, [215–217](#)
publish-subscribe, [226–229](#)

service-oriented architecture, [222–226](#)
shared-data, [230–231](#)
types, [13–14](#)
views, [335–339](#), [344](#), [406](#)

Components, [5](#)
independently developed, [35–36](#)
replacing for testability, [167](#)
substituting in variation mechanism, [492](#)

Comprehensive models for behavior documentation, [351](#), [353–354](#)

Computer Measurement Group (CMG), [524](#)

Computer science knowledge of architects, [466](#)

Concepts and terms, [368–369](#)

Conceptual integrity of architecture, [189](#)

Concrete quality attribute scenarios, [69](#)

Concurrency
component-and-connector views, [13–14](#), [337](#)
handling, [132–133](#)

Condition monitoring tactic, [89](#)

Confidence in usability, [175](#)

Confidentiality, integrity, and availability (CIA) approach, [147–148](#)

Configurability quality attribute, [307](#)

Configuration manager roles, [422](#)

Configurators, [492](#)

Conformance, [380–381](#)
by analysis, [389–392](#)
architectural, [48](#)
by construction, [389](#)

Conformance checkers, [54](#)

Conformity Monkey, [161](#)

Connectors
component-and-connector views, [335–339](#)
multi-tier pattern, [236](#)
peer-to-peer systems, [220](#)
REST, [225](#)
UML, [369](#)

Consistency
CAP theorem, [523](#)
code and architecture, [366–368](#)
databases, [520](#)

Consolidation in QAW, [295](#)

Constraints
ADD method, [322–323](#)
allocation views, [339](#)
broker pattern, [211](#)
client-server pattern, [218](#)
component-and-connector views, [337](#)
conformance, [390](#)
defining, [32–33](#)
layered pattern, [207](#)
map-reduce patterns, [235](#)
Model-View-Controller pattern, [213](#)
modular views, [333](#)
multi-tier pattern, [236–237](#)

peer-to-peer pattern, [222](#)
pipe-and-filter pattern, [216](#)
publish-subscribe pattern, [227](#)
requirements, [64–65](#)
service-oriented architecture pattern, [225](#)
shared-data pattern, [231](#)
Construction, conformance by, [389](#)
COnstructive COst MOdel II (COCOMO II) scale factor, [279](#)
Content-based publish-subscribe pattern, [229](#)
Contention for resources tactic, [136](#)
Context diagrams
 ATAM presentations, [406](#)
 in documentation, [347](#)
Contexts
 architecture influence, [56–58](#)
 business, [49–51, 58](#)
 decision-making, [438–439](#)
 professional, [51–52](#)
 project life-cycle, [44–48](#)
 in relationships, [204–205](#)
 stakeholders, [52–55](#)
 summary, [59](#)
 technical, [40–43](#)
 thought experiments, [263](#)
 types, [39–40](#)
Contextual factors in evaluation, [399–400](#)
Continuity as business goal, [298](#)
Control relation in map-reduce patterns, [235](#)
Control resource demand tactic, [137–138](#)
Control tactics for testability, [164–167](#)
Controllers in Model-View-Controller pattern, [213–214](#)
Conway's law, [38](#)
Coordination model category
 ASRs, [293](#)
 availability, [96](#)
 global development, [426](#)
 interoperability, [114](#)
 modifiability, [126](#)
 performance, [143](#)
 quality design decisions, [73–74](#)
 security, [155](#)
 testability, [169](#)
 usability, [181](#)
CORBA (Common Object Request Broker Architecture), [212](#)
Core asset units, [497](#)
Core requirements, [531–532](#)
Core vs. periphery in Metropolis model, [534](#)
Correlation logic for faults, [81](#)
Cost Benefit Analysis Method (CBAM), [442](#)
 cost determination, [444](#)
 results, [456–457](#)
 steps, [445–447](#)
 utility curve determination, [442–443](#)

variation points, [448–450](#)
weighting determination, [444](#)

Costs
CBAM, [444](#)
of change, [118](#)
estimates, [34](#)
global development, [423–424](#)
independently developed components for, [36](#)
power, [507](#)
resources, [244](#)
thought experiments, [263](#)
value for, [442](#)

Costs to complete measure, [430](#)

Coupling
in modifiability, [121–124](#)
in testability, [167](#)

Crashes and availability, [85](#)

Credit cards, [147, 157, 260, 268](#)

Crisis, syncing at, [368](#)

Criteria for ASRs, [306](#)

Crowd management in Metropolis model, [534](#)

Crowdsourcing, [528](#)

CRUD operations, [109](#)

CruiseControl tool, [172](#)

Crystal Clear method, [44, 287](#)

Cummins, Inc., [480, 490](#)

Cunningham, Ward, [286](#)

Customers
communication with, [29](#)
edge-dominant systems, [529](#)

Customization of user interface, [180](#)

Darwin, Charles, [275](#)

Data Access Working Group (DAWG), [451](#)

Data accessors in shared-data pattern, [230–231](#)

Data latency, utility trees for, [306](#)

Data model category, [13](#)
ASRs, [293](#)
availability, [96](#)
interoperability, [114](#)
modifiability, [126](#)
performance, [143](#)
quality design decisions, [74](#)
security, [155](#)
testability, [169](#)
usability, [182](#)

Data reading and writing in shared-data pattern, [230–231](#)

Data replication, [139](#)

Data sets
map-reduce pattern, [232–233](#)
for testability, [170–171](#)

Data stores in shared-data pattern, [230–231](#)

Data transformation systems, [215](#)
Database administrators, [54](#)
Database systems
 cloud, [517–520](#)
 in reconstruction, [386–387](#)
DataNodes, [512–514](#)
DAWG (Data Access Working Group), [451](#)
Deadline monotonic prioritization strategy, [140](#)
Deadlines in processing, [134](#)
Debugging brokers, [211](#)
Decision makers on ATAM teams, [401](#)
Decision-making context, [438–439](#)
Decisions
 evaluating, [398](#)
 mapping to quality requirements, [402–403](#)
 quality design, [72–76](#)
Decomposition
 description, [311–312](#)
 module, [5, 12, 16](#)
 views, [16, 343, 345](#)
Dedicated finite resources, [530](#)
Defects
 analysis, [374](#)
 eliminating, [486](#)
 tracking, [430](#)
Defer binding
 modifiability, [124–125](#)
 user interface, [178](#)
Degradation tactic, [93](#)
Delegation connectors, [369](#)
Demilitarized zones (DMZs), [152](#)
Denial-of-service attacks, [79, 521, 533](#)
Dependencies
 basis set elements, [261](#)
 on computations, [136](#)
 intermediary tactic for, [123–124](#)
 user interface, [178](#)
Dependent events in probability, [257](#)
Depends-on relation
 layered pattern, [207](#)
 modules, [332–333](#)
Deploy on relation, [235](#)
Deployability attribute, [129, 187](#)
Deployers, [54](#)
Deployment models for cloud, [506](#)
Deployment structure, [14](#)
Deployment views
 ATAM presentations, [406](#)
 combining, [345](#)
 purpose, [332](#)
Depth first ADD strategy, [319](#)
Design and design strategy, [311](#)
 ADD. See [Attribute-Driven Design \(ADD\) method](#)

architecturally significant requirements, [311–312](#)
in code, [364](#)
decomposition, [311–312](#)
early decisions, [31–32](#)
generate and test process, [313–316](#)
initial hypotheses, [314–315](#)
next hypotheses, [315](#)
quality attributes, [197](#)
summary, [325](#)
test choices, [315](#)

Design checklists
availability, [96–98](#)
design strategy hypotheses, [315](#)
interoperability, [114–115](#)
modifiability, [125–127](#)
performance, [142–144](#)
quality attributes, [199](#)
security, [154–156](#)
summary, [183](#)
testability, [169–170](#)
usability, [181–182](#)

Designers
description and interests, [54](#)
evaluation by, [397–398](#)

Detect attacks tactic, [151](#)
Detect faults tactic, [87–91](#)
Detect intrusion tactic, [151](#)
Detect message delay tactic, [151](#)
Detect service denial tactic, [151](#)
Deutsche Bank, [480](#)

Developers
edge-dominant systems, [529](#)
roles, [422](#)

Development
business context, [50–51](#)
global, [423–426](#)
incremental, [428](#)
project life-cycle context, [44–45](#)
tests, [374](#)

Development distributability attribute, [186](#)

Deviation
failure from, [80](#)
measuring, [429](#)

Devices in ADD method, [317](#)

DiNucci, Darcy, [527](#)

dir_contains_dir relationship, [384](#)
dir_contains_file relationship, [384](#)

Directed system of systems, [106](#)

Directories in documentation, [349](#)

DiscoTect system, [391](#)

Discover service tactic, [111](#)

Discovery in interoperability, [105](#)

Discovery services, [533](#)

Distributed computing, [221](#)
Distributed development, [427](#)
Distributed testing in Metropolis model, [535](#)
DMZs (demilitarized zones), [152](#)
DNS (domain name server), [514](#)
Doctor Monkey, [161](#)
Documentation
 Agile development projects, [356–357](#)
 architect duties, [462](#)
 architectural structures, [17–18](#)
 architecture, [47, 347–349](#)
 behavior, [351–354](#)
 changing architectures, [355–356](#)
 distributed development, [427](#)
 global development, [426](#)
 introduction, [327–328](#)
 notations, [329–331](#)
 online, [350](#)
 packages, [345–351](#)
 patterns, [350–351](#)
 and quality attributes, [354–355](#)
 services, [533](#)
 summary, [359](#)
 uses and audiences for, [328–329](#)
 views. See [Views](#)
 YAGNI, [282](#)
Documentation maps, [347–349](#)
Documents, control information, [347](#)
Domain decomposition, [315](#)
Domain knowledge of architects, [467](#)
Domain name server (DNS), [514](#)
Drivers
 ATAM, [404–405](#)
 Lightweight Architecture Evaluation, [416](#)
 PALM method, [305](#)
 QAW, [295](#)
DSK (Duties, Skills, and Knowledge) model of competence, [476](#)
Duke's Bank application, [391–392](#)
Duties
 architects, [460–464](#)
 competence, [472](#)
 professional context, [51](#)
Duties, Skills, and Knowledge (DSK) model of competence, [476](#)
Dynamic allocation views, [340](#)
Dynamic analysis with fault trees, [83](#)
Dynamic priority scheduling strategies, [140–141](#)
Dynamic structures, [5](#)
Dynamic system information, [385–386](#)

Earliest-deadline-first scheduling strategy, [141](#)
Early design decisions, [31–32](#)
Earth Observing System Data Information System (EOSDIS) Core System (ECS). See [NASA ECS project](#)

eBay, [234](#)
EC2 cloud service, [81](#), [160](#), [522](#), [532](#)
Eclipse platform, [228](#)
Economic analysis
 basis, [439–442](#)
 benefit and normalization, [441–442](#)
 case study, [451–457](#)
 CBAM. See [Cost Benefit Analysis Method \(CBAM\)](#)
 cost value, [442](#)
 decision-making context, [438–439](#)
 introduction, [437](#)
 scenario weighting, [441](#)
 side effects, [441](#)
 summary, [457](#)
 utility-response curves, [439–441](#)
Economics
 cloud, [506–509](#)
 issues, [543](#)
Economies of scale in cloud, [507–508](#)
Ecosystems, [528–530](#)
ECS system. See [NASA ECS project](#)
Edge-dominant systems, [528–530](#)
Edison, Thomas, [203](#)
eDonkey networks, [221](#)
Education, documentation as, [328–329](#)
Effective resource utilization, [187](#)
Effectiveness category for quality, [189](#)
Efficiency category for quality, [189–190](#)
Einstein, Albert, [175](#)
EJB (Enterprise Java Beans), [212](#)
Elasticity, rapid, [504–505](#)
Elasticity property, [187](#)
Electric grids, [106](#)
Electricity, [191](#), [570](#)
Electronic communication in global development, [426](#)
Elements
 ADD method, [318–319](#)
 allocation views, [339–340](#)
 broker pattern, [211](#)
 catalogs, [346–347](#)
 client-server pattern, [218](#)
 component-and-connector views, [337](#)
 defined, [5](#)
 layered pattern, [207](#)
 map-reduce patterns, [235](#)
 mapping, [75](#)
 Model-View-Controller pattern, [213](#)
 modular views, [333](#)
 multi-tier pattern, [237](#)
 peer-to-peer pattern, [222](#)
 pipe-and-filter pattern, [216](#)
 product reuse, [484](#)
 publish-subscribe pattern, [227](#)

service-oriented architecture pattern, [225](#)
shared-data pattern, [231](#)

Employees
as goal-object, [302](#)
responsibilities to, [299](#)

Enabling quality attributes, [26–27](#)

Encapsulation tactic, [123](#)

Encrypt data tactic, [152](#)

End users in edge-dominant systems, [529](#)

Enterprise architecture vs. system architecture, [7–8](#)

Enterprise Java Beans (EJB), [212](#)

Enterprise resource planning (ERP) systems, [228](#)

Enterprise service bus (ESB), [223, 225, 369](#)

Environment
ADD method, [317](#)
allocation views, [339–340](#)
availability, [85–86](#)
business goals, [300](#)
interoperability, [107–108](#)
modifiability, [119–120](#)
performance, [134](#)
quality attributes expressions, [68–70](#)
security, [149–150](#)
technical context, [41–42](#)
testability, [162–163](#)
usability, [176](#)
variability, [489](#)

Environmental change as business goal, [299](#)

ERP (enterprise resource planning) systems, [228](#)

Errors, [80](#)
core handling of, [532](#)
detection by services, [533](#)
error-handling views, [341](#)
in usability, [175](#)

ESB (enterprise service bus), [223, 225, 369](#)

Escalating restart tactic, [94](#)

Estimates, cost and schedule, [34](#)

Evaluation
architect duties, [462–463](#)
architecture, [47–48](#)
ATAM. See [Architecture Tradeoff Analysis Method \(ATAM\)](#)
contextual factors, [399–400](#)
by designer, [397–398](#)
Lightweight Architecture Evaluation, [415–417](#)
outsider analysis, [399](#)
peer review, [398–399](#)
questions, [472](#)
software product lines, [493–494](#)
summary, [417](#)

Evaluators, [54](#)

Event bus in publish-subscribe pattern, [227](#)

Events
Model-View-Controller pattern, [214](#)

performance, [131](#), [133](#)
probability, [257](#)

Eventual consistency model, [168](#), [523](#)
Evolutionary prototyping, [33](#)–[34](#)
Evolving software product lines, [496](#)–[497](#)
Exception detection tactic, [90](#)
Exception handling tactic, [92](#)
Exception prevention tactic, [95](#)
Exception views, [341](#)
Exchanging information via interfaces, [104](#)–[105](#)
EXCLUSIVE OR gate symbol, [84](#)
Executable assertions for system state, [166](#)
Execution of tests, [374](#)
Exemplar systems, [485](#)
Exercise conclusion in PALM method, [305](#)
Existing systems in design, [314](#)
Expected quality attribute response levels, [453](#)
Experience of architects, [51](#)–[52](#)
Experiments in quality attribute modeling, [264](#)–[265](#)
Expressing business goals, [299](#)–[301](#)
Extensibility quality attribute, [307](#)
Extensible programming environments, [228](#)
Extension points for variation, [491](#)
External sources for product lines, [496](#)
External system representatives, [55](#)
External systems in ADD method, [317](#)
Externalizing change, [125](#)
Extract-transform-load functions, [235](#)
Extraction, raw view, [382](#)–[386](#)
Extreme Programming development methodology, [44](#)

Facebook, [527](#)–[528](#)
 map-reduce patterns, [234](#)
 users, [518](#)

Fail fast principle, [522](#)

Failure Mode, Effects, and Criticality Analysis (FMECA), [83](#)–[84](#)

Failures, [80](#)
 availability. *See Availability*
 planning for, [82](#)–[85](#)
 probabilities and effects, [84](#)–[85](#)

Fairbanks, George, [279](#), [364](#)

Fallbacks principle, [522](#)

Fault tree analysis, [82](#)–[84](#)

Faults, [80](#)
 correlation logic, [81](#)
 detection, [87](#)–[91](#)
 prevention, [94](#)–[95](#)
 recovery from, [91](#)–[94](#)

Feature removal principle, [522](#)

FIFO (first-in/first-out) queues, [140](#)

File system managers, [516](#)

Filters in pipe-and-filter pattern, [215](#)–[217](#)

Financial objectives as business goal, [298](#)
Finding violations, [389–392](#)
Fire-and-forget information exchange, [223](#)
Firefox, [531](#)
First-in/first-out (FIFO) queues, [140](#)
First principles from tactics, [72](#)
Fixed-priority scheduling, [140](#)
Flex software development kit, [215](#)
Flexibility
 defer binding tactic, [124](#)
 independently developed components for, [36](#)
Flickr service, [527, 536](#)
Flight control software, [192–193](#)
FMEA (Failure Mode, Effects, and Criticality Analysis), [83–84](#)
Focus on architecture in Metropolis model, [534–535](#)
Follow-up phase in ATAM, [403–404](#)
Folsonomy, [528](#)
Ford, Henry, [479](#)
Formal documentation notations, [330](#)
Frameworks
 design strategy hypotheses, [314–315](#)
 implementation, [364–365](#)
Frankl, Viktor E., [63](#)
Freedom from risk category for quality, [189](#)
Functional redundancy tactic, [90](#)
Functional requirements, [64, 66](#)
Functional responsibility in ADD method, [322–323](#)
Functional suitability quality attribute, [193](#)
Functionality
 component-and-connector views, [336](#)
 description, [65](#)
Fused views, [388–389](#)

Gamma, E., [212](#)
Gate symbols, [83–84](#)
General Motors product line, [487](#)
Generalization structure, [13](#)
Generate and test process, [313–316](#)
Generators of variation, [492](#)
Get method for system state, [165](#)
Global development, [423–426](#)
Global metrics, [429–430](#)
Gnutella networks, [221](#)
Goal components in business goals, [300](#)
Goals. See [Business goals](#)
Goldberg, Rube, [102](#)
Good architecture, [19–21](#)
Good enough vs. perfect, [398](#)
Google
 database system, [518](#)
 Google App Engine, [517](#)
 Google Maps, [105–107](#)

greenhouse gas from, [190–191](#)
map-reduce patterns, [234](#)
power sources, [507](#)
Governance, [430–431](#)
Government, responsibilities to, [299](#)
Graceful degradation, [522](#)
Graphical user interfaces in publish-subscribe pattern, [228](#)
Gray-box testing, [373](#)
Green computing, [190–191](#)
Greenspan, Alan, [443](#)
Growth and continuity as business goal, [298](#)
Guerrilla movements, [543–544](#)

Hadoop Distributed File System (HDFS), [512](#)
Hardware costs for cloud, [507](#)
Harel, David, [353](#)
Harnesses for tests, [374](#)
Hazard analysis, [82](#)
Hazardous failures, [82](#)
HBase database system, [518–519](#)
HDFS (Hadoop Distributed File System), [512](#)
Heartbeat tactic, [89, 256, 408](#)
Helm, R., [212](#)
Hewlett-Packard, [480](#)
Hiatus stage in ATAM, [409](#)
High availability. See [Availability](#)
Highway systems, [142](#)
Horizontal scalability, [187](#)
Hot spare tactic, [91](#)
HTTP (HyperText Transfer Protocol), [219](#)
Hudson tool, [172](#)
Hufstedler, Shirley, [363](#)
Human body structure, [9](#)
Human Performance model of competence, [476](#)
Human Performance Technology model, [469–473](#)
Human resource management in global development, [425](#)
Hybertsson, Henrik, [42–43](#)
Hybrid clouds, [506](#)
Hydroelectric power station catastrophe, [188, 192](#)
Hypertext for documentation, [350](#)
HyperText Transfer Protocol (HTTP), [219](#)
Hypervisors, [510–512](#)
Hypotheses
 conformance, [390](#)
 design strategy, [314–315](#)
 fused views, [388](#)

IaaS (Infrastructure as a Service) model, [505–506, 515–517](#)
Identify actors tactic, [152](#)
Ignore faulty behavior tactic, [93](#)
Implementation, [363–364, 427](#)
 architect duties, [463](#)

code and architecture consistency, [366–368](#)
code templates, [365–367](#)
design in code, [364](#)
frameworks, [364–365](#)
incremental development, [428](#)
modules, [333–334](#)
structure, [14](#)
summary, [376](#)
testing, [370–376](#)
tracking progress, [428–429](#)
tradeoffs, [427](#)

Implementors, [55](#)

In-service software upgrade (ISSU), [92](#)

Includes relationship, [384](#)

Inclusion of elements for variation, [491](#)

Increase cohesion tactic, [123](#)

Increase competence set tactic, [95](#)

Increase efficiency tactic, [142](#)

Increase resource efficiency tactic, [138](#)

Increase resources tactic, [138–139](#), [142](#)

Increase semantic coherence tactic, [123](#), [239](#)

Incremental Commitment Model, [286](#)

Incremental development, [428](#)

Incremental integration, [371](#)

Incremental models in adoption strategies, [495–496](#)

Independent events in probability, [257](#)

Independently developed components, [35–36](#)

Inflexibility of methods, [277](#)

Inform actors tactic, [153](#)

Informal contacts in global development, [426](#)

Informal notations for documentation, [330](#)

Information handling skills, [465](#)

Information sharing in cloud, [520](#)

Infrastructure as a Service (IaaS) model, [505–506](#), [515–517](#)

Infrastructure in map-reduce patterns, [235](#)

Infrastructure labor costs in cloud, [507](#)

Inheritance variation mechanism, [492](#)

Inherits from relation, [13](#)

INHIBIT gate symbol, [84](#)

Inhibiting quality attributes, [26–27](#)

Initial hypotheses in design strategy, [314–315](#)

Inputs in ADD method, [316](#), [321–323](#)

Instantiate relation, [235](#)

Integration management in global development, [424](#)

Integration testing, [371–372](#)

Integrators, [55](#)

Integrity

- architecture, [189](#)
- CIA approach, [147](#)

Interchangeable parts, [35–36](#), [480](#)

Interfaces

- exchanging information via, [104–105](#)
- separating, [178](#)

Intermediary tactic, [123](#)
Intermediate states in failures, [80](#)
Internal sources of product lines, [496–497](#)
Internet Protocol (IP) addresses
 automatic reallocation, [516](#)
 overview, [514](#)
Interoperability
 analytic model space, [259](#)
 design checklist, [114–115](#)
 general scenario, [107–110](#)
 introduction, [103–106](#)
 service-oriented architecture pattern, [224](#)
 and standards, [112–113](#)
 summary, [115](#)
 tactics, [110–113](#)
Interpersonal skills, [465](#)
Interpolation in CBAM, [446](#)
Interviewing stakeholders, [294–296](#)
Introduce concurrency tactic, [139](#)
Invokes-services role, [335](#)
Involvement, [542–543](#)
Iowability, [195–196](#)
IP (Internet Protocol) addresses
 automatic reallocation, [516](#)
 overview, [514](#)
Is a relation, [332–333](#)
Is-a-submodule-of relation, [12](#)
Is an instance of relation, [13](#)
Is part of relation
 modules, [332–333](#)
 multi-tier pattern, [237](#)
ISO 25010 standard, [66, 193–195](#)
ISSU (in-service software upgrade), [92](#)
Iterative approach
 description, [44](#)
 reconstruction, [382](#)
 requirements, [56](#)

Janitor Monkey, [161](#)
JavaScript Object Notation (JSON) form, [519](#)
Jitter, [134](#)
Jobs, Steve, [311](#)
Johnson, R., [212](#)
JSON (JavaScript Object Notation) form, [519](#)
Just Enough Architecture (Fairbanks), [279, 364](#)

Keys in map-reduce pattern, [232](#)
Knowledge
 architects, [460, 466–467](#)
 competence, [472–473](#)
 professional context, [51](#)
Kroc, Ray, [291](#)

Kruchten, Philippe, [327](#)
KSLOC (thousands of source lines of code), [279–281](#)
Kundra, Vivek, [503](#)

Labor availability in global development, [423](#)
Labor costs
 cloud, [507](#)
 global development, [423](#)
Language, [542](#)
Larger data sets in map-reduce patterns, [234](#)
Late binding, [385](#), [388](#)
Latency
 CAP theorem, [523](#)
 performance, [133](#), [255](#)
 queuing models for, [198–199](#)
 utility trees for, [306](#)
Latency Monkey, [161](#)
Lattix tool, [387](#)
Lawrence Livermore National Laboratory, [71](#)
Layer bridging, [206](#)
Layer structures, [13](#)
Layer views in ATAM presentations, [406](#)
Layered patterns, [19](#), [205–210](#)
Layered views, [331–332](#)
Leaders on ATAM teams, [401](#)
Leadership skills, [464–465](#)
Learning issues in usability, [175](#)
Least-slack-first scheduling strategy, [141](#)
LePatner, Barry, [3](#)
Letterman, David, [443](#)
Levels
 failure, [258](#)
 restart, [94](#)
 testing, [370–372](#)
Leveson, Nancy, [200](#)
Lexical analyzers, [386](#)
Life cycle
 architecture in, [271–274](#)
 changes, [530–531](#)
 Metropolis model, [537](#)
 project. See [Project life-cycle context](#)
 quality attribute analysis, [265–266](#)
Life-cycle milestones, syncing at, [368](#)
Lightweight Architecture Evaluation method, [415–417](#)
Likelihood of change, [117](#)
Limit access tactic, [152](#)
Limit complexity tactic, [167](#)
Limit event response tactic, [137](#)
Limit exposure tactic, [152](#)
Limit structural complexity tactic, [167–168](#)
Linux, [531](#)
List-based publish-subscribe pattern, [229](#)

Load balancers, [139](#)
Local changes, [27–28](#)
Local knowledge of markets in global development, [423](#)
Localize state storage for testability, [165](#)
Locate tactic, [111](#)
Location independence, [504](#)
Lock computer tactic, [153](#)
Logical threads in concurrency, [13–14](#)

Macros for testability, [167](#)
Mailing lists in publish-subscribe pattern, [228](#)
Maintain multiple copies tactic, [142](#)
Maintain multiple copies of computations tactic, [139](#)
Maintain multiple copies of data tactic, [139](#)
Maintain system model tactic, [180](#)
Maintain task model tactic, [180](#)
Maintain user model tactic, [180](#)
Maintainability quality attribute, [195, 307](#)
Maintainers, [55](#)
Major failures, [82](#)
Manage event rate tactic, [142](#)
Manage resources tactic, [137–139](#)
Manage sampling rate tactic
 performance, [137](#)
 quality attributes, [72](#)
Management and governance
 architect skills, [464](#)
 governance, [430–431](#)
 implementing, [427–429](#)
 introduction, [419](#)
 measuring, [429–430](#)
 organizing, [422–426](#)
 planning, [420–421](#)
 summary, [432](#)
Management information in modules, [334](#)
Managers, communication with, [29](#)
Managing interfaces tactic, [111](#)
Manifesto for Agile software development, [276](#)
Map architectural strategies in CBAM, [446](#)
Map-reduce pattern, [232–235](#)
Mapping
 to requirements, [355, 402–403](#)
 to source code units, [334](#)
Mapping among architectural elements category
 ASRs, [293](#)
 availability, [97](#)
 interoperability, [114](#)
 modifiability, [127](#)
 performance, [144](#)
 quality design decisions, [75](#)
 security, [155](#)
 testability, [169](#)

usability, [182](#)
Maps, documentation, [347–349](#)
Market position as business goal, [299](#)
Marketability category for quality, [190](#)
Markov analysis, [83](#)
Matrixed team members, [422](#)
McGregor, John, [448](#)
Mean time between failures (MTBF), [80](#), [255–259](#)
Mean time to repair (MTTR), [80](#), [255–259](#)
Measured services, [505](#)
Measuring, [429–430](#)
Meetings
 global development, [426](#)
 progress tracking, [428](#)
Methods in product reuse, [484](#)
Metrics, [429–430](#)
Metropolis structure
 edge-dominant systems, [528–530](#)
 implications, [533–537](#)
Microsoft Azure, [517](#)
Microsoft Office [365](#), [509](#)
Migrates-to relation, [14](#)
Mill, John Stuart, [527](#)
Minimal cut sets, [83](#)
Minor failures, [82](#)
Missile defense system, [104](#)
Missile warning system, [192](#)
Mixed initiative in usability, [177](#)
Mobility attribute, [187](#)
Model driven development, [45](#)
Model-View-Controller (MVC) pattern
 overview, [212–215](#)
 performance analysis, [252–254](#)
 user interface, [178](#)
Models
 product reuse, [484](#)
 quality attributes, [197–198](#)
 transferable and reusable, [35](#)
Modifiability
 analytic model space, [259](#)
 component-and-connector views, [337](#)
 design checklist, [125–127](#)
 general scenario, [119–120](#)
 introduction, [117–119](#)
 managing, [27](#)
 ping/echo, [243](#)
 restrict dependencies tactic, [246](#)
 scheduling policy tactic, [244–245](#)
 summary, [128](#)
 tactics, [121–125](#)
 and time-to-market, [284](#)
 unit testing, [371](#)
 in usability, [179](#)

Modularity of core, [532](#)
Modules and module patterns, [10](#), [205–210](#)
 coupling, [121](#)
 decomposition structures, [5](#)
 description, [4–5](#)
 types, [12–13](#)
 views, [332–335](#), [406](#)
MongoDB database, [519](#)
Monitor relation in map-reduce patterns, [235](#)
Monitor tactic, [88–89](#)
Monitorability attribute, [188](#)
MoSCoW style, [292](#)
MSMQ product, [224](#)
MTBF (mean time between failures), [80](#), [255–259](#)
MTTR (mean time to repair), [80](#), [255–259](#)
Multi-tenancy
 cloud, [509](#), [520](#)
 description, [505](#)
Multi-tier patterns, [19](#), [235–237](#)
Multitasking, [132–133](#)
Musket production, [35–36](#)
MVC (Model-View-Controller) pattern
 overview, [212–215](#)
 performance analysis, [252–254](#)
 user interface, [178](#)
Mythical Man-Month (Brooks), [47](#)

NameNode process, [512–513](#)
Names for modules, [333](#)
NASA ECS project, [451](#)
 architectural strategies, [452–456](#)
 assign utility, [452](#)
 collate scenarios, [451](#)
 expected quality attribute response level, [453](#)
 prioritizing scenarios, [452](#)
 refining scenarios, [451–452](#)
Nation as goal-object, [302](#)
National Reconnaissance Office, [481](#)
.NET platform, [212](#)
Netflix
 cloud, [522](#)
 Simian Army, [160–161](#)
Network administrators, [55](#)
Networked services, [36](#)
Networks, cloud, [514](#)
Nightingale application, [306–307](#)
No effect failures, [82](#)
Node managers, [516](#)
Nokia, [480](#)
non-ASR requirements, [312–313](#)
Non-stop forwarding (NSF) tactic, [94](#)
Nondeterminism in testability, [168](#)

Nonlocal changes, [27](#)
Nonrepudiation in CIA approach, [148](#)
Nonisks in ATAM, [402](#)
Normalization
 databases, [520](#)
 economic analysis, [441–442](#)
NoSQL database systems, [518–520](#), [523](#)
NoSQL movement, [248](#)
Notations
 component-and-connector views, [338–339](#)
 documentation, [329–331](#)
Notifications
 failures, [80](#)
 Model-View-Controller pattern, [214](#)
NSF (non-stop forwarding) tactic, [94](#)
Number of events not processed measurement, [134](#)

Object-oriented systems
 in testability, [167](#)
 use cases, [46](#)
Objects in sequence diagrams, [352](#)
Observability of failures, [80](#)
Observe system state tactics, [164–167](#)
Off-the-shelf components, [36](#)
Omissions
 availability faults from, [85](#)
 for variation, [491](#)
On-demand self-service, [504](#)
1+1 redundancy tactic, [91](#)
Online documentation, [350](#)
Ontologies, [368–369](#)
OPC (Order Processing Center) component, [224](#), [226](#)
Open content systems, [529](#)
Open Group
 certification program, [477](#)
 governance responsibilities, [430–431](#)
Open source software, [36](#), [238](#)
Operation Desert Storm, [104](#)
OR gate symbol, [84](#)
Orchestrate tactic, [111](#)
Orchestration servers, [223](#), [225](#)
Order Processing Center (OPC) component, [224](#), [226](#)
Organization
 global development, [423–426](#)
 project manager and software architect responsibilities, [422–423](#)
 software development teams, [422](#)
Organizational Coordination model, [470](#), [473](#), [476](#)
Organizational Learning model, [470](#), [474](#), [476](#)
Organizations
 activities for success, [468](#)
 architect skills, [464](#)
 architecture influence on, [33](#)

as goal-object, [302](#)
security processes, [157](#)
structural strategies for products, [497](#)

Outages. See [Availability](#)

Outputs

- ADD method, [317–318](#)
- ATAM, [402–403](#)

Outsider analysis, [399](#)

Overlay views, [343](#)

Overloading for variation, [491](#)

Overview presentations in PALM method, [305](#)

P2P (peer-to-peer) pattern, [220–222](#)

PaaS (Platform as a Service) model, [505](#), [517](#)

Page mappers, [510–512](#)

PALM (Pedigreed Attribute eLicitation Method), [304–305](#)

Parameter fence tactic, [90](#)

Parameter typing tactic, [90](#)

Parameters for variation mechanism, [492](#)

Parser tool, [386](#)

Partitioning CAP theorem, [523](#)

Partnership and preparation phase in ATAM, [403–404](#)

Passive redundancy, [91–92](#), [256–259](#)

Patterns, [18–19](#)

- allocation, [232–237](#)
- component-and-connector. See [Component-and-connector \(C&C\) patterns and structures](#)
- documenting, [350–351](#)
- introduction, [203–204](#)
- module, [205–210](#)
- relationships, [204–205](#)
- summary, [247–248](#)
- and tactics, [238–247](#), [315](#)

Paulish, Dan, [420](#)

Pause/resume command, [179](#)

Payment Card Industry (PCI), [260](#)

PDF (probability density function), [255](#)

PDM (platform-definition model), [45](#)

Pedigree and value component of business goals, [301](#)

Pedigreed Attribute eLicitation Method (PALM), [304–305](#)

Peer nodes, [220](#)

Peer review, [398–399](#)

Peer-to-peer (P2P) pattern, [220–222](#)

Penalties in Incremental Commitment Model, [286](#)

People

- managing, [464](#)
- in product reuse, [485](#)

Perfect vs. good enough, [398](#)

Performance

- analytic model space, [259](#)
- analyzing, [252–255](#)
- broker pattern, [241](#)
- cloud, [521](#)

component-and-connector views, [336](#)
control resource demand tactics, [137–138](#)
design checklist, [142–144](#)
general scenario, [132–134](#)
introduction, [131–132](#)
manage resources tactics, [138–139](#)
map-reduce pattern, [232](#)
ping/echo, [243](#)
and quality, [191](#)
quality attributes tactics, [72](#)
queuing models for, [198–199](#)
resource effects, [244, 246](#)
summary, [145](#)
tactics overview, [135–137](#)
views, [341](#)

Performance quality attribute, [307](#)
Performance efficiency quality attribute, [193](#)
Periodic events, [133](#)

Periphery
Metropolis model, [535](#)
requirements, [532](#)

Persistent object managers, [515–516](#)
Personal objectives as business goal, [298](#)
Personnel availability in ADD method, [320](#)
Petrov, Stanislav Yevgrafovich, [192](#)

Phases
ATAM, [403–404](#)
metrics, [430](#)
Metropolis model, [534](#)

Philips product lines, [480–481, 487](#)
Physical security, [191](#)
PIM (platform-independent model), [45](#)
Ping/echo tactic, [87–88, 243](#)
Pipe-and-filter pattern, [215–217](#)
Planned increments, [530](#)

Planning
for failure, [82–85](#)
incremental development, [428](#)
overview, [420–421](#)
tests, [374](#)

Platform as a Service (PaaS) model, [505, 517](#)
Platform-definition model (PDM), [45](#)
Platform-independent model (PIM), [45](#)

Platforms
architect knowledge about, [467](#)
frameworks in, [365](#)
patterns, [19, 238](#)
services for, [532–533](#)

Plug-in architectures, [34](#)
PMBOK (Project Management Body of Knowledge), [423–425](#)
Pointers, smart, [95](#)
Policies, scheduling, [140](#)
Pooling resources, [504](#)

Portability quality attributes, [67](#), [186](#), [195](#)
Portfolio as goal-object, [302](#)
Ports in component-and-connector views, [335](#), [337–338](#)
Potential alternatives, [398](#)
Potential problems, peer review for, [399](#)
Potential quality attributes, [305](#)
Power station catastrophe, [188](#), [192](#)
Predicting system qualities, [28](#)
Predictive model tactic, [95](#)
Preemptible processes, [141](#)
Preparation-and-repair tactic, [91–93](#)
Preprocessor macros, [167](#)
Presentation
 ATAM, [402–406](#)
 documentation, [346](#)
 Lightweight Architecture Evaluation, [416](#)
 PALM method, [305](#)
 QAW, [295](#)
Prevent faults tactics, [94–95](#)
Primary presentations in documentation, [346](#)
Principles
 Agile, [276–277](#)
 cloud failures, [522](#)
 design fragments from, [72](#)
 Incremental Commitment Model, [286](#)
Prioritize events tactic, [137–138](#), [142](#)
Prioritizing
 ATAM scenarios, [410](#)
 CBAM scenarios, [445–446](#)
 CBAM weighting, [444](#)
 Lightweight Architecture Evaluation scenarios, [416](#)
 NASA ECS project scenarios, [452](#)
 QAW, [295–296](#)
 risk, [429](#)
 schedules, [140–141](#)
 views, [343](#)
PRIORITY AND gate symbol, [84](#)
Private clouds, [506](#)
Private IP addresses, [514](#)
Proactive enforcement in Metropolis model, [535](#)
Proactive product line models, [495](#)
Probability density function (PDF), [255](#)
Probability for availability, [256–259](#)
Problem relationships in patterns, [204–205](#)
Proceedings scribes, [401](#)
Processes
 development, [44–45](#)
 product reuse, [484](#)
 recommendations, [20](#)
 security, [157](#)
Processing time in performance, [136](#)
Procurement management, [425](#)
Product-line managers, [55](#)

Product lines. See [Software product lines](#)

Product manager roles, [422](#)

Productivity metrics, [429–430](#)

Professional context, [51–52](#), [58](#)

Profiler tools, [386](#)

Programming knowledge of architects, [466](#)

Project context, [57](#)

Project life-cycle context

- architecturally significant requirements, [46–47](#)
- architecture analysis and evaluation, [47–48](#)
- architecture documentation and communication, [47](#)
- architecture selection, [47](#)
- business cases, [46](#)
- development processes, [44–45](#)
- implementation conformance, [48](#)

Project Management Body of Knowledge (PMBOK), [423–425](#)

Project managers

- description and interests, [55](#)
- responsibilities, [422–423](#)

Project planning artifacts in product reuse, [484](#)

Propagation costs of change, [288](#)

Prosumers in edge-dominant systems, [529](#)

Protection groups, [91](#)

Prototypes

- evolutionary, [33–34](#)
- quality attribute modeling and analysis, [264–265](#)
- for requirements, [47](#)

Provides-services role, [335](#)

Proxy servers, [146](#), [211](#)

Public clouds, [506](#)

Public IP addresses, [514](#)

Publicly available apps, [36](#)

Publish-subscribe connector, [336](#)

Publish-subscribe pattern, [226–229](#)

Publisher role, [336](#)

QAW (Quality Attribute Workshop), [294–296](#)

Qt framework, [215](#)

Quality attribute modeling and analysis, [251–252](#)

- analytic model space, [259–260](#)
- availability analysis, [255–259](#)
- checklists, [260–262](#)
- experiments, simulations, and prototypes, [264–265](#)
- life cycle stages, [265–266](#)
- performance analysis, [252–255](#)
- summary, [266–267](#)
- thought experiments and back-of-the-envelope analysis, [262–264](#)

Quality Attribute Workshop (QAW), [294–296](#)

Quality attributes, [185](#)

- ADD method, [322–323](#)
- ASRs, [294–296](#)
- ATAM, [407](#)

capture scenarios, [196–197](#)
categories, [189–190](#)
checklists, [199, 260–262](#)
considerations, [65–67](#)
design approaches, [197](#)
and documentation, [354–355](#)
grand unified theory, [261](#)
important, [185–188](#)
inhibiting and enabling, [26–27](#)
introduction, [63–64](#)
Lightweight Architecture Evaluation, [416](#)
models, [197–198](#)
NASA ECS project, [453](#)
peer review, [398](#)
quality design decisions, [72–76](#)
requirements, [64, 68–70](#)
software and system, [190–193](#)
standard lists, [193–196](#)
summary, [76–77](#)
tactics, [70–72, 198–199](#)
technical context, [40–41](#)
variability, [488–489](#)
X-ability, [196–199](#)
Quality design decisions, [72–73](#)
allocation of responsibilities, [73](#)
binding time, [75–76](#)
coordination models, [73–74](#)
data models, [74](#)
element mapping, [75](#)
resource management, [74–75](#)
technology choices, [76](#)
Quality management in global development, [424](#)
Quality of products as business goal, [299](#)
Quality requirements, mapping decisions to, [402–403](#)
Quality views, [340–341](#)
Questioners on ATAM teams, [401](#)
Questions for organizational competence, [470, 472–474](#)
Queue sizes tactic, [139](#)
Queuing models for performance, [198–199, 252–255](#)
Quick Test Pro tool, [172](#)

Race conditions, [133](#)
Random access in equipment utilization, [508](#)
Rapid elasticity, [504–505](#)
Rate monotonic prioritization strategy, [140](#)
Rational Unified Process, [44](#)
Rationale in documentation, [347, 349](#)
Raw view extraction in reconstruction, [382–386](#)
RDBMSs (relational database management systems), [518](#)
React to attacks tactics, [153](#)
Reactive enforcement in Metropolis model, [536](#)
Reactive product line models, [495](#)

Reader role in component-and-connector views, [335](#)
Reconfiguration tactic, [93](#)
Reconstruction and conformance, [380–381](#)
 database construction, [386–387](#)
 finding violations, [389–392](#)
 guidelines, [392–393](#)
 process, [381–382](#)
 raw view extraction, [382–386](#)
 summary, [393–394](#)
 view fusion, [388–389](#)
Record/playback method for system state, [165](#)
Recover from attacks tactics, [153–154](#)
Recover-from-faults tactics, [91–94](#)
Reduce computational overhead tactic, [142](#)
Reduce function in map-reduce pattern, [232–235](#)
Reduce overhead tactic, [138](#)
Redundancy tactics, [90](#), [256–259](#)
Refactor tactic, [124](#)
Refined scenarios
 NASA ECS project, [451–452](#)
 QAW, [296](#)
Reflection for variation, [491](#)
Reflection pattern, [262](#)
Registry of services, [225](#)
Regression testing, [372](#)
Reintroduction tactics, [91](#), [93–94](#)
Rejuvenation tactic, [95](#)
Relational database management systems (RDBMSs), [518](#)
Relations
 allocation views, [339–340](#)
 architectural structures, [14](#), [16–17](#)
 broker pattern, [211](#)
 client-server pattern, [218](#)
 component-and-connector views, [337](#)
 conformance, [390](#)
 in documentation, [346](#)
 layered pattern, [207](#)
 map-reduce patterns, [235](#)
 Model-View-Controller pattern, [213](#)
 modular views, [333](#)
 multi-tier pattern, [237](#)
 peer-to-peer pattern, [222](#)
 pipe-and-filter pattern, [216](#)
 publish-subscribe pattern, [227](#)
 service-oriented architecture pattern, [225](#)
 shared-data pattern, [231](#)
 view extraction, [384](#)
Release strategy for documentation, [350](#)
Reliability
 cloud, [507](#)
 component-and-connector views, [336](#)
 core, [532](#)
 independently developed components for, [36](#)

vs. safety, [188](#)
SOAP, [109](#)
views, [341](#)

Reliability quality attribute, [195](#)

Remote procedure call (RPC) model, [109](#)

Removal from service tactic, [94–95](#)

Replicated elements in variation, [491](#)

Replication tactic, [90](#)

Report method for system state, [165](#)

Reporting tests, [374](#)

Repository patterns, [19](#)

Representation of architecture, [6](#)

Representational State Transfer (REST), [108–110](#), [223–225](#)

Reputation of products as business goal, [299](#)

Request/reply connectors

- client-server pattern, [218](#)
- peer-to-peer pattern, [222](#)

Requirements

- ASRs. See [Architecturally significant requirements \(ASRs\)](#)
- categories, [64–65](#)
- from goals, [49](#)
- mapping to, [355](#), [402–403](#)
- Metropolis model, [534](#)
- product reuse, [483](#)
- prototypes for, [47](#)
- quality attributes, [68–70](#)
- software development life cycle changes, [530](#)
- summary, [308–310](#)
- tying methods together, [308](#)

Requirements documents

- ASRs from, [292–293](#)
- Waterfall model, [56](#)

Reset method for system state, [165](#)

Resisting attacks tactics, [152–153](#)

RESL scale factor, [279](#)

Resource management category

- ASRs, [293](#)
- availability, [97](#)
- interoperability, [115](#)
- modifiability, [127](#)
- performance, [144](#)
- quality design decisions, [74–75](#)
- security, [155](#)
- software development life cycle changes, [530](#)
- testability, [170](#)
- usability, [182](#)

Resources

- component-and-connector views, [336](#)
- equipment utilization, [508](#)
- pooling, [504](#)
- sandboxing, [166](#)
- software development life cycle changes, [530](#)

Response

availability, [85–86](#)
interoperability, [105, 107–108](#)
modifiability, [119–120](#)
performance, [134](#)
quality attributes expressions, [68–70](#)
security, [149–150](#)
testability, [162–163](#)
usability, [176](#)
variability, [489](#)

Response measure
availability, [85–86](#)
interoperability, [107–108](#)
modifiability, [119–120](#)
performance, [134](#)
quality attributes expressions, [68–70](#)
security, [149–150](#)
testability, [162–163](#)
usability, [176](#)
variability, [489](#)

Responsibilities
as business goal, [299](#)
modules, [333](#)
quality design decisions, [73](#)

REST (Representational State Transfer), [108–110, 223–225](#)

Restart tactic, [94](#)

Restrict dependencies tactic, [124, 239, 246–247](#)

Restrictions on vocabulary, [36](#)

Results
ATAM, [411](#)
CBAM, [447, 456–457](#)
evaluation, [400](#)
Lightweight Architecture Evaluation, [416](#)

Retry tactic, [93](#)

Reusable models, [35](#)

Reuse of software architecture, [479, 483–486](#)

Reviews
back door, [544–545](#)
peer, [398–399](#)

Revision history of modules, [334](#)

Revoke access tactic, [153](#)

Rework in agility, [279](#)

Risk
ADD method, [320](#)
ATAM, [402](#)
global development, [425](#)
progress tracking, [429](#)

Risk-based testing, [373–374](#)

Robustness of core, [532](#)

Roles
component-and-connector views, [335](#)
product line architecture, [488–490](#)
software development teams, [422](#)
testing, [375–376](#)

Rollback tactic, [92](#)
Round-robin scheduling strategy, [140–141](#)
Rozanski, Nick, [170](#)
RPC (remote procedure call) model, [109](#)
Runtime conditionals, [492](#)
Rutan, Burt, [159](#)

SaaS (Software as a Service) model, [505](#)

Safety
checklists, [260, 268](#)
use cases, [46](#)

Safety attribute, [188](#)

Safety Integrity Level, [268](#)

Salesforce.com, [509](#)

Sample technologies in cloud, [514–520](#)

Sampling rate tactic, [137](#)

Sandbox tactic, [165–166](#)

Sanity checking tactic, [89](#)

Satisfaction in usability, [175](#)

Saxe, John Godfrey, [379](#)

Scalability

kinds, [187](#)
peer-to-peer systems, [220](#)
WebArrow web-conferencing system, [285](#)

Scalability attribute, [187](#)

Scaling, automatic, [516](#)

Scenario scribes, [401](#)

Scenarios

ATAM, [408, 410](#)
availability, [85–86](#)
business goals, [301–303](#)
CBAM, [445–446](#)
interoperability, [107–110](#)
Lightweight Architecture Evaluation, [416](#)
modifiability, [119–120](#)
NASA ECS project, [451–452](#)
performance, [132–134](#)
QAW, [295–296](#)
quality attributes, [67–70, 196–197](#)
security, [148–150](#)
for structures, [12](#)
testability, [162–163](#)
usability, [176](#)
weighting, [441, 444](#)

Schedule resources tactic

performance, [139](#)
quality attributes, [72](#)

Scheduled downtimes, [81](#)

Schedulers, hypervisor, [512](#)

Schedules

deviation measurements, [429](#)
estimates, [34](#)

policies, [140–141](#)
policy tactic, [244–245](#)
top-down and bottom-up, [420–421](#)

Schemas, database, [519](#)

Scope, product line, [486–488](#)

Scope and summary section in documentation maps, [347](#)

Scrum development methodology, [44](#)

SDL (Specification and Description Language), [354](#)

Security

- analytic model space, [259](#)
- broker pattern, [242](#)
- cloud, [507, 520–521](#)
- component-and-connector views, [336](#)
- design checklist, [154–156](#)
- general scenario, [148–150](#)
- introduction, [147–148](#)
- ping/echo, [243](#)
- quality attributes checklists, [260](#)
- summary, [156](#)
- tactics, [150–154](#)
- views, [341](#)

Security Monkey, [161](#)

Security quality attribute, [195, 307](#)

SEI (Software Engineering Institute), [59](#)

Selecting

- architecture, [47](#)
- tools and technology, [463](#)

Selenium tool, [172](#)

Self-organization in Agile, [277](#)

Self-test tactic, [91](#)

Semantic coherence, [178](#)

Semantic importance, [140](#)

Semiformal documentation notations, [330](#)

Sensitivity points in ATAM, [403](#)

Separate entities tactic, [153](#)

Separation of concerns in testability, [167](#)

Sequence diagrams

- thought experiments, [263](#)
- for traces, [351–352](#)

Servers

- client-server pattern, [217–219](#)
- proxy, [146, 211](#)
- SAO pattern, [223, 225](#)

Service consumer components, [222, 225](#)

Service discovery in SOAP, [108](#)

Service impact of faults, [81](#)

Service-level agreements (SLAs)

- Amazon, [81, 522](#)
- availability in, [81](#)
- IaaS, [506](#)
- PaaS, [505](#)
- SOA, [222](#)

Service-oriented architecture (SOA) pattern, [222–226](#)

Service providers, [222–225](#)
Service registry, [223](#)
Service structure, [13](#)
Services for platforms, [532–533](#)
Set method for system state, [165](#)
Shadow tactic, [93](#)
Shared-data patterns, [19, 230–231](#)
Shared documents in documentation, [350](#)
Shareholders, responsibilities to, [299](#)
Siberian hydroelectric plant catastrophe, [188, 192](#)
Siddhartha, Gautama, [251](#)
Side-channel attacks, [521](#)
Side effects in economic analysis, [439, 441](#)
Simian Army, [160–161](#)
Simulations, [264–265](#)
Size
 modules, [121](#)
 queue, [139](#)
Skeletal systems, [34](#)
Skeletal view of human body, [9](#)
Skills
 architects, [460, 463, 465](#)
 global development, [423](#)
 professional context, [51](#)
SLAs. See [Service-level agreements \(SLAs\)](#)
Small victories, [544](#)
Smart pointers, [95](#)
SOA (service-oriented architecture) pattern, [222–226](#)
SOAP
 vs. REST, [108–110](#)
 SOA pattern, [223–225](#)
Social networks in publish-subscribe pattern, [229](#)
Socializing in Incremental Commitment Model, [286](#)
Society
 as goal-object, [302](#)
 service to, [299](#)
Software architecture importance, [25–26](#)
 change management, [27–28](#)
 constraints, [32–33](#)
 cost and schedule estimates, [34](#)
 design decisions, [31–32](#)
 evolutionary prototyping, [33–34](#)
 independently developed components, [35–36](#)
 organizational structure, [33](#)
 quality attributes, [26–27](#)
 stakeholder communication, [29–31](#)
 summary, [37](#)
 system qualities prediction, [28](#)
 training basis, [37](#)
 transferable, reusable models, [35](#)
 vocabulary restrictions, [36](#)
Software architecture overview, [3–4](#). See also [Architecture](#)
 as abstraction, [5–6](#)

behavior in, [6–7](#)
competence, [467–475](#)
contexts. See [Contexts](#)
definitions, [4](#)
good and bad, [19–21](#)
patterns, [18–19](#)
selecting, [7](#)
as set of software structures, [4–5](#)
structures and views, [9–18](#)
summary, [21–22](#)
system architecture vs. enterprise, [7–8](#)

Software as a Service (SaaS) model, [505](#)

Software Engineering Body of Knowledge (SWEBOK), [292](#)

Software Engineering Institute (SEI), [59, 479](#)

Software Product Line Conference (SPLC), [498](#)

Software Product Line Hall of Fame, [498](#)

Software product lines

- adoption strategies, [494–496](#)
- evaluating, [493–494](#)
- evolving, [496–497](#)
- failures, [481–482](#)
- introduction, [479–481](#)
- key issues, [494–497](#)
- organizational structure, [497](#)
- quality attribute of variability, [488](#)
- reuse potential, [483–486](#)
- role of, [488–490](#)
- scope, [486–488](#)
- successful, [483–486](#)
- summary, [497–498](#)
- variability, [482–483](#)
- variation mechanisms, [490–493](#)

Software quality attributes, [190–193](#)

Software rejuvenation tactic, [95](#)

Software upgrade tactic, [92–93](#)

Solutions in relationships, [204–205](#)

SonarJ tool, [387–391](#)

Sorting in map-reduce pattern, [232](#)

SoS (system of systems), [106](#)

Source code

- KSLOC, [279–281](#)
- mapping to, [334](#)

Source in security scenario, [150](#)

Source of stimulus

- availability, [85–86](#)
- interoperability, [107–108](#)
- modifiability, [119–120](#)
- performance, [134](#)
- quality attributes expressions, [68–70](#)
- security, [148](#)
- testability, [162–163](#)
- usability, [176](#)
- variability, [489](#)

Spare tactics, [91–92](#), [256–259](#)
Specialized interfaces tactic, [165](#)
Specification and Description Language (SDL), [354](#)
Spikes in Agile, [284–285](#)
SPLC (Software Product Line Conference), [498](#)
Split module tactic, [123](#)
Sporadic events, [133](#)
Spring framework, [166](#)
Staging views, [343](#)
Stakeholders
 on ATAM teams, [401](#)
 communication among, [29–31](#), [329](#)
 documentation for, [348–349](#)
 evaluation process, [400](#)
 interests, [52–55](#)
 interviewing, [294–296](#)
 for methods, [272](#)
 utility tree reviews, [306](#)
 views, [342](#)
Standard lists for quality attributes, [193–196](#)
Standards and interoperability, [112–113](#)
State, system, [164–167](#)
State machine diagrams, [353](#)
State resynchronization tactic, [93](#)
Stateless services in cloud, [522](#)
States, responsibilities to, [299](#)
Static allocation views, [340](#)
Static scheduling, [141](#)
Status meetings, [428](#)
Stein, Gertrude, [142](#)
Steinberg, Saul, [39](#)
Stimulus
 availability, [85–86](#)
 interoperability, [107–108](#)
 modifiability, [119–120](#)
 performance, [134](#)
 quality attributes expressions, [68–70](#)
 security, [148](#), [150](#)
 source. See [Source of stimulus](#)
 testability, [162–163](#)
 usability, [176](#)
 variability, [489](#)
Stochastic events, [133](#)
Stonebraker, Michael, [518](#)
Storage
 for testability, [165](#)
 virtualization, [512–513](#)
Strategies in NASA ECS project, [452–456](#)
Strictly layered patterns, [19](#)
Structural complexity in testability, [167–168](#)
Structure101 tool, [387](#)
Stuxnet virus, [80](#)
Subarchitecture in component-and-connector views, [335](#)

Submodules, [333](#)
Subscriber role, [336](#)
Subsystems, [9](#)
Supernodes in peer-to-peer pattern, [220](#)
Support and development software, [358–359](#)
Support system initiative tactic, [180–181](#)
Support user initiative tactic, [179–180](#)
SWEBOK (Software Engineering Body of Knowledge), [292](#)
Swing classes, [215](#)
Syncing code and architecture, [368](#)
System analysis and construction, documentation for, [329](#)
System architecture vs. enterprise architecture, [7–8](#)
System as goal-object, [302](#)
System availability requirements, [81](#)
System efficiency in usability, [175](#)
System engineers, [55](#)
System exceptions tactic, [90](#)
System Generation Module, [358](#)
System initiative in usability, [177](#)
System of systems (SoS), [106](#)
System overview in documentation, [349](#)
System qualities, predicting, [28](#)
System quality attributes, [190–193](#)
System test manager roles, [422](#)
System testing, [371](#)

Tactics

availability, [87–96](#)
interactions, [242–247](#)
interoperability, [110–113](#)
modifiability, [121–125](#)
patterns relationships with, [238–242](#)
performance, [135–142](#)
quality attributes, [70–72, 198–199](#)
security, [150–154](#)
testability, [164–168](#)
usability, [177–181](#)

Tailor interface tactic, [111](#)

Team building skills, [463, 465](#)

Team leader roles, [422](#)

TeamCity tool, [172](#)

Teams

ATAM, [400–401](#)

organizing, [422](#)

Technical contexts

architecture influence, [57](#)

environment, [41–42](#)

quality attributes, [40–41](#)

Vasa ship, [42–43](#)

Technical debt, [286](#)

Technical processes in security, [157](#)

Technology choices, [76](#)

Technology knowledge of architects, [467](#)

Templates

ATAM, [406](#)

code, [365–367](#)

scenarios. See [Scenarios](#)

variation mechanism, [492](#)

10-18 Monkey, [161](#)

Terminating generate and test process, [316](#)

Terms and concepts, [368–369](#)

Test harnesses, [160](#)

Testability

analytic model space, [259](#)

automation, [171–172](#)

broker pattern, [241](#)

design checklist, [169–170](#)

general scenario, [162–163](#)

introduction, [159–162](#)

summary, [172](#)

tactics, [164–168](#)

test data, [170–171](#)

Testable requirements, [292](#)

TestComplete tool, [172](#)

Testers, [55](#)

Tests and testing

activities, [374–375](#)

architect role, [375–376](#), [463](#)

black-box and white-box, [372–373](#)

choices, [315](#)

in incremental development, [428](#)

levels, [370–372](#)

modules, [334](#)

product reuse, [484](#)

risk-based, [373–374](#)

summary, [376](#)

Therac-25 fatal overdose, [192](#)

Thought experiments, [262–264](#)

Thousands of source lines of code (KSLOC), [279–281](#)

Threads in concurrency, [132–133](#)

Throughput of systems, [134](#)

Tiers

component-and-connector views, [337](#)

multi-tier pattern, [235–237](#)

Time and time management

basis sets, [261](#)

global development, [424](#)

performance, [131](#)

Time boxing, [264](#)

Time of day factor in equipment utilization, [508](#)

Time of year factor in equipment utilization, [508](#)

Time-sharing, [503](#)

Time stamp tactic, [89](#)

Time to market

independently developed components for, [36](#)

and modifiability, [284](#)
Timeout tactic, [91](#)
Timing in availability, [85](#)
TMR (triple modular redundancy), [89](#)
Tools
 for product reuse, [484](#)
 selecting, [463](#)
Top-down adoption, [495](#)
Top-down analysis mode, [284](#)
Top-down schedules, [420–421](#)
Topic-based publish-subscribe patterns, [229](#)
Topological constraints, [236](#)
Torvalds, Linus, [530](#), [535](#), [538](#)
Total benefit in CBAM, [446](#)
Traces for behavior documentation, [351–353](#)
Tracking progress, [428–429](#)
Tradeoffs
 ATAM, [403](#)
 implementation, [427](#)
Traffic systems, [142](#)
Training, architecture for, [37](#)
Transactions
 availability, [95](#)
 databases, [519–520](#)
 SOAP, [108](#)
Transferable models, [35](#)
Transformation systems, [215](#)
Transforming existing systems, [462](#)
Transitions in state machine diagrams, [354](#)
Triple modular redundancy (TMR), [89](#)
Troeh, Eve, [190](#)
Turner, R., [279](#), [281](#), [288](#)
Twitter, [528](#)
Two-phase commits, [95](#)

Ubiquitous network access, [504](#)
UDDI (Universal Description, Discovery and Integration) language, [108](#)
UML
 activity diagrams, [353](#)
 communication diagrams, [353](#)
 component-and-connector views, [338–339](#)
 connectors, [369](#)
 sequence diagrams, [351–352](#)
 state machine diagrams, [353](#)
Unambiguous requirements, [292](#)
Uncertainty in equipment utilization, [508–509](#)
Undo command, [179](#)
Unified Process, [44](#)
Unit testing, [370–371](#)
Unity of purpose in modules, [121](#)
Universal Description, Discovery and Integration (UDDI) language, [108](#)
Up-front planning vs. agility, [278–281](#)

Usability
analytic model space, [259](#)
design checklist, [181–182](#)
general scenario, [176](#)
introduction, [175](#)
quality attributes checklists, [260](#)
tactics, [177–181](#)
Usability quality attribute, [193](#), [307](#)
Usage
allocation views, [339](#)
component-and-connector views, [337](#)
modular views, [333](#)
Use an intermediary tactic, [245](#)
modifiability, [123](#)
quality attributes, [72](#)
Use cases
ATAM presentations, [406](#)
thought experiments, [263](#)
for traces, [351](#)
“User beware” proviso, [372](#)
User initiative in usability, [177](#)
User interface
exchanging information via, [104–105](#)
separating, [178](#)
User needs in usability, [175](#)
User stories in Agile, [278](#)
Users
communication with, [29](#)
description and interests, [55](#)
Uses
for documentation, [328–329](#)
views for, [332](#)
Uses relation in layered patterns, [19](#)
Uses structure in decomposition, [12](#)
Utility
assigning, [452](#)
CBAM, [448](#)
Utility-response curves, [439–443](#)
Utility trees
ASRs, [304–307](#)
ATAM, [407](#), [410](#)
Lightweight Architecture Evaluation, [416](#)
Utilization of equipment in cloud, [508–509](#)

Value component
business goals, [301](#)
utility trees, [306](#)
Value for cost (VFC), [438](#), [442](#)
Variability
product line, [482–483](#)
quality attributes, [488–489](#)
Variability attribute, [186](#)

Variability guides, [347](#), [493](#)
Variation
 binding time, [75](#)
 software product lines, [490–493](#)
Variation points
 CBAM, [448–450](#)
 identifying, [490](#)
Vasa ship, [42–43](#)
Vascular view of human body, [9](#)
Vehicle cruise control systems, [353](#)
Verify and refine requirements in ADD, [321–323](#)
Verify message integrity tactic, [151](#)
Vertical scalability, [187](#)
VFC (value for cost), [438](#), [442](#)
Views, [331–332](#)
 allocation, [339–340](#)
 architectural structures, [9–10](#)
 choosing, [341–343](#)
 combining, [343–345](#)
 component-and-connector, [335–339](#), [344](#), [406](#)
 documenting, [345–347](#)
 fused, [388–389](#)
 Model-View-Controller pattern, [213–214](#)
 module, [332–335](#), [406](#)
 quality, [340–341](#)
Views and Beyond approach, [282](#), [356–357](#)
Villa, Pancho, [541](#)
Violations, finding, [389–392](#)
Virtual resource managers, [515](#)
Virtual system of systems, [106](#)
Virtualization and virtual machines
 cloud, [509–514](#), [520–521](#)
 layers as, [13](#)
 in sandboxing, [166](#)
Visibility of interfaces, [333](#)
Vitruvius, [459](#)
Vlissides, J., [212](#)
Vocabulary
 quality attributes, [67](#)
 restrictions, [36](#)
Voting tactic, [89](#)
Vulnerabilities in security views, [341](#)

Walking skeleton method, [287](#)
War ship example, [42–43](#)
Warm spare tactic, [91–92](#)
Watchdogs, [89](#)
Waterfall model
 description, [44](#)
 requirements documents, [56](#)
Weaknesses
 broker pattern, [211](#), [240–242](#)

client-server pattern, [218](#)
layered pattern, [207](#)
map-reduce patterns, [235](#)
Model-View-Controller pattern, [213](#)
multi-tier pattern, [237](#)
peer-to-peer pattern, [222](#)
pipe-and-filter pattern, [216](#)
publish-subscribe pattern, [227](#)
service-oriented architecture pattern, [225](#)
shared-data pattern, [231](#)
Wealth of Networks (Benkler), [528](#)
Web 2.0 movement, [527](#)
Web-based system events, [131](#)
Web-conferencing systems
 Agile example, [283–285](#)
 considerations, [265](#)
Web Services Description Language (WSDL), [110](#)
WebArrow web-conferencing system, [284–285](#)
WebSphere MQ product, [224](#)
Weighting scenarios, [441](#), [444](#)
Wells, H. G., [117](#)
West, Mae, [131](#)
“What if” questions in performance analysis, [255](#)
White-box testing, [372–373](#)
Whitney, Eli, [35–36](#), [480](#)
Wikipedia, [528](#)
Wikis for documentation, [350](#)
Wisdom of crowds, [537](#)
Woods, Eoin, [25](#), [170](#)
Work assignment structures, [14](#)
Work-breakdown structures, [33](#)
Work skills of architect, [465](#)
World Wide Web as client-server pattern, [219](#)
Wrappers, [129](#)
Writer role in component-and-connector views, [335](#)
WS*, [108–110](#)
WSDL (Web Services Description Language), [110](#)

X-ability, [196–199](#)
X-ray view of human body, [9](#)

YAGNI principle, [282](#)
Yahoo! map-reduce patterns, [234](#)
Young, Toby, [39](#)
YouTube, [528](#)

Zoning policies analogy in Metropolis model, [536](#)

Special permission to reproduce portions of the following works copyright by Carnegie Mellon University is granted by the Software Engineering Institute:
Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith A. Stafford. “Software Architecture Documentation in Practice: Documenting Architectural Layers,” CMU/SEI-2000-SR-004, March 2000.

Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith A. Stafford. "Documenting Software Architectures: Organization of Documentation Package," CMU/SEI-2001-TN-010, August 2001.

Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith A. Stafford. "Documenting Software Architecture: Documenting Behavior," CMU/SEI-2002-TN-001, January 2002.

Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith A. Stafford. "Documenting Software Architecture: Documenting Interfaces," CMU/SEI-2002-TN-015, June 2002.

Felix Bachmann and Paul Clements. "Variability in Product Lines," CMU/SEI-2005-TR-012, September 2005.

Felix Bachmann, Len Bass, and Robert Nord. "Modifiability Tactics," CMU/SEI-2007-TR-002, September 2007.

Mario R. Barbacci, Robert Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. "Quality Attribute Workshops (QAWs), Third Edition," CMU/SEI-2003-TR-016, August 2003.

Len Bass, Paul Clements, Rick Kazman, and Mark Klein. "Models for Evaluating and Improving Architecture Competence," CMU/SEI-2008-TR-006, March 2008.

Len Bass, Paul Clements, Rick Kazman, John Klein, Mark Klein, and Jeannine Siviy. "A Workshop on Architecture Competence," CMU/SEI-2009-TN-005, April 2009.

Lisa Brownsword, David Carney, David Fisher, Grace Lewis, Craig Meyers, Edwin Morris, Patrick Place, James Smith, and Lutz Wrage. "Current Perspectives on Interoperability," CMU/SEI-2004-TR-009, March 2004.

Paul Clements and Len Bass. "Relating Business Goals to Architecturally Significant Requirements for Software Systems," CMU/SEI-2010-TN-018, May 2010.

Rick Kazman and Jeremy Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence," CMU/SEI-97-TR-010, October 1997.

Rick Kazman, Mark Klein, and Paul Clements. "ATAM: Method for Architecture Evaluation," CMU/SEI-2000-TR-004, August 2000.

Rick Kazman, Jai Asundi, and Mark Klein, "Making Architecture Design Decisions, An Economic Approach," CMU/SEI-2002-TR-035, September 2002.

Rick Kazman, Liam O'Brien, and Chris Verhoef, "Architecture Reconstruction Guidelines, Third Edition," CMU/SEI-2002-TR-034, November 2003.

Robert L. Nord, Paul C. Clements, David Emery, and Rich Hilliard. "A Structured Approach for Reviewing Architecture Documentation," CMU/SEI-2009-TN-030, December 2009.

James Scott and Rick Kazman. "Realizing and Refining Architectural Tactics: Availability," CMU/SEI-2009-TR-006 and ESC-TR-2009-006, August 2009.