

SEI SERIES IN SOFTWARE ENGINEERING

Software Architecture in Practice

FOURTH EDITION

Len Bass

Paul Clements

Rick Kazman



Software Architecture in Practice

Fourth Edition

The SEI Series in Software Engineering



Software Engineering Institute

Carnegie Mellon University



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

The SEI Series in Software Engineering is 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.

Titles 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 titles in the series address critical problems in software engineering for which practical solutions are available.



Make sure to connect with us!
informit.com/socialconnect

Software Architecture in Practice

Fourth Edition

Len Bass

Paul Clements

Rick Kazman

◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town

Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City

São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo



Software Engineering Institute

Carnegie Mellon

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 437, 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021934450

Copyright © 2022 Pearson Education, Inc.

Cover image: [Zhernosek_FFMstudio.com/Shutterstock](https://www.shutterstock.com/image-vector/zhernosek_FFMstudio.com/Shutterstock)

Hand/input icon: [In-Finity/Shutterstock](https://www.shutterstock.com/image-vector/in-finity/Shutterstock)

Figure 1.1: [GraphicsRF.com/Shutterstock](https://www.shutterstock.com/image-vector/graphicsrf.com/Shutterstock)

Figure 15.2: [Shutterstock Vector/Shutterstock](https://www.shutterstock.com/image-vector/shutterstock-vector/Shutterstock)

Figure 17.1: [Oleksiy Mark/Shutterstock](https://www.shutterstock.com/image-vector/oleksiy-mark/Shutterstock)

Figure 17.2, cloud icon: [luckyguy/123RF](https://www.shutterstock.com/image-vector/luckyguy/123RF)

Figures 17.2, 17.4, and 17.5 computer icons: [Dacian G/Shutterstock](https://www.shutterstock.com/image-vector/dacian-g/Shutterstock)

All rights reserved. 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. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-688609-9

ISBN-10: 0-13-688609-4

ScoutAutomatedPrintCode

Contents

Preface *xv*

Acknowledgments *xvii*

PART I INTRODUCTION 1

CHAPTER 1 What Is Software Architecture? 1

- 1.1 What Software Architecture Is and What It Isn't 2
- 1.2 Architectural Structures and Views 5
- 1.3 What Makes a “Good” Architecture? 19
- 1.4 Summary 21
- 1.5 For Further Reading 21
- 1.6 Discussion Questions 22

CHAPTER 2 Why Is Software Architecture Important? 25

- 2.1 Inhibiting or Enabling a System's Quality Attributes 26
- 2.2 Reasoning about and Managing Change 27
- 2.3 Predicting System Qualities 28
- 2.4 Communication among Stakeholders 28
- 2.5 Early Design Decisions 31
- 2.6 Constraints on Implementation 31
- 2.7 Influences on Organizational Structure 32
- 2.8 Enabling Incremental Development 33
- 2.9 Cost and Schedule Estimates 33
- 2.10 Transferable, Reusable Model 34
- 2.11 Architecture Allows Incorporation of Independently Developed Elements 34

2.12 Restricting the Vocabulary of Design Alternatives	35
2.13 A Basis for Training	36
2.14 Summary	36
2.15 For Further Reading	37
2.16 Discussion Questions	37

PART II QUALITY ATTRIBUTES 39

CHAPTER 3 Understanding Quality Attributes 39

3.1 Functionality	40
3.2 Quality Attribute Considerations	41
3.3 Specifying Quality Attribute Requirements: Quality Attribute Scenarios	42
3.4 Achieving Quality Attributes through Architectural Patterns and Tactics	45
3.5 Designing with Tactics	46
3.6 Analyzing Quality Attribute Design Decisions: Tactics-Based Questionnaires	48
3.7 Summary	49
3.8 For Further Reading	49
3.9 Discussion Questions	50

CHAPTER 4 Availability 51

4.1 Availability General Scenario	53
4.2 Tactics for Availability	55
4.3 Tactics-Based Questionnaire for Availability	62
4.4 Patterns for Availability	66
4.5 For Further Reading	68
4.6 Discussion Questions	69

CHAPTER 5 Deployability 71

5.1 Continuous Deployment	72
5.2 Deployability	75
5.3 Deployability General Scenario	76
5.4 Tactics for Deployability	78

5.5	Tactics-Based Questionnaire for Deployability	80
5.6	Patterns for Deployability	81
5.7	For Further Reading	87
5.8	Discussion Questions	87
CHAPTER 6	Energy Efficiency	89
6.1	Energy Efficiency General Scenario	90
6.2	Tactics for Energy Efficiency	92
6.3	Tactics-Based Questionnaire for Energy Efficiency	95
6.4	Patterns	97
6.5	For Further Reading	98
6.6	Discussion Questions	99
CHAPTER 7	Integrability	101
7.1	Evaluating the Integrability of an Architecture	102
7.2	General Scenario for Integrability	104
7.3	Integrability Tactics	105
7.4	Tactics-Based Questionnaire for Integrability	110
7.5	Patterns	112
7.6	For Further Reading	114
7.7	Discussion Questions	115
CHAPTER 8	Modifiability	117
8.1	Modifiability General Scenario	120
8.2	Tactics for Modifiability	121
8.3	Tactics-Based Questionnaire for Modifiability	125
8.4	Patterns	126
8.5	For Further Reading	130
8.6	Discussion Questions	131
CHAPTER 9	Performance	133
9.1	Performance General Scenario	134
9.2	Tactics for Performance	137
9.3	Tactics-Based Questionnaire for Performance	145
9.4	Patterns for Performance	146

9.5	For Further Reading	149
9.6	Discussion Questions	150

CHAPTER 10 Safety 151

10.1	Safety General Scenario	154
10.2	Tactics for Safety	156
10.3	Tactics-Based Questionnaire for Safety	160
10.4	Patterns for Safety	163
10.5	For Further Reading	165
10.6	Discussion Questions	166

CHAPTER 11 Security 169

11.1	Security General Scenario	170
11.2	Tactics for Security	172
11.3	Tactics-Based Questionnaire for Security	176
11.4	Patterns for Security	179
11.5	For Further Reading	180
11.6	Discussion Questions	180

CHAPTER 12 Testability 183

12.1	Testability General Scenario	186
12.2	Tactics for Testability	187
12.3	Tactics-Based Questionnaire for Testability	192
12.4	Patterns for Testability	192
12.5	For Further Reading	194
12.6	Discussion Questions	195

CHAPTER 13 Usability 197

13.1	Usability General Scenario	198
13.2	Tactics for Usability	200
13.3	Tactics-Based Questionnaire for Usability	202
13.4	Patterns for Usability	203
13.5	For Further Reading	205
13.6	Discussion Questions	205

CHAPTER 14	Working with Other Quality Attributes	207
14.1	Other Kinds of Quality Attributes	207
14.2	Using Standard Lists of Quality Attributes—Or Not	209
14.3	Dealing with “X-Ability”: Bringing a New QA into the Fold	212
14.4	For Further Reading	215
14.5	Discussion Questions	215
PART III	ARCHITECTURAL SOLUTIONS	217
CHAPTER 15	Software Interfaces	217
15.1	Interface Concepts	218
15.2	Designing an Interface	222
15.3	Documenting the Interface	228
15.4	Summary	230
15.5	For Further Reading	230
15.6	Discussion Questions	231
CHAPTER 16	Virtualization	233
16.1	Shared Resources	234
16.2	Virtual Machines	235
16.3	VM Images	238
16.4	Containers	239
16.5	Containers and VMs	241
16.6	Container Portability	242
16.7	Pods	242
16.8	Serverless Architecture	243
16.9	Summary	244
16.10	For Further Reading	245
16.11	Discussion Questions	245
CHAPTER 17	The Cloud and Distributed Computing	247
17.1	Cloud Basics	248
17.2	Failure in the Cloud	251

17.3	Using Multiple Instances to Improve Performance and Availability	253
17.4	Summary	261
17.5	For Further Reading	262
17.6	Discussion Questions	262

CHAPTER 18 Mobile Systems 263

18.1	Energy	264
18.2	Network Connectivity	266
18.3	Sensors and Actuators	267
18.4	Resources	268
18.5	Life Cycle	270
18.6	Summary	273
18.7	For Further Reading	274
18.8	Discussion Questions	275

PART IV SCALABLE ARCHITECTURE PRACTICES 277

CHAPTER 19 Architecturally Significant Requirements 277

19.1	Gathering ASRs from Requirements Documents	278
19.2	Gathering ASRs by Interviewing Stakeholders	279
19.3	Gathering ASRs by Understanding the Business Goals	282
19.4	Capturing ASRs in a Utility Tree	284
19.5	Change Happens	286
19.6	Summary	286
19.7	For Further Reading	287
19.8	Discussion Questions	287

CHAPTER 20 Designing an Architecture 289

20.1	Attribute-Driven Design	289
20.2	The Steps of ADD	292
20.3	More on ADD Step 4: Choose One or More Design Concepts	295
20.4	More on ADD Step 5: Producing Structures	298

20.5	More on ADD Step 6: Creating Preliminary Documentation during the Design	301
20.6	More on ADD Step 7: Perform Analysis of the Current Design and Review the Iteration Goal and Achievement of the Design Purpose	304
20.7	Summary	306
20.8	For Further Reading	306
20.9	Discussion Questions	307
CHAPTER 21	Evaluating an Architecture	309
21.1	Evaluation as a Risk Reduction Activity	309
21.2	What Are the Key Evaluation Activities?	310
21.3	Who Can Perform the Evaluation?	311
21.4	Contextual Factors	312
21.5	The Architecture Tradeoff Analysis Method	313
21.6	Lightweight Architecture Evaluation	324
21.7	Summary	326
21.8	For Further Reading	327
21.9	Discussion Questions	327
CHAPTER 22	Documenting an Architecture	329
22.1	Uses and Audiences for Architecture Documentation	330
22.2	Notations	331
22.3	Views	332
22.4	Combining Views	339
22.5	Documenting Behavior	340
22.6	Beyond Views	345
22.7	Documenting the Rationale	346
22.8	Architecture Stakeholders	347
22.9	Practical Considerations	350
22.10	Summary	353
22.11	For Further Reading	353
22.12	Discussion Questions	354

CHAPTER 23	Managing Architecture Debt	355
23.1	Determining Whether You Have an Architecture Debt Problem	356
23.2	Discovering Hotspots	358
23.3	Example	362
23.4	Automation	363
23.5	Summary	364
23.6	For Further Reading	364
23.7	Discussion Questions	365
PART V	ARCHITECTURE AND THE ORGANIZATION	367
CHAPTER 24	The Role of Architects in Projects	367
24.1	The Architect and the Project Manager	367
24.2	Incremental Architecture and Stakeholders	369
24.3	Architecture and Agile Development	370
24.4	Architecture and Distributed Development	373
24.5	Summary	376
24.6	For Further Reading	376
24.7	Discussion Questions	377
CHAPTER 25	Architecture Competence	379
25.1	Competence of Individuals: Duties, Skills, and Knowledge of Architects	379
25.2	Competence of a Software Architecture Organization	386
25.3	Become a Better Architect	387
25.4	Summary	388
25.5	For Further Reading	388
25.6	Discussion Questions	389

PART VI	CONCLUSIONS	391
CHAPTER 26	A Glimpse of the Future: Quantum Computing	391
26.1	Single Qubit	392
26.2	Quantum Teleportation	394
26.3	Quantum Computing and Encryption	394
26.4	Other Algorithms	395
26.5	Potential Applications	396
26.6	Final Thoughts	397
26.7	For Further Reading	398
	<i>References</i>	399
	<i>About the Authors</i>	415
	<i>Index</i>	417

This page intentionally left blank

Preface

When we set out to write the fourth edition of *Software Architecture in Practice*, our first question to ourselves was: Does architecture still matter? With the rise of cloud infrastructures, microservices, frameworks, and reference architectures for every conceivable domain and quality attribute, one might think that architectural knowledge is hardly needed anymore. All the architect of today needs to do is select from the rich array of tools and infrastructure alternatives out there, instantiate and configure them, and voila! An architecture.

We were (and are) pretty sure this is not true. Admittedly, we are somewhat biased. So we spoke to some of our colleagues—working architects in the healthcare and automotive domains, in social media and aviation, in defense and finance and e-commerce—none of whom can afford to let dogmatic bias rule them. What we heard confirmed our belief—that architecture is just as relevant today as it was more than 20 years ago, when we wrote the first edition.

Let’s examine a few of the reasons that we heard. First, the rate of new requirements has been accelerating for many years, and it continues to accelerate even now. Architects today are faced with a nonstop and ever-increasing stream of feature requests and bugs to fix, driven by customer and business needs and by competitive pressures. If architects aren’t paying attention to the modularity of their system (and, no, microservices are not a panacea here), that system will quickly become an anchor—hard to understand, change, debug, and modify, and weighing down the business.

Second, while the level of abstraction in systems is increasing—we can and do regularly use many sophisticated services, blissfully unaware of how they are implemented—the complexity of the systems we are being asked to create is increasing at least as quickly. This is an arms race, and the architects aren’t winning! Architecture has always been about taming complexity, and that just isn’t going to go away anytime soon.

Speaking of raising the level of abstraction, model-based systems engineering (MBSE) has emerged as a potent force in the engineering field over the last decade or so. MBSE is the formalized application of modeling to support (among other things) system design. The International Council on Systems Engineering (INCOSE) ranks MBSE as one of a select set of “transformational enablers” that underlie the entire discipline of systems engineering. A model is a graphical, mathematical, or physical representation of a concept or a construct that can be reasoned about. INCOSE is trying to move the engineering field from a document-based mentality to a model-based mentality, where structural models, behavioral models, performance models, and more are all used consistently to build systems better, faster, and cheaper. MBSE per se is beyond the scope of this book, but we can’t help but notice that what is being modeled is architecture. And who builds the models? Architects.

Third, the meteoric growth (and unprecedented levels of employee turnover) that characterizes the world of information systems means that no one understands everything in any real-world system. Just being smart and working hard aren't good enough.

Fourth, despite having tools that automate much of what we used to do ourselves—think about all of the orchestration, deployment, and management functions baked into Kubernetes, for example—we still need to understand the quality attribute properties of these systems that we depend upon, and we need to understand the emergent quality attribute properties when we combine systems together. Most quality attributes—performance, security, availability, safety, and so on—are susceptible to “weakest link” problems, and those weakest links may only emerge and bite us when we compose systems. Without a guiding hand to ward off disaster, the composition is very likely to fail. That guiding hand belongs to an architect, regardless of their title.

Given these considerations, we felt safe and secure that there was indeed a need for this book.

But was there a need for a fourth edition? Again (and this should be abundantly obvious), we concluded an emphatic “yes”! Much has changed in the computing landscape since the last edition was published. Some quality attributes that were not previously considered have risen to importance in the daily lives of many architects. As software continues to pervade all aspects of our society, *safety* considerations have become paramount for many systems; think about all of the ways that software controls the cars that we now drive. Likewise, *energy efficiency* is a quality that few architects considered a decade ago, but now must pay attention to, from massive data centers with unquenchable needs for energy to the small (even tiny) battery-operated mobile and IoT devices that surround us. Also, given that we are, more than ever, building systems by leveraging preexisting components, the quality attribute of *integrability* is consuming ever-increasing amounts of our attention.

Finally, we are building different kinds of systems, and building them in different ways than a decade ago. Systems these days are often built on top of virtualized resources that reside in a cloud, and they need to provide and depend on explicit interfaces. Also, they are increasingly mobile, with all of the opportunities and challenges that mobility brings. So, in this edition we have added chapters on virtualization, interfaces, mobility, and the cloud.

As you can see, we convinced ourselves. We hope that we have convinced you as well, and that you will find this fourth edition a useful addition to your (physical or electronic) bookshelf.

Register your copy of *Software Architecture in Practice, Fourth Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780136886099) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

We are profoundly grateful to all the people with whom we collaborated to produce this book.

First and foremost, we extend our gratitude to the co-authors of individual chapters. Their knowledge and insights in these areas were invaluable. Our thanks go to Cesare Pautasso of the Faculty of Informatics, University of Lugano; Yazid Hamdi of Siemens Mobile Systems; Greg Hartman of Google; Humberto Cervantes of Universidad Autonoma Metropolitana—Iztapalapa; and Yuanfang Cai of Drexel University. Thanks to Eduardo Miranda of Carnegie Mellon University’s Institute for Software Research, who wrote the sidebar on the Value of Information technique.

Good reviewers are essential to good work, and we are fortunate to have had John Hudak, Mario Benitez, Grace Lewis, Robert Nord, Dan Justice, and Krishna Guru lend their time and talents toward improving the material in this book. Thanks to James Ivers and Ipek Ozkaya for overseeing this book from the perspective of the SEI Series in Software Engineering.

Over the years, we have benefited from our discussions and writings with colleagues and we would like to explicitly acknowledge them. In particular, in addition to those already mentioned, our thanks go to David Garlan, Reed Little, Paulo Merson, Judith Stafford, Mark Klein, James Scott, Carlos Paradis, Phil Bianco, Jungwoo Ryoo, and Phil Laplante. Special thanks go to John Klein, who contributed one way or another to many of the chapters in this book.

In addition, we are grateful to everyone at Pearson for all their work and attention to detail in the countless steps involved in turning our words into the finished product that you are now reading. Thanks especially to Haze Humbert, who oversaw the whole process.

Finally, thanks to the many, many researchers, teachers, writers, and practitioners who have, over the years, worked to turn software architecture from a good idea into an engineering discipline. This book is for you.

This page intentionally left blank

1



What Is Software Architecture?

We are called to be architects of the future, not its victims.
—R. Buckminster Fuller

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 reasonable software architecture is important to the successful development of a software system and
2. there is a sufficient body of knowledge about software architecture to fill up a book.

There was a time when both of these assumptions needed justification. Early editions of this book tried to convince readers that both of these assumptions are true and, once you were convinced, supply you with basic knowledge so that you could apply the practice of architecture yourself. Today, there seems to be little controversy about either aim, and so this book is more about the supplying than the convincing.

The basic principle of software architecture is every software system is constructed to satisfy an organization's business goals, and that the architecture of a system 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, and documented using known techniques that will support the achievement of these business goals. The complexity can be tamed, made tractable.

These, then, are the topics for this book: the design, analysis, and documentation of architectures. We will also examine the influences, principally in the form of business goals that lead to quality attribute requirements, that 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 business and organizational perspectives.

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:

The software architecture of a system is the set of structures needed to reason about the system. These structures 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" or "important" decisions. While it is true that many architectural decisions are made early, not all are—especially in Agile and spiral-development projects. It's also true that many decisions that are made early are not what we would consider architectural. Also, it's hard to look at a decision and tell whether it's "major." Sometimes only time will tell. And since deciding on an architecture is one of the architect's most important obligations, we need to know which decisions an architecture comprises.

Structures, by contrast, are fairly easy to identify in software, and they form a powerful tool for system design and analysis.

So, there we are: Architecture is about reasoning-enabling structures.

Let's 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 can lay claim to being *the* architecture. Structures can be grouped into categories, and the categories themselves provide useful ways to think about the architecture. Architectural structures can be organized into three useful categories, which will play an important role in the design, documentation, and analysis of architectures:

1. Component-and-connector structures
2. Module structures
3. Allocation structures

We'll delve more into these types of structures in the next section.

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(s). These include properties such as the functionality achieved by the system, the system's ability to keep operating usefully in the face of faults or attempts to take it down, the ease or difficulty of making specific changes to the system, the system's

responsiveness to user requests, and many others. We will spend a great deal of time in this book exploring the relationship between architecture and *quality attributes* like these.

Thus the set of architectural structures is neither fixed nor limited. What is architectural depends on what is useful to reason about in your context for your system.

Architecture Is an Abstraction

Since architecture consists of structures, and structures consist of elements¹ and relations, it follows that an architecture comprises software elements and how those elements relate to each other. This means that architecture specifically and intentionally omits certain information about elements that is not useful for reasoning about the system. 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. This abstraction is essential to taming the complexity of an architecture: We simply cannot, and do not want to, deal with all of the complexity all of the time. We want—and need—the understanding of a system’s architecture to be many orders of magnitude easier than understanding every detail about that system. You can’t keep every detail of a system of even modest size in your head; the point of architecture is to make it so you don’t have to.

Architecture versus Design

Architecture is design, but not all design is architecture. That is, many design decisions are left unbound by the architecture—it is, after all, an abstraction—and depend on the discretion and good judgment of downstream designers and even implementers.

Every Software System Has a Software Architecture

Every system has an architecture, because every system has elements and relations. However, it does not 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 at hand is the executing binary code. This reveals the difference between the architecture of a system and the *representation* of that architecture. Given that an architecture can exist independently of its description or specification, this raises the importance of *architecture documentation*, which is described in Chapter 22.

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

Not All Architectures Are Good Architectures

Our definition is indifferent as to whether the architecture for a system is a good one or a bad one. An architecture may either support or hinder achieving the important requirements for a system. 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 20 and *architecture evaluation*, which will be dealt with in Chapter 21.

Architecture Includes Behavior

The behavior of each element is part of the architecture insofar as that behavior can help you reason about the system. The behavior of elements embodies how they interact with each other and with the environment. This is clearly part of our definition of architecture and will have an effect on the properties exhibited by the system, such as its runtime performance.

Some aspects of behavior are below the architect's level of concern. Nevertheless, to the extent that an element's behavior influences the acceptability of the system as a whole, this behavior must be considered part of the system's architectural design, and should be documented as such.

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, and its architect, must live.

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 the totality of hardware, software, and humans.

A system architecture will influence, for example, the functionality that is assigned to different processors and the types of networks that connect those processors. The software architecture will determine how this functionality is structured and how the software programs residing on the various processors interact.

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 dimensions.

When designing a particular system, there is frequently negotiation between the system architect and the software architect over 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. An enterprise architecture need not include computerized 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 software systems support the enterprise's business processes and goals. Typically included in this set of concerns is a process for deciding which systems with which functionality the enterprise should support.

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

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

Sometimes the software infrastructure that supports communication among systems and with the external world is considered a portion of the enterprise architecture; at 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.

Are These Disciplines in Scope for This Book? Yes! (Well, No.)

The system and the enterprise provide environments for, and constraints on, the software architecture. The software architecture must live within the system and the enterprise, and increasingly is the focus for achieving the organization's business goals. Enterprise and system 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 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 exist that do.

1.2 Architectural Structures and Views

Because architectural structures are at the heart of our definition and treatment of software architecture, this section will explore these concepts in more depth. These concepts are dealt with in much greater depth in Chapter 22, where we discuss architecture documentation.

Architectural structures have counterparts in nature. For example, the neurologist, the orthopedist, the hematologist, and the dermatologist all have different views of the various

structures of a human body, as illustrated in Figure 1.1. Ophthalmologists, cardiologists, and podiatrists concentrate on specific subsystems. Kinesiologists and psychiatrists 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 and interconnected: Together they describe the architecture of the human body.

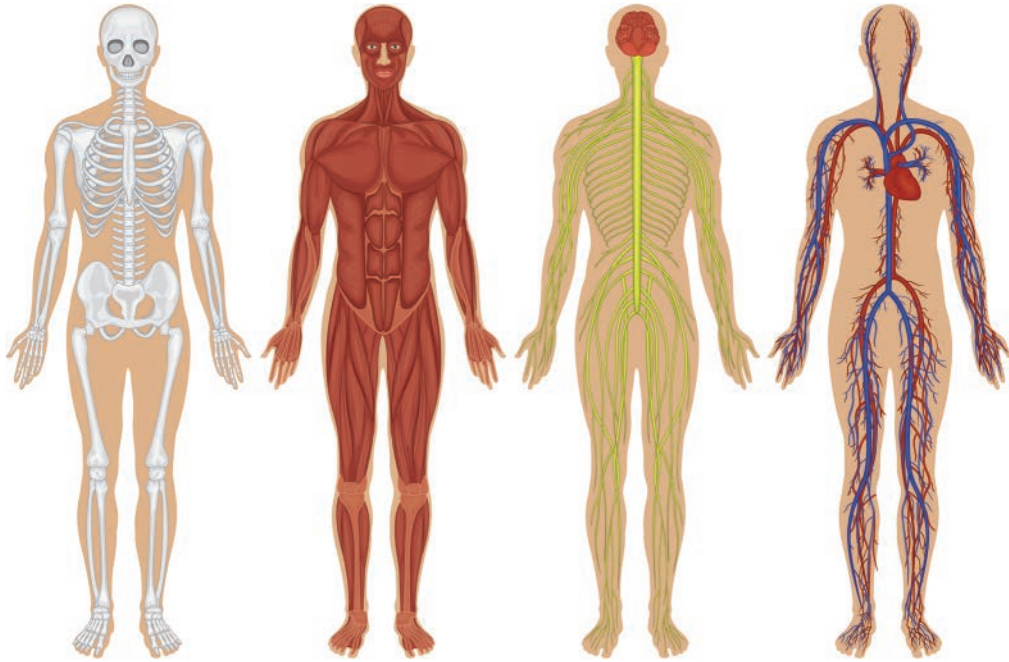


FIGURE 1.1 Physiological structures

Architectural structures also have counterparts in human endeavors. For example, electricians, plumbers, heating and air conditioning specialists, roofers, and framers are each concerned with different structures in a building. You can readily see the qualities that are the focus of each of these structures.

So it is with software.

Three Kinds of Structures

Architectural structures can be divided into three major categories, depending on the broad nature of the elements they show and the kinds of reasoning they support:

1. *Component-and-connector (C&C) structures* focus on the way the elements interact with each other at runtime to carry out the system's functions. They describe how the system is structured as a set of elements that have runtime behavior (components) and interactions (connectors). Components are the principal units of computation and could be services, peers, clients, servers, filters, or many other types of runtime element. Connectors are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others. C&C structures help answer questions such as the following:

- 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?
- Which parts of the system can run in parallel?
- Can the system's structure change as it executes and, if so, how?

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

C&C structures are the most common ones that we see, but two other categories of structures are important and should not be overlooked.

Figure 1.2 shows a sketch of a C&C structure of a system using an informal notation that is explained in the figure's key. The system contains a shared repository that is accessed by servers and an administrative component. A set of client tellers can interact with the account servers and communicate among themselves using a publish-subscribe connector.

2. *Module structures* partition systems into implementation units, which in this book we call *modules*. Module structures show how a system is structured as a set of code or data units that have to be constructed or procured. Modules are assigned specific computational responsibilities and are the basis of work assignments for programming teams. In any module structure, the elements are modules of some kind (perhaps classes, packages, 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 implementations include packages, classes, and layers. Relations among modules in a module structure include uses, generalization (or "is-a"), and "is part of." Figures 1.3 and 1.4 show examples of module elements and relations, respectively, using the Unified Modeling Language (UML) notation.

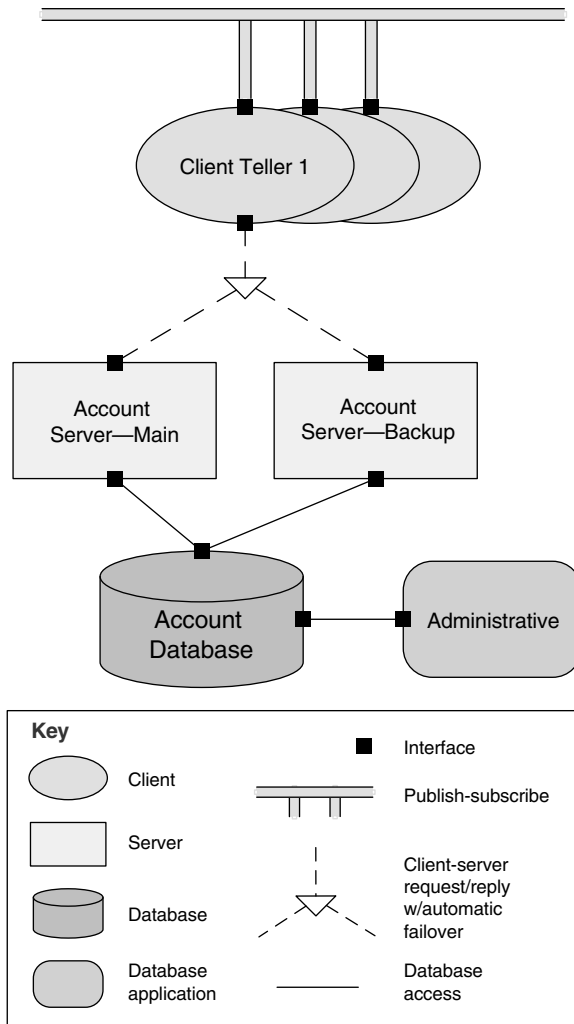


FIGURE 1.2 A component-and-connector structure

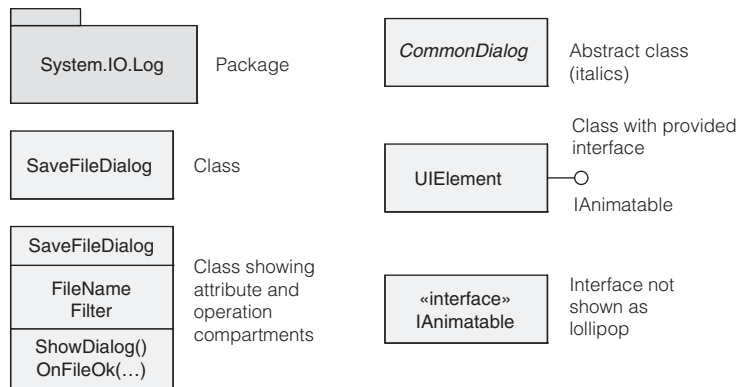


FIGURE 1.3 Module elements in UML

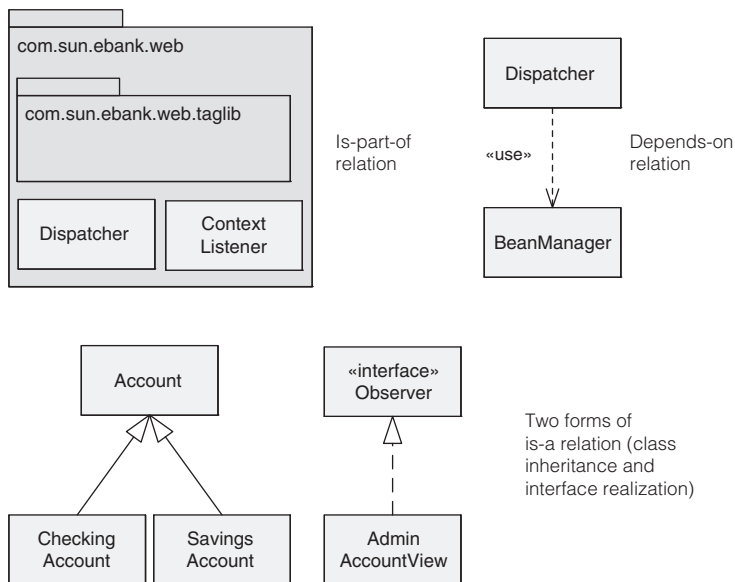


FIGURE 1.4 Module relations in UML

Module structures allow us to answer questions such as the following:

- 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 to answer questions about the impact on the system when the responsibilities assigned to each module change. Thus module structures are the primary tools for reasoning about a system's modifiability.

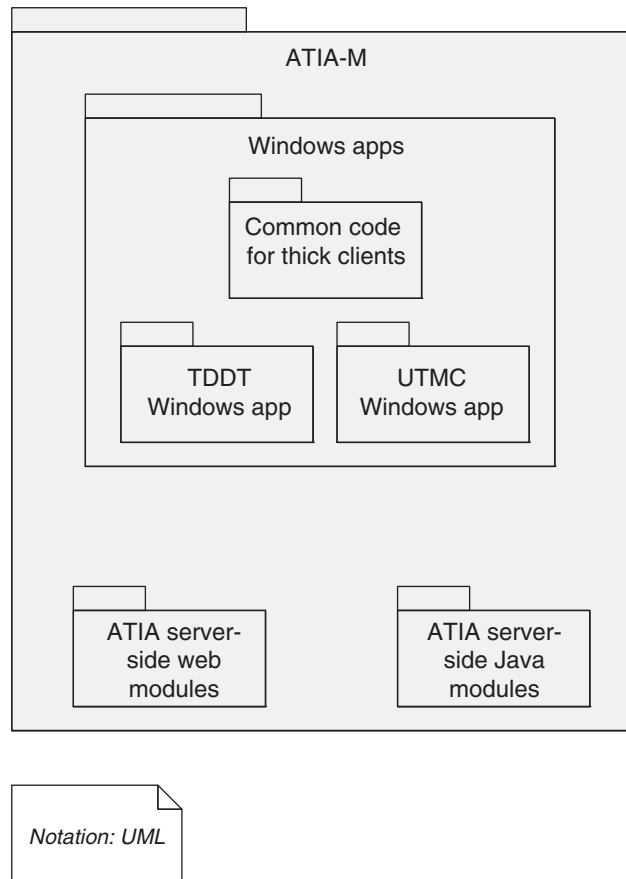
3. *Allocation structures* establish the mapping from software structures to the system's nonsoftware structures, such as its organization, or its development, test, and execution environments. Allocation structures answer questions such as the following:

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

Some Useful Module Structures

Useful module structures include:

- *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, and test plans) associated with them. The decomposition structure determines, to a large degree, the system's modifiability. That is, do changes fall within the purview of 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. Figure 1.5 shows an example of a decomposition structure.
- *Uses structure.* In this important but often overlooked structure, the units are also modules, and perhaps classes. The units are related by the *uses* relation, a specialized form of dependency. One 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. This structure is also the basis

**FIGURE 1.5** A decomposition structure

for measuring social debt—the amount of communication that actually is, as opposed to merely should be, taking place among teams—as it defines which teams should be talking to each other. Figure 1.6 shows a uses structure and highlights the modules that must be present in an increment if the module `admin.client` is present.

- **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 managed fashion; in strictly layered systems, a layer is only allowed to use a single other layer. This structure imbues a system with portability—that is, the ability to change the underlying virtual machine. Figure 1.7 shows a layer structure of the UNIX System V operating system.

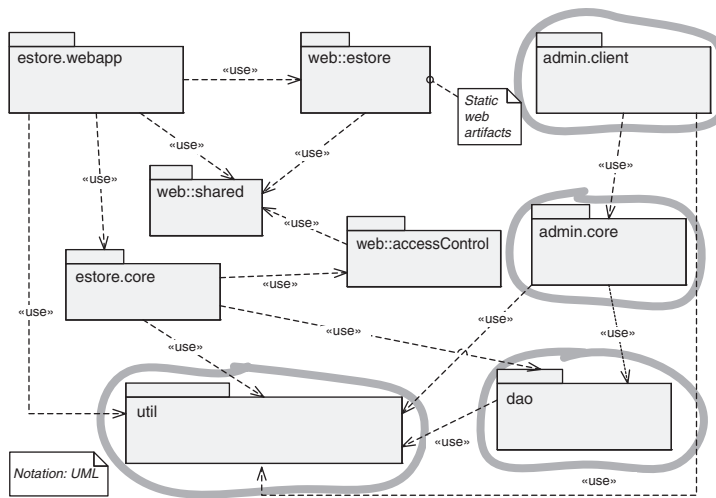


FIGURE 1.6 Uses structure

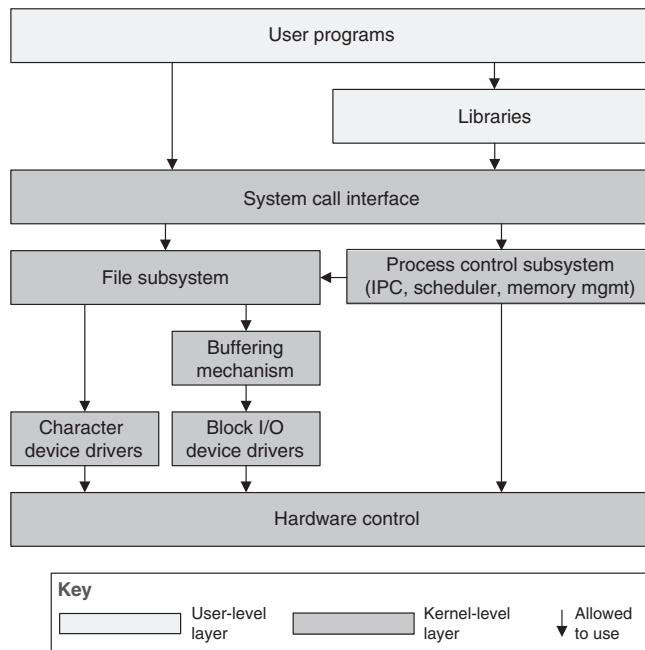


FIGURE 1.7 Layer structure

- *Class (or generalization) structure.* The modules in this structure are called classes, and they are related through an “inherits-from” or “is-an-instance-of” relation. This view supports reasoning about collections of similar behavior or capability 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. Figure 1.8 shows a generalization structure taken from an architectural expert tool.

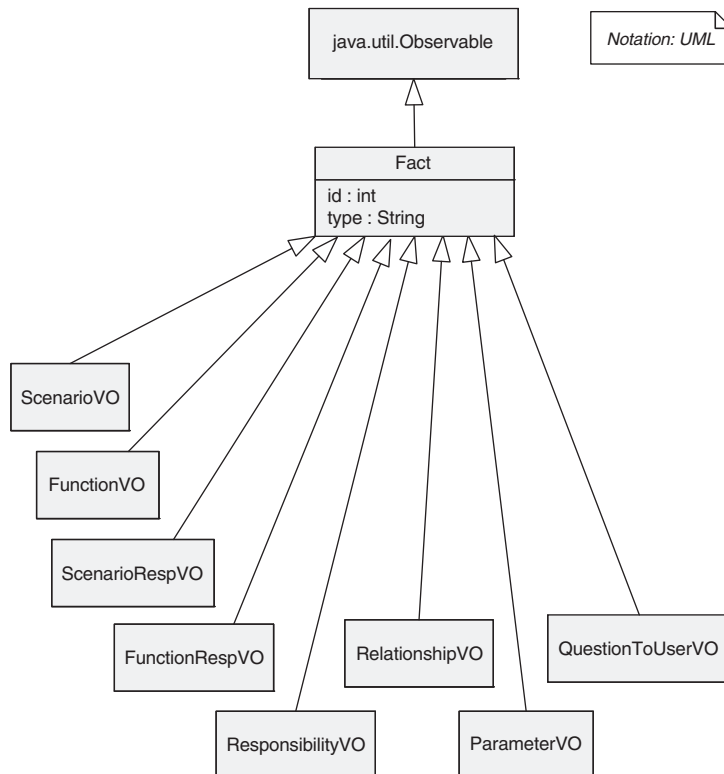


FIGURE 1.8 Generalization structure

- *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 with one or more customers. Figure 1.9 shows an example of a data model.

Some Useful C&C Structures

The relation in all C&C 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:

- *Service structure.* The units here are services that interoperate through a service coordination mechanism, such as messages. The service structure is an important structure to help engineer a system composed of components that may have been developed independently of each other.
- *Concurrency structure.* This C&C 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 and manage 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 (possibly virtualized), teams, and file systems. Useful allocation structures include:

- *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 structure), 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 systems and is the key structure involved in the achievement of the quality attribute of deployability (see Chapter 5). Figure 1.10 shows a simple deployment structure in UML.

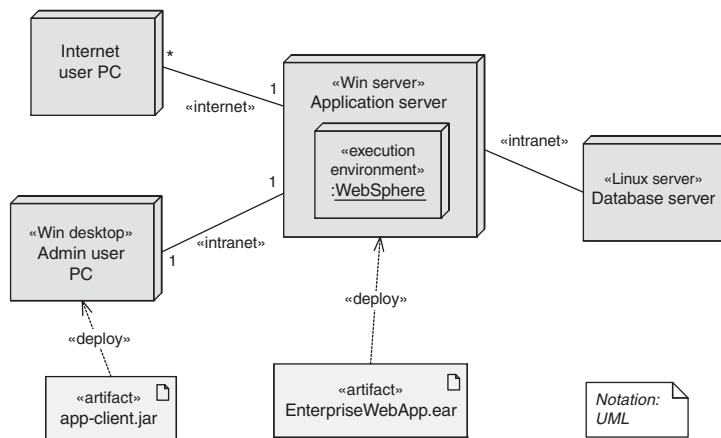


FIGURE 1.10 Deployment structure

- *Implementation structure.* This structure shows how software elements (usually modules) are mapped to the file structures in the system’s development, integration, test, or configuration control environments. This is critical for the management of development activities and build processes.
- *Work assignment structure.* This structure assigns responsibility for implementing and integrating the modules to the teams that will carry out these tasks. 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. Amazon’s decision to devote a single

team to each of its microservices, for example, is a statement about its work assignment structure. On large development projects, it is useful to identify units of functional commonality and assign those to a single team, rather than having them be implemented by everyone who needs them. This structure will also determine the major communication pathways among the teams: regular web conferences, wikis, email lists, and so forth.

Table 1.1 summarizes these structures. It lists the meaning of the elements and relations in each structure and tells what each might be used for.

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 C&C structures, reflecting its runtime alter-ego. In general, mappings between structures are many to many.

Figure 1.11 shows a simple example of how two structures might relate to each other. The image 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 image on the right shows a C&C view of the same system. At runtime, ten clients are running and accessing the server. Thus this little system has two modules and eleven components (and ten connectors).

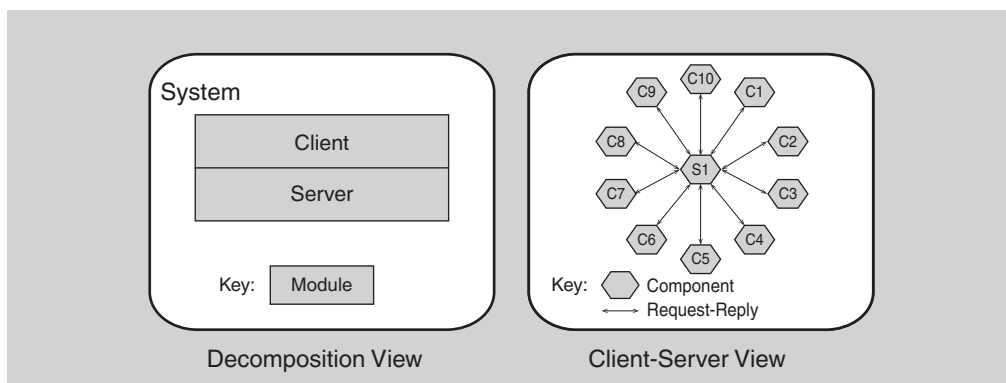


FIGURE 1.11 Two views of a client-server system

TABLE 1.1 Useful Architectural Structures

Software Structure		Element Types	Relations	Useful for	Quality Concerns Affected
Module structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; encapsulation	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Designing subsets and extensions	"Subsetability," extensibility
	Layers	Layer	Allowed to use the services of; provides abstraction to	Incremental development; implementing systems on top of "virtual machines"	Portability, modifiability
	Class	Class, object	Is an instance of; is a generalization of	In object-oriented systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
C&C structures	Data model	Data entity	{one, many}-to-{one, many}; generalizes; specializes	Engineering global data structures for consistency and performance	Modifiability, performance
	Service	Service, service registry	Attachment (via message-passing)	Scheduling analysis; performance analysis; robustness analysis	Interoperability, availability, modifiability
	Concurrency	Processes, threads	Attachment (via communication and synchronization mechanisms)	Identifying locations where resource contention exists, opportunities for parallelism	Performance
	Deployment	Components, hardware elements	Allocated to; migrates to	Mapping software elements to system elements	Performance, security, energy, availability, deployability
Allocation structures	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

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 9, 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 to be dominant and cast other structures, when possible, in terms of the dominant structure. Often, the dominant structure is the module decomposition structure, and for good reason: It tends to spawn the project structure, since 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 at runtime.

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 structures. For example, instead of working with each of several C&C 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 no distribution will occur (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, you should 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 many more are certainly possible. Which ones should an architect choose to work on? Which ones should the architect choose to document? Surely not all of them. 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.

Architectural Patterns

In some cases, architectural elements are composed in ways that solve particular problems. These compositions have been found to be useful over time and over many different domains, so they have been documented and disseminated. These compositions of architectural elements, which provide packaged strategies for solving some of the problems facing a system, are called patterns. Architectural patterns are discussed in detail in Part II of this book.

1.3 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 throw-away 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 such evaluation only makes sense in the context of specific stated goals.

Nevertheless, some rules of thumb should be followed when designing most architectures. Failure to apply any of these guidelines 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. These rules can be applied proactively for greenfield development, to help build the system “right.” Or they can be applied as analysis heuristics, to understand the potential problem areas in existing systems and to guide the direction of its evolution.

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

1. A software (or system) architecture should be the product of a single architect or a small group of architects with an identified technical leader. This approach is important to give 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 architects and the development team, to avoid “ivory tower,” impractical designs.
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*. (A view is simply a representation of one or more architectural structures.) The views should address the concerns of the most important stakeholders in support of the project timeline. This might mean minimal documentation at first, with the documentation then being elaborated later. Concerns usually are related to construction, analysis, and maintenance of the system, as well as education of new stakeholders.
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 via the creation of 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, thereby 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 by using well-known architectural patterns and tactics (described in Chapters 4 through 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, multiple instances of a component may be 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. This is a driving force in the increasing trends toward virtualization and cloud deployment, as we will discuss in Chapters 16 and 17.
7. The architecture should feature a small number of simple component interaction patterns. That is, the system should do the same things in the same way throughout. This practice 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, whose resolution 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 acceptable levels of network traffic. If performance is a concern, the architect should produce (and enforce) time budgets.

1.4 Summary

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

There are three categories of structures:

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

Structures represent the primary engineering leverage points of an architecture. Each structure brings with it the power to manipulate one or more quality attributes. Collectively, structures 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 22, 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 or may not be documented and disseminated.

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

1.5 For Further Reading

If you're keenly interested in software architecture as a field of study, you might be interested in reading some of the pioneering work. Most of it does not mention "software architecture" at all, as this phrase evolved only in the mid-1990s, so you'll have to read between the lines.

Edsger Dijkstra's 1968 paper on the T.H.E. operating system introduced the concept of layers [Dijkstra 68]. The early work of David Parnas laid many conceptual foundations, including information hiding [Parnas 72], program families [Parnas 76], the structures inherent in software systems [Parnas 74], and the uses structure to build subsets and supersets of systems [Parnas 79]. All of Parnas's papers can be found in the more easily accessible collection of his important papers [Hoffman 00]. Modern distributed systems owe their existence to the concept of cooperating sequential processes that (among others) Sir C. A. R. (Tony) Hoare was instrumental in conceptualizing and defining [Hoare 85].

In 1972, Dijkstra and Hoare, along with Ole-Johan Dahl, argued that programs should be decomposed into independent components with small and simple interfaces. They called their

approach structured programming, but arguably this was the debut of software architecture [Dijkstra 72].

Mary Shaw and David Garlan, together and separately, produced a major body of work that helped create the field of study we call software architecture. They established some of its fundamental principles and, among other things, catalogued a seminal family of architectural styles (a concept similar to patterns), several of which appear in this chapter as architectural structures. Start with [Garlan 95].

Software architectural patterns have been extensively catalogued in the series *Pattern-Oriented Software Architecture* [Buschmann 96 and others]. We also deal with architectural patterns throughout Part II of this book.

Early papers on architectural views as used in industrial development projects are [Soni 95] and [Kruchten 95]. The former grew into a book [Hofmeister 00] that presents a comprehensive picture of using views in development and analysis.

A number of books have focused on practical implementation issues associated with architectures, such as George Fairbanks' *Just Enough Software Architecture* [Fairbanks 10], Woods and Rozanski's *Software Systems Architecture* [Woods 11], and Martin's *Clean Architecture: A Craftsman's Guide to Software Structure and Design* [Martin 17].

1.6 Discussion Questions

1. 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?
2. Discuss how an architecture serves as a basis for analysis. What about decision making? What kinds of decision making does an architecture empower?
3. What is architecture's role in project risk reduction?
4. Find a commonly accepted definition of *system architecture* and discuss what it has in common with software architecture. Do the same for *enterprise architecture*.
5. Find a published example of a software architecture. Which structures are shown? Given its purpose, which structures *should* have been shown? What analysis does the architecture support? Critique it: What questions do you have that the representation does not answer?
6. 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.

7. Aircraft have architectures that can be characterized by how they resolve some major design questions, such as engine location, wing location, landing gear layout, and more. For many decades, most jet aircraft designed for passenger transport have the following characteristics:

- Engines housed in nacelles slung underneath the wing (as opposed to engines built into the wings, or engines mounted on the rear of the fuselage)
- Wings that join the fuselage at the bottom (as opposed to the top or middle)

First, do an online search to find an example and a counter-example of this type of design from each of the following manufacturers: Boeing, Embraer, Tupolev, and Bombardier. Next, do some online research and answer the following question: What qualities important to aircraft does this design provide?

This page intentionally left blank

2



Why Is Software Architecture Important?

*Ah, to build, to build!
That is the noblest art of all the arts.*
—Henry Wadsworth Longfellow

If architecture is the answer, what was the question?

This chapter focuses on why architecture matters from a technical perspective. We will examine a baker's dozen of the most important reasons. You can use these reasons to motivate the creation of a new architecture, or the analysis and evolution of an existing system's architecture.

1. An architecture can either 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 incremental development.
9. An architecture is the key artifact that allows the architect and the 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 of a new team member.

Even if you already believe us that architecture is important and don't need that point hammered home 13 more times, think of these 13 points (which form the outline for this chapter) as 13 useful ways to use architecture in a project, or to justify the resources devoted to architecture.

2.1 Inhibiting or Enabling a System's Quality Attributes

A system's ability to meet its desired (or required) quality attributes is substantially determined by its architecture. If you remember nothing else from this book, remember that.

This relationship is so important that we've devoted all of Part II of this book to expound on 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 their interelement communication.
- If modifiability is important, then you need to pay attention to assigning responsibilities to elements and limiting the interactions (coupling) of those 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 interelement 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 to set up a strong "perimeter" to guard against intrusion.
- If you want your system to be safe and secure, you need to design in safeguards and recovery mechanisms.
- If you believe that scalability of performance will be important to the success of your system, then you need to localize the use of resources to facilitate the 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 manage intercomponent usage.
- If you want the elements from your system to be reusable in other systems, then you need to restrict interelement 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): What the architecture giveth, the implementation may taketh away. Decisions at all stages of the life cycle—from architectural design to coding and implementation and testing—affect system quality. Therefore, quality is not completely a function of an architectural design. But that's where it starts.

2.2 Reasoning about and Managing Change

This 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. 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 lifetimes, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But the reality is that these changes are often fraught with difficulty.

Every architecture, no matter what it is, 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 applying 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 single-threaded to multi-threaded.

Obviously, local changes are the most desirable, so an *effective* architecture is one in which the most common changes are local, and hence easy to make. Nonlocal changes are not as desirable but do have the virtue that they can usually be staged—that is, rolled out—in an orderly manner over time. For example, you might first make changes to add a new pricing rule, then make the changes to actually deploy the new rule.

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 the relationships, performance, and behaviors of system software elements. These tasks are all part of the job description for an architect. Reasoning about the architecture and analyzing the architecture can provide the insights necessary to make decisions about anticipated changes. If you do not take this step, and if you do not pay attention to maintaining the conceptual integrity of your architecture, then you will almost certainly accumulate *architecture debt*. We deal with this subject in Chapter 23.

2.3 Predicting System Qualities

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

This may seem obvious, but it need not be the case. Then designing an architecture would consist of making a series of pretty much random design decisions, building the system, testing for quality attributes, and hoping for the best. Oops—not fast enough or hopelessly vulnerable to attacks? Start hacking.

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 determine whether those decisions have been made and confidently predict that the architecture will exhibit the associated qualities.

This point and the previous point, taken together, mean that architecture largely determines system qualities and—even better!—we know how it does so, and we know how to make it do so.

Even if you don't perform 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 early.

2.4 Communication among Stakeholders

One point made in Chapter 1 is that an architecture is an abstraction, and that is useful because it represents a simplified model of the whole system that (unlike the infinite details of the whole system) you can keep in your head. So can others on your team. 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—are sufficiently abstract that most nontechnical people can understand it to the extent they need to, particularly with some coaching from the architect, and yet that abstraction can be refined into sufficiently rich technical specifications to guide implementation, integration, testing, 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 (who pays for the system) is concerned that the architecture can be implemented on schedule and according to budget;

- the manager is worried that (in addition to cost and schedule concerns) the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways; and
- 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.

Chapter 22, on architecture documentation, 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 it was large enough that these lapses were attracting the U.S. Congress’s 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 presentation of the software architecture. 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 slide 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, as 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 turn 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—an absolutely essential conversation that should have happened when the requirements were being formulated but, for whatever reason, had not.

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 system understanding?

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, 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 possibility 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 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 sequence of decisions. When painting a picture, an artist decides on the material for the canvas and 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, and 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. 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 an avalanche.

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 information that flows through the system be encrypted?
- Which operating system will we use?
- Which communication protocol will we choose?

Imagine the nightmare of having to change any of these or a myriad of other related decisions. Decisions like these begin to flesh out some of the structures of the architecture and their interactions.

2.6 Constraints on Implementation

If you want your implementation to conform to an architecture, then it must conform to the design decisions prescribed by the architecture. It must have the set of elements prescribed by the architecture, these elements must interact with each other in the fashion prescribed by the architecture, and each element must fulfill its responsibility to the other elements as prescribed 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 is when an architect assigns performance budgets 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, but 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. Architects, however, are the people responsible for establishing, analyzing, and enforcing the architectural decisions and tradeoffs.

2.7 Influences on 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 so-called work-breakdown structure of a system is manifested in the architecture in the work assignment structure described in Chapter 1. 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; inter-team 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 their elements. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific 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 may 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 upon, it becomes very costly—for managerial and business reasons—to significantly modify it. This is one argument (among many) for analyzing the software architecture for a large system before settling on a specific choice.

2.8 Enabling Incremental Development

Once an architecture has been defined, it can serve as the basis for incremental development. The first increment can be a skeletal system 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 present, but much of the system’s application functionality is not.

Building the infrastructure and building the application functionality can go hand in hand. Design and build a little infrastructure to support a little end-to-end functionality; repeat until done.

Many systems are built as skeletal systems that can be extended using plug-ins, packages, or extensions. Examples include the R language, Visual Studio Code, and most web browsers. The extensions, when added, provide additional functionality over and above what is present in the skeleton. This approach aids the development process by ensuring that the system is executable early in the product’s life cycle. The fidelity of the system increases as extensions are added, or early versions are replaced by more complete versions of these parts of the software. In some cases, the parts may be low-fidelity versions or prototypes of the final functionality; in other cases, they may be *surrogates* that consume and produce data at the appropriate rates but do little else. Among other things, this allows potential performance (and other) problems to be identified early in the product’s life cycle.

This practice gained attention in the early 2000s through the ideas of Alistair Cockburn and his notion of a “walking skeleton.” More recently, it has been adopted by those employing MVP (minimum viable product) as a strategy for risk reduction.

The benefits of incremental development include a reduction of the potential risk in the project. If the architecture is for a family of related systems, the infrastructure can be reused across the family, lowering the per-system cost of each.

2.9 Cost and Schedule Estimates

Cost and schedule estimates are an important tool for the project manager. They help the project manager acquire the necessary resources as well as monitor progress on the project. One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project’s life cycle. While top-down estimates are useful for setting goals and apportioning budgets, cost estimations based on a bottom-up understanding of the system’s pieces are typically more accurate than those 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 can, and will feel more ownership in making those 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 the project manager) and the bottom-up estimates (created by the developers). The discussion and negotiation that result from this process create a far more accurate estimate than the use of 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 24 delves into the use of architecture in project management.

2.10 Transferable, Reusable Model

The earlier in the life cycle reuse is applied, the greater the benefit that can be achieved from this practice. While code reuse offers a benefit, reuse of architectures provides opportunities for tremendous leverage for systems with similar requirements. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we described in earlier sections are also transferred to those systems.

A product line or family is a set of systems that are all built using the same set of shared assets—software components, requirements documents, test cases, and so forth. 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.

Product lines represent a powerful approach to multi-system development that has shown order-of-magnitude payoffs in time to market, cost, productivity, and product quality. The power of architecture lies at the heart of this paradigm. Similar to other capital investments, architectures for product lines become a developing organization's shared asset.

2.11 Architecture Allows Incorporation of Independently Developed Elements

Whereas earlier software paradigms 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, which data they consume and produce, how they access data, and which protocols they use for communication and resource sharing. We elaborate on these ideas in Chapter 15.

Commercial off-the-shelf components, open source software, publicly available apps, and networked services are all examples of independently developed elements. The complexity and ubiquity of integrating many independently developed elements into your system have spawned an entire industry of software tools, such as Apache Ant, Apache Maven, MSBuild, and Jenkins.

For software, the payoffs can take the following forms:

- Decreased time to market (It should be easier to use someone else’s ready solution than to 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 its customer base.)
- Flexibility (If the element you want to buy is not terribly special-purpose, it’s likely to be available from several sources, which in turn increases your buying leverage.)

An *open system* is one that defines a set of standards for software elements—how they behave, how they interact with other elements, how they share data, and so forth. The goal of an open system is to enable, and even encourage, many different suppliers to be able to produce elements. This can avoid “vendor lock-in,” a situation in which a single vendor is the only one who can provide an element and charges a premium price for doing so. Open systems are enabled by an architecture that defines the elements and their interactions.

2.12 Restricting the Vocabulary of Design Alternatives

As useful architectural solutions 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* where creativity and freedom are paramount. Instead, engineering is about discipline, and discipline comes, in part, by *restricting* the vocabulary of alternatives to proven solutions. Examples of these proven design solutions include tactics and patterns, which will be discussed extensively in Part II. Reusing off-the-shelf elements is another approach to restricting your design vocabulary.

Restricting your design vocabulary to proven solutions can yield the following benefits:

- Enhanced reuse
- More regular and simpler designs that are more easily understood and communicated, and bring more reliably predictable outcomes
- Easier analysis with greater confidence
- Shorter selection time
- Greater interoperability

Unprecedented designs are risky. Proven designs are, well, proven. This is not to say that software design can never be innovative or offer new and exciting solutions. It can. But these solutions should not be invented for the sake of novelty; rather, they should be sought when existing solutions are insufficient to solve the problem at hand.

Properties of software follow from the choice of architectural tactics or patterns. Tactics and patterns that are more desirable for a particular problem should improve the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insights into poorly understood design contexts, and by helping surface inconsistencies in requirements. We will discuss architectural tactics and patterns in Part II.

2.13 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 important use of software architecture is to support and encourage communication among the various stakeholders. The architecture serves as a common reference point for all of these people.

Module views are excellent means of 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 choices for explaining how the system is expected to work and accomplish its job. Allocation views show a new project member where their assigned part fits into the project's development or deployment environment.

2.14 Summary

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

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 incremental development.

9. An architecture is the key artifact that allows the architect and the 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 productively 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

The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise by Gregor Hohpe describes the unique ability of architects to interact with people at all levels inside and outside an organization, and facilitate stakeholder communication [Hohpe 20].

The granddaddy of papers about architecture and organization is by [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."

Cockburn's notion of the walking skeleton is described in *Agile Software Development: The Cooperative Game* [Cockburn 06].

A good example of an open systems architecture standard is AUTOSAR, developed for the automotive industry (autosar.org).

For a comprehensive treatment on building software product lines, see [Clements 16]. Feature-based product line engineering is a modern, automation-centered approach to building product lines that expands the scope from software to systems engineering. A good summary may be found at [INCOSE 19].

2.16 Discussion Questions

1. If you remember nothing else from this book, remember . . . what? Extra credit for not peeking.
2. For each of the 13 reasons why architecture is important articulated in this chapter, 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 13 reasons.)
3. 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 13 points?

4. 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?
5. Prioritize the list of 13 reasons in this chapter according to some criteria that are 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



Understanding Quality Attributes

Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution.

—William A. Foster

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 the achievement of 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 4–14, we discuss how various qualities are supported by architectural design decisions. In Chapter 20, we show how to integrate all of your drivers, including quality attribute decisions, into a coherent 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 beyond the basic function of the system. You can think of a quality attribute as measuring the “utility” 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 exhibit
- How to achieve the qualities through architectural means
- How to determine the design decisions we might make with respect to the qualities

This chapter provides the context for the discussions of individual quality attributes in Chapters 4–14.

3.1 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 sub-pieces 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 examine in the remaining sections of this chapter, and in the subsequent quality attribute chapters in Part II.

Although functionality is independent of any particular structure, it is achieved by assigning responsibilities to architectural elements. This process results in one of the most basic architectural structures—module decomposition.

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

Functional Requirements

After more than 30 years of writing about 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 system's timing behavior, modifiability has to do with the system's ability to support changes in its behavior or other qualities after initial deployment, availability has to do with the system's ability to survive failures, and so forth.

Function, however, is a much more slippery concept. 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 by requiring a user name/password combination not a function, even though it is not the purpose of any system?

I much prefer using 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”—which is probably equally ineffectually.

—LB

3.2 Quality Attribute Considerations

Just as a system’s functions do not stand on their own without due consideration of 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 is allowed to fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Quality attributes as a distinct topic have been studied by the software community at least since the 1970s. A variety of taxonomies and definitions have been published (we discuss some of these in Chapter 14), many of which have their own research and practitioner communities. However, there are three problems with most 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 will 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 issue belongs to. Is a denial-of-service attack on a system 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 the denial-of-service attack. All are, to some extent, correct. But this debate over categorization doesn’t help us, as architects, understand and create architectural solutions to actually 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 “faults” arriving, 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 problems (untestable definitions and overlapping issues) is to use *quality attribute scenarios* as a means of characterizing quality attributes (see Section 3.3). A solution to the third problem is to illustrate the concepts that are fundamental to that attribute community in a common form, which we do in Chapters 4–14.

We will focus on two categories of quality attributes. The first category includes those attributes that describe some property of the system at runtime, such as availability, performance, or usability. The second category includes those that describe some property of the development of the system, such as modifiability, testability, or deployability.

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, which then hurts performance. Determining a design that may satisfy quality attribute requirements is partially a matter of making the appropriate tradeoffs; we discuss design in Chapter 21.

In the next three sections, 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.

3.3 Specifying Quality Attribute Requirements: Quality Attribute Scenarios

We use a common form to specify all QA requirements as scenarios. This addresses the vocabulary problems we identified previously. The common form is testable and unambiguous; it is not sensitive to whims of categorization. Thus it provides regularity in how we treat all quality attributes.

Quality attribute scenarios have six parts:

- *Stimulus*. We use the term “stimulus” to describe an event arriving at the system or the project. The stimulus can be an *event* to the performance community, a *user operation* to the usability community, or an *attack* to the security community, and so forth. 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 unit of development.
- *Stimulus source*. A stimulus must have a source—it must come from somewhere. Some entity (a human, a computer system, or any other actor) must have generated the

stimulus. 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.* The response is the activity that occurs as the result of the arrival of the stimulus. The response is something the architect undertakes to satisfy. It 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.* When the response occurs, it should be measurable in some fashion so that the scenario can be tested—that is, so that we can determine if the architect achieved it. 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 two more characteristics are important, yet often overlooked: environment and artifact.

- *Environment.* The environment is the set of circumstances in which the scenario takes place. Often this refers to a runtime state: 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. But the environment can also refer to states in which the system is not running at all: when it is in development, or testing, or refreshing its data, or recharging its battery between runs. The environment sets the context for the rest of the scenario. 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. The fifth successive failure of a component may be treated differently than the first failure of that component.
- *Artifact.* The stimulus arrives at some target. This is often captured as just the system or project itself, but it’s helpful to be more precise if possible. The artifact may be a collection of systems, the whole system, or one or more pieces of the system. A failure or a change request may affect just a small portion of the system. 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, we capture quality attribute requirements 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 of the parts are there forces the architect to consider whether each part is relevant.

We have created a general scenario for each of the quality attributes presented in Chapters 4–13 to facilitate brainstorming and elicitation of concrete scenarios. We distinguish

general quality attribute scenarios—general scenarios—which are system independent and can pertain to any system, from *concrete* quality attribute scenarios—concrete scenarios—which are specific to the particular system under consideration.

To translate these generic attribute characterizations into requirements for a particular system, the general scenarios need to be made system specific. But, as we have found, it is much easier for a stakeholder to tailor a general scenario into one that fits their system than it is for them to generate a scenario from thin air.

Figure 3.1 shows the parts of a quality attribute scenario just discussed. Figure 3.2 shows an example of a general scenario, in this instance for availability.

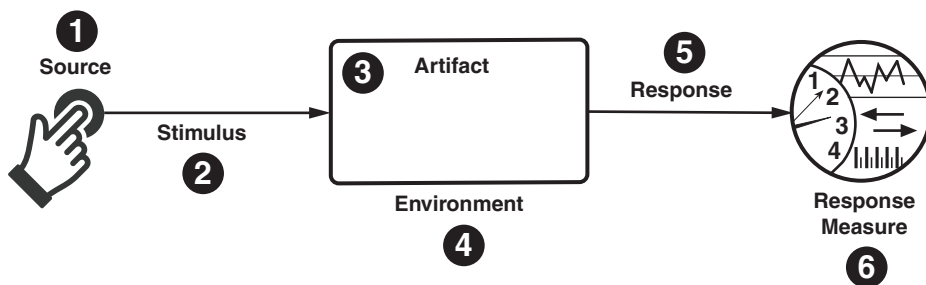


FIGURE 3.1 The parts of a quality attribute scenario

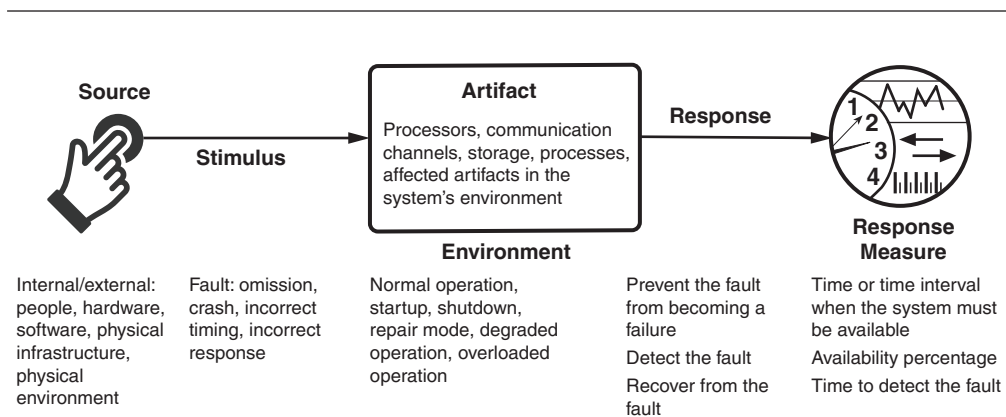


FIGURE 3.2 A general scenario for availability

Not My Problem

Some time ago I was doing an architecture analysis on a complex system created by and for Lawrence Livermore National Laboratory. If you visit this organization's website (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 my clients 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 not the software architects. The lesson here is that the software architect may not bear the responsibility for every QA requirement.

—RK

3.4 Achieving Quality Attributes through Architectural Patterns and Tactics

We now turn to the techniques an architect can use to *achieve* the required quality attributes: architectural patterns and tactics.

A tactic is a design decision that influences the achievement of a quality attribute response—it directly affects the system's response to some stimulus. Tactics may impart portability to one design, high performance to another, and integrability to a third.

An architectural pattern describes a particular recurring design problem that arises in specific design contexts and presents a well-proven architectural solution for the problem. The solution is specified by describing the roles of its constituent elements, their responsibilities and relationships, and the ways in which they collaborate. Like the choice of tactics, the choice of an architectural pattern has a profound effect on quality attributes—usually more than one.

Patterns typically comprise multiple design decisions and, in fact, often comprise multiple quality attribute tactics. We say that patterns often bundle tactics and, consequently, frequently make tradeoffs among quality attributes.

We will look at example relationships between tactics and patterns in each of our quality attribute-specific chapters. Chapter 14 explains how a set of tactics for any quality attribute can be constructed; those tactics are, in fact, the steps we used to produce the sets found in this book.

While we discuss patterns and tactics as though they were foundational design decisions, the reality is that architectures often emerge and evolve as a result of many small decisions and business forces. For example, a system that was once tolerably modifiable may deteriorate over time, through the actions of developers adding features and fixing bugs. Similarly, a system's performance, availability, security, and any other quality may (and typically does) deteriorate over time, again through the well-intentioned actions of programmers who are focused on their immediate tasks and not on preserving architectural integrity.

This “death by a thousand cuts” is common on software projects. Developers may make suboptimal decisions due to a lack of understanding of the structures of the system, schedule pressures, or perhaps a lack of clarity in the architecture from the start. This kind of deterioration is a form of technical debt known as architecture debt. We discuss architecture debt in Chapter 23. To reverse this debt, we typically refactor.

Refactoring may be done for many reasons. For example, you might refactor a system to improve its security, placing different modules into different subsystems based on their security properties. Or you might refactor a system to improve its performance, removing bottlenecks and rewriting slow portions of the code. Or you might refactor to improve the system's modifiability. For example, when two modules are affected by the same kinds of changes over and over because they are (at least partial) duplicates of each other, the common functionality could be factored out into its own module, thereby improving cohesion and reducing the number of places that need to be changed when the next (similar) change request arrives.

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.

Successfully achieving quality attributes often involves process-related decisions, in addition to architecture-related decisions. For example, a great security architecture is worthless if your employees are susceptible to phishing attacks or do not choose strong passwords. We are not dealing with the process aspects in this book, but be aware that they are important.

3.5 Designing with Tactics

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 depict this relationship in Figure 3.3. Tactics, like patterns, are design techniques that architects have been using for years. In this book, we isolate, catalog, and describe them. We are not inventing tactics here, but rather just capturing what good architects do in practice.

Why do we focus on tactics? There are three reasons:

1. Patterns are foundational for many architectures, but sometimes there may be no pattern that solves your problem completely. For example, you might need the high-availability high-security broker pattern, not the textbook broker pattern. Architects frequently need to modify and adapt patterns to their particular context, and tactics provide a systematic means for augmenting an existing pattern to fill the gaps.

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. Tactics provide a way of making design and analysis more systematic within some limitations. We'll explore this idea in the next section.

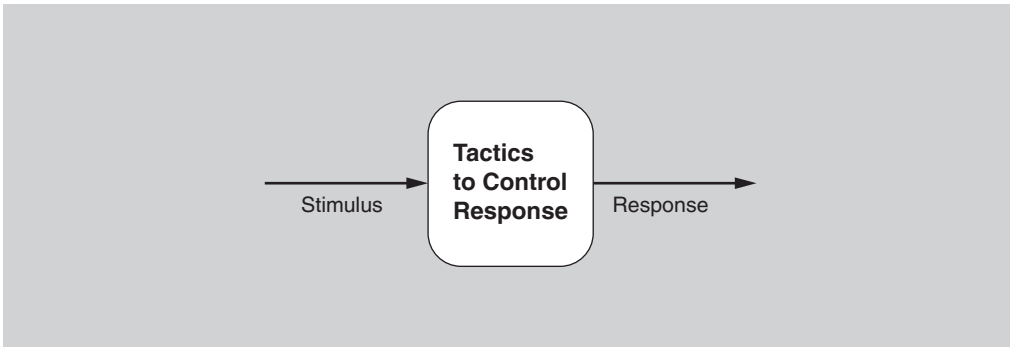


FIGURE 3.3 Tactics are intended to control responses to stimuli.

Like any design concept, the tactics that we present here can and should be refined as they are applied to design a system. 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, proxies, and tiers, to name just a few), which are realized in different ways. Thus a designer will employ refinements to make each tactic concrete.

In addition, the application of a tactic depends on the context. Again, consider performance: *Manage sampling rate* is relevant in some real-time systems but not in all real-time systems, and certainly not in database systems or stock-trading systems where losing a single event is highly problematic.

Note that there are some “super-tactics”—tactics that are so fundamental and so pervasive that they deserve special mention. For example, the modifiability tactics of encapsulation, restricting dependencies, using an intermediary, and abstracting common services are found in the realization of almost every pattern ever! But other tactics, such as the scheduling tactic from performance, also appear in many places. For example, a load balancer is an intermediary that does scheduling. We see monitoring appearing in many quality attributes: We monitor aspects of a system to achieve energy efficiency, performance, availability, and safety. Thus we should not expect a tactic to live in only one place, for just a single quality attribute. Tactics are design primitives and, as such, are found over and over in different aspects of design. This is actually an argument for why tactics are so powerful and deserving of our attention—and yours. Get to know them; they’ll be your friends.

3.6 Analyzing Quality Attribute Design Decisions: Tactics-Based Questionnaires

In this section, we introduce a tool the analyst can use to understand potential quality attribute behavior at various stages through the architecture's design: tactics-based questionnaires.

Analyzing how well quality attributes have been achieved is a critical part of the task of designing an architecture. And (no surprise) you shouldn't wait until your design is complete before you begin to do it. Opportunities for quality attribute analysis crop up at many different points in the software development life cycle, even very early ones.

At any point, the analyst (who might be the architect) needs to respond appropriately to whatever artifacts have been made available for analysis. The accuracy of the analysis and expected degree of confidence in the analysis results will vary according to the maturity of the available artifacts. But no matter the state of the design, we have found tactics-based questionnaires to be helpful in gaining insights into the architecture's ability (or likely ability, as it is refined) to provide the needed quality attributes.

In Chapters 4–13, we include a tactics-based questionnaire for each quality attribute covered in the chapters. For each question in the questionnaire, the analyst records the following information:

- Whether each tactic is supported by the system's architecture.
- Whether there are any obvious risks in the use (or nonuse) of this tactic. If the tactic has been used, record how it is realized in the system, or how it is intended to be realized (e.g., via custom code, generic frameworks, or externally produced components).
- The specific design decisions made to realize the tactic and where in the code base the implementation (realization) may be found. This is useful for auditing and architecture reconstruction purposes.
- Any rationale or assumptions made in the realization of this tactic.

To use these questionnaires, simply follow these four steps:

1. For each tactics question, fill the "Supported" column with "Y" if the tactic is supported in the architecture and with "N" otherwise.
2. If the answer in the "Supported" column is "Y," then in the "Design Decisions and Location" column describe the specific design decisions made to support the tactic and enumerate where these decisions are, or will be, manifested (located) in the architecture. For example, indicate which code modules, frameworks, or packages implement this tactic.
3. In the "Risk" column indicate the risk of implementing the tactic using a (H = High, M = Medium, L = Low) scale.
4. In the "Rationale" column, describe the rationale for the design decisions made (including a decision to *not* use this tactic). Briefly explain the implications of this decision. For example, explain the rationale and implications of the decision in terms of the effort on cost, schedule, evolution, and so forth.

While this questionnaire-based approach might sound simplistic, it can actually be very powerful and insightful. Addressing the set of questions forces the architect to take a step back and consider the bigger picture. This process can also be quite efficient: A typical questionnaire for a single quality attribute takes between 30 and 90 minutes to complete.

3.7 Summary

Functional requirements are satisfied by including an appropriate set of responsibilities within the design. *Quality attribute* requirements are satisfied by the structures and behaviors of the architecture.

One challenge in architectural design is that these requirements are often captured poorly, if at all. To capture and express a quality attribute requirement, we recommend the use of a quality attribute scenario. Each scenario consists of six parts:

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. An architectural pattern describes a particular recurring design problem that arises in specific design contexts and presents a well-proven architectural solution for the problem. Architectural patterns can be seen as “bundles” of tactics.

An analyst can understand the decisions made in an architecture through the use of a tactics-based checklist. This lightweight architecture analysis technique can provide insights into the strengths and weaknesses of the architecture in a very short amount of time.

3.8 For Further Reading

Some extended case studies showing how tactics and patterns are used in design can be found in [Cervantes 16].

A substantial catalog of architectural patterns can be found in the five-volume set *Pattern-Oriented Software Architecture*, by Frank Buschmann et al.

Arguments showing that many different architectures can provide the same functionality—that is, that architecture and functionality are largely orthogonal—can be found in [Shaw 95].

3.9 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. Enumerate the set of responsibilities that an automatic teller machine should support and propose a design to accommodate that set of responsibilities. Justify your proposal.
4. Choose an architecture that you are familiar with (or choose the ATM architecture you defined in question 3) and walk through the performance tactics questionnaire (found in Chapter 9). What insight did these questions provide into the design decisions made (or not made)?

4



Availability

*Technology does not always rhyme
with perfection and reliability.
Far from it in reality!*
—Jean-Michel Jarre

Availability refers to a property of software—namely, 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). Availability builds on 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, as we’ll see in this chapter.

Availability also encompasses the ability of a system to mask or repair faults such that they do not become failures, thereby ensuring that the cumulative service outage period does not exceed a required value over a specified time interval. This definition subsumes concepts of reliability, robustness, and any other quality attribute that involves a concept of unacceptable failure.

A failure is the deviation of the system from its specification, where that deviation is externally visible. Determining that a failure has occurred requires some external observer in the environment.

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. Through these actions, 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.

Availability is closely related to, but clearly distinct from, 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, since it may be difficult to tell when a system has failed

and when it is simply being egregiously 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.

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. Once those are understood, mitigation strategies can be designed into the system.

Since a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be an imperceptible delay in a user's 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" is critical here: If a failure *could have been observed*, then it is a failure, whether or not it was actually observed.

In addition, we are often concerned with the level of capability that remains when a failure has occurred—a degraded operating mode.

Distinguishing between faults and failures allows us to discuss repair strategies. If code containing a fault is executed but the system is able to recover from the fault without any observable deviation from the otherwise specified behavior, we say that no failure has occurred.

The availability of a system can be measured as the probability that it will provide the specified services within the required bounds over a specified time interval. A well-known expression is used to derive steady-state availability (which came from the world of hardware):

$$MTBF/(MTBF + MTTR)$$

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 it is that such an event will occur, and how much time will be required to repair it.

From this formula, it is possible to calculate probabilities and make claims like "the system exhibits 99.999 percent availability" or "there is 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) should not be considered when calculating availability, since the system is deemed "not needed" then; of course, this is dependent on the specific requirements for the system, which are often encoded in a service level agreement (SLA). This 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.

Detected faults can be categorized prior to being reported and repaired. This categorization is commonly based on the fault's severity (critical, major, or minor) and service impact (service-affecting or non-service-affecting). It provides the system operator with a timely and accurate system status and allows for an appropriate repair strategy to be employed. The repair strategy may be automated or may require manual intervention.

As just mentioned, the availability expected of a system or service is frequently expressed as an SLA. The SLA specifies the availability level that is guaranteed and, usually, the

penalties that the provider will suffer if the SLA is violated. For example, Amazon provides the following SLA for its EC2 cloud service:

AWS will use commercially reasonable efforts to make the Included Services each available for each AWS region with a Monthly Uptime Percentage of at least 99.99%, in each case during any monthly billing cycle (the “Service Commitment”). In the event any of the Included Services do not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Table 4.1 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. As mentioned earlier, only unscheduled outages contribute to system downtime.

TABLE 4.1 System Availability Requirements

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

4.1 Availability General Scenario

We can now describe the individual portions of an availability general scenario as summarized in Table 4.2.

TABLE 4.2 Availability General Scenario

Portion of Scenario	Description	Possible Values
Source	This specifies where the fault comes from.	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	The stimulus to an availability scenario is a fault.	Fault: omission, crash, incorrect timing, incorrect response
Artifact	This specifies which portions of the system are responsible for and affected by the fault.	Processors, communication channels, storage, processes, affected artifacts in the system’s environment

continues

TABLE 4.2 Availability General Scenario *continued*

Portion of Scenario	Description	Possible Values
Environment	We may be interested in not only how a system behaves in its “normal” environment, but also how it behaves in situations such as when it is already recovering from a fault.	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	The most commonly desired response is to prevent the fault from becoming a failure, but other responses may also be important, such as notifying people or logging the fault for later analysis. This section specifies the desired system response.	Prevent the fault from becoming a failure Detect the fault: <ul style="list-style-type: none"> ▪ Log the fault ▪ Notify the appropriate entities (people or systems) ▪ Recover from the fault ▪ Disable the source of events causing the fault ▪ Be temporarily unavailable while a repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while a repair is being effected
Response measure	We may focus on a number of measures of availability, depending on the criticality of the service being provided.	<ul style="list-style-type: none"> ▪ Time or time interval when the system must be available ▪ Availability percentage (e.g., 99.999 percent) ▪ Time to detect the fault ▪ Time to repair the fault ▪ Time or time interval in which system can be in degraded mode ▪ Proportion (e.g., 99 percent) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

An example concrete availability scenario derived from the general scenario in Table 4.2 is shown in Figure 4.1. The scenario is this: *A server in a server farm fails during normal operation, and the system informs the operator and continues to operate with no downtime.*

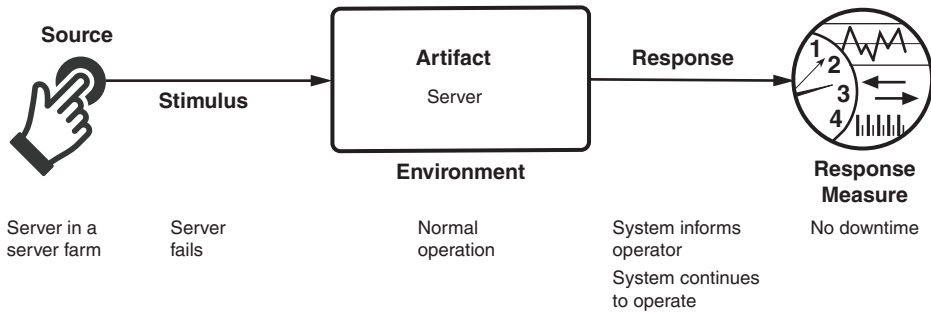


FIGURE 4.1 Sample concrete availability scenario

4.2 Tactics for Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification and this failure is observable by the system's actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, in turn, are designed to enable a system to prevent or 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, as illustrated in Figure 4.2.

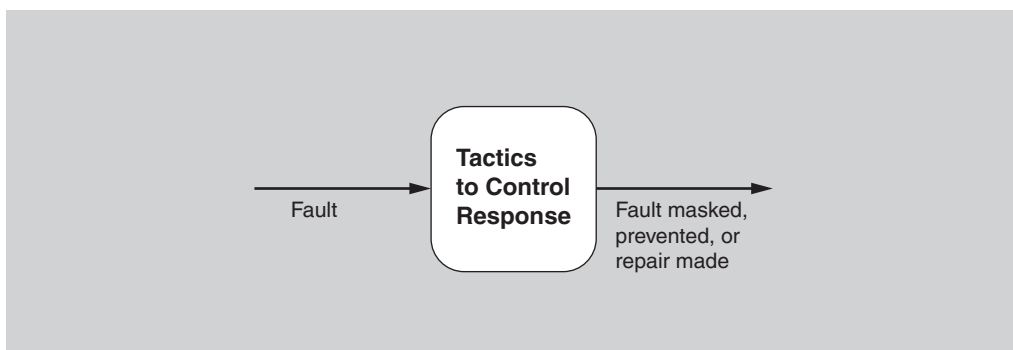


FIGURE 4.2 Goal of availability tactics

Availability tactics have one of three purposes: fault detection, fault recovery, or fault prevention. The tactics for availability are shown in Figure 4.3. These tactics will often be provided by a software infrastructure, such as a middleware package, so your job as an architect may be choosing and assessing (rather than implementing) the right availability tactics and the right combination of tactics.

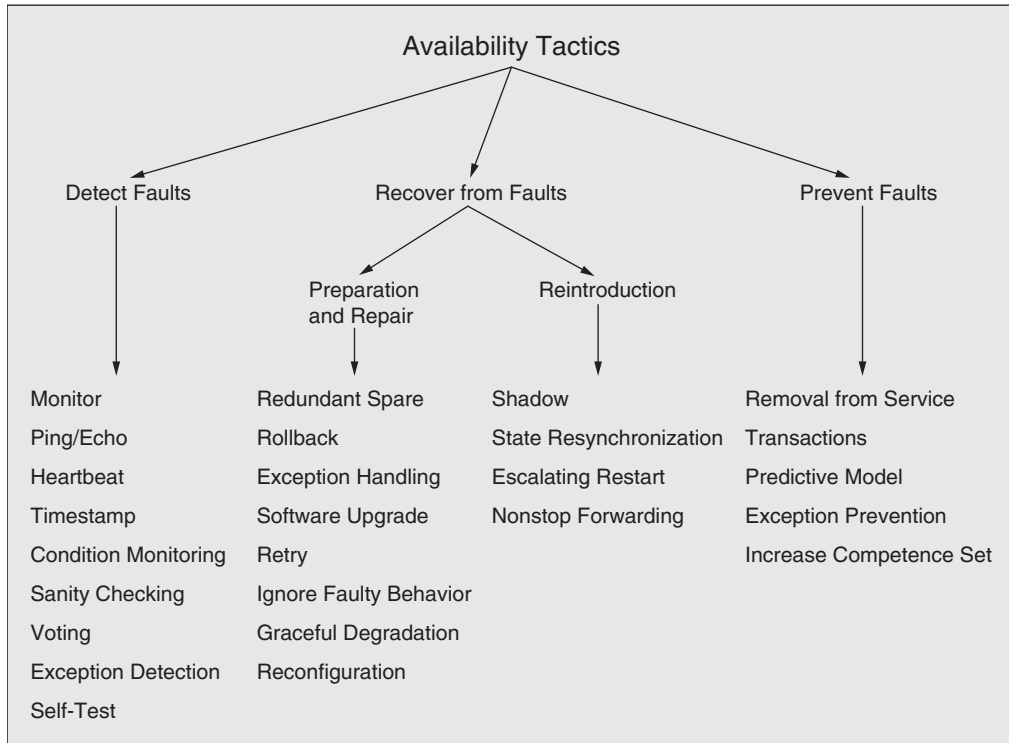


FIGURE 4.3 Availability tactics

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:

- *Monitor.* This component is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so forth. 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 *timestamps* or missed *heartbeats*.¹

- *Ping/echo*. In this tactic, an asynchronous request/response message pair is exchanged between nodes; it is used to determine reachability and the round-trip delay through the associated network path. In addition, the echo indicates that the pinged component is alive. 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 Internet Protocol (IP).
- *Heartbeat*. This fault detection mechanism 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 onto other control messages being exchanged. The difference between heartbeat and ping/echo lies in who holds the responsibility for initiating the health check—the monitor or the component itself.
- *Timestamp*. This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A timestamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Sequence numbers can also be used for this purpose, since timestamps in a distributed system may be inconsistent across different processors. See Chapter 17 for a fuller discussion of the topic of time in a distributed system.
- *Condition monitoring*. This tactic 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, provably correct) to ensure that it does not introduce new software errors.
- *Sanity checking*. This tactic checks the validity or reasonableness of specific operations or outputs of a component. It 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.
- *Voting*. Voting involves comparing computational results from multiple sources that should be producing the same results and, if they are not, deciding which results to use. 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. Voting

1. When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of the 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.”

also depends critically on having multiple sources to evaluate. Typical schemes include the following:

- *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 cannot protect against design or implementation errors, in hardware or software, since there is no form of diversity embedded in this tactic.
- *Functional redundancy*, in contrast, is intended to address the issue of common-mode failures (where replicas exhibit the same fault at the same time because they share the same implementation) in hardware or software components, by implementing design diversity. This tactic attempts to deal with the systematic nature of design faults by adding diversity to redundancy. The outputs of functionally redundant components should be the same given the same input. The functional redundancy tactic is still vulnerable to specification errors—and, of course, functional replicas will be more expensive to develop and verify.
- *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 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, with 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*. This tactic focuses on the detection of a system condition that alters the normal flow of execution. It can be further refined as follows:
 - *System exceptions* will vary according to the processor hardware architecture employed. They include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.
 - The *parameter fence* tactic incorporates a known 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 provide functions to build and parse messages. Use of parameter typing ensures that the sender and the receiver of messages agree on the type of the content, and detects cases where they don't.
 - *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:

- *Redundant spare.* This tactic refers to a configuration in which one or more duplicate components can step in and take over the work if the primary component fails. This tactic is at the heart of the hot spare, warm spare, and cold spare patterns, which differ primarily in how up-to-date the backup component is at the time of its takeover.
- *Rollback.* A rollback 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 the transactions tactic and the redundant spare tactic 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.
- *Exception handling.* Once an exception has been detected, the system will 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, the origin of the exception, and the cause of the exception. Software can then use this information to mask or repair the fault.
- *Software upgrade.* The goal of this tactic is to achieve in-service upgrades to executable code images in a non-service-affecting manner. Strategies include the following:
 - *Function patch.* This kind of patch, which is used in procedural programming, 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.

- *Class patch.* This kind of upgrade is applicable for targets executing object-oriented code, where the class definitions include a backdoor mechanism that enables the run-time addition of member data and functions.
- *Hitless in-service software upgrade (ISSU).* This leverages the redundant spare tactic 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 ISSU is used to deliver new features and capabilities.

- *Retry.* The retry tactic assumes that the fault that caused a failure is transient, and that retrying the operation may lead to success. It is used in networks and in server farms where failures are expected and common. A limit should be placed 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 emanating from the live failure of a sensor.
- *Graceful degradation.* This tactic maintains the most critical system functions in the presence of component failures, while 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.* Reconfiguration attempts to recover from failures by reassigning responsibilities to the (potentially restricted) resources or components left functioning, while maintaining as much functionality as possible.

Reintroduction occurs when a failed component is reintroduced after it has been repaired. Reintroduction tactics include the following:

- *Shadow.* This 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.* This reintroduction tactic is a partner to the redundant spare tactic. When used with active redundancy—a version of the redundant spare tactic—the state resynchronization occurs organically, since 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 version of the redundant 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.
- *Escalating restart.* This reintroduction tactic allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affectation. For example, consider a system that supports four levels of restart, numbered 0–3. The lowest level of restart (Level 0) has the least impact on services and

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 is untouched. The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. The final level of restart (Level 3) involves 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.

- *Nonstop forwarding.* This concept originated in router design, and assumes that functionality is split into two parts: the supervisory or control plane (which manages connectivity and routing information) and the 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 a “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 it might sound as if some measure of clairvoyance would be required, it turns out that in many cases it is possible to do just that.²

- *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. For example, a component of a system might be taken out of service and reset to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults reaches the service-affecting level, resulting in system failure. Other terms for this tactic are *software rejuvenation* and *therapeutic reboot*. If you reboot your computer every night, you are practicing removal from service.
- *Transactions.* Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable—properties collectively referred to as the “ACID properties.” The most common realization of the transactions tactic is the “two-phase commit” (2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item at the same time.

2. 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.

- *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 the system nears a critical threshold. The operational performance metrics monitored are used to predict the onset of faults; examples include the session establishment rate (in an HTTP server), threshold crossing (monitoring high and low watermarks for some constrained, shared resource), statistics on the process state (e.g., in-service, out-of-service, under maintenance, idle), and message queue length statistics.
- *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 earlier. Other examples of exception prevention include error-correcting code (used in telecommunications), abstract data types such as smart pointers, and the use of wrappers to prevent faults such as dangling pointers or semaphore access violations. Smart pointers prevent exceptions by doing bounds checking on pointers, and by ensuring that resources are automatically de-allocated when no data refers to them, thereby avoiding resource leaks.
- *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.

4.3 Tactics-Based Questionnaire for Availability

Based on the tactics described in Section 4.2, we can create a set of availability tactics—inspired questions, as presented in Table 4.3. To gain an overview of the architectural choices made to support availability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 4.3 Tactics-Based Questionnaire for Availability

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect Faults	<p>Does the system use ping/echo to detect failure of a component or connection, or network congestion?</p> <p>Does the system use a component to monitor the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.</p> <p>Does the system use a heartbeat—a periodic message exchange between a system monitor and a process—to detect failure of a component or connection, or network congestion?</p> <p>Does the system use a timestamp to detect incorrect sequences of events in distributed systems?</p> <p>Does the system use voting to check that replicated components are producing the same results?</p> <p>The replicated components may be identical replicas, functionally redundant, or analytically redundant.</p> <p>Does the system use exception detection to detect a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, timeout)?</p> <p>Can the system do a self-test to test itself for correct operation?</p>				

continues

TABLE 4.3 Tactics-Based Questionnaire for Availability *continued*

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Recover from Faults (Preparation and Repair)	<p>Does the system employ redundant spares?</p> <p>Is a component's role as active versus spare fixed, or does it change in the presence of a fault? What is the switchover mechanism? What is the trigger for a switchover? How long does it take for a spare to assume its duties?</p> <p>Does the system employ exception handling to deal with faults?</p> <p>Typically the handling involves either reporting, correcting, or masking the fault.</p> <p>Does the system employ rollback, so that it can revert to a previously saved good state (the "rollback line") in the event of a fault?</p> <p>Can the system perform in-service software upgrades to executable code images in a non-service-affecting manner?</p> <p>Does the system systematically retry in cases where the component or connection failure may be transient?</p> <p>Can the system simply ignore faulty behavior (e.g., ignore messages when it is determined that those messages are spurious)?</p> <p>Does the system have a policy of degradation when resources are compromised, maintaining the most critical system functions in the presence of component failures, and dropping less critical functions?</p> <p>Does the system have consistent policies and mechanisms for reconfiguration after failures, reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible?</p>				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Recover from Faults (Reintroduction)	<p>Can the system operate a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role?</p> <p>If the system uses active or passive redundancy, does it also employ state resynchronization to send state information from active components to standby components?</p> <p>Does the system employ escalating restart to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected?</p> <p>Can message processing and routing portions of the system employ nonstop forwarding, where functionality is split into supervisory and data planes?</p>				
Prevent Faults	<p>Can the system remove components from service, temporarily placing a system component in an out-of-service state for the purpose of preempting potential system failures?</p> <p>Does the system employ transactions—bundling state updates so that asynchronous messages exchanged between distributed components are <i>atomic, consistent, isolated, and durable</i>?</p> <p>Does the system use a predictive model to monitor the state of health of a component to ensure that the system is operating within nominal parameters?</p> <p>When conditions are detected that are predictive of likely future faults, the model initiates corrective action.</p>				

4.4 Patterns for Availability

This section presents a few of the most important architectural patterns for availability.

The first three patterns are all centered on the redundant spare tactic, and will be described as a group. They differ primarily in the degree to which the backup components' state matches that of the active component. (A special case occurs when the components are stateless, in which case the first two patterns become identical.)

- *Active redundancy (hot spare)*. For stateful components, this refers to a configuration in which all of the nodes (active or redundant spare) in a protection group³ receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain a 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 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.
- *Passive redundancy (warm spare)*. For stateful components, this refers to a configuration in which 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 period of the state updates), the redundant nodes are referred to as warm spares. Passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy pattern and the less available but significantly less complex cold spare pattern (which is also significantly cheaper).
- *Spare (cold spare)*. Cold sparing refers to a configuration in which redundant spares remain out of service until a failover occurs, at which point a power-on-reset⁴ procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, and hence its high mean time to repair, this pattern is poorly suited to systems having high-availability requirements.

Benefits:

- The benefit of a redundant spare is a system that continues to function correctly after only a brief delay in the presence of a failure. The alternative is a system that stops functioning correctly, or stops functioning altogether, until the failed component is repaired. This repair could take hours or days.

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

4. A power-on-reset ensures that a device starts operating in a known state.

Tradeoffs:

- The tradeoff with any of these patterns is the additional cost and complexity incurred in providing a spare.
- The tradeoff among the three alternatives is the time to recover from a failure versus the runtime cost incurred to keep a spare up-to-date. A hot spare carries the highest cost but leads to the fastest recovery time, for example.

Other patterns for availability include the following.

- *Triple modular redundancy (TMR)*. This widely used implementation of the voting tactic employs three components that do the same thing. Each component receives identical inputs and forwards its output to the voting logic, which detects any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide which output to use, and different instantiations of this pattern use different decision rules. Typical choices are letting the majority rule or choosing some computed average of the disparate outputs.

Of course, other versions of this pattern that employ 5 or 19 or 53 redundant components are also possible. However, in most cases, 3 components are sufficient to ensure a reliable result.

Benefits:

- TMR is simple to understand and to implement. It is blissfully independent of what might be causing disparate results, and is only concerned about making a reasonable choice so that the system can continue to function.

Tradeoffs:

- There is a tradeoff between increasing the level of replication, which raises the cost, and the resulting availability. In systems employing TMR, the statistical likelihood of two or more components failing is vanishingly small, and three components represents a sweet spot between availability and cost.
- *Circuit breaker*. A commonly used availability tactic is retry. In the event of a timeout or fault when invoking a service, the invoker simply tries again—and again, and again. A circuit breaker keeps the invoker from trying countless times, waiting for a response that never comes. In this way, it breaks the endless retry cycle when it deems that the system is dealing with a fault. That’s the signal for the system to begin handling the fault. Until the circuit break is “reset,” subsequent invocations will return immediately without passing along the service request.

Benefits:

- This pattern can remove from individual components the policy about how many retries to allow before declaring a failure.
- At worst, endless fruitless retries would make the invoking component as useless as the invoked component that has failed. This problem is especially acute in distributed systems, where you could have many callers calling an unresponsive component and effectively going out of service themselves, causing the failure to cascade across the

whole system. The circuit breaker, *in conjunction with software that listens to it and begins recovery procedures*, prevents that problem.

Tradeoffs:

- Care must be taken in choosing timeout (or retry) values. If the timeout is too long, then unnecessary latency is added. But if the timeout is too short, then the circuit breaker will be tripping when it does not need to—a kind of “false positive”—which can lower the availability and performance of these services.

Other availability patterns that are commonly used include the following:

- *Process pairs*. This pattern employs checkpointing and rollback. In case of failure, the backup has been checkpointing and (if necessary) rolling back to a safe state, so is ready to take over when a failure occurs.
- *Forward error recovery*. This pattern provides a way to get out of an undesirable state by *moving forward* to a desirable state. This often relies upon built-in error-correction capabilities, such as data redundancy, so that errors may be corrected without the need to fall back to a previous state or to retry. Forward error recovery finds a safe, possibly degraded state from which the operation can move forward.

4.5 For Further Reading

Patterns for availability:

- You can read about patterns for fault tolerance in [Hanmer 13].

General tactics for availability:

- 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 2004], *Ping/Echo (ICMP)* [IETF 1981] or *ICMPv6* [RFC 2006b] *Echo Request/Response*, and MPLS (LSP Ping) networks [IETF 2006a].

Tactics for availability—fault detection:

- Triple modular redundancy (TMR) was developed in the early 1960s by Lyons [Lyons 62].
- The fault detection in the voting tactic 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 of the seven-layer OSI (Open Systems

Interconnection) model [Bellcore 98, 99; Telcordia 00] and the network/link layer [IETF 2005].

- Some examples of how a system can degrade through use (degradation) are given in [Nygard 18].
- 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). [Utas 05] has also written about escalating restart.
- 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 in 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].

Disaster recovery:

- A disaster is an event such as an earthquake, flood, or hurricane that destroys an entire data center. The U.S. National Institute of Standards and Technology (NIST) identifies eight different types of plans that should be considered in the event of a disaster. See Section 2.2 of NIST Special Publication 800-34, *Contingency Planning Guide for Federal Information Systems*, <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-34r1.pdf>.

4.6 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) driverless car.
3. Write a concrete availability scenario for a program like Microsoft Word.
4. Redundancy is a key strategy for achieving high availability. Look at the patterns and 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 and deployability? How would you make a change to a system that is required to have 24/7 availability (i.e., no scheduled or unscheduled down time, ever)?

6. Consider the fault detection tactics (ping/echo, heartbeat, system monitor, voting, and exception detection). What are the performance implications of using these tactics?
7. Which tactics are used by a load balancer (see Chapter 17) when it detects a failure of an instance?
8. Look up recovery point objective (RPO) and recovery time objective (RTO), and explain how these can be used to set a checkpoint interval when using the rollback tactic.

5



Deployability

*From the day we arrive on the planet
And blinking, step into the sun
There's more to be seen than can ever be seen
More to do than can ever be done
—The Lion King*

There comes a day when software, like the rest of us, must leave home and venture out into the world and experience real life. Unlike the rest of us, software typically makes the trip many times, as changes and updates are made. This chapter is about making that transition as orderly and as effective and—most of all—as *rapid* as possible. That is the realm of continuous deployment, which is most enabled by the quality attribute of deployability.

Why has deployability come to take a front-row seat in the world of quality attributes?

In the “bad old days,” releases were infrequent—large numbers of changes were bundled into releases and scheduled. A release would contain new features and bug fixes. One release per month, per quarter, or even per year was common. Competitive pressures in many domains—with the charge being led by e-commerce—resulted in a need for much shorter release cycles. In these contexts, releases can occur at any time—possibly hundreds of releases per day—and each can be instigated by a different team within an organization. Being able to release frequently means that bug fixes in particular do not have to wait until the next scheduled release, but rather can be made and released as soon as a bug is discovered and fixed. It also means that new features do not need to be bundled into a release, but can be put into production at any time.

This is not desirable, or even possible, in all domains. If your software exists in a complex ecosystem with many dependencies, it may not be possible to release just one part of it without coordinating that release with the other parts. In addition, many embedded systems, systems in hard-to-access locations, and systems that are not networked would be poor candidates for a continuous deployment mindset.

This chapter focuses on the large and growing numbers of systems for which just-in-time feature releases are a significant competitive advantage, and just-in-time bug fixes are essential to safety or security or continuous operation. Often these systems are microservice and cloud-based, although the techniques here are not limited to those technologies.

5.1 Continuous Deployment

Deployment is a process that starts with coding and ends with real users interacting with the system in a production environment. If this process is fully automated—that is, if there is no human intervention—then it is called *continuous deployment*. If the process is automated up to the point of placing (portions of) the system into production and human intervention is required (perhaps due to regulations or policies) for this final step, the process is called *continuous delivery*.

To speed up releases, we need to introduce the concept of a *deployment pipeline*: the sequence of tools and activities that begin when you check your code into a version control system and end when your application has been deployed for users to send it requests. In between those points, a series of tools integrate and automatically test the newly committed code, test the integrated code for functionality, and test the application for concerns such as performance under load, security, and license compliance.

Each stage in the deployment pipeline takes place in an environment established to support isolation of the stage and perform the actions appropriate to that stage. The major environments are as follows:

- Code is developed in a *development environment* for a single module where it is subject to standalone unit tests. Once it passes the tests, and after appropriate review, the code is committed to a version control system that triggers the build activities in the integration environment.
- An *integration environment* builds an executable version of your service. A continuous integration server compiles¹ your new or changed code, along with the latest compatible versions of code for other portions of your service and constructs an executable image for your service.² Tests in the integration environment include the unit tests from the various modules (now run against the built system), as well as integration tests designed specifically for the whole system. When the various tests are passed, the built service is promoted to the staging environment.
- A *staging environment* tests for various qualities of the total system. These include performance testing, security testing, license conformance checks, and possibly user testing. For embedded systems, this is where simulators of the physical environment (feeding synthetic inputs to the system) are brought to bear. An application that passes all staging environment tests—which may include field testing—is deployed to the production environment, using either a blue/green model or a rolling upgrade (see Section 5.6). In some cases, partial deployments are used for quality control or to test the market response to a proposed change or offering.
- Once in the *production environment*, the service is monitored closely until all parties have some level of confidence in its quality. At that point, it is considered a normal part of the system and receives the same amount of attention as the other parts of the system.

1. If you are developing software using an interpreted language such as Python or JavaScript, there is no compilation step.

2. In this chapter, we use the term “service” to denote any independently deployable unit.

You perform a different set of tests in each environment, expanding the testing scope from unit testing of a single module in the development environment, to functional testing of all the components that make up your service in the integration environment, and ending with broad quality testing in the staging environment and usage monitoring in the production environment.

But not everything always goes according to plan. If you find problems after the software is in its production environment, it is often necessary to roll back to a previous version while the defect is being addressed.

Architectural choices affect deployability. For example, by employing the microservice architecture pattern (see Section 5.6), each team responsible for a microservice can make its own technology choices; this removes incompatibility problems that would previously have been discovered at integration time (e.g., incompatible choices of which version of a library to use). Since microservices are independent services, such choices do not cause problems.

Similarly, a continuous deployment mindset forces you to think about the testing infrastructure earlier in the development process. This is necessary because designing for continuous deployment requires continuous automated testing. In addition, the need to be able to roll back or disable features leads to architectural decisions about mechanisms such as feature toggles and backward compatibility of interfaces. These decisions are best taken early on.

The Effect of Virtualization on the Different Environments

Before the widespread use of virtualization technology, the environments that we describe here were physical facilities. In most organizations, the development, integration, and staging environments comprised hardware and software procured and operated by different groups. The development environment might consist of a few desktop computers that the development team repurposed as servers. The integration environment was operated by the test or quality-assurance team, and might consist of some racks, populated with previous-generation equipment from the data center. The staging environment was operated by the operations team and might have hardware similar to that used in production.

A lot of time was spent trying to figure out why a test that passed in one environment failed in another environment. One benefit of environments that employ virtualization is the ability to have *environment parity*, where environments may differ in scale but not in type of hardware or fundamental structure. A variety of provisioning tools support environment parity by allowing every team to easily build a common environment and by ensuring that this common environment mimics the production environment as closely as possible.

Three important ways to measure the quality of the pipeline are as follows:

- *Cycle time* is the pace of progress through the pipeline. Many organizations will deploy to production several or even hundreds of times a day. Such rapid deployment is not possible if human intervention is required. It is also not possible if one team must coordinate

with other teams before placing its service in production. Later in this chapter, we will see architectural techniques that allow teams to perform continuous deployment without consulting other teams.

- *Traceability* is the ability to recover all of the artifacts that led to an element having a problem. That includes all the code and dependencies that are included in that element. It also includes the test cases that were run on that element and the tools that were used to produce the element. Errors in tools used in the deployment pipeline can cause problems in production. Typically, traceability information is kept in an *artifact database*. This database will contain code version numbers, version numbers of elements the system depends on (such as libraries), test version numbers, and tool version numbers.
- *Repeatability* is getting the same result when you perform the same action with the same artifacts. This is not as easy as it sounds. For example, suppose your build process fetches the latest version of a library. The next time you execute the build process, a new version of the library may have been released. As another example, suppose one test modifies some values in the database. If the original values are not restored, subsequent tests may not produce the same results.

DevOps

DevOps—a portmanteau of “development” and “operations”—is a concept closely associated with continuous deployment. It is a movement (much like the Agile movement), a description of a set of practices and tools (again, much like the Agile movement), and a marketing formula touted by vendors selling those tools. The goal of DevOps is to shorten time to market (or time to release). The goal is to dramatically shorten the time between a developer making a change to an existing system—implementing a feature or fixing a bug—and the system reaching the hands of end users, as compared with traditional software development practices.

A formal definition of DevOps captures both the frequency of releases and the ability to perform bug fixes on demand:

DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality. [Bass 15]

Implementing DevOps is a process improvement effort. DevOps encompasses not only the cultural and organizational elements of any process improvement effort, but also a strong reliance on tools and architectural design. All environments are different, of course, but the tools and automation we describe are found in the typical tool chains built to support DevOps.

The continuous deployment strategy we describe here is the conceptual heart of DevOps. Automated testing is, in turn, a critically important ingredient of continuous deployment, and the tooling for that often represents the highest technological hurdle for DevOps. Some forms of DevOps include logging and post-deployment monitoring of those logs, for automatic detection of errors back at the “home office,” or even monitoring to understand the user experience. This, of course, requires a “phone home” or log delivery capability in the system, which may or may not be possible or allowable in some systems.

DevSecOps is a flavor of DevOps that incorporates approaches for security (for the infrastructure and for the applications it produces) into the entire process. DevSecOps is increasingly popular in aerospace and defense applications, but is also valid in any application area where DevOps is useful and a security breach would be particularly costly. Many IT applications fall in this category.

5.2 Deployability

Deployability refers to a property of software indicating that it may be deployed—that is, allocated to an environment for execution—within a predictable and acceptable amount of time and effort. Moreover, if the new deployment is not meeting its specifications, it may be rolled back, again within a predictable and acceptable amount of time and effort. As the world moves increasingly toward virtualization and cloud infrastructures, and as the scale of deployed software-intensive systems inevitably increases, it is one of the architect’s responsibilities to ensure that deployment is done in an efficient and predictable way, minimizing overall system risk.³

To achieve these goals, an architect needs to consider how an executable is updated on a host platform, and how it is subsequently invoked, measured, monitored, and controlled. Mobile systems in particular present a challenge for deployability in terms of how they are updated because of concerns about bandwidth. Some of the issues involved in deploying software are as follows:

- How does it arrive at its host (i.e., push, where updates deployed are unbidden, or pull, where users or administrators must explicitly request updates)?
- How is it integrated into an existing system? Can this be done while the existing system is executing?
- What is the medium, such as DVD, USB drive, or Internet delivery?
- What is the packaging (e.g., executable, app, plug-in)?
- What is the resulting integration into an existing system?
- What is the efficiency of executing the process?
- What is the controllability of the process?

With all of these concerns, the architect must be able to assess the associated risks. Architects are primarily concerned with the degree to which the architecture supports deployments that are:

- *Granular.* Deployments can be of the whole system or of elements within a system. If the architecture provides options for finer granularity of deployment, then certain risks can be reduced.

3. The quality attribute of testability (see Chapter 12) certainly plays a critical role in continuous deployment, and the architect can provide critical support for continuous deployment by ensuring that the system is testable, in all the ways just mentioned. However, our concern here is the quality attribute directly related to continuous deployment over and above testability: deployability.

- *Controllable.* The architecture should provide the capability to deploy at varying levels of granularity, monitor the operation of the deployed units, and roll back unsuccessful deployments.
- *Efficient.* The architecture should support rapid deployment (and, if needed, rollback) with a reasonable level of effort.

These characteristics will be reflected in the response measures of the general scenario for deployability.

5.3 Deployability General Scenario

Table 5.1 enumerates the elements of the general scenario that characterize deployability.

TABLE 5.1 General Scenario for Deployability

Portion of Scenario	Description	Possible Values
Source	The trigger for the deployment	End user, developer, system administrator, operations personnel, component marketplace, product owner.
Stimulus	What causes the trigger	A new element is available to be deployed. This is typically a request to replace a software element with a new version (e.g., fix a defect, apply a security patch, upgrade to the latest release of a component or framework, upgrade to the latest version of an internally produced element). New element is approved for incorporation. An existing element/set of elements needs to be rolled back.
Artifacts	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. Thus the artifact might be a single software element, multiple software elements, or the entire system.
Environment	Staging, production (or a specific subset of either)	Full deployment. Subset deployment to a specified portion of users, VMs, containers, servers, platforms.
Response	What should happen	Incorporate the new components. Deploy the new components. Monitor the new components. Roll back a previous deployment.

Portion of Scenario	Description	Possible Values
Response measure	A measure of cost, time, or process effectiveness for a deployment, or for a series of deployments over time	<p>Cost in terms of:</p> <ul style="list-style-type: none"> ▪ Number, size, and complexity of affected artifacts ▪ Average/worst-case effort ▪ Elapsed clock or calendar time ▪ Money (direct outlay or opportunity cost) ▪ New defects introduced <p>Extent to which this deployment/rollback affects other functions or quality attributes.</p> <p>Number of failed deployments.</p> <p>Repeatability of the process.</p> <p>Traceability of the process.</p> <p>Cycle time of the process.</p>

Figure 5.1 illustrates a concrete deployability scenario: “A new release of an authentication/authorization service (which our product uses) is made available in the component marketplace and the product owner decides to incorporate this version into the release. The new service is tested and deployed to the production environment within 40 hours of elapsed time and no more than 120 person-hours of effort. The deployment introduces no defects and no SLA is violated.”

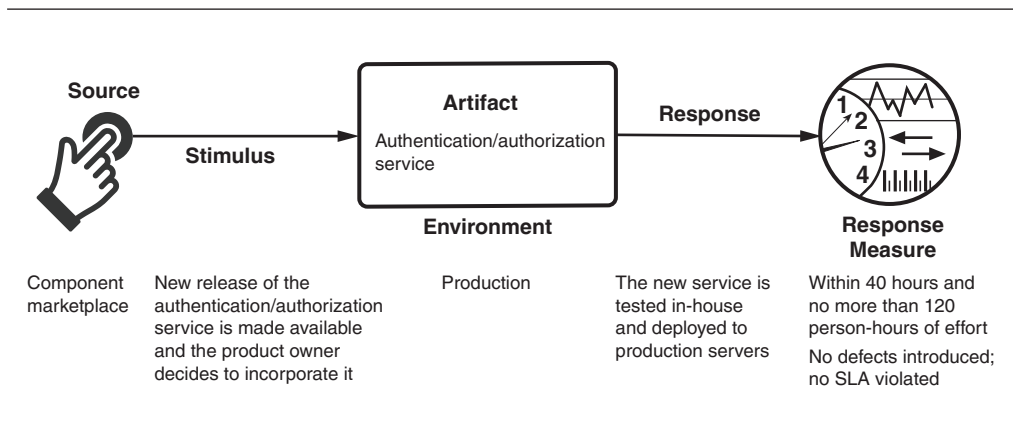


FIGURE 5.1 Sample concrete deployability scenario

5.4 Tactics for Deployability

A deployment is catalyzed by the release of a new software or hardware element. The deployment is successful if these new elements are deployed within acceptable time, cost, and quality constraints. We illustrate this relationship—and hence the goal of deployability tactics—in Figure 5.2.

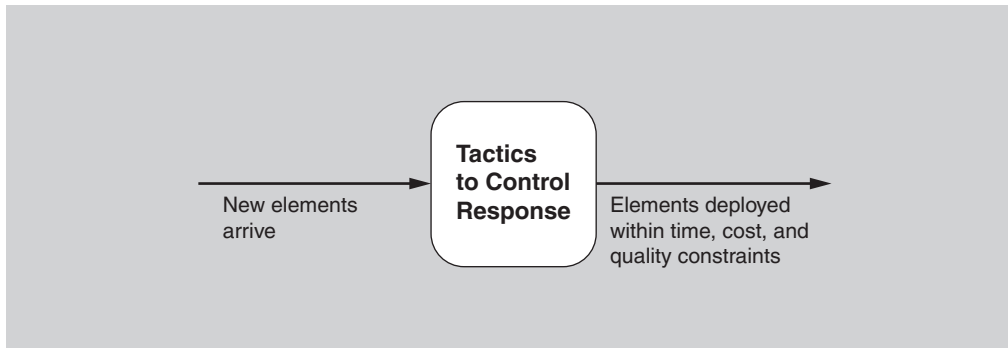


FIGURE 5.2 Goal of deployability tactics

The tactics for deployability are shown in Figure 5.3. In many cases, these tactics will be provided, at least in part, by a CI/CD (continuous integration/continuous deployment) infrastructure that you buy rather than build. In such a case, your job as an architect is often one of choosing and assessing (rather than implementing) the right deployability tactics and the right combination of tactics.

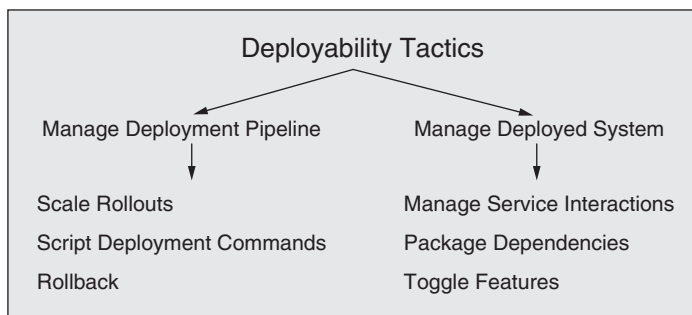


FIGURE 5.3 Deployability tactics

Next, we describe these six deployability tactics in more detail. The first category of deployability tactics focuses on strategies for managing the deployment pipeline, and the second category deals with managing the system as it is being deployed and once it has been deployed.

Manage Deployment Pipeline

- *Scale rollouts.* Rather than deploying to the entire user base, scaled rollouts deploy a new version of a service gradually, to controlled subsets of the user population, often with no explicit notification to those users. (The remainder of the user base continues to use the previous version of the service.) By gradually releasing, the effects of new deployments can be monitored and measured and, if necessary, rolled back. This tactic minimizes the potential negative impact of deploying a flawed service. It requires an architectural mechanism (not part of the service being deployed) to route a request from a user to either the new or old service, depending on that user's identity.
- *Roll back.* If it is discovered that a deployment has defects or does not meet user expectations, then it can be “rolled back” to its prior state. Since deployments may involve multiple coordinated updates of multiple services and their data, the rollback mechanism must be able to keep track of all of these, or must be able to reverse the consequences of any update made by a deployment, ideally in a fully automated fashion.
- *Script deployment commands.* Deployments are often complex and require many steps to be carried out and orchestrated precisely. For this reason, deployment is often scripted. These deployment scripts should be treated like code—documented, reviewed, tested, and version controlled. A scripting engine executes the deployment script automatically, saving time and minimizing opportunities for human error.

Manage Deployed System

- *Manage service interactions.* This tactic accommodates simultaneous deployment and execution of multiple versions of system services. Multiple requests from a client could be directed to either version in any sequence. Having multiple versions of the same service in operation, however, may introduce version incompatibilities. In such cases, the interactions between services need to be mediated so that version incompatibilities are proactively avoided. This tactic is a resource management strategy, obviating the need to completely replicate the resources so as to separately deploy the old and new versions.
- *Package dependencies.* This tactic packages an element together with its dependencies so that they get deployed together and so that the versions of the dependencies are consistent as the element moves from development into production. The dependencies may include libraries, OS versions, and utility containers (e.g., sidecar, service mesh), which we will discuss in Chapter 9. Three means of packaging dependencies are using containers, pods, or virtual machines; these are discussed in more detail in Chapter 16.

- *Feature toggle.* Even when your code is fully tested, you might encounter issues after deploying new features. For that reason, it is convenient to be able to integrate a “kill switch” (or feature toggle) for new features. The kill switch automatically disables a feature in your system at runtime, without forcing you to initiate a new deployment. This provides the ability to control deployed features without the cost and risk of actually redeploying services.

5.5 Tactics-Based Questionnaire for Deployability

Based on the tactics described in Section 5.4, we can create a set of deployability tactics—inspired questions, as presented in Table 5.2. To gain an overview of the architectural choices made to support deployability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of subsequent activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 5.2 Tactics-Based Questionnaire for Deployability

Tactics Groups	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage deployment pipeline	Do you scale rollouts , rolling out new releases gradually (in contrast to releasing in an all-or-nothing fashion)?				
	Are you able to automatically roll back deployed services if you determine that they are not operating in a satisfactory fashion?				
	Do you script deployment commands to automatically execute complex sequences of deployment instructions?				
Manage deployed system	Do you manage service interactions so that multiple versions of services can be safely deployed simultaneously?				
	Do you package dependencies so that services are deployed along with all of the libraries, OS versions, and utility containers that they depend on?				

Tactics Groups	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Do you employ feature toggles to automatically disable a newly released feature (rather than rolling back the newly deployed service) if the feature is determined to be problematic?				

5.6 Patterns for Deployability

Patterns for deployability can be organized into two categories. The first category contains patterns for structuring services to be deployed. The second category contains patterns for how to deploy services, which can be parsed into two broad subcategories: all-or-nothing or partial deployment. The two main categories for deployability are not completely independent of each other, because certain deployment patterns depend on certain structural properties of the services.

Patterns for Structuring Services

Microservice Architecture

The microservice architecture pattern structures the system as a collection of independently deployable services that communicate only via messages through service interfaces. There is no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. Services are usually stateless, and (because they are developed by a single relatively small team⁴) are relatively small—hence the term *microservice*. Service dependencies are acyclic. An integral part of this pattern is a discovery service so that messages can be appropriately routed.

Benefits:

- Time to market is reduced. Since each service is small and independently deployable, a modification to a service can be deployed without coordinating with teams that own other services. Thus, once a team completes its work on a new version of a service and that version has been tested, it can be deployed immediately.

4. At Amazon, service teams are constrained in size by the “two pizza rule”: The team must be no larger than can be fed by two pizzas.

- Each team can make its own technology choices for its service, as long as the technology choices support message passing. No coordination is needed with respect to library versions or programming languages. This reduces errors due to incompatibilities that arise during integration—and which are a major source of integration errors.
- Services are more easily scaled than coarser-grained applications. Since each service is independent, dynamically adding instances of the service is straightforward. In this way, the supply of services can be more easily matched to the demand.

Tradeoffs:

- Overhead is increased, compared to in-memory communication, because all communication among services occurs via messages across a network. This can be mitigated somewhat by using the service mesh pattern (see Chapter 9), which constrains the deployment of some services to the same host to reduce network traffic. Furthermore, because of the dynamic nature of microservice deployments, discovery services are heavily used, adding to the overhead. Ultimately, those discovery services may become a performance bottleneck.
- Microservices are less suitable for complex transactions because of the difficulty of synchronizing activities across distributed systems.
- The freedom for every team to choose its own technology comes at a cost—the organization must maintain those technologies and the required experience base.
- Intellectual control of the total system may be difficult because of the large number of microservices. This introduces a requirement for catalogs and databases of interfaces to assist in maintaining intellectual control. In addition, the process of properly combining services to achieve a desired outcome may be complex and subtle.
- Designing the services to have appropriate responsibilities and an appropriate level of granularity is a formidable design task.
- To achieve the ability to deploy versions independently, the architecture of the services must be designed to allow for that deployment strategy. Using the manage service interactions tactic described in Section 5.4 can help achieve this goal.

Organizations that have heavily employed the microservice architecture pattern include Google, Netflix, PayPal, Twitter, Facebook, and Amazon. Many other organizations have adopted the microservice architecture pattern as well; books and conferences exist that focus on how an organization can adopt the microservice architecture pattern for its own needs.

Patterns for Complete Replacement of Services

Suppose there are N instances of Service A and you wish to replace them with N instances of a new version of Service A, leaving no instances of the original version. You wish to do this with no reduction in quality of service to the clients of the service, so there must always be N instances of the service running.

Two different patterns for the complete replacement strategy are possible, both of which are realizations of the scale rollouts tactic. We'll cover them both together:

1. *Blue/green.* In a blue/green deployment, N new instances of the service would be created and each populated with new Service A (let's call these the green instances). After the N instances of new Service A are installed, the DNS server or discovery service would be changed to point to the new version of Service A. Once it is determined that the new instances are working satisfactorily, then and only then are the N instances of the original Service A removed. Before this cutoff point, if a problem is found in the new version, it is a simple matter of switching back to the original (the blue services) with little or no interruption.
2. *Rolling upgrade.* A rolling upgrade replaces the instances of Service A with instances of the new version of Service A one at a time. (In practice, you can replace more than one instance at a time, but only a small fraction are replaced in any single step.) The steps of the rolling upgrade are as follows:
 - a. Allocate resources for a new instance of Service A (e.g., a virtual machine).
 - b. Install and register the new version of Service A.
 - c. Begin to direct requests to the new version of Service A.
 - d. Choose an instance of the old Service A, allow it to complete any active processing, and then destroy that instance.
 - e. Repeat the preceding steps until all instances of the old version have been replaced.

Figure 5.4 shows a rolling upgrade process as implemented by Netflix's Asgard tool on Amazon's EC2 cloud platform.

Benefits:

- The benefit of these patterns is the ability to completely replace deployed versions of services without having to take the system out of service, thus increasing the system's availability.

Tradeoffs:

- The peak resource utilization for a blue/green approach is $2N$ instances, whereas the peak utilization for a rolling upgrade is $N + 1$ instances. In either case, resources to host these instances must be procured. Before the widespread adoption of cloud computing, procurement meant purchase: An organization had to purchase physical computers to perform the upgrade. Most of the time there was no upgrade in progress, so these additional computers largely sat idle. This made the financial tradeoff clear, and rolling upgrade was the standard approach. Now that computing resources can be rented on an as-needed basis, rather than purchased, the financial tradeoff is less compelling but still present.
- Suppose you detect an error in the new Service A when you deploy it. Despite all the testing you did in the development, integration, and staging environments, when your service is deployed to production, there may still be latent errors. If you are using blue/green deployment, by the time you discover an error in the new Service A, all of the original instances may have been deleted and rolling back to the old version could take considerable time. In contrast, a rolling upgrade may allow you to discover an error in the new version of the service while instances of the old version are still available.

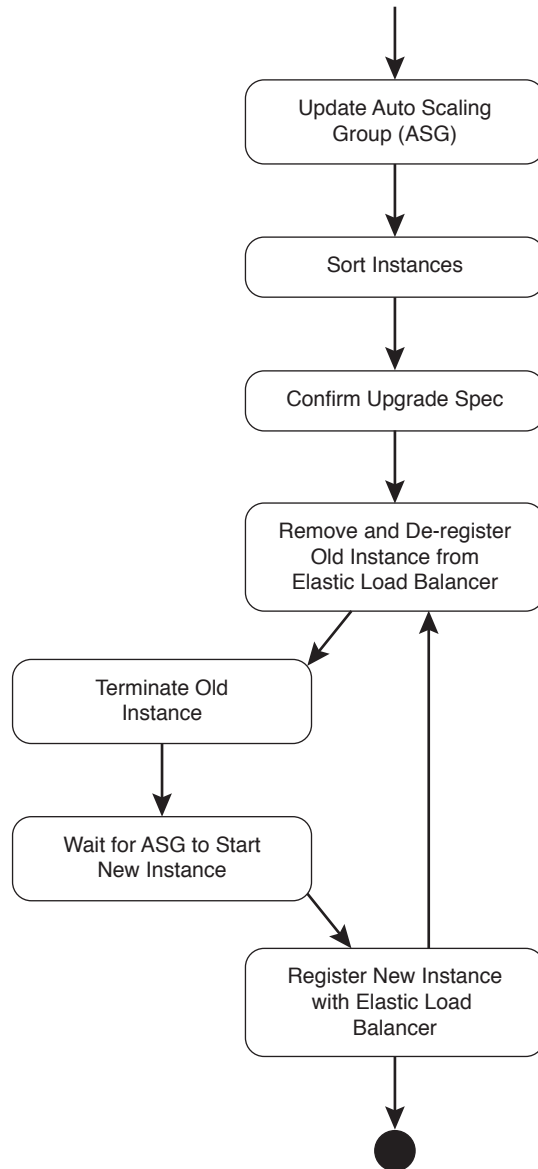


FIGURE 5.4 A flowchart of the rolling upgrade pattern as implemented by Netflix's Asgard tool

- From a client's perspective, if you are using the blue/green deployment model, then at any point in time either the new version or the old version is active, but not both. If you are using the rolling upgrade pattern, both versions are simultaneously active. This introduces the possibility of two types of problems: *temporal inconsistency* and *interface mismatch*.
 - *Temporal inconsistency*. In a sequence of requests by Client C to Service A, some may be served by the old version of the service and some may be served by the new version. If the versions behave differently, this may cause Client C to produce erroneous, or at least inconsistent, results. (This can be prevented by using the manage service interactions tactic.)
 - *Interface mismatch*. If the interface to the new version of Service A is different from the interface to the old version of Service A, then invocations by clients of Service A that have not been updated to reflect the new interface will produce unpredictable results. This can be prevented by extending the interface but not modifying the existing interface, and using the mediator pattern (see Chapter 7) to translate from the extended interface to an internal interface that produces correct behavior. See Chapter 15 for a fuller discussion.

Patterns for Partial Replacement of Services

Sometimes changing all instances of a service is undesirable. Partial-deployment patterns aim at providing multiple versions of a service simultaneously for different user groups; they are used for purposes such as quality control (canary testing) and marketing tests (A/B testing).

Canary Testing

Before rolling out a new release, it is prudent to test it in the production environment, but with a limited set of users. *Canary testing* is the continuous deployment analog of beta testing.⁵ Canary testing designates a small set of users who will test the new release. Sometimes, these testers are so-called power users or preview-stream users from outside your organization who are more likely to exercise code paths and edge cases that typical users may use less frequently. Users may or may not know that they are being used as guinea pigs—er, that is, canaries. Another approach is to use testers from within the organization that is developing the software. For example, Google employees almost never use the release that external users would be using, but instead act as testers for upcoming releases. When the focus of the testing is on determining how well new features are accepted, a variant of canary testing called dark launch is used.

In both cases, the users are designated as canaries and routed to the appropriate version of a service through DNS settings or through discovery-service configuration. After testing is

5. Canary testing is named after the 19th-century practice of bringing canaries into coal mines. Coal mining releases gases that are explosive and poisonous. Because canaries are more sensitive to these gases than humans, coal miners brought canaries into the mines and watched them for signs of reaction to the gases. The canaries acted as early warning devices for the miners, indicating an unsafe environment.

complete, users are all directed to either the new version or the old version, and instances of the deprecated version are destroyed. Rolling upgrade or blue/green deployment could be used to deploy the new version.

Benefits:

- Canary testing allows real users to “bang on” the software in ways that simulated testing cannot. This allows the organization deploying the service to collect “in use” data and perform controlled experiments with relatively low risk.
- Canary testing incurs minimal additional development costs, because the system being tested is on a path to production anyway.
- Canary testing minimizes the number of users who may be exposed to a serious defect in the new system.

Tradeoffs:

- Canary testing requires additional up-front planning and resources, and a strategy for evaluating the results of the tests needs to be formulated.
- If canary testing is aimed at power users, those users have to be identified and the new version routed to them.

A/B Testing

A/B testing is used by marketers to perform an experiment with real users to determine which of several alternatives yields the best business results. A small but meaningful number of users receive a different treatment from the remainder of the users. The difference can be minor, such as a change to the font size or form layout, or it can be more significant. For example, HomeAway (now Vrbo) has used A/B testing to vary the format, content, and look-and-feel of its worldwide websites, tracking which editions produced the most rentals. The “winner” would be kept, the “loser” discarded, and another contender designed and deployed. Another example is a bank offering different promotions to open new accounts. An oft-repeated story is that Google tested 41 different shades of blue to decide which shade to use to report search results.

As in canary testing, DNS servers and discovery-service configurations are set to send client requests to different versions. In A/B testing, the different versions are monitored to see which one provides the best response from a business perspective.

Benefits:

- A/B testing allows marketing and product development teams to run experiments on, and collect data from, real users.
- A/B testing can allow for targeting of users based on an arbitrary set of characteristics.

Tradeoffs:

- A/B testing requires the implementation of alternatives, one of which will be discarded.
- Different classes of users, and their characteristics, need to be identified up front.

5.7 For Further Reading

Much of the material in this chapter is adapted from *Deployment and Operations for Software Engineers* by Len Bass and John Klein [Bass 19] and from [Kazman 20b].

A general discussion of deployability and architecture in the context of DevOps can be found in [Bass 15].

The tactics for deployability owe much to the work of Martin Fowler and his colleagues, which can be found in [Fowler 10], [Lewis 14], and [Sato 14].

Deployment pipelines are described in much more detail in [Humble 10]

Microservices and the process of migrating to microservices are described in [Newman 15].

5.8 Discussion Questions

1. Write a set of concrete scenarios for deployability using each of the possible responses in the general scenario.
2. Write a concrete deployability scenario for the software for a car (such as a Tesla).
3. Write a concrete deployability scenario for a smartphone app. Now write one for the server-side infrastructure that communicates with this app.
4. If you needed to display the results of a search operation, would you perform A/B testing or simply use the color that Google has chosen? Why?
5. Referring to the structures described in Chapter 1, which structures would be involved in implementing the package dependencies tactic? Would you use the uses structure? Why or why not? Are there other structures you would need to consider?
6. Referring to the structures described in Chapter 1, which structures would be involved in implementing the manage service interactions tactic? Would you use the uses structure? Why or why not? Are there other structures you would need to consider?
7. Under what circumstances would you prefer to roll forward to a new version of service, rather than to roll back to a prior version? When is roll forward a poor choice?

This page intentionally left blank

6



Energy Efficiency

Energy is a bit like money: If you have a positive balance, you can distribute it in various ways, but according to the classical laws that were believed at the beginning of the century, you weren't allowed to be overdrawn.

—Stephen Hawking

Energy used by computers used to be free and unlimited—or at least that's how we behaved. Architects rarely gave much consideration to the energy consumption of software in the past. But those days are now gone. With the dominance of mobile devices as the primary form of computing for most people, with the increasing adoption of the Internet of Things (IoT) in industry and government, and with the ubiquity of cloud services as the backbone of our computing infrastructure, energy has become an issue that architects can no longer ignore. Power is no longer “free” and unlimited. The energy efficiency of mobile devices affects us all. Likewise, cloud providers are increasingly concerned with the energy efficiency of their server farms. In 2016, it was reported that data centers globally accounted for more energy consumption (by 40 percent) than the entire United Kingdom—about 3 percent of all energy consumed worldwide. More recent estimates put that share up as high as 10 percent. The energy costs associated with running and, more importantly, cooling large data centers have led people to calculate the cost of putting whole data centers in space, where cooling is free and the sun provides unlimited power. At today's launch prices, the economics are actually beginning to look favorable. Notably, server farms located underwater and in arctic climates are already a reality.

At both the low end and the high end, energy consumption of computational devices has become an issue that we should consider. This means that we, as architects, now need to add *energy efficiency* to the long list of competing qualities that we consider when designing a system. And, as with every other quality attribute, there are nontrivial tradeoffs to consider: energy usage versus performance or availability or modifiability or time to market. Thus considering energy efficiency as a first-class quality attribute is important for the following reasons:

1. An architectural approach is necessary to gain control over *any* important system quality attribute, and energy efficiency is no different. If system-wide techniques for monitoring

and managing energy are lacking, then developers are left to invent them on their own. This will, in the best case, result in an ad hoc approach to energy efficiency that produces a system that is hard to maintain, measure, and evolve. In the worst case, it will yield an approach that simply does not predictably achieve the desired energy efficiency goals.

2. Most architects and developers are unaware of energy efficiency as a quality attribute of concern, and hence do not know how to go about engineering and coding for it. More fundamentally, they lack an understanding of energy efficiency requirements—how to gather them and analyze them for completeness. Energy efficiency is not taught, or typically even mentioned, as a programmer’s concern in today’s educational curricula. In consequence, students may graduate with degrees in engineering or computer science without ever having been exposed to these issues.
3. Most architects and developers lack suitable design concepts—models, patterns, tactics, and so forth—for designing for energy efficiency, as well as managing and monitoring it at runtime. But since energy efficiency is a relatively recent concern for the software engineering community, these design concepts are still in their infancy and no catalog yet exists.

Cloud platforms typically do not have to be concerned with running out of energy (except in disaster scenarios), whereas this is a daily concern for users of mobile devices and some IoT devices. In cloud environments, scaling up and scaling down are core competencies, so decisions must be made on a regular basis about optimal resource allocation. With IoT devices, their size, form factors, and heat output all constrain their design space—there is no room for bulky batteries. In addition, the sheer number of IoT devices projected to be deployed in the next decade makes their energy usage a concern.

In all of these contexts, energy efficiency must be balanced with performance and availability, requiring engineers to consciously reason about such tradeoffs. In the cloud context, greater allocation of resources—more servers, more storage, and so on—creates improved performance capabilities as well as improved robustness against failures of individual devices, but at the cost of energy and capital outlays. In the mobile and IoT contexts, greater allocation of resources is typically not an option (although shifting the computational burden from a mobile device to a cloud back-end is possible), so the tradeoffs tend to center on energy efficiency versus performance and usability. Finally, in all contexts, there are tradeoffs between energy efficiency, on the one hand, and buildability and modifiability, on the other hand.

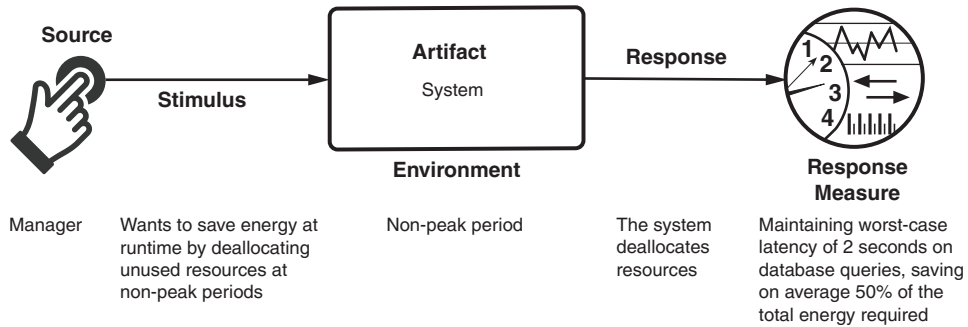
6.1 Energy Efficiency General Scenario

From these considerations, we can now determine the various portions of the energy efficiency general scenario, as presented in Table 6.1.

TABLE 6.1 Energy Efficiency General Scenario

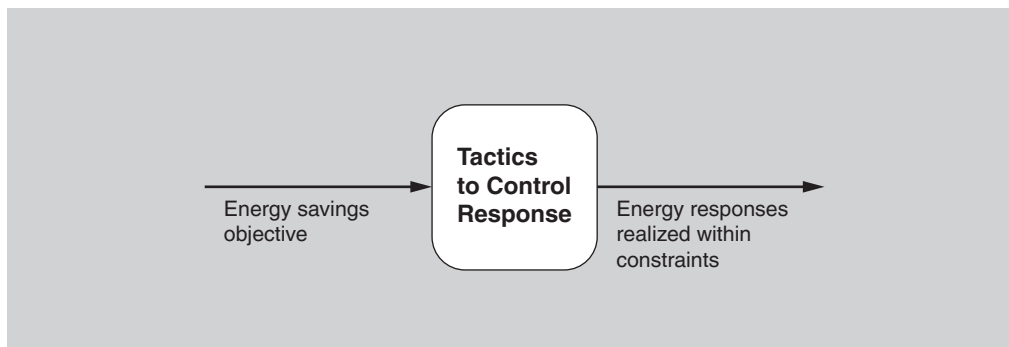
Portion of Scenario	Description	Possible Values
Source	This specifies who or what requests or initiates a request to conserve or manage energy.	End user, manager, system administrator, automated agent
Stimulus	A request to conserve energy.	Total usage, maximum instantaneous usage, average usage, etc.
Artifacts	This specifies what is to be managed.	Specific devices, servers, VMs, clusters, etc.
Environment	Energy is typically managed at runtime, but many interesting special cases exist, based on system characteristics.	Runtime, connected, battery-powered, low-battery mode, power-conservation mode
Response	What actions the system takes to conserve or manage energy usage.	One or more of the following: <ul style="list-style-type: none"> ▪ Disable services ▪ Deallocate runtime services ▪ Change allocation of services to servers ▪ Run services at a lower consumption mode ▪ Allocate/deallocate servers ▪ Change levels of service ▪ Change scheduling
Response measure	The measures revolve around the amount of energy saved or consumed and the effects on other functions or quality attributes.	Energy managed or saved in terms of: <ul style="list-style-type: none"> ▪ Maximum/average kilowatt load on the system ▪ Average/total amount of energy saved ▪ Total kilowatt hours used ▪ Time period during which the system must stay powered on ... while still maintaining a required level of functionality and acceptable levels of other quality attributes

Figure 6.1 illustrates a concrete energy efficiency scenario: *A manager wants to save energy at runtime by deallocating unused resources at non-peak periods. The system deallocates resources while maintaining worst-case latency of 2 seconds on database queries, saving on average 50 percent of the total energy required.*

**FIGURE 6.1** Sample energy efficiency scenario

6.2 Tactics for Energy Efficiency

An energy efficiency scenario is catalyzed by the desire to conserve or manage energy while still providing the required (albeit not necessarily full) functionality. This scenario is successful if the energy responses are achieved within acceptable time, cost, and quality constraints. We illustrate this simple relationship—and hence the goal of energy efficiency tactics—in Figure 6.2.

**FIGURE 6.2** Goal of energy efficiency tactics

Energy efficiency is, at its heart, about effectively utilizing resources. We group the tactics into three broad categories: resource monitoring, resource allocation, and resource adaptation (Figure 6.3). By “resource,” we mean a computational device that consumes energy while providing its functionality. This is analogous to the definition of a *hardware* resource in Chapter 9, which includes CPUs, data stores, network communications, and memory.

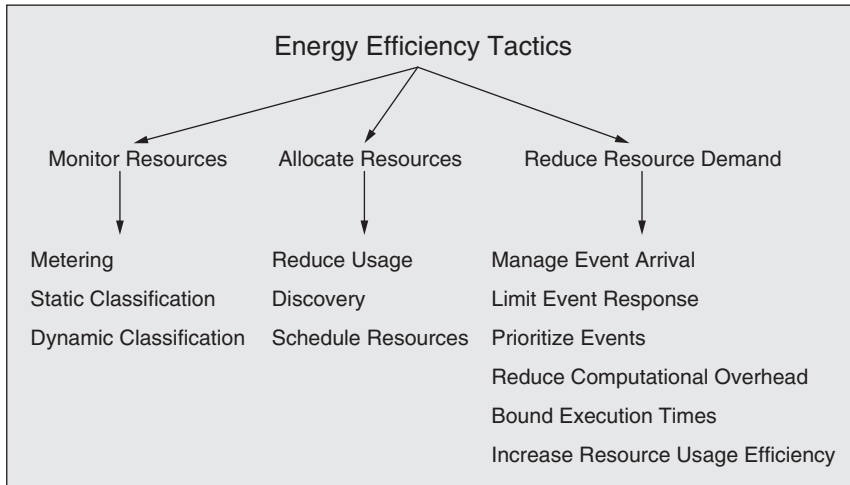


FIGURE 6.3 Energy efficiency tactics

Monitor Resources

You can’t manage what you can’t measure, and so we begin with resource monitoring. The tactics for resource monitoring are metering, static classification, and dynamic classification.

- *Metering.* The metering tactic involves collecting data about the energy consumption of computational resources via a sensor infrastructure, in near real time. At the coarsest level, the energy consumption of an entire data center can be measured from its power meter. Individual servers or hard drives can be measured using external tools such as amp meters or watt-hour meters, or using built-in tools such as those provided with metered rack PDUs (power distribution units), ASICs (application-specific integrated circuits), and so forth. In battery-operated systems, the energy remaining in a battery can be determined through a battery management system, which is a component of modern batteries.
- *Static classification.* Sometimes real-time data collection is infeasible. For example, if an organization is using an off-premises cloud, it might not have direct access to real-time energy data. Static classification allows us to estimate energy consumption by

cataloging the computing resources used and their known energy characteristics—the amount of energy used by a memory device per fetch, for example. These characteristics are available as benchmarks, or from manufacturers’ specifications.

- *Dynamic classification.* In cases where a static model of a computational resource is inadequate, a dynamic model might be required. Unlike static models, dynamic models estimate energy consumption based on knowledge of transient conditions such as workload. The model could be a simple table lookup, a regression model based on data collected during prior executions, or a simulation.

Allocate Resources

Resource allocation means assigning resources to do work in a way that is mindful of energy consumption. The tactics for resource allocation are to reduce usage, discovery, and scheduling.

- *Reduce usage.* Usage can be reduced at the device level by device-specific activities such as reducing the refresh rate of a display or darkening the background. Removing or deactivating resources when demands no longer require them is another method for decreasing energy consumption. This may involve spinning down hard drives, turning off CPUs or servers, running CPUs at a slower clock rate, or shutting down current to blocks of the processor that are not in use. It might also take the form of moving VMs onto the minimum number of physical servers (consolidation), combined with shutting down idle computational resources. In mobile applications, energy savings may be realized by sending part of the computation to the cloud, assuming that the energy consumption of communication is lower than the energy consumption of computation.
- *Discovery.* As we will see in Chapter 7, a discovery service matches service requests (from clients) with service providers, supporting the identification and remote invocation of those services. Traditionally discovery services have made these matches based on a description of the service request (typically an API). In the context of energy efficiency, this request could be annotated with energy information, allowing the requestor to choose a service provider (resource) based on its (possibly dynamic) energy characteristics. For the cloud, this energy information can be stored in a “green service directory” populated by information from metering, static classification, or dynamic classification (the resource monitoring tactics). For a smartphone, the information could be obtained from an app store. Currently such information is ad hoc at best, and typically nonexistent in service APIs.
- *Schedule resources.* Scheduling is the allocation of tasks to computational resources. As we will see in Chapter 9, the schedule resources tactic can increase performance. In the energy context, it can be used to effectively manage energy usage, given task constraints and respecting task priorities. Scheduling can be based on data collected using one or more resource monitoring tactics. Using an energy discovery service in a cloud context, or a controller in a multi-core context, a computational task can dynamically switch among computational resources, such as service providers, selecting the ones that offer better energy efficiency or lower energy costs. For example, one provider may be more lightly loaded than another, allowing it to adapt its energy usage, perhaps using some of the tactics described earlier, and consume less energy, on average, per unit of work.

Reduce Resource Demand

This category of tactics is detailed in Chapter 9. Tactics in this category—manage event arrival, limit event response, prioritize events (perhaps letting low-priority events go unserved), reduce computational overhead, bound execution times, and increase resource usage efficiency—all directly increase energy efficiency by doing less work. This is a complementary tactic to reduce usage, in that the reduce usage tactic assumes that the demand stays the same, whereas the reduce resource demand tactics are a means of explicitly managing (and reducing) the demand.

6.3 Tactics-Based Questionnaire for Energy Efficiency

As described in Chapter 3, this tactics-based questionnaire is intended to very quickly understand the degree to which an architecture employs specific tactics to manage energy efficiency.

Based on the tactics described in Section 6.2, we can create a set of tactics-inspired questions, as presented in Table 6.2. To gain an overview of the architectural choices made to support energy efficiency, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 6.2 Tactics-Based Questionnaire for Energy Efficiency

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Resource Monitoring	<p>Does your system meter the use of energy?</p> <p>That is, does the system collect data about the actual energy consumption of computational devices via a sensor infrastructure, in near real time?</p> <p>Does the system statically classify devices and computational resources?</p> <p>That is, does the system have reference values to estimate the energy consumption of a device or resource (in cases where real-time metering is infeasible or too computationally expensive)?</p>				

continues

TABLE 6.2 Tactics-Based Questionnaire for Energy Efficiency *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Resource Monitoring	Does the system dynamically classify devices and computational resources? In cases where static classification is not accurate due to varying load or environmental conditions, does the system use dynamic models, based on prior data collected, to estimate the varying energy consumption of a device or resource at runtime?				
Resource Allocation	<p>Does the system reduce usage to scale down resource usage? That is, can the system deactivate resources when demands no longer require them, in an effort to save energy? This may involve spinning down hard drives, darkening displays, turning off CPUs or servers, running CPUs at a slower clock rate, or shutting down memory blocks of the processor that are not being used.</p> <p>Does the system schedule resources to more effectively utilize energy, given task constraints and respecting task priorities, by switching computational resources, such as service providers, to the ones that offer better energy efficiency or lower energy costs? Is scheduling based on data collected (using one or more resource monitoring tactics) about the state of the system?</p> <p>Does the system make use of a discovery service to match service requests to service providers? In the context of energy efficiency, a service request could be annotated with energy requirement information, allowing the requestor to choose a service provider based on its (possibly dynamic) energy characteristics.</p>				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Reduce Resource Demand	Do you consistently attempt to reduce resource demand ? Here, you may insert the questions in this category from the Tactics-Based Questionnaire for Performance from Chapter 9.				

6.4 Patterns

Some examples of patterns used for energy efficiency include sensor fusion, kill abnormal tasks, and power monitor.

Sensor Fusion

Mobile apps and IoT systems often collect data from their environment using multiple sensors. In this pattern, data from low-power sensors can be used to infer whether data needs to be collected from higher-power sensors. A common example in the mobile phone context is using accelerometer data to assess if the user has moved and, if so, to update the GPS location. This pattern assumes that accessing the low-power sensor is much cheaper, in terms of energy consumption, than accessing the higher-power sensor.

Benefits:

- The obvious benefit of this pattern is the ability to minimize the usage of more energy-intensive devices in an intelligent way rather than, for example, just reducing the frequency of consulting the more energy-intensive sensor.

Tradeoffs:

- Consulting and comparing multiple sensors adds up-front complexity.
- The higher-energy-consuming sensor will provide higher-quality data, albeit at the cost of increased power consumption. And it will provide this data more quickly, since using the more energy-intensive sensor alone takes less time than first consulting a secondary sensor.
- In cases where the inference frequently results in accessing the higher-power sensor, this pattern could result in overall higher energy usage.

Kill Abnormal Tasks

Mobile systems, because they are often executing apps of unknown provenance, may end up unknowingly running some exceptionally power-hungry apps. This pattern provides a way

to monitor the energy usage of such apps and to interrupt or kill energy-greedy operations. For example, if an app is issuing an audible alert and vibrating the phone and the user is not responding to these alerts, then after a predetermined timeout period the task is killed.

Benefits:

- This pattern provides a “fail-safe” option for managing the energy consumption of apps with unknown energy properties.

Tradeoffs:

- Any monitoring process adds a small amount of overhead to system operations, which may affect performance and, to a small extent, energy usage.
- The usability of this pattern needs to be considered. Killing energy-hungry tasks may be counter to the user’s intention.

Power Monitor

The power monitor pattern monitors and manages system devices, minimizing the time during which they are active. This pattern attempts to automatically disable devices and interfaces that are not being actively used by the application. It has long been used within integrated circuits, where blocks of the circuit are shut down when they are not being used, in an effort to save energy.

Benefits:

- This pattern can allow for intelligent savings of power at little to no impact to the end user, assuming that the devices being shut down are truly not needed.

Tradeoffs:

- Once a device has been switched off, switching it on adds some latency before it can respond, as compared with keeping it continually running. And, in some cases, the startup may be more energy expensive than a certain period of steady-state operation.
- The power monitor needs to have knowledge of each device and its energy consumption characteristics, which adds up-front complexity to the system design.

6.5 For Further Reading

The first published set of energy tactics appeared in [Procaccianti 14]. These were, in part, the inspiration for the tactics presented here. The 2014 paper subsequently inspired [Paradis 21]. Many of the tactics presented in this chapter owe a debt to these two papers.

For a good general introduction to energy usage in software development—and what developers do not know—you should read [Pang 16].

Several research papers have investigated the consequences of design choices on energy consumption, such as [Kazman 18] and [Chowdhury 19].

A general discussion of the importance of creating “energy-aware” software can be found in [Fonseca 19].

Energy patterns for mobile devices have been catalogued by [Cruz 19] and [Schaarschmidt 20].

6.6 Discussion Questions

1. Write a set of concrete scenarios for energy efficiency using each of the possible responses in the general scenario.
2. Create a concrete energy efficiency scenario for a smartphone app (for example, a health monitoring app).
3. Create a concrete energy efficiency scenario for a cluster of data servers in a data center. What are the important distinctions between this scenario and the one you created for question 2?
4. Enumerate the energy efficiency techniques that are currently employed by your laptop or smartphone.
5. What are the energy tradeoffs in your smartphone between using Wi-Fi and the cellular network?
6. Calculate the amount of greenhouse gases in the form of carbon dioxide that you, over an average lifetime, will exhale into the atmosphere. How many Google searches does this equate to?
7. Suppose Google reduced its energy usage per search by 1 percent. How much energy would that save per year?
8. How much energy did you use to answer question 7?

This page intentionally left blank

7



Integrability

Integration is a basic law of life; when we resist it, disintegration is the natural result, both inside and outside of us. Thus we come to the concept of harmony through integration.

—Norman Cousins

According to the Merriam-Webster dictionary, the adjective *integrable* means “capable of being integrated.” We’ll give you a moment to catch your breath and absorb that profound insight. But for practical software systems, software architects need to be concerned about more than just making separately developed components cooperate; they are also concerned with the *costs* and *technical risks* of anticipated and (to varying degrees) unanticipated future integration tasks. These risks may be related to schedule, performance, or technology.

A general, abstract representation of the integration problem is that a project needs to integrate a unit of software C , or a set of units C_1, C_2, \dots, C_n , into a system S . S might be a platform, into which we integrate $\{C_i\}$, or it might be an existing system that already contains $\{C_1, C_2, \dots, C_n\}$ and our task is to design for, and analyze the costs and technical risks of, integrating $\{C_{n+1}, \dots, C_m\}$.

We assume we have control over S , but the $\{C_i\}$ may be outside our control—supplied by external vendors, for example, so our level of understanding of each C_i may vary. The clearer our understanding of C_i , the more capable the design and accurate the analysis will be.

Of course, S is not static but will evolve, and this evolution may require reanalysis. Integrability (like other quality attributes such as modifiability) is challenging because it is about planning for a future when we have incomplete information at our disposal. Simply put, some integrations will be simpler than others because they have been anticipated and accommodated in the architecture, whereas others will be more complex because they have not been.

Consider a simple analogy: To plug a North American plug (an example of a C_i) into a North American socket (an interface provided by the electrical system S), the “integration” is trivial. However, integrating a North American plug into a British socket will require an adapter. And the device with the North American plug may only run on 110-volt power, requiring further adaptation before it will work in a British 220-volt socket. Furthermore, if the component was designed to run at 60 Hz and the system provides 70 Hz, the component may not operate as intended even though it plugs in just fine. The architectural decisions made

by the creators of S and C_i —for example, to provide plug adapters or voltage adapters, or to make the component operate identically at different frequencies—will affect the cost and risk of the integration.

7.1 Evaluating the Integrability of an Architecture

Integration difficulty—the costs and the technical risks—can be thought of as a function of the size of and the “distance” between the interfaces of $\{C_i\}$ and S :

Size is the number of potential dependencies between $\{C_i\}$ and S .

Distance is the difficulty of resolving differences at each of the dependencies.

Dependencies are often measured syntactically. For example, we say that module A is dependent on component B if A calls B , if A inherits from B , or if A uses B . But while syntactic dependency is important, and will continue to be important in the future, dependency can occur in forms that are not detectable by any syntactic relation. Two components might be coupled *temporally* or through *resources* because they share and compete for a finite resource at runtime (e.g., memory, bandwidth, CPU), share control of an external device, or have a timing dependency. Or they might be coupled *semantically* because they share knowledge of the same protocol, file format, unit of measure, metadata, or some other aspect. The reason that these distinctions are important is that temporal and semantic dependencies are not often well understood, explicitly acknowledged, or properly documented. Missing or implicit knowledge is always a risk for a large, long-lived project, and such knowledge gaps will inevitably increase the costs and risks of integration and integration testing.

Consider the trend toward services and microservices in computation today. This approach is fundamentally about decoupling components to reduce the number and distance of their dependencies. Services only “know” each other via their published interfaces and, if that interface is an appropriate abstraction, changes to one service have less chance to ripple to other services in the system. The ever-increasing decoupling of components is an industry-wide trend that has been going on for decades. Service orientation, by itself, addresses (that is, reduces) only the syntactic aspects of dependency; it does not address the temporal or semantic aspects. Supposedly decoupled components that have detailed knowledge of each other and make assumptions about each other are in fact tightly coupled, and changing them in the future may well be costly.

For integrability purposes, “interfaces” must be understood as much more than simply APIs. They must characterize all of the relevant dependencies between the elements. When trying to understand dependencies between components, the concept of “distance” is helpful. As components interact, how aligned are they with respect to how they cooperate to successfully carry out an interaction? Distance may mean:

- *Syntactic distance.* The cooperating elements must agree on the number and type of the data elements being shared. For example, if one element sends an integer and the other

expects a floating point, or perhaps the bits within a data field are interpreted differently, this discrepancy presents a syntactic distance that must be bridged. Differences in data types are typically easy to observe and predict. For example, such type mismatches could be caught by a compiler. Differences in bit masks, while similar in nature, are often more difficult to detect, and the analyst may need to rely on documentation or scrutiny of the code to identify them.

- *Data semantic distance.* The cooperating elements must agree on the data semantics; that is, even if two elements share the same data type, their values are interpreted differently. For example, if one data value represents altitude in meters and the other represents altitude in feet, this presents a data semantic distance that must be bridged. This kind of mismatch is typically difficult to observe and predict, although the analyst's life is improved somewhat if the elements involved employ metadata. Mismatches in data semantics may be discovered by comparing interface documentation or metadata descriptions, if available, or by checking the code, if available.
- *Behavioral semantic distance.* The cooperating elements must agree on behavior, particularly with respect to the states and modes of the system. For example, a data element may be interpreted differently in system startup, shutdown, or recovery mode. Such states and modes may, in some cases, be explicitly captured in protocols. As another example, C_i and C_j may make different assumptions regarding control, such as each expecting the other to initiate interactions.
- *Temporal distance.* The cooperating elements must agree on assumptions about time. Examples of temporal distance include operating at different rates (e.g., one element emits values at a rate of 10 Hz and the other expects values at 60 Hz) or making different timing assumptions (e.g., one element expects event A to follow event B and the other element expects event A to follow event B with no more than 50 ms latency). While this might be considered to be a subcase of behavioral semantics, it is so important (and often subtle) that we call it out explicitly.
- *Resource distance.* The cooperating elements must agree on assumptions about shared resources. Examples of resource distance may involve devices (e.g., one element requires exclusive access to a device, whereas another expects shared access) or computational resources (e.g., one element needs 12 GB of memory to run optimally and the other needs 10 GB, but the target CPU has only 16 GB of physical memory; or three elements are simultaneously producing data at 3 Mbps each, but the communication channel offers a peak capacity of just 5 Mbps). Again, this distance may be seen as related to behavioral distance, but it should be consciously analyzed.

Such details are not typically mentioned in a programming language interface description. In the organizational context, however, these unstated, implicit interfaces often add time and complexity to integration tasks (and modification and debugging tasks). This is why interfaces are architectural concerns, as we will discuss further in Chapter 15.

In essence, integrability is about discerning and bridging the distance between the elements of each potential dependency. This is a form of planning for modifiability. We will revisit this topic in Chapter 8.

7.2 General Scenario for Integrability

Table 7.1 presents the general scenario for integrability.

TABLE 7.1 General Scenario for Integrability

Portion of Scenario	Description	Possible Values
Source	Where does the stimulus come from?	One or more of the following: <ul style="list-style-type: none"> ▪ Mission/system stakeholder ▪ Component marketplace ▪ Component vendor
Stimulus	What is the stimulus? That is, what kind of integration is being described?	One of the following: <ul style="list-style-type: none"> ▪ Add new component ▪ Integrate new version of existing component ▪ Integrate existing components together in a new way
Artifact	What parts of the system are involved in the integration?	One of the following: <ul style="list-style-type: none"> ▪ Entire system ▪ Specific set of components ▪ Component metadata ▪ Component configuration
Environment	What state is the system in when the stimulus occurs?	One of the following: <ul style="list-style-type: none"> ▪ Development ▪ Integration ▪ Deployment ▪ Runtime
Response	How will an “integrable” system respond to the stimulus?	One or more of the following: <ul style="list-style-type: none"> ▪ Changes are {completed, integrated, tested, deployed} ▪ Components in the new configuration are successfully and correctly (syntactically and semantically) exchanging information ▪ Components in the new configuration are successfully collaborating ▪ Components in the new configuration do not violate any resource limits
Response measure	How is the response measured?	One or more of the following: <ul style="list-style-type: none"> ▪ Cost, in terms of one or more of: <ul style="list-style-type: none"> ▪ Number of components changed ▪ Percentage of code changed ▪ Lines of code changed ▪ Effort ▪ Money ▪ Calendar time ▪ Effects on other quality attribute response measures (to capture allowable tradeoffs)

Figure 7.1 illustrates a sample integrability scenario constructed from the general scenario: *A new data filtering component has become available in the component marketplace. The new component is integrated into the system and deployed in 1 month, with no more than 1 person-month of effort.*

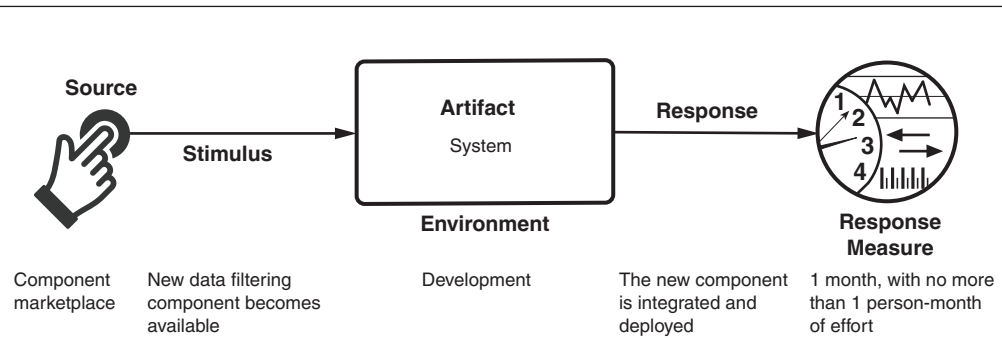


FIGURE 7.1 Sample integrability scenario

7.3 Integrability Tactics

The goals for the integrability tactics are to reduce the costs and risks of adding new components, reintegrating changed components, and integrating sets of components together to fulfill evolutionary requirements, as illustrated in Figure 7.2.

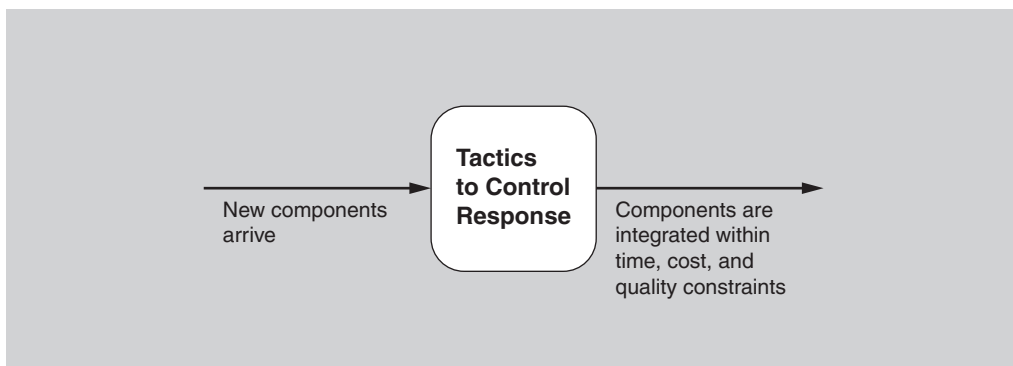


FIGURE 7.2 Goal of integrability tactics

The tactics achieve these goals either by reducing the number of potential dependencies between components or by reducing the expected distance between components. Figure 7.3 shows an overview of the integrability tactics.

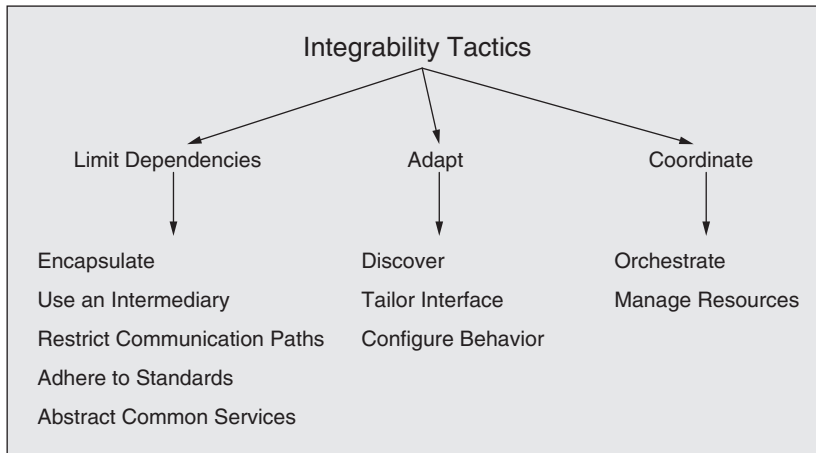


FIGURE 7.3 Integrability tactics

Limit Dependencies

Encapsulate

Encapsulation is the foundation upon which all other integrability tactics are built. It is therefore seldom seen on its own, but its use is implicit in the other tactics described here.

Encapsulation introduces an explicit interface to an element and ensures that all access to the element passes through this interface. Dependencies on the element internals are eliminated, because all dependencies must flow through the interface. Encapsulation reduces the probability that a change to one element will propagate to other elements, by reducing either the number of dependencies or their distances. These strengths are, however, reduced because the interface limits the ways in which external responsibilities can interact with the element (perhaps through a wrapper). In consequence, the external responsibilities can only directly interact with the element through the exposed interface (indirect interactions, such as dependence on quality of service, will likely remain unchanged).

Encapsulation may also hide interfaces that are not relevant for a particular integration task. An example is a library used by a service that can be completely hidden from all consumers and changed without these changes propagating to the consumers.

Encapsulation, then, can reduce the number of dependencies as well as the syntactic, data, and behavior semantic distances between *C* and *S*.

Use an Intermediary

Intermediaries are used for breaking dependencies between a set of components C_i or between C_i and the system S . Intermediaries can be used to resolve different types of dependencies. For example, intermediaries such as a publish–subscribe bus, shared data repository, or dynamic service discovery all reduce dependencies between data producers and consumers by removing any need for either to know the identity of the other party. Other intermediaries, such as data transformers and protocol translators, resolve forms of syntactic and data semantic distance.

Determining the specific benefits of a particular intermediary requires knowledge of what the intermediary actually does. An analyst needs to determine whether the intermediary reduces the number of dependencies between a component and the system and which dimensions of distance, if any, it addresses.

Intermediaries are often introduced during integration to resolve specific dependencies, but they can also be included in an architecture to promote integrability with respect to anticipated scenarios. Including a communication intermediary such as a publish–subscribe bus in an architecture, and then restricting communication paths to and from sensors to this bus, is an example of using an intermediary with the goal of promoting integrability of sensors.

Restrict Communication Paths

This tactic restricts the set of elements with which a given element can communicate. In practice, this tactic is implemented by restricting an element’s visibility (when developers cannot see an interface, they cannot employ it) and by authorization (i.e., restricting access to only authorized elements). The restrict communication paths tactic is seen in service-oriented architectures (SOAs), in which point-to-point requests are discouraged in favor of forcing all requests to go through an enterprise service bus so that routing and preprocessing can be done consistently.

Adhere to Standards

Standardization in system implementations is a primary enabler of integrability and interoperability, across both platforms and vendors. Standards vary considerably in terms of the scope of what they prescribe. Some focus on defining syntax and data semantics. Others include richer descriptions, such as those describing protocols that include behavioral and temporal semantics.

Standards similarly vary in their scope of applicability or adoption. For example, standards published by widely recognized standards-setting organizations such as the Institute of Electrical and Electronics Engineers (IEEE), the International Organization for Standardization (ISO), and the Object Management Group (OMG) are more likely to be broadly adopted. Conventions that are local to an organization, particularly if well documented and enforced, can provide similar benefits as “local standards,” though with less expectation of benefits when integrating components from outside the local standard’s sphere of adoption.

Adopting a standard can be an effective integrability tactic, although its effectiveness is limited to benefits based on the dimensions of difference addressed in the standard and how likely it is that future component suppliers will conform to the standard. Restricting communication with a system S to require use of the standard often reduces the number of potential

dependencies. Depending on what is defined in a standard, it may also address syntactic, data semantic, behavioral semantic, and temporal dimensions of distance.

Abstract Common Services

Where two elements provide services that are similar but not quite the same, it may be useful to hide both specific elements behind a common abstraction for a more general service. This abstraction might be realized as a common interface implemented by both, or it might involve an intermediary that translates requests for the abstract service to more specific requests for the elements hidden behind the abstraction. The resulting encapsulation hides the details of the elements from other components in the system. In terms of integrability, this means that future components can be integrated with a single abstraction rather than separately integrated with each of the specific elements.

When the abstract common services tactic is combined with an intermediary (such as a wrapper or adapter), it can also normalize syntactic and semantic variations among the specific elements. For example, we see this when systems use many sensors of the same type from different manufacturers, each with its own device drivers, accuracy, or timing properties, but the architecture provides a common interface to them. As another example, your browser may accommodate various kinds of ad-blocking plug-ins, yet because of the plug-in interface the browser itself can remain blissfully unaware of your choice.

Abstracting common services allows for consistency when handling common infrastructure concerns (e.g., translations, security mechanisms, and logging). When these features change, or when new versions of the components implementing these features change, the changes can be made in a smaller number of places. An abstract service is often paired with an intermediary that may perform processing to hide syntactic and data semantic differences among specific elements.

Adapt

Discover

A discovery service is a catalog of relevant addresses, which comes in handy whenever there is a need to translate from one form of address to another, whenever the target address may have been dynamically bound, or when there are multiple targets. It is the mechanism by which applications and services locate each other. A discovery service may be used to enumerate variants of particular elements that are used in different products.

Entries in a discovery service are there because they were registered. This registration can happen statically, or it can happen dynamically when a service is instantiated. Entries in the discovery service should be de-registered when they are no longer relevant. Again, this can be done statically, such as with a DNS server, or dynamically. Dynamic de-registration can be handled by the discovery service itself performing health checks on its entries, or it can be carried out by an external piece of software that knows when a particular entry in the catalog is no longer relevant.

A discovery service may include entries that are themselves discovery services. Likewise, entries in a discovery service may have additional attributes, which a query may reference. For

example, a weather discovery service may have an attribute of “cost of forecast”; you can then ask a weather discovery service for a service that provides free forecasts.

The discover tactic works by reducing the dependencies between cooperating services, which should be written without knowledge of each other. This enables flexibility in the binding between services, as well as when that binding occurs.

Tailor Interface

Tailoring an interface is a tactic that adds capabilities to, or hides capabilities in, an existing interface without changing the API or implementation. Capabilities such as translation, buffering, and data smoothing can be added to an interface without changing it. An example of removing capabilities is hiding particular functions or parameters from untrusted users. A common dynamic application of this tactic is intercepting filters that add functionality such as data validation to help prevent SQL injections or other attacks, or to translate between data formats. Another example is using techniques from aspect-oriented programming that weave in preprocessing and postprocessing functionality at compile time.

The tailor interface tactic allows functionality that is needed by many services to be added or hidden based on context and managed independently. It also enables services with syntactic differences to interoperate without modification to either service.

This tactic is typically applied during integration; however, designing an architecture so that it facilitates interface tailoring can support integrability. Interface tailoring is commonly used to resolve syntactic and data semantic distance during integration. It can also be applied to resolve some forms of behavioral semantic distance, though it can be more complex to do (e.g., maintaining a complex state to accommodate protocol differences) and is perhaps more accurately categorized as introducing an intermediary.

Configure Behavior

The tactic of configuring behavior is used by software components that are implemented to be configurable in prescribed ways that allow them to more easily interact with a range of components. The behavior of a component can be configured during the build phase (recompile with a different flag), during system initialization (read a configuration file or fetch data from a database), or during runtime (specify a protocol version as part of your requests). A simple example is configuring a component to support different versions of a standard on its interfaces. Ensuring that multiple options are available increases the chances that the assumptions of S and a future C will match.

Building configurable behavior into portions of S is an integrability tactic that allows S to support a wider range of potential C s. This tactic can potentially address syntactic, data semantic, behavioral semantic, and temporal dimensions of distance.

Coordinate

Orchestrate

Orchestrate is a tactic that uses a control mechanism to coordinate and manage the invocation of particular services so that they can remain unaware of each other.

Orchestration helps with the integration of a set of loosely coupled reusable services to create a system that meets a new need. Integration costs are reduced when orchestration is included in an architecture in a way that supports the services that are likely to be integrated in the future. This tactic allows future integration activities to focus on integration with the orchestration mechanism instead of point-to-point integration with multiple components.

Workflow engines commonly make use of the orchestrate tactic. A workflow is a set of organized activities that order and coordinate software components to complete a business process. It may consist of other workflows, each of which may itself consist of aggregated services. The workflow model encourages reuse and agility, leading to more flexible business processes. Business processes can be managed under a philosophy of business process management (BPM) that views processes as a set of competitive assets to be managed. Complex orchestration can be specified in a language such as BPEL (Business Process Execution Language).

Orchestration works by reducing the number of dependencies between a system S and new components $\{C_i\}$, and eliminating altogether the explicit dependencies among the components $\{C_i\}$, by centralizing those dependencies at the orchestration mechanism. It may also reduce syntactic and data semantic distance if the orchestration mechanism is used in conjunction with tactics such as adherence to standards.

Manage Resources

A resource manager is a specific form of intermediary that governs access to computing resources; it is similar to the restrict communication paths tactic. With this tactic, software components are not allowed to directly access some computing resources (e.g., threads or blocks of memory), but instead request those resources from a resource manager. Resource managers are typically responsible for allocating resource access across multiple components in a way that preserves some invariants (e.g., avoiding resource exhaustion or concurrent use), enforces some fair access policy, or both. Examples of resource managers include operating systems, transaction mechanisms in databases, use of thread pools in enterprise systems, and use of the ARINC 653 standard for space and time partitioning in safety-critical systems.

The manage resource tactic works by reducing the resource distance between a system S and a component C , by clearly exposing the resource requirements and managing their common use.

7.4 Tactics-Based Questionnaire for Integrability

Based on the tactics described in Section 7.3, we can create a set of integrability tactics-inspired questions, as presented in Table 7.2. To gain an overview of the architectural choices made to support integrability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 7.2 Tactics-Based Questionnaire for Integrability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Limit Dependencies	<p>Does the system encapsulate functionality of each element by introducing explicit interfaces and requiring that all access to the elements passes through these interfaces?</p> <p>Does the system broadly use intermediaries for breaking dependencies between components—for example, removing a data producer's knowledge of its consumers?</p> <p>Does the system abstract common services, providing a general, abstract interface for similar services?</p> <p>Does the system provide a means to restrict communication paths between components?</p> <p>Does the system adhere to standards in terms of how components interact and share information with each other?</p>				
Adapt	<p>Does the system provide the ability to statically (i.e., at compile time) tailor interfaces—that is, the ability to add or hide capabilities of a component's interface without changing its API or implementation?</p> <p>Does the system provide a discovery service, cataloguing and disseminating information about services?</p> <p>Does the system provide a means to configure the behavior of components at build, initialization, or runtime?</p>				

continues

TABLE 7.2 Tactics-Based Questionnaire for Integrability *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Coordinate	<p>Does the system include an orchestration mechanism that coordinates and manages the invocation of components so they can remain unaware of each other?</p> <p>Does the system provide a resource manager that governs access to computing resources?</p>				

7.5 Patterns

The first three patterns are all centered on the tailor interface tactic, and are described here as a group:

- *Wrappers.* A wrapper is a form of encapsulation whereby some component is encased within an alternative abstraction. A wrapper 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 Imperial measures but find itself in a system in which all of the other components produce metric measures. Wrappers can:
 - Translate an element of a component interface into an alternative element
 - Hide an element of a component interface
 - Preserve an element of a component's base interface without change
- *Bridges.* A bridge translates some “requires” assumptions of one arbitrary component to some “provides” assumptions of another component. The key difference between a bridge and a wrapper is that a bridge is independent of any particular component. Also, the bridge must be explicitly invoked by some external agent—possibly but not necessarily by one of the components the bridge spans. This last point should convey the idea that bridges are usually transient and that the specific translation is defined at the time of bridge construction (e.g., bridge compile time). The significance of both of these distinctions will be made clear in the discussion of mediators.

Bridges typically focus on a narrower range of interface translations than do wrappers because bridges address specific assumptions. The more assumptions a bridge tries to address, the fewer components to which it applies.

- *Mediators.* Mediators exhibit properties of both bridges and wrappers. The major distinction between bridges and mediators, is that mediators incorporate a planning function that results in runtime determination of the translation, whereas bridges establish this translation at bridge construction time.

A mediator is also similar to a wrapper insofar as it becomes an explicit component in the system architecture. That is, semantically primitive, often transient bridges can be thought of as incidental repair mechanisms whose role in a design can remain implicit. In contrast, mediators have sufficient semantic complexity and runtime autonomy (persistence) to play a first-class role in a software architecture.

Benefits:

- All three patterns allow access to an element without forcing a change to the element or its interface.

Tradeoffs:

- Creating any of the patterns requires up-front development work.
- All of the patterns will introduce some performance overhead while accessing the element, although typically this overhead is small.

Service-Oriented Architecture Pattern

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 entities: 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), which may sometimes be legally binding. Components perform their computations by requesting services from one another. Communication among the services is typically performed by using web services standards such as WSDL (Web Services Description Language) or SOAP (Simple Object Access Protocol).

The SOA pattern is related to the microservice architecture pattern (see Chapter 5). Microservice architectures are assumed to compose a single system and be managed by a single organization, however, whereas SOAs provide reusable components that are assumed to be heterogeneous and managed by distinct organizations.

Benefits:

- Services are designed to be used by a variety of clients, leading them to be more generic. Many commercial organizations will provide and market their service with the goal of broad adoption.
- Services are independent. The only method for accessing a service is through its interface and through messages over a network. Consequently, a service and the rest of the system do not interact, except through their interfaces.

- Services can be implemented heterogeneously, using whatever languages and technologies are most appropriate.

Tradeoffs:

- SOAs, because of their heterogeneity and distinct ownership, come with a great many interoperability features such as WSDL and SOAP. This adds complexity and overhead.

Dynamic Discovery

Dynamic discovery applies the discovery tactic to enable the discovery of service providers at runtime. Consequently, a runtime binding can occur between a service consumer and a concrete service.

Use of a dynamic discovery capability sets the expectation that the system will clearly advertise both the services available for integration with future components and the minimal information that will be available for each service. The specific information available will vary, but typically comprises data that can be mechanically searched during discovery and runtime integration (e.g., identifying a specific version of an interface standard by string match).

Benefits:

- This pattern allows for flexibility in binding services together into a cooperating whole. For example, services may be chosen at startup or runtime based on their pricing or availability.

Tradeoffs:

- Dynamic discovery registration and de-registration must be automated, and tools for this purpose must be acquired or generated.

7.6 For Further Reading

Much of the material for this chapter was inspired by and drawn from [Kazman 20a].

An in-depth discussion of the quality attribute of integrability can be found in [Henttonen 07].

[MacCormack 06] and [Mo 16] define and provide empirical evidence for architecture-level coupling metrics, which can be useful in measuring designs for integrability.

The book *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma 94] defines and distinguishes the bridge, wrapper, and adapter patterns.

7.7 Discussion Questions

1. Think about an integration that you have done in the past—perhaps integrating a library or a framework into your code. Identify the various “distances” that you had to deal with, as discussed in Section 7.1. Which of these required the greatest effort to resolve?
2. Write a concrete integrability scenario for a system that you are working on (perhaps an exploratory scenario for some component that you are considering integrating).
3. Which of the integrability tactics do you think would be the easiest to implement in practice, and why? Which would be the most difficult, and why?
4. Many of the integrability tactics are similar to the modifiability tactics. If you make your system highly modifiable, does that automatically mean that it will be easy to integrate into another context?
5. A standard use of SOA is to add a shopping cart feature to an e-commerce site. Which commercially available SOA platforms provide different shopping cart services? What are the attributes of the shopping carts? Can these attributes be discovered at runtime?
6. Write a program that accesses the Google Play Store, via its API, and returns a list of weather forecasting applications and their attributes.
7. Sketch a design for a dynamic discovery service. Which types of distances does this service help to mitigate?

This page intentionally left blank

8



Modifiability

It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.

—Charles Darwin

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 alter 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 is to lower 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 will be supported and which will 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 aspect of the system. Equally clear, it is not in the

same category as changing the system so that it uses a different database management system. Changes can be made to the implementation (by modifying the source code), during compilation (using compile-time switches), during the build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting), or during execution (by parameter settings, plug-ins, allocation to hardware, and so forth). A change can also be made by a developer, an end user, or a system administrator. Systems that learn and adapt supply a whole different answer to the question of when a change is made and “who” makes it—it is the system itself that is the agent for change.

- *What is the cost of the change?* Making a system more modifiable involves two types of costs:
 - 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. In such a case, the cost of introducing the mechanism is zero (since there is no special mechanism); the cost of exercising it is the cost of changing the source code and revalidating the system.

Toward 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 designed UI produced through direct manipulation techniques and which may then produce source code. The cost of introducing the mechanism is the cost of acquiring the UI builder, which may be substantial. The cost of using the mechanism is the cost of producing the input to feed the builder (this cost can be either substantial or negligible), the cost of running the builder (close to zero), and finally the cost of whatever testing is performed on the result (usually much less than for hand-coding).

Still further along the spectrum are software systems that discover their environments, learn, and modify themselves to accommodate any changes. For those systems, the cost of making the modification is zero, but that ability was purchased along with implementing and testing the learning mechanisms, which may have been quite costly.

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

$$N * \text{Cost of making change without the mechanism} \leq \\ \text{Cost of creating the mechanism} + (N * \text{cost of making the change using the mechanism})$$

Here, N is the anticipated number of modifications that will use the modifiability mechanism—but it is also 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 (opportunity cost)—in adding new functionality, in improving the performance, or even in non-software investments such as hiring or training. 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 its completion. However, if your code is modified frequently, not introducing some architectural mechanism and simply piling change on top of change typically leads to substantial technical debt. We address the topic of architectural debt in Chapter 23.

Change is so prevalent in the life of software systems that special names have been given to specific flavors of modifiability. Some of the common ones are highlighted here:

- *Scalability* is about accommodating more of something. In terms of performance, scalability means adding more resources. Two kinds of performance 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 unduly disrupt operations. In cloud-based 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 17 for further discussion of such environments).
- *Variability* refers to the ability of a system and its supporting artifacts, such as code, requirements, test plans, and documentation, 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 product line, which is a family of systems that are similar but vary in features and functions. If the engineering assets associated with these systems can be shared among members of the family, then the overall cost of the product line plummets. This is achieved by introducing mechanisms that allow the artifacts to be selected and/or adapt to usages in the different product contexts that are within the product line's scope. The goal of variability in a software product line is to make it easy to build and maintain products in that family over a period of time.
- *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” (for example, a Java Virtual Machine) that encapsulates all the platform dependencies. 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. Architectural approaches to dealing with portability are intertwined with those for *deployability*, a topic addressed in Chapter 5.
- *Location independence* refers to the case where two pieces of distributed software interact and the location of one or both of the pieces is not known prior to runtime. Alternatively, the location of these pieces may change during runtime. In distributed systems, services are often deployed to arbitrary locations, and clients of those services must discover their location dynamically. In addition, services in a distributed system must often make their location discoverable once they have been deployed to a location. Designing the system for location independence means that the location will be easy to modify with minimal impact on the rest of the system.

8.1 Modifiability General Scenario

From these considerations, we can construct the general scenario for modifiability. Table 8.1 summarizes this scenario.

TABLE 8.1 General Scenario for Modifiability

Portion of Scenario	Description	Possible Values
Source	The agent that causes a change to be made. Most are human actors, but the system might be one that learns or self-modifies, in which case the source is the system itself.	End user, developer, system administrator, product line owner, the system itself
Stimulus	The change that the system needs to accommodate. (For this categorization, we regard fixing a defect as a change, to something that presumably wasn't working correctly.)	A directive to add/delete/modify functionality, or change a quality attribute, capacity, platform, or technology; a directive to add a new product to a product line; a directive to change the location of a service to another location
Artifacts	The artifacts that are modified. Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates.	Code, data, interfaces, components, resources, test cases, configurations, documentation
Environment	The time or stage at which the change is made.	Runtime, compile time, build time, initiation time, design time
Response	Make the change and incorporate it into the system.	One or more of the following: <ul style="list-style-type: none"> ▪ Make modification ▪ Test modification ▪ Deploy modification ▪ Self-modify
Response measure	The resources that were expended to make the change.	Cost in terms of: <ul style="list-style-type: none"> ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Elapsed time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced ▪ How long it took the system to adapt

Figure 8.1 illustrates a concrete modifiability scenario: *A developer wishes to change the user interface. This change will be made to the code at design time, it will take less than three hours to make and test the change, and no side effects will occur.*

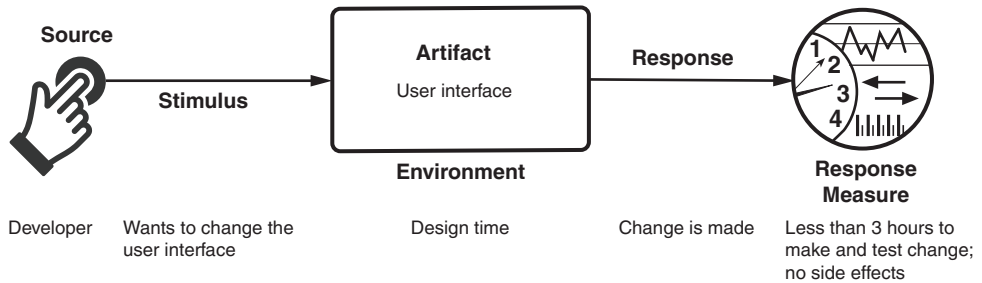


FIGURE 8.1 Sample concrete modifiability scenario

8.2 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 8.2 shows this relationship.

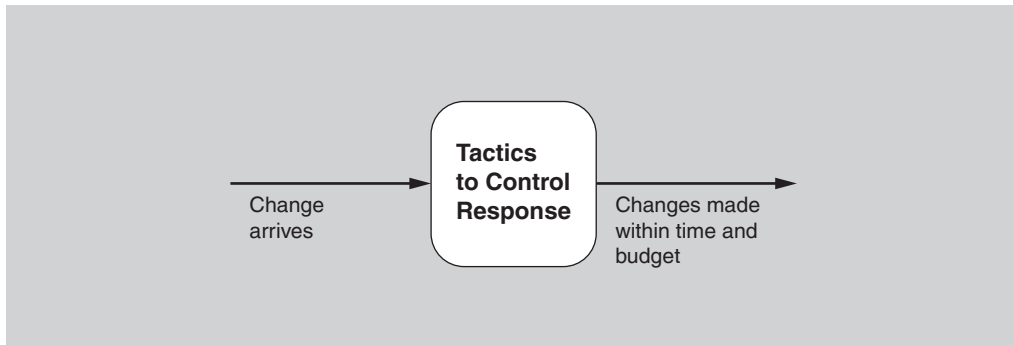


FIGURE 8.2 Goal of modifiability tactics

To understand modifiability, we begin with some of the earliest and most fundamental complexity measures of software design—coupling and cohesion—which were first described in the 1960s.

Generally, a change that affects one module is easier and less expensive than a change that affects more than one module. However, if two modules' responsibilities overlap in some way,

then a single change may well affect them both. We can quantify this overlap by measuring the probability that a modification to one module will propagate to the other. This relationship is called *coupling*, and high coupling is an enemy of modifiability. Reducing the coupling between two modules will decrease the expected cost of any modification that affects either one. Tactics that reduce coupling are those that place intermediaries of various sorts between the two otherwise highly coupled modules.

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 modules. High cohesion is good for modifiability; low cohesion is bad for it. If module A has a low cohesion, then cohesion can be improved by removing responsibilities unaffected by anticipated changes.

A third characteristic that affects the cost and complexity of a change is the *size of a module*. All other things being equal, larger modules are more difficult and more costly to change, and are more prone to have bugs.

Finally, we need to be concerned with the point in the software development life cycle where 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 be successfully made (i.e., quickly and at low cost) late in the life cycle only if the architecture is suitably prepared to accommodate them. Thus the fourth and final parameter in a model of modifiability is *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 modifications that occur late in the life cycle.

Now we can understand tactics and their consequences as affecting one or more of these parameters: reducing size, increasing cohesion, reducing coupling, and deferring binding time. These tactics are shown in Figure 8.3.

Increase Cohesion

Several tactics involve redistributing responsibilities among modules. This step is taken to reduce the likelihood that a single change will affect multiple modules.

- *Split module*. If the module being modified includes responsibilities that are not cohesive, the modification costs will likely be high. Refactoring the module into several more cohesive modules should reduce the average cost of future changes. Splitting a module should not simply consist of placing half of the lines of code into each submodule; instead, it should sensibly and appropriately result in a series of submodules that are cohesive on their own.
- *Redistribute responsibilities*. If responsibilities A, A', and A'' (all similar responsibilities) are sprinkled across several distinct modules, they should be placed together. This

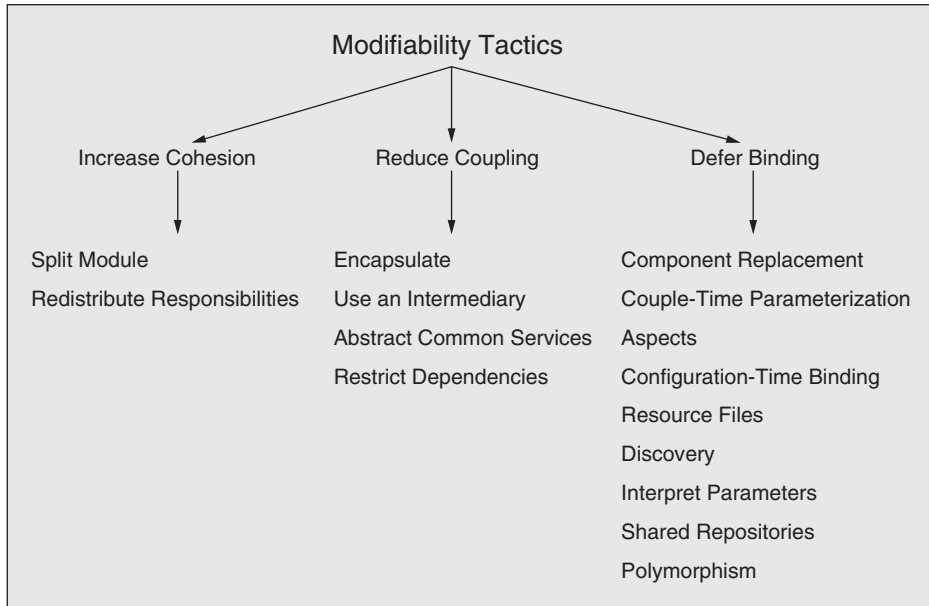


FIGURE 8.3 Modifiability tactics

refactoring may involve creating a new module, or it may involve moving responsibilities to existing modules. One method for identifying responsibilities to be moved is to hypothesize a set of likely changes as scenarios. If the scenarios consistently affect just one part of a module, then perhaps the other parts have separate responsibilities and should be moved. Alternatively, if some scenarios require modifications to multiple modules, then perhaps the responsibilities affected should be grouped together into a new module.

Reduce Coupling

We now turn to tactics that reduce the coupling between modules. These tactics overlap with the integrability tactics described in Chapter 7, because reducing dependencies among independent components (for integrability) is similar to reducing coupling among modules (for modifiability).

- *Encapsulate.* See the discussion in Chapter 7.
- *Use an intermediary.* See the discussion in Chapter 7.
- *Abstract common services.* See the discussion in Chapter 7.

- *Restrict dependencies.* This tactic restricts which modules a given module interacts with or depends on. In practice, this tactic is implemented 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). The restrict dependencies tactic is seen in layered architectures, in which a layer is allowed to use only lower layers (sometimes only the next lower layer), and with the use of wrappers, where external entities can see (and hence depend on) only the wrapper, and not the internal functionality that it wraps.

Defer Binding

Because the work of people is almost always more expensive error-prone 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 their use 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 deferring binding.

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—a well-known tradeoff. And so the equation given earlier in the chapter comes into play. We want to bind as late as possible, as long as the mechanism that allows it is cost-effective.

The following tactics can be used to bind values at compile time or build time:

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

The following tactics are available to bind values at deployment, startup time, or initialization time:

- Configuration-time binding
- Resource files

Tactics to bind values at runtime include the following:

- Discovery (see Chapter 7)
- Interpret parameters
- Shared repositories
- Polymorphism

Separating the building of a mechanism for modifiability from the use of that 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 administrator or installer) 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.

8.3 Tactics-Based Questionnaire for Modifiability

Based on the tactics described in Section 8.2, we can create a set of tactics-inspired questions, as presented in Table 8.2. To gain an overview of the architectural choices made to support modifiability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 8.2 Tactics-Based Questionnaire for Modifiability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk?	Design Decisions and Location	Rationale and Assumptions
Increase Cohesion	Do you make modules more cohesive by splitting the module ? For example, if you have a large, complex module, can you split it into two (or more) more cohesive modules? Do you make modules more cohesive by redistributing responsibilities ? For example, if responsibilities in a module do not serve the same purpose, they should be placed in other modules.				
Reduce Coupling	Do you consistently encapsulate functionality? This typically involves isolating the functionality under scrutiny and introducing an explicit interface to it. Do you consistently use an intermediary to keep modules from being too tightly coupled? For example, if A calls concrete functionality C, you might introduce an abstraction B that mediates between A and C. Do you restrict dependencies between modules in a systematic way? Or is any system module free to interact with any other module?				

continues

TABLE 8.2 Tactics-Based Questionnaire for Modifiability *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk?	Design Decisions and Location	Rationale and Assumptions
Reduce Coupling	Do you abstract common services , in cases where you are providing several similar services? For example, this technique is often used when you want your system to be portable across operating systems, hardware, or other environmental variations.				
Defer Binding	Does the system regularly defer binding of important functionality so that it can be replaced later in the life cycle? For example, are there plug-ins, add-ons, resource files, or configuration files that can extend the functionality of the system?				

8.4 Patterns

Patterns for modifiability divide the system into modules in such a way that the modules can be developed and evolved separately with little interaction among them, thereby supporting portability, modifiability, and reuse. There are probably more patterns designed to support modifiability than for any other quality attribute. We present a few that are among the most commonly used here.

Client-Server Pattern

The client-server pattern consists of a server providing services simultaneously to multiple distributed clients. The most common example is a web server providing information to multiple simultaneous users of a website.

The interactions between a server and its clients follow this sequence:

- *Discovery*:
 - Communication is initiated by a client, which uses a discovery service to determine the location of the server.
 - The server responds to the client using an agreed-upon protocol.

- *Interaction:*

- The client sends requests to the server.
- The server processes the requests and responds.

Several points about this sequence are worth noting:

- The server may have multiple instances if the number of clients grows beyond the capacity of a single instance.
- If the server is stateless with respect to the clients, each request from a client is treated independently.
- If the server maintains state with respect to the clients, then:
 - Each request must identify the client in some fashion.
 - The client should send an “end of session” message so that the server can remove resources associated with that particular client.
 - The server may time out if the client has not sent a request in a specified time so that resources associated with the client can be removed.

Benefits:

- The connection between a server and its clients is established dynamically. The server has no a priori knowledge of its clients—that is, there is low coupling between the server and its clients.
- There is no coupling among the clients.
- The number of clients can easily scale and is constrained only by the capacity of the server. The server functionality can also scale if its capacity is exceeded.
- Clients and servers can evolve independently.
- Common services can be shared among multiple clients.
- The interaction with a user is isolated to the client. This factor has resulted in the development of specialized languages and tools for managing the user interface.

Tradeoffs:

- This pattern is implemented such that communication occurs over a network, perhaps even the Internet. Thus messages may be delayed by network congestion, leading to degradation (or at least unpredictability) of performance.
- For clients that communicate with servers over a network shared by other applications, special provisions must be made for achieving security (especially confidentiality) and maintaining integrity.

Plug-in (Microkernel) Pattern

The plug-in pattern has two types of elements—elements that provide a core set of functionality and specialized variants (called plug-ins) that add functionality to the core via a fixed set of interfaces. The two types are typically bound together at build time or later.

Examples of usage include the following cases:

- The core functionality may be a stripped-down operating system (the microkernel) that provides the mechanisms needed to implement operating system services, such as low-level address space management, thread management, and interprocess communication (IPC). The plug-ins provide the actual operating system functionality, such as device drivers, task management, and I/O request management.
- The core functionality is a product providing services to its users. The plug-ins provide portability, such as operating system compatibility or supporting library compatibility. The plug-ins can also provide additional functionality not included in the core product. In addition, they can act as adapters to enable integration with external systems (see Chapter 7).

Benefits:

- Plug-ins provide a controlled mechanism to extend a core product and make it useful in a variety of contexts.
- The plug-ins can be developed by different teams or organizations than the developers of the microkernel. This allows for the development of two different markets: for the core product and for the plug-ins.
- The plug-ins can evolve independently from the microkernel. Since they interact through fixed interfaces, as long as the interfaces do not change, the two types of elements are not otherwise coupled.

Tradeoffs:

- Because plug-ins can be developed by different organizations, it is easier to introduce security vulnerabilities and privacy threats.

Layers Pattern

The layers pattern divides the system in such a way that the modules can be developed and evolved separately with little interaction among the parts, which supports portability, modifiability, and reuse. To achieve this separation of concerns, the layers pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The *allowed-to-use* relationship among the layers is subject to a key constraint: 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 software assigned to layer A is allowed to use any of the public facilities provided by layer B. (In a vertically arranged representation of layers, which is almost ubiquitous, A will be drawn higher than B.) In some cases, modules in one layer are required to directly use modules in a nonadjacent lower layer, although 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*. Upward usages are not allowed in this pattern.

Benefits:

- Because a layer is constrained to use only lower layers, software in lower layers can be changed (as long as the interface does not change) without affecting the upper layers.
- Lower-level layers may be reused across different applications. For example, suppose a certain layer allows portability across operating systems. This layer would be useful in any system that must run on multiple, different operating systems. The lowest layers are often provided by commercial software—an operating system, for example, or network communications software.
- Because the *allowed-to-use* relations are constrained, the number of interfaces that any team must understand is reduced.

Tradeoffs:

- 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.
- Layering often adds a performance penalty to a system. If a call is made from a function in the top-most layer, it may have to traverse many lower layers before being executed by the hardware.
- If many instances of layer bridging occur, the system may not meet its portability and modifiability goals, which strict layering helps to achieve.

Publish-Subscribe Pattern

Publish-subscribe is an architectural pattern in which components communicate primarily through asynchronous messages, sometimes referred to as “events” or “topics.” The publishers have no knowledge of the subscribers, and subscribers are only aware of message types. Systems using the publish-subscribe pattern rely on implicit invocation; that is, the component publishing a message does not directly invoke any other component. Components publish messages on one or more events or topics, and other components register an interest in the publication. At runtime, when a message is published, the publish–subscribe (or event) bus notifies all of the elements that registered an interest in the event or topic. In this way, the message publication causes an implicit invocation of (methods in) other components. The result is loose coupling between the publishers and the subscribers.

The publish-subscribe pattern has three types of elements:

- *Publisher component.* Sends (publishes) messages.
- *Subscriber component.* Subscribes to and then receives messages.
- *Event bus.* Manages subscriptions and message dispatch as part of the runtime infrastructure.

Benefits:

- Publishers and subscribers are independent and hence loosely coupled. Adding or changing subscribers requires only registering for an event and causes no changes to the publisher.

- System behavior can be easily changed by changing the event or topic of a message being published, and consequently which subscribers might receive and act on this message. This seemingly small change can have large consequences, as features may be turned on or off by adding or suppressing messages.
- Events can be logged easily to allow for record and playback and thereby reproduce error conditions that can be challenging to recreate manually.

Tradeoffs:

- Some implementations of the publish-subscribe pattern can negatively impact performance (latency). Use of a distributed coordination mechanism will ameliorate the performance degradation.
- In some cases, a component cannot be sure how long it will take to receive a published message. In general, system performance and resource management are more difficult to reason about in publish-subscribe systems.
- Use of this pattern can negatively impact the determinism produced by synchronous systems. The order in which methods are invoked, as a result of an event, can vary in some implementations.
- Use of the publish-subscribe pattern can negatively impact testability. Seemingly small changes in the event bus—such as a change in which components are associated with which events—can have a wide impact on system behavior and quality of service.
- Some publish-subscribe implementations limit the mechanisms available to flexibly implement security (integrity). Since publishers do not know the identity of their subscribers, and vice versa, end-to-end encryption is limited. Messages from a publisher to the event bus can be uniquely encrypted, and messages from the event bus to a subscriber can be uniquely encrypted; however, any end-to-end encrypted communication requires all publishers and subscribers involved to share the same key.

8.5 For Further Reading

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

More patterns for modifiability are given in *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* [Woods 11].

The Decoupling Level metric [Mo 16] is an architecture-level coupling metric that can give insights into how globally coupled an architecture is. This information can be used to track coupling over time, as an early warning indicator of technical debt.

A fully automated way of detecting modularity violations—and other kinds of design flaws—has been described in [Mo 19]. The detected violations can be used as a guide to refactoring, so as to increase cohesion and reduce coupling.

Software modules intended for use in a software product line are often imbued with variation mechanisms that allow them to be quickly modified to serve in different applications—that is, in different members of the product line. Lists of variation mechanisms for components in a product line can be found in the works by Bachmann and Clements [Bachmann 05], Jacobson and colleagues [Jacobson 97], and Anastasopoulos and colleagues [Anastasopoulos 00].

The layers pattern comes in many forms and variations—“layers with a sidecar,” for example. Section 2.4 of [DSA2] sorts them all out, and discusses why (surprisingly for an architectural pattern invented more than a half-century ago) most layer diagrams for software that you’ve ever seen are very ambiguous. If you don’t want to spring for the book, then [Bachmann 00a] is a good substitute.

8.6 Discussion Questions

1. Modifiability comes in many flavors and is known by many names; we discussed a few in the opening section of this chapter, but that discussion only scratches the surface. 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. In the list you compiled for question 1, which tactics and patterns are especially helpful for each?
3. For each quality attribute that you discovered as a result of question 2, write a modifiability scenario that expresses it.
4. In many laundromats, washing machines and dryers accept coins but do not give change. Instead, separate machines dispense change. In an average laundromat, there are six or eight washers and dryers for every change machine. What modifiability tactics do you see at work in this arrangement? What can you say about availability?
5. For the laundromat in question 4, describe the specific form of modifiability (using a modifiability scenario) that seems to be the aim of arranging the machines as described.
6. A *wrapper*, introduced in Chapter 7, is a common architectural pattern to aid modifiability. Which modifiability tactics does a wrapper embody?
7. Other common architectural patterns that can increase a system’s modifiability include blackboard, broker, peer-to-peer, model-view-controller, and reflection. Discuss each in terms of the modifiability tactics it packages.
8. 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 an undesirable circumvention of an intermediary. Discuss some non-architectural means as well.

9. The abstract common services tactic is intended to reduce coupling but might also reduce cohesion. Discuss.
10. Discuss the proposition that the client-server pattern is the microkernel pattern with runtime binding.

9



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. The melancholy fact is that operations on computers take time. Computations take time on the order of thousands of nanoseconds, disk access (whether solid state or rotating) takes time on the order of tens of milliseconds, and network access takes time ranging from hundreds of microseconds within the same data center to upward of 100 milliseconds for intercontinental messages. Time must be taken into consideration when designing your system for performance.

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's 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. Services get events from other services. 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, a database-centric system, or a system processing input signals from its environment, the desired response might be expressed as the number of requests that can be processed in a unit of time. 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, which began when computers were slow and expensive and the tasks to perform dwarfed the ability to do them, performance has been the driving factor in 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.

But performance remains of fundamental importance. There are still (and will likely always be) important problems that we know *how* to solve with computers, but that we can't solve fast enough to be useful.

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 you 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. They're certainly linked, although technically scalability is making your system easy to change in a particular way, and so is a kind of modifiability, as discussed in Chapter 8. In addition, scalability of services in the cloud is discussed explicitly in Chapter 17.

Often, performance improvement happens after you have constructed a version of your system and found its performance to be inadequate. You can anticipate this by architecting your system with performance in mind. For example, if you have designed the system with a scalable resource pool, and you subsequently determine that this pool is a bottleneck (from your instrumented data), then you can easily increase the size of the pool. If not, your options are limited—and mostly all bad—and they may involve considerable rework.

It is not useful to spend a lot of your time optimizing a portion of the system that is responsible for only a small percentage of the total time. Instrumenting the system by logging timing information will help you determine where the actual time is spent and allow you to focus on improving the performance of critical portions of the system.

9.1 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 topics 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 anytime your system creates a new thread, because threads, by definition, are independent sequences of control. Multitasking on your system is supported by independent threads. Multiple users are simultaneously supported on your system through the use of threads. Concurrency also occurs anytime your system is executing on more than one processor, whether those processors are packaged separately or as multi-core processors. In addition, you must consider concurrency when you use parallel algorithms, parallelizing infrastructures such as map-reduce, or NoSQL databases, or when you use 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.

As our example shows, race conditions can occur when two threads of control are present 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 we have two instances of *x*, *x* is not shared by the two threads and no race condition will occur.

Race conditions are among 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 more than 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 (or take actions to ensure) that race conditions will not occur in those sections.

—LB

Table 9.1 summarizes the general scenario for performance.

TABLE 9.1 Performance General Scenario

Portion of Scenario	Description	Possible Values
Source	The stimulus can come from a user (or multiple users), from an external system, or from some portion of the system under consideration.	<p>External:</p> <ul style="list-style-type: none"> ▪ User request ▪ Request from external system ▪ Data arriving from a sensor or other system <p>Internal:</p> <ul style="list-style-type: none"> ▪ One component may make a request of another component. ▪ A timer may generate a notification.
Stimulus	The stimulus is the arrival of an event. The event can be a request for service or a notification of some state of either the system under consideration or an external system.	<p>Arrival of a periodic, sporadic, or stochastic event:</p> <ul style="list-style-type: none"> ▪ A periodic event arrives at a predictable interval. ▪ A stochastic event arrives according to some probability distribution. ▪ A sporadic event arrives according to a pattern that is neither periodic nor stochastic.
Artifact	The artifact stimulated may be the whole system or just a portion of the system. For example, a power-on event may stimulate the whole system. A user request may arrive at (stimulate) the user interface.	<ul style="list-style-type: none"> ▪ Whole system ▪ Component within the system
Environment	The state of the system or component when the stimulus arrives. Unusual modes—error mode, overloaded mode—will affect the response. For example, three unsuccessful login attempts are allowed before a device is locked out.	<p>Runtime. The system or component can be operating in:</p> <ul style="list-style-type: none"> ▪ Normal mode ▪ Emergency mode ▪ Error correction mode ▪ Peak load ▪ Overload mode ▪ Degraded operation mode ▪ Some other defined mode of the system
Response	The system will process the stimulus. Processing the stimulus will take time. This time may be required for computation, or it may be required because processing is blocked by contention for shared resources. Requests can fail to be satisfied because the system is overloaded or because of a failure somewhere in the processing chain.	<ul style="list-style-type: none"> ▪ System returns a response ▪ System returns an error ▪ System generates no response ▪ System ignores the request if overloaded ▪ System changes the mode or level of service ▪ System services a higher-priority event ▪ System consumes resources

Portion of Scenario	Description	Possible Values
Response measure	Timing measures can include latency or throughput. Systems with timing deadlines can also measure jitter of response and ability to meet the deadlines. Measuring how many of the requests go unsatisfied is also a type of measure, as is how much of a computing resource (e.g., a CPU, memory, thread pool, buffer) is utilized.	<ul style="list-style-type: none"> ▪ The (maximum, minimum, mean, median) time the response takes (latency) ▪ The number or percentage of satisfied requests over some time interval (throughput) or set of events received ▪ The number or percentage of requests that go unsatisfied ▪ The variation in response time (jitter) ▪ Usage level of a computing resource

Figure 9.1 gives an example concrete performance scenario: *Five hundred users initiate 2,000 requests in a 30-second interval, under normal operations. The system processes all of the requests with an average latency of two seconds.*

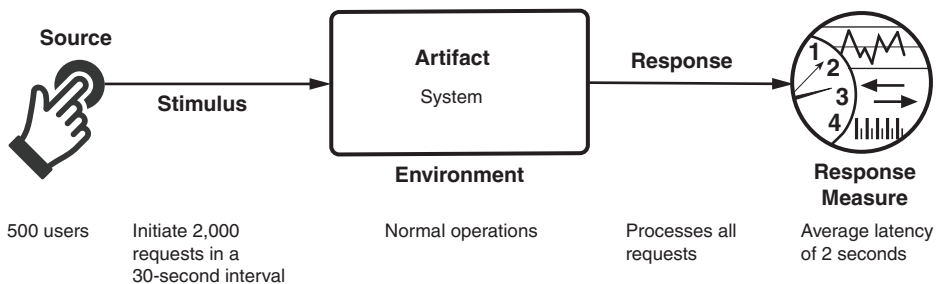
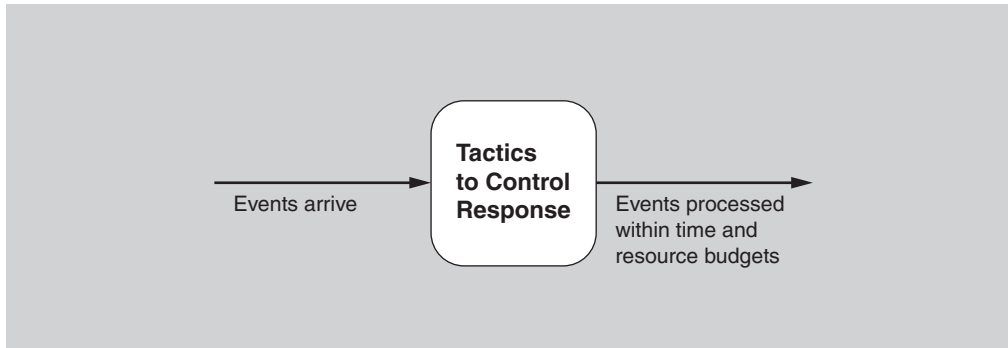


FIGURE 9.1 Sample performance scenario

9.2 Tactics for Performance

The goal of performance tactics is to generate a response to events arriving at the system under some time-based or resource-based constraint. The event can be a single event or a stream, and is the trigger to perform computation. Performance tactics control the time or resources used to generate a response, as illustrated in Figure 9.2.

**FIGURE 9.2** 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 and resource usage: processing time (when the system is working to respond and actively consuming resources) and blocked time (when the system is unable to respond).

- *Processing time and resource usage.* 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, thread pools and buffers must be managed and access to critical sections must be made sequential.

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 some component, another system, or some actor. Each of these steps contributes to the overall latency and resource consumption 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. In contrast, when you start to run out of memory, at some point the page swapping becomes overwhelming and performance crashes suddenly.

- *Blocked time and resource contention.* 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 be used by only a single client at a time. As a consequence, other clients must wait for access to those resources. Figure 9.2

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 that occurs, the more latency grows.

- *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 for any reason.
- *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 when the called component is at the other end of a network (as opposed to co-located on the same processor), or when the called component is heavily loaded.

Whatever the cause, you must identify places in the architecture where resource limitations might cause a significant contribution to overall latency.

With this background, we turn to our tactic categories. We can either reduce demand for resources (control resource demand) or make the resources we have available handle the demand more effectively (manage resources).

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 or by limiting the rate at which the system responds to events. In addition, a number of techniques can be applied to ensure that the resources that you do have are applied judiciously:

- *Manage work requests.* One way to reduce work is to reduce the number of requests coming into the system to do work. Ways to do that include the following:
 - *Manage event arrival.* A common way to manage event arrivals from an external system is to put in place a service level agreement (SLA) that specifies the maximum event arrival rate that you are willing to support. An SLA is an agreement of the form “The system or component will process X events arriving per unit time with a response time of Y.” This agreement constrains both the system—it must provide that response—and the client—if it makes more than X requests per unit time, the response is not guaranteed. Thus, from the client’s perspective, if it needs more than X requests per unit time to be serviced, it must utilize multiple instances of the element processing the requests. SLAs are one method for managing scalability for Internet-based systems.
 - *Manage sampling rate.* In cases where the system cannot maintain adequate response levels, you can reduce the sampling frequency of the stimuli—for example, the rate at which data is received from a sensor or the number of video frames per second that you process. Of course, the price paid here is the fidelity of the video stream or the information you gather from the sensor data. Nevertheless, this is a viable strategy if the result is “good enough.” Such an approach is commonly used in signal processing

systems where, for example, different codices can be chosen with different sampling rates and data formats. This design choice seeks to maintain predictable levels of latency; you must decide whether having a lower fidelity but consistent stream of data is preferable to having erratic latency. Some systems manage the sampling rate dynamically in response to latency measures or accuracy needs.

- *Limit event response.* When discrete events arrive at the system (or component) too rapidly to be processed, then the events must be queued until they can be processed, or they are simply discarded. You may choose to process events only up to a set maximum rate, thereby ensuring predictable processing for the events that are actually processed. This tactic could be triggered by a queue size or processor utilization exceeding some warning level. Alternatively, it could be triggered by an event rate that violates an SLA. 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. Conversely, if you choose to drop events, then you need to choose a policy: 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 insufficient resources are available to service them when they arise, low-priority events might be ignored. Ignoring events consumes minimal resources (including time), thereby increasing 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 being too cold.
- *Reduce computational overhead.* For events that do make it into the system, the following approaches can be implemented to reduce the amount of work involved in handling each event:
 - *Reduce indirection.* The use of intermediaries (so important for modifiability, as we saw in Chapter 8) increases the computational overhead in processing an event stream, 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. You may be able to realize the best of both worlds, however: Clever code optimization can let you *program* using the intermediaries and interfaces that support encapsulation (and thus keep the modifiability) but reduce, or in some cases eliminate, the costly indirection at runtime. Similarly, some brokers allow for direct communication between a client and a server (after initially establishing the relationship via the broker), thereby eliminating the indirection step for all subsequent requests.
 - *Co-locate communicating resources.* Context switching and intercomponent communication costs add up, especially when the components are on different nodes on a network. One 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; or it may mean placing tiers of a multi-tier architecture on the same rack in the data center.

- *Periodic cleaning.* A special case when 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. Many system administrators and even regular computer users do a periodic reboot of their systems for exactly this reason.
- *Bound execution times.* You can 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, however, 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 efficiency of resource usage.* Improving the efficiency of algorithms used in critical areas can decrease latency and improve throughput and resource consumption. This is, for some programmers, their *primary* performance tactic. If the system does not perform adequately, they try to “tune up” their processing logic. As you can see, this approach is actually just one of many tactics available.

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 which resources are more critical: time, space, or network bandwidth. Here are some resource management tactics:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential to improve performance. Cost is usually a consideration in the choice of resources, but increasing the resources is, in many cases, 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, you can choose scheduling policies to achieve the goals you find desirable using the schedule resources tactic.)
- *Maintain multiple copies of computations.* This tactic reduces the contention that would occur if all requests for service were allocated to a single instance. Replicated services in a microservice architecture or replicated web servers in a server pool are examples of replicas of computation. 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 a round-robin scheme or assigning the next request to the least busy server. The load balancer pattern is discussed in detail in Section 9.4.

- *Maintain multiple copies of data.* Two common examples of maintaining multiple copies of data are data replication and caching. *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses. Because the data being replicated is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume. *Caching* also involves keeping copies of data (with one set of data possibly being a subset of the other), but on storage with different access speeds. The different access speeds may be due to memory speed versus secondary storage speed, or the speed of local versus remote communication. Another responsibility with caching is choosing 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 to begin the calculations or prefetches necessary to comply with those requests before the user has made them.
- *Bound queue sizes.* This tactic controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to establish 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 contention for a resource occurs, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your concern as an architect is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it. (See the "Scheduling Policies" sidebar.)

Figure 9.3 summarizes the tactics for performance.

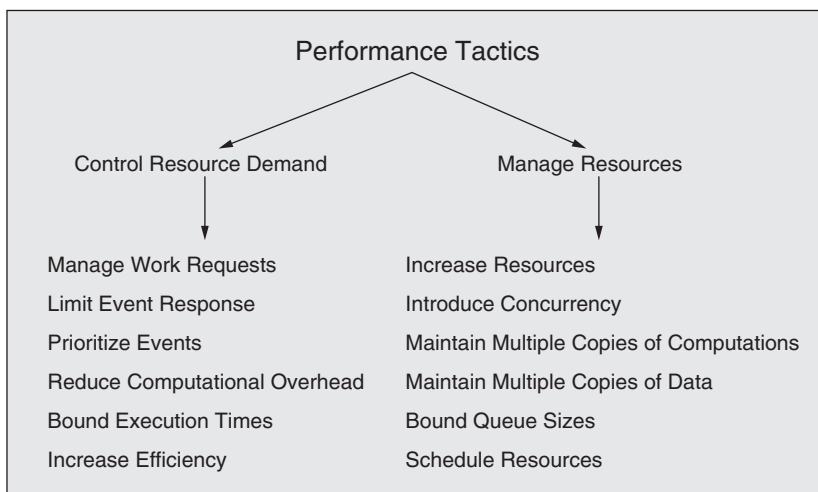


FIGURE 9.3 Performance tactics

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 scheduling policy has on the system's ability to meet them.

A high-priority event stream can be dispatched—assigned to a resource—only if that resource 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, or 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 creates a challenge.
- *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. However, it also admits the possibility that a lower-priority, but still important request might take 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.* Semantic importance assigns a priority statically according to some domain characteristic of the task that generates it.
 - *Deadline monotonic.* Deadline monotonic is a static priority assignment that assigns a higher priority to streams with shorter deadlines. This scheduling policy is used when scheduling streams of different priorities with real-time deadlines.
 - *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns a 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.* The round-robin scheduling strategy 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 possible assignment times are designated at fixed time intervals.
 - *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, both the earliest-deadline-first and least-slack-first scheduling strategies are optimal choices. 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 in which 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, users of the system can employ their own tricks:

- *Increase resources.* Buy a Ferrari, for example. All other things being equal, being 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.* Drive closer to the car in front of you, or load more people into the same vehicle (i.e., carpooling).

What is the point of this discussion? To paraphrase Gertrude Stein: Performance is performance is performance. Engineers have been analyzing and optimizing complex 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

9.3 Tactics-Based Questionnaire for Performance

Based on the tactics described in Section 9.2, we can create a set of tactics-inspired questions, as presented in Table 9.2. To gain an overview of the architectural choices made to support performance, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 9.2 Tactics-Based Questionnaire for Performance

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Control Resource Demand	<p>Do you have in place a service level agreement (SLA) that specifies the maximum event arrival rate that you are willing to support?</p> <p>Can you manage the rate at which you sample events arriving at the system?</p> <p>How will the system limit the response (amount of processing) for an event?</p> <p>Have you defined different categories of requests and defined priorities for each category?</p> <p>Can you reduce computational overhead by, for example, co-location, cleaning up resources, or reducing indirection?</p> <p>Can you bound the execution time of your algorithms?</p> <p>Can you increase computational efficiency through your choice of algorithms?</p>				
Manage Resources	<p>Can you allocate more resources to the system or its components?</p> <p>Are you employing concurrency? If requests can be processed in parallel, the blocked time can be reduced.</p> <p>Can computations be replicated on different processors?</p>				

continues

TABLE 9.2 Tactics-Based Questionnaire for Performance *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage Resources	<p>Can data be cached (to maintain a local copy that can be quickly accessed) or replicated (to reduce contention)?</p> <p>Can queue sizes be bounded to place an upper bound on the resources needed to process stimuli?</p> <p>Have you ensured that the scheduling strategies you are using are appropriate for your performance concerns?</p>				

9.4 Patterns for Performance

Performance concerns have plagued software engineers for decades, so it comes as no surprise that a rich set of patterns have been developed for managing various aspects of performance. In this section, we sample just a few of them. Note that some patterns serve multiple purposes. For example, we saw the circuit breaker pattern in Chapter 4, where it was identified as an availability pattern, but it also has a benefit for performance—since it reduces the time that you wait around for nonresponsive services.

The patterns we will introduce here are service mesh, load balancer, throttling, and map-reduce.

Service Mesh

The service mesh pattern is used in microservice architectures. The main feature of the mesh is a sidecar—a kind of proxy that accompanies each microservice, and which provides broadly useful capabilities to address application-independent concerns such as interservice communications, monitoring, and security. A sidecar executes alongside each microservice and handles all interservice communication and coordination. (As we will describe in Chapter 16, these elements are often packaged into *pods*.) They are deployed together, which cuts down on the latency due to networking, thereby boosting performance.

This approach allows developers to separate the functionality—the core business logic—of the microservice from the implementation, management, and maintenance of cross-cutting concerns, such as authentication and authorization, service discovery, load balancing, encryption, and observability.

Benefits:

- Software to manage cross-cutting concerns can be purchased off the shelf or implemented and maintained by a specialist team that does nothing else, allowing developers of the business logic to focus on only that concern.
- A service mesh enforces the deployment of utility functions onto the same processor as the services that use those utility functions. This cuts down on communication time between the service and its utilities since the communication does not need to use network messages.
- The service mesh can be configured to make communication dependent on context, thus simplifying functions such as the canary and A/B testing described in Chapter 3.

Tradeoffs:

- The sidecars introduce more executing processes, and each of these will consume some processing power, adding to the system's overhead.
- A sidecar typically includes multiple functions, and not all of these will be needed in every service or every invocation of a service.

Load Balancer

A load balancer is a kind of intermediary that handles messages originating from some set of clients and determines which instance of a service should respond to those messages. The key to this pattern is that the load balancer serves as a single point of contact for incoming messages—for example, a single IP address—but it then farms out requests to a pool of providers (servers or services) that can respond to the request. In this way, the load can be balanced across the pool of providers. The load balancer implements some form of the schedule resources tactic. The scheduling algorithm may be very simple, such as round-robin, or it may take into account the load on each provider, or the number of requests awaiting service at each provider.

Benefits:

- Any failure of a server is invisible to clients (assuming there are still some remaining processing resources).
- By sharing the load among several providers, latency can be kept lower and more predictable for clients.
- It is relatively simple to add more resources (more servers, faster servers) to the pool available to the load balancer, and no client needs to be aware of this.

Tradeoffs:

- The load balancing algorithm must be very fast; otherwise, it may itself contribute to performance problems.
- The load balancer is a potential bottleneck or single point of failure, so it is itself often replicated (and even load balanced).

Load balancers are discussed in much more detail in Chapter 17.

Throttling

The throttling pattern is a packaging of the manage work requests tactic. It is used to limit access to some important resource or service. In this pattern, there is typically an intermediary—a throttler—that monitors (requests to) the service and determines whether an incoming request can be serviced.

Benefits:

- By throttling incoming requests, you can gracefully handle variations in demand. In doing so, services never become overloaded; they can be kept in a performance “sweet spot” where they handle requests efficiently.

Tradeoffs:

- The throttling logic must be very fast; otherwise, it may itself contribute to performance problems.
- If client demand regularly exceeds capacity, buffers will need to be very large, or there is a risk of losing requests.
- This pattern can be difficult to add to an existing system where clients and servers are tightly coupled.

Map-Reduce

The map-reduce pattern efficiently performs a distributed and parallel sort of a large data set and provides a simple means for the programmer to specify the analysis to be done. Unlike our other patterns for performance, which are independent of any application, the map-reduce pattern is specifically designed to bring high performance to a specific kind of recurring problem: sort and analyze a large data set. This problem is experienced by any organization dealing with massive data—think Google, Facebook, Yahoo, and Netflix—and all of these organizations do in fact use map-reduce.

The map-reduce pattern has three parts:

- First is a specialized infrastructure that 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 virtual machine, 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 and a data set. It uses the key to hash the data into a set of buckets. For example, if our data set consisted of playing cards, the key could be the suit. The map function is also used to filter the data—that is, determine whether a data record is to be involved in further processing or discarded. Continuing our card example, we might choose to discard jokers or letter cards (A, K, Q, J), keeping only numeric cards, and we could then map each card into a bucket, based on its suit. The performance of the map phase of the map-reduce pattern is enhanced by having multiple map instances, each of which processes a different portion of the data

set. An input file is divided into portions, and a number of map instances are created to process each portion. Continuing our example, let's consider that we have 1 billion playing cards, not just a single deck. Since each card can be examined in isolation, the map process can be carried out by tens or hundreds of thousands of instances in parallel, with no need for communication among them. Once all of the input data has been mapped, these buckets are shuffled by the map-reduce infrastructure, and then assigned to new processing nodes (possibly reusing the nodes used in the map phase) for the reduce phase. For example, all of the clubs could be assigned to one cluster of instances, all of the diamonds to another cluster, and so forth.

- All of the heavy analysis takes place in the *reduce* function. The number of reduce instances corresponds to the number of buckets output by the map function. The reduce phase does some programmer-specified analysis and then emits the results of that analysis. For example, we could count the number of clubs, diamonds, hearts, and spades, or we could sum the numeric values of all of the cards in each bucket. The output set is almost always much smaller than the input sets—hence the name “reduce.”

The map instances are stateless and do not communicate with each other. The only communication between the map instances and the reduce instances is the data emitted from the map instances as `<key, value>` pairs.

Benefits:

- Extremely large, unsorted data sets can be efficiently analyzed through the exploitation of parallelism.
- A failure of any instance has only a small impact on the processing, since map-reduce typically breaks large input datasets into many smaller ones for processing, allocating each to its own instance.

Tradeoffs:

- If you do not have large data sets, the overhead incurred by the map-reduce pattern is not justified.
- If you cannot divide your data set into similarly sized subsets, the advantages of parallelism are lost.
- Operations that require multiple reduces are complex to orchestrate.

9.5 For Further Reading

Performance is the subject of a rich body of literature. Here are some books we recommend as general overviews of performance:

- *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice* [Bondi 14]. This book provides a comprehensive overview of performance engineering, ranging from technical practices to organizational ones.

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

To get an overview of some of the many patterns for performance, see *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* [Douglass 99] and *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management* [Kircher 03]. In addition, Microsoft has published a catalog of performance and scalability patterns for cloud-based applications: <https://docs.microsoft.com/en-us/azure/architecture/patterns/category/performance-scalability>.

9.6 Discussion Questions

1. “Every system has real-time performance constraints.” Discuss. Can you provide a counterexample?
2. Write a concrete performance scenario that describes the average on-time flight arrival performance for an airline.
3. Write several performance scenarios for an online auction site. Think about whether your major concern is worst-case latency, average-case latency, throughput, or some other response measure. Which tactics would you use to satisfy your 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. A system may include 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 interaction mechanisms is whether interaction is synchronous or asynchronous. Discuss the advantages and disadvantages of each with respect to each of these performance responses: latency, deadline, throughput, jitter, miss rate, data loss, or any other required performance-related response you may be used to.
6. Find physical-world (that is, non-software) 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? What are the performance implications? (Hint: Think about race conditions.)

10



Safety

Giles: Well, for god's sake, be careful. . . . If you should be hurt or killed, shall take it amiss.

Willow: Well, we try not to get killed. That's part of our whole mission statement: Don't get killed.

Giles: Good.

—*Buffy the Vampire Slayer*, Season 3, episode “Anne”

“Don’t kill anyone” should be a part of every software architect’s mission statement.

The thought that software could kill people or cause injury or damage used to belong solidly in the realm of computers-run-amok science fiction; think of HAL politely declining to open the pod bay doors in the now-aged but still-classic movie *2001: A Space Odyssey*, leaving Dave stranded in space.

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.

The thought that software (strings of 0s and 1s) can kill or maim or destroy is still an unnatural notion. To be fair, it’s not the 0s and 1s that wreak havoc—at least, not directly. It’s what they’re connected to. Software, and the computer 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.

In 2009, an employee of the Shushenskaya hydroelectric power station used a cybernetwork to remotely—and accidentally—activate an unused turbine with a few errant keystrokes. The offline turbine created a “water hammer” that flooded and then destroyed the plant and killed dozens of workers.

There are many other equally notorious examples. The Therac 25 fatal radiation overdose, the Ariane 5 explosion, and a hundred lesser-known accidents all caused harm because the computer was connected to the environment: a turbine, an X-ray emitter, and a rocket’s steering controls, in the examples just cited. The infamous Stuxnet virus was created to intentionally cause damage and destruction. In these cases, software commanded some hardware in its environment to take a disastrous action, and the hardware obeyed. Actuators are devices

that connect hardware to software; they are the bridge between the world of 0s and 1s 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 connecting to the outside world doesn't have to mean robot arms or uranium centrifuges or missile launchers: Connecting to a simple display screen is enough. 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 computers, believing them to be in error. He thought it extremely unlikely that the United States would have fired just a few missiles, thereby inviting mass 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. You and/or your parents may well owe your life to Lieutenant Colonel Petrov.

Of course, the humans don't always get it right when the computers get it wrong. On the stormy night of June 1, 2009, Air France flight 447 from Rio de Janeiro to Paris plummeted into the Atlantic Ocean, killing all 228 people on board, despite the aircraft's engines and flight controls working perfectly. The Airbus A-330's flight recorders, which were not recovered until May 2011, showed 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 autopilot disengaged as a result. 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 3-minute-plus plunge from 35,000 feet, the pilots kept trying to pull the nose up and throttle back to lower the speed, when all they needed to do was lower the nose to increase the speed and resume normal flying. Very probably 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 software deactivates the stall warning when it "thinks" that the angle of attack measurements are invalid. This can occur when the airspeed readings are very low. That is what happened with AF447: Its forward speed dropped below 60 knots, and the angle of attack was extremely high. As a consequence of this flight control software rule, the stall warning stopped and started several times. Worse, it came on whenever the pilot pushed forward on the stick (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 exactly the wrong feedback, and vice versa. Was this an unsafe system, or a safe system operated unsafely? Ultimately questions like this are decided in the courts.

As this edition was going to publication, Boeing was still reeling from the grounding of its 737 MAX aircraft after two crashes that appear to have been caused at least partly by a piece of software called MCAS, which pushed the aircraft's nose down at the wrong time. Faulty sensors seem to be involved here, too, as well as a baffling design decision that caused the software to rely on only one sensor to determine its behavior, instead of the two available on the aircraft. It also appears that Boeing never tested the software in question under the

conditions of a sensor failure. The company did provide a way to disable the system in flight, although remembering how to do that when your airplane is doing its best to kill you may be asking a lot of a flight crew—especially when they were never made aware of the existence of the MCAS in the first place. In total, 346 people died in the two crashes of the 737 MAX.

Okay, enough scary stories. Let's talk about the principles behind them as they affect software and architectures.

Safety is concerned with a system's ability to avoid straying into states that cause or lead to damage, injury, or loss of life to actors in its environment. These unsafe states can be caused by a variety of factors:

- *Omissions* (the failure of an event to occur).
- *Commission* (the spurious occurrence of an undesirable event). The event could be acceptable in some system states but undesirable in others.
- *Timing*. Early (the occurrence of an event before the time required) or late (the occurrence of an event after the time required) timing can both be potentially problematic.
- *Problems with system values*. These come in two categories: Coarse incorrect values are incorrect but detectable, whereas subtle incorrect values are typically undetectable.
- *Sequence omission and commission*. In a sequence of events, either an event is missing (omission) or an unexpected event is inserted (commission).
- *Out of sequence*. A sequence of events arrive, but not in the prescribed order.

Safety is also concerned with detecting and recovering from these unsafe states to prevent or at least minimize resulting harm.

Any portion of the system can lead to an unsafe state: The software, the hardware portions, or the environment can behave in an unanticipated, unsafe fashion. Once an unsafe state is detected, the potential system responses are similar to those enumerated for availability (in Chapter 4). The unsafe state should be recognized and the system should be made through

- Continuing operations after recovering from the unsafe state or placing the system in a safe mode, or
- Shutting down (fail safe), or
- Transitioning to a state requiring manual operation (e.g., manual steering if the power steering in a car fails).

In addition, the unsafe state should be reported immediately and/or logged.

Architecting for safety begins by identifying the system's safety-critical functions—those functions that could cause harm as just outlined—using techniques such as failure mode and effects analysis (FMEA; also called hazard analysis) and fault tree analysis (FTA). FTA is a top-down deductive approach to identify failures that could result in moving the system into an unsafe state. Once the failures have been identified, the architect needs to design mechanisms to detect and mitigate the fault (and ultimately the hazard).

The techniques outlined in this chapter are intended to discover possible hazards that could result from the system's operation and help in creating strategies to cope with these hazards.

10.1 Safety General Scenario

With this background, we can construct the general scenario for safety, shown in Table 10.1.

TABLE 10.1 Safety General Scenario

Portion of Scenario	Description	Possible Values
Source	A data source (a sensor, a software component that calculates a value, a communication channel), a time source (clock), or a user action	Specific instances of a: <ul style="list-style-type: none"> ▪ Sensor ▪ Software component ▪ Communication channel ▪ Device (such as a clock)
Stimulus	An omission, commission, or occurrence of incorrect data or timing	<p>A specific instance of an omission:</p> <ul style="list-style-type: none"> ▪ A value never arrives. ▪ A function is never performed. <p>A specific instance of a commission:</p> <ul style="list-style-type: none"> ▪ A function is performed incorrectly. ▪ A device produces a spurious event. ▪ A device produces incorrect data. <p>A specific instance of incorrect data:</p> <ul style="list-style-type: none"> ▪ A sensor reports incorrect data. ▪ A software component produces incorrect results. <p>A timing failure:</p> <ul style="list-style-type: none"> ▪ Data arrives too late or too early. ▪ A generated event occurs too late or too early or at the wrong rate. ▪ Events occur in the wrong order.
Environment	System operating mode	<ul style="list-style-type: none"> ▪ Normal operation ▪ Degraded operation ▪ Manual operation ▪ Recovery mode
Artifacts	The artifact is some part of the system.	Safety-critical portions of the system
Response	The system does not leave a safe state space, or the system returns to a safe state space, or the system continues to operate in a degraded mode to prevent (further) injury or damage or to minimize injury or damage. Users are advised of the unsafe state or the prevention of entry into the unsafe state. The event is logged.	<p>Recognize the unsafe state and one or more of the following:</p> <ul style="list-style-type: none"> ▪ Avoid the unsafe state ▪ Recover ▪ Continue in degraded or safe mode ▪ Shut down ▪ Switch to manual operation ▪ Switch to a backup system ▪ Notify appropriate entities (people or systems) ▪ Log the unsafe state (and the response to it)

Portion of Scenario	Description	Possible Values
Response measure	Time to return to safe state space; damage or injury caused	One or more of the following: <ul style="list-style-type: none">▪ Amount or percentage of entries into unsafe states that are avoided▪ Amount or percentages of unsafe states from which the system can (automatically) recover▪ Change in risk exposure: size(loss) * prob(loss)▪ Percentage of time the system can recover▪ Amount of time the system is in a degraded or safe mode▪ Amount or percentage of time the system is shut down▪ Elapsed time to enter and recover (from manual operation, from a safe or degraded mode)

A sample safety scenario is: *A sensor in the patient monitoring system fails to report a life-critical value after 100 ms. The failure is logged, a warning light is illuminated on the console, and a backup (lower-fidelity) sensor is engaged. The system monitors the patient using the backup sensor after no more than 300 ms.* Figure 10.1 illustrates this scenario.

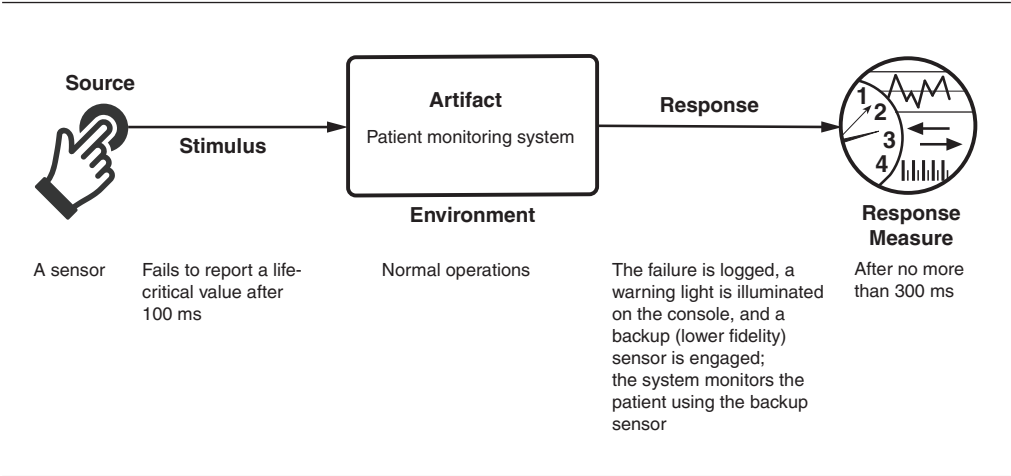


FIGURE 10.1 Sample concrete safety scenario

10.2 Tactics for Safety

Safety tactics may be broadly categorized as unsafe state avoidance, unsafe state detection, or unsafe state remediation. Figure 10.2 shows the goal of the set of safety tactics.

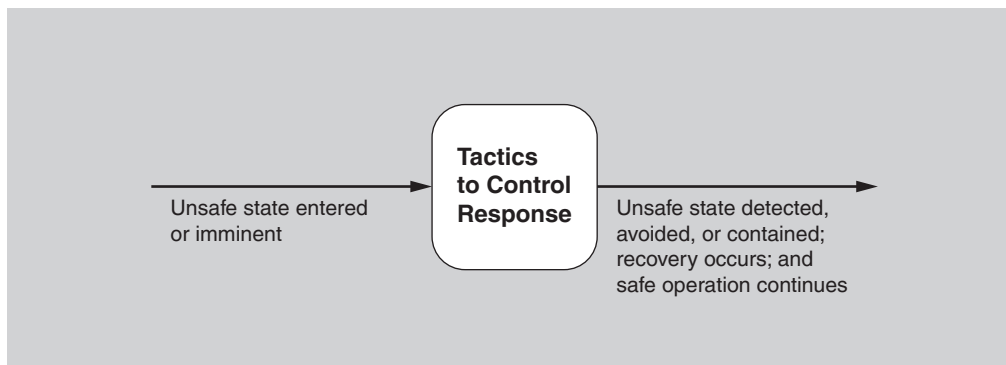


FIGURE 10.2 Goal of safety tactics

A logical precondition to avoid or detect entry into an unsafe state is the ability to recognize what constitutes an unsafe state. The following tactics assume that capability, which means that you should perform your own hazard analysis or FTA once you have your architecture in hand. Your design decisions may themselves have introduced new safety vulnerabilities not accounted for during requirements analysis.

You will note a substantial overlap between the tactics presented here and those presented in Chapter 4 on availability. This overlap occurs because availability problems may often lead to safety problems, and because many of the design solutions for repairing these problems are shared between the qualities.

Figure 10.3 summarizes the architectural tactics to achieve safety.

Unsafe State Avoidance

Substitution

This tactic employs protection mechanisms—often hardware-based—for potentially dangerous software design features. For example, hardware protection devices such as watchdogs, monitors, and interlocks can be used in lieu of software versions. Software versions of these mechanisms can be starved of resources, whereas a separate hardware device provides and controls its own resources. Substitution is typically beneficial only when the function being replaced is relatively simple.

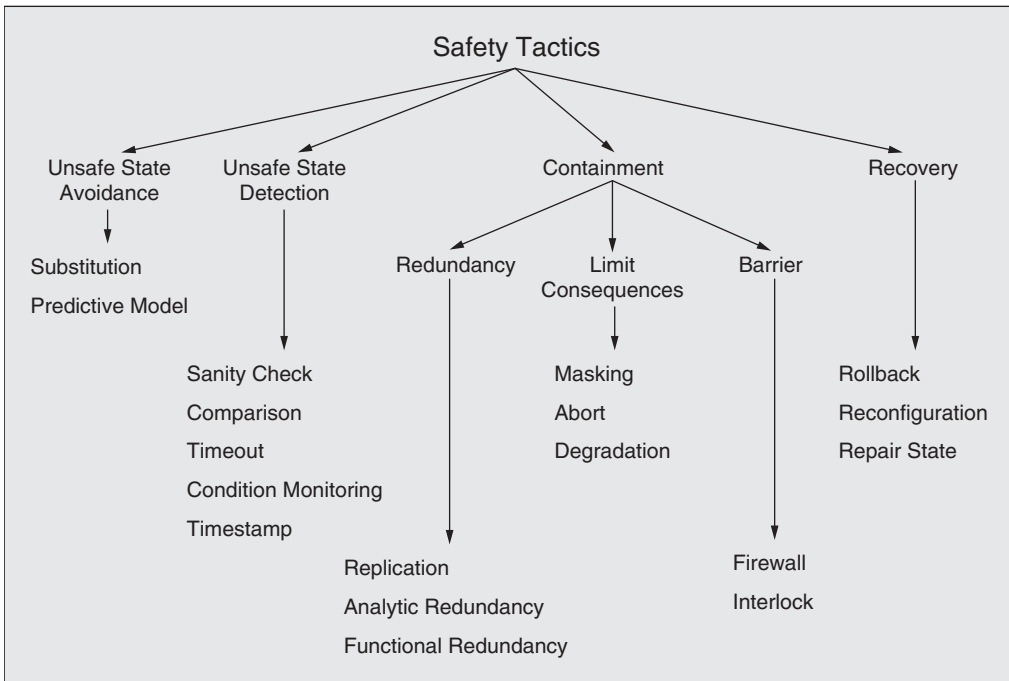


FIGURE 10.3 Safety tactics

Predictive Model

The predictive model tactic, as introduced in Chapter 4, predicts the state of health of system processes, resources, or other properties (based on monitoring the state), not only to ensure that the system is operating within its nominal operating parameters but also to provide early warning of a potential problem. For example, some automotive cruise control systems calculate the closing rate between the vehicle and an obstacle (or another vehicle) ahead and warn the driver before the distance and time become too small to avoid a collision. A predictive model is typically combined with condition monitoring, which we discuss later.

Unsafe State Detection

Timeout

The timeout tactic is used to determine whether the operation of a component is meeting its timing constraints. This might be realized in the form of an exception being raised, to indicate the failure of a component if its timing constraints are not met. Thus this tactic can detect late timing and omission failures. Timeout is a particularly common tactic in real-time or

embedded systems and distributed systems. It is related to the availability tactics of system monitor, heartbeat, and ping-echo.

Timestamp

As described in Chapter 4, the timestamp tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A timestamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Sequence numbers can also be used for this purpose, since timestamps in a distributed system may be inconsistent across different processors.

Condition Monitoring

This tactic involves checking conditions in a process or device, or validating assumptions made during the design, perhaps by using assertions. Condition monitoring identifies system states that may lead to hazardous behavior. However, the monitor should be simple (and, ideally, provable) to ensure that it does not introduce new software errors or contribute significantly to overall workload. Condition monitoring provides the input to a predictive model and to sanity checking.

Sanity Checking

The sanity checking tactic checks the validity or reasonableness of specific operation results, or inputs 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.

Comparison

The comparison tactic allows the system to detect unsafe states by comparing the outputs produced by a number of synchronized or replicated elements. Thus the comparison tactic works together with a redundancy tactic, typically the active redundancy tactic presented in the discussion of availability. When the number of replicants is three or greater, the comparison tactic can not only detect an unsafe state but also indicate which component has led to it. Comparison is related to the voting tactic used in availability. However, a comparison may not always lead to a vote; another option is to simply shut down if outputs differ.

Containment

Containment tactics seek to limit the harm associated with an unsafe state that has been entered. This category includes three subcategories: redundancy, limit consequences, and barrier.

Redundancy

The redundancy tactics, at first glance, appear to be similar to the various sparing/redundancy tactics presented in the discussion of availability. Clearly, these tactics overlap, but since the goals of safety and availability are different, the use of backup components differs. In

the realm of safety, redundancy enables the system to continue operation in the case where a total shutdown or further degradation would be undesirable.

Replication is the simplest redundancy tactic, as it just involves having clones of a component. Having multiple copies of identical components can be effective in protecting against random failures of hardware, but it cannot protect against design or implementation errors in hardware or software since there is no form of diversity embedded in this tactic.

Functional redundancy, by contrast, is intended to address the issue of common-mode failures (where replicas exhibit the same fault at the same time because they share the same implementation) in hardware or software components, by implementing design diversity. This tactic attempts to deal with the systematic nature of design faults by adding diversity to redundancy. The outputs of functionally redundant components should be the same given the same input. The functional redundancy tactic is still vulnerable to specification errors, however, and of course, functional replicas will be more expensive to develop and verify.

Finally, the analytic redundancy tactic permits not only diversity of components, but also a higher-level diversity that is visible at the input and output level. As a consequence, it can tolerate specification errors by using separate requirement specifications. Analytic redundancy often involves partitioning the system into high assurance and high performance (low assurance) portions. The high assurance portion is designed to be simple and reliable, whereas the high performance portion is typically designed to be more complex and more accurate, but less stable: It changes more rapidly, and may not be as reliable as the high assurance portion. (Hence, here we do not mean high performance in the sense of latency or throughput; rather, this portion “performs” its task better than the high assurance portion.)

Limit Consequences

The second subcategory of containment tactics is called limit consequences. These tactics are all intended to limit the bad effects that may result from the system entering an unsafe state.

The abort tactic is conceptually the simplest. If an operation is determined to be unsafe, it is aborted before it can cause damage. This technique is widely employed to ensure that systems fail safely.

The degradation tactic maintains the most critical system functions in the presence of component failures, dropping or replacing functionality in a controlled way. This approach allows individual component failures to gracefully reduce system functionality in a planned, deliberate, and safe way, rather than causing a complete system failure. For example, a car navigation system may continue to operate using a (less accurate) dead reckoning algorithm in a long tunnel where it has lost its GPS satellite signal.

The masking tactic masks a fault by comparing the results of several redundant components and employing a voting procedure in case one or more of the components differ. For this tactic to work as intended, the voter must be simple and highly reliable.

Barrier

The barrier tactics contain problems by keeping them from propagating.

The firewall tactic is a specific realization of the limit access tactic, which is described in Chapter 11. A firewall limits access to specified resources, typically processors, memory, and network connections.

The interlock tactic protects against failures arising from incorrect sequencing of events. Realizations of this tactic provide elaborate protection schemes by controlling all access to protected components, including controlling the correct sequencing of events affecting those components.

Recovery

The final category of safety tactics is recovery, which acts to place the system in a safe state. It encompasses three tactics: rollback, repair state, and reconfiguration.

The rollback tactic permits the system to revert to a saved copy of a previous known good state—the rollback line—upon the detection of a failure. This tactic is often combined with checkpointing and transactions, to ensure that the rollback is complete and consistent. Once the good state is reached, then execution can continue, potentially employing other tactics such as retry or degradation to ensure that the failure does not reoccur.

The repair state tactic repairs an erroneous state—effectively increasing the set of states that a component can handle competently (i.e., without failure)—and then continues execution. For example, a vehicle’s lane keep assist feature will monitor whether a driver is staying within their lane and actively return the vehicle to a position between the lines—a safe state—if it drifts out. This tactic is inappropriate as a means of recovery from unanticipated faults.

Reconfiguration attempts to recover from component failures by remapping the logical architecture onto the (potentially limited) resources left functioning. Ideally, this remapping allows full functionality to be maintained. When this is not possible, the system may be able to maintain partial functionality in combination with the degradation tactic.

10.3 Tactics-Based Questionnaire for Safety

Based on the tactics described in Section 10.2, we can create a set of tactics-inspired questions, as presented in Table 10.2. To gain an overview of the architectural choices made to support safety, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

Prior to beginning the tactics-based questionnaire for safety, you should assess whether the project under review has performed a hazard analysis or FTA to identify what constitutes an unsafe state (to be detected, avoided, contained, or recovered from) in your system. Without this analysis, designing for safety is likely to be less effective.

TABLE 10.2 Tactics-Based Questionnaire for Safety

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Unsafe State Avoidance	Do you employ substitution —that is, safer, often hardware-based protection mechanisms for potentially dangerous software design features? Do you use a predictive model to predict the state of health of system processes, resources, or other properties—based on monitored information—not only to ensure that the system is operating within its nominal operating parameters, but also to provide early warning of a potential problem?				
Unsafe State Detection	Do you use timeouts to determine whether the operation of a component meets its timing constraints? Do you use timestamps to detect incorrect sequences of events? Do you employ condition monitoring to check conditions in a process or device, particularly to validate assumptions made during design? Is sanity checking employed to check the validity or reasonableness of specific operation results, or inputs or outputs of a component? Does the system employ comparison to detect unsafe states, by comparing the outputs produced based on the number of synchronized or replicated elements?				
Containment: Redundancy	Do you use replication —clones of a component—to protect against random failures of hardware? Do you use functional redundancy to address the common-mode failures by implementing diversely designed components?				

continues

TABLE 10.2 Tactics-Based Questionnaire for Safety *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Containment: Redundancy	Do you use analytic redundancy —functional “replicas” that include high assurance/high performance and low assurance/low performance alternatives—to be able to tolerate specification errors?				
Containment: Limit Consequences	Can the system abort an operation that is determined to be unsafe before it can cause damage? Does the system provide controlled degradation , where the most critical system functions are maintained in the presence of component failures, while less critical functions are dropped or degraded? Does the system mask a fault by comparing the results of several redundant components and employ a voting procedure in case one or more of the components differ?				
Containment: Barrier	Does the system support limiting access to critical resources (e.g., processors, memory, and network connections) through a firewall ? Does the system control access to protected components and protect against failures arising from incorrect sequencing of events through interlocks ?				
Recovery	Is the system able to roll back —that is, to revert to a previous known good state—upon the detection of a failure? Can the system repair a state determined to be erroneous, without failure, and then continue execution? Can the system reconfigure resources, in the event of failures, by remapping the logical architecture onto the resources left functioning?				

10.4 Patterns for Safety

A system that unexpectedly stops operating, or starts operating incorrectly, or falls into a degraded mode of operation is likely to affect safety negatively, if not catastrophically. Hence, the first place to look for safety patterns is in patterns for availability, such as the ones described in Chapter 4. They all apply here.

- *Redundant sensors.* If the data produced by a sensor is important to determine whether a state is safe or unsafe, that sensor should be replicated. This protects against the failure of any single sensor. Also, independent software should monitor each sensor—in essence, the redundant spare tactic from Chapter 4 applied to safety-critical hardware.

Benefits:

- This form of redundancy, which is applied to sensors, guards against the failure of a single sensor.

Tradeoffs:

- Redundant sensors add cost to the system, and processing the inputs from multiple sensors is more complicated than processing the input from a single sensor.
- *Monitor-actuator.* This pattern focuses on two software elements—a monitor and an actuator controller—that are employed before sending a command to a physical actuator. The actuator controller performs the calculations necessary to determine the values to send to the physical actuator. The monitor checks these values for reasonableness before sending them. This separates the computation of the value from the testing of the value.

Benefits:

- In this form of redundancy applied to actuator control, the monitor acts as a redundant check on the actuator controller computations.

Tradeoffs:

- The development and maintenance of the monitor take time and resources.
- Because of the separation this pattern achieves between actuator control and monitoring, this particular tradeoff is easy to manipulate by making the monitor as simple (easy to produce but may miss errors) or as sophisticated (more complex but catches more errors) as required.
- *Separated safety.* Safety-critical systems must frequently be certified as safe by some authority. Certifying a large system is expensive, but dividing a system into safety-critical portions and non-safety-critical portions can reduce those costs. The safety-critical portion must still be certified. Likewise, the division into safety-critical and non-critical portions must be certified to ensure that there is no influence on the safety-critical portion from the non-safety-critical portion.

Benefits:

- The cost of certifying the system is reduced because you need to certify only a (usually small) portion of the total system.
- Cost and safety benefits accrue because the effort focuses on just those portions of the system that are germane to safety.

Tradeoffs:

- The work involved in performing the separation can be expensive, such as installing two different networks in a system to partition safety-critical and non-safety-critical messages. However, this approach limits the risk and consequences of bugs in the non-safety-critical portion from affecting the safety-critical portion.
- Separating the system and convincing the certification agency that the separation was performed correctly and that there are no influences from the non-safety-critical portion on the safety-critical portion is difficult, but is far easier than the alternative: having the agency certify everything to the same rigid level.

Design Assurance Levels

The separated safety pattern emphasizes dividing the software system into safety-critical portions and non-safety-critical portions. In avionics, the distinction is finer-grained. DO-178C, “Software Considerations in Airborne Systems and Equipment Certification,” is the primary document by which certification authorities such as Federal Aviation Administration (FAA), European Union Aviation Safety Agency (EASA), and Transport Canada approve all commercial software-based aerospace systems. It defines a ranking called Design Assurance Level (DAL) for each software function. The DAL is determined from the safety assessment process and hazard analysis by examining the effects of a failure condition in the system. The failure conditions are categorized by their effects on the aircraft, crew, and passengers:

- *A: Catastrophic.* Failure may cause deaths, usually with loss of the airplane.
- *B: Hazardous.* Failure has a large negative impact on safety or performance, or reduces the crew’s ability to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
- *C: Major.* Failure significantly reduces the safety margin or significantly increases crew workload, and may result in passenger discomfort (or even minor injuries).
- *D: Minor.* Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change.
- *E: No effect.* Failure has no impact on safety, aircraft operation, or crew workload.

Software validation and testing is a terrifically expensive task, undertaken with very finite budgets. DALs help you decide where to put your limited testing resources. The next time you’re on a commercial airline flight, if you see a glitch in the entertainment system or your reading light keeps blinking off, take comfort by thinking of all the validation money spent on making sure the flight control system works just fine.

—PC

10.5 For Further Reading

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, available at risks.org. This has been moderated by Peter Neumann since 1985 and is still going strong.

Two prominent standard safety *processes* are described in ARP-4761, “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment,” developed by SAE International, and MIL STD 882E, “Standard Practice: System Safety,” developed by the U.S. Department of Defense.

Wu and Kelly [Wu 04] published a set of safety tactics in 2004, based on a survey of existing architectural approaches, which inspired much of the thinking in this chapter.

Nancy Leveson is a 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, socio-technical, software-intensive systems.

The Federal Aviation Administration is the U.S. government agency charged with oversight of the U.S. airspace system and is extremely concerned about safety. Its 2019 *System Safety Handbook* is a good practical overview of the topic. Chapter 10 of this handbook deals with software safety. You can download it from faa.gov/regulations_policies/handbooks_manuals/aviation/risk_management/ss_handbook/.

Phil Koopman is well known in the automotive safety field. He has several tutorials available online that deal with safety-critical patterns. See, for example, youtube.com/watch?v=JA5wdyOjoXg and youtube.com/watch?v=4Tdh3jq6W4Y. Koopman’s book, *Better Embedded System Software*, gives much more detail about safety patterns [Koopman 10].

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. NASA’s 2002 *Fault Tree Handbook with Aerospace Applications* is an updated comprehensive primer of the NRC handbook. Both are available online as downloadable PDF files.

Similar to Design Assurance Levels, Safety Integrity Levels (SILs) provide definitions of how safety-critical various functions are. These definitions create a common understanding among the architects involved in designing the system, but also assist with safety evaluation. The IEC 61508 Standard titled “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems” defines four SILs, with SIL 4 being the most dependable and SIL 1 being the least dependable. This standard is instantiated through domain-specific standards such as IEC 62279 for the railway industry, titled “Railway Applications: Communication, Signaling and Processing Systems: Software for Railway Control and Protection Systems.”

In a world where semi-autonomous and autonomous vehicles are the subject of much research and development, functional safety is becoming increasingly more prominent. For a long time, ISO 26026 has been the standard in functional safety of road vehicles. There is

also a wave of new norms such as ANSI/UL 4600, “Standard for Safety for the Evaluation of Autonomous Vehicles and Other Products,” which tackle the challenges that emerge when software takes the wheel, figuratively and literally.

10.6 Discussion Questions

1. List 10 computer-controlled devices that are part of your everyday life right now, and hypothesize ways that a malicious or malfunctioning system could use them to hurt you.
2. Write a safety scenario that is designed to prevent a stationary robotic device (such as an assembly arm on a manufacturing line) from injuring someone, and discuss tactics to achieve it.
3. The U.S. Navy’s F/A-18 Hornet fighter aircraft was one of the early applications of fly-by-wire technology, in which onboard computers send digital commands to the control surfaces (ailerons, rudder, etc.) based on the pilot’s input to the control stick and rudder pedals. The flight control software was programmed to prevent the pilot from commanding certain violent maneuvers that might cause the aircraft to enter an unsafe flight regime. During early flight testing, which often involves pushing the aircraft to (and beyond) its utmost limits, an aircraft entered an unsafe state and “violent maneuvers” were exactly what were needed to save it—but the computers dutifully prevented them. The aircraft crashed into the ocean because of software designed to keep it safe. Write a safety scenario to address this situation, and discuss the tactics that would have prevented this outcome.
4. 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, resulting in several minor injuries and untold mayhem. Is Facebook unsafe? Discuss.
5. Write a safety scenario to protect the unfortunate girl in Germany from Facebook.
6. On February 25, 1991, during the Gulf War, a U.S. Patriot missile battery failed to intercept an incoming Scud missile, which struck a barracks, killing 28 soldiers and injuring dozens. The cause of the failure was an inaccurate calculation of the time since boot due to arithmetic errors in the software that accumulated over time. Write a safety scenario that addresses the Patriot failure and discuss tactics that might have prevented it.
7. Author James Gleick (“A Bug and a Crash,” 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 safety tends to “trade off” against the quality attributes of performance, availability, and interoperability.
9. Discuss the relationship between safety and testability.
10. What is the relationship between safety and modifiability?
11. With the Air France flight 447 story in mind, discuss the relationship between safety and usability.
12. Create a list of faults or 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 use tactics to accommodate these faults?

This page intentionally left blank

11



Security

*If you reveal your secrets to the wind, you should not blame the wind
for revealing them to the trees.*

—Kahlil Gibran

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 attack—that is, an action taken against a computer system with the intention of doing harm—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 focuses on three characteristics: confidentiality, integrity, and availability (CIA):

- *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.
- *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.
- *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 *this* book from an online bookstore.

We will use these characteristics in our general scenario for security.

One technique that is used in the security domain is threat modeling. An “attack tree,” which is similar to the fault tree discussed in Chapter 4, is used by security engineers to determine possible threats. The root of the tree 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 compromise CIA, with the leaves of attack trees being the stimulus in the scenario. The response to the attack is to preserve CIA or deter attackers through monitoring of their activities.

Privacy

An issue closely related to security is the quality of privacy. Privacy concerns have become more important in recent years and are enshrined into law in the European Union through the General Data Protection Regulation (GDPR). Other jurisdictions have adopted similar regulations.

Achieving privacy is about limiting access to information, which in turn is about which information should be access-limited and to whom access should be allowed. The general term for information that should be kept private is personally identifiable information (PII). The National Institute of Standards and Technology (NIST) defines PII as “any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individual’s identity, such as name, social security number, date and place of birth, mother’s maiden name, or biometric records; and (2) any other information that is linked or linkable to an individual, such as medical, educational, financial, and employment information.”

The question of who is permitted access to such data is more complicated. Users are routinely asked to review and agree to privacy agreements initiated by organizations. These privacy agreements detail who, outside of the collecting organization, is entitled to see PII. The collecting organization itself should have policies that govern who within that organization can have access to such data. Consider, for example, a tester for a software system. To perform tests, realistic data should be used. Does that data include PII? Generally, PII is obscured for testing purposes.

Frequently the architect, perhaps acting for the project manager, is asked to verify that PII is hidden from members of the development team who do not need to have access to PII.

11.1 Security General Scenario

From these considerations, we can now describe the individual portions of a security general scenario, which is summarized in Table 11.1.

TABLE 11.1 Security General Scenario

Portion of Scenario	Description	Possible Values
Source	The attack may be from outside the organization or from inside the organization. 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.	<ul style="list-style-type: none">▪ Human▪ Another system which is: <ul style="list-style-type: none">▪ Inside the organization▪ Outside the organization▪ Previously identified▪ Unknown

Portion of Scenario	Description	Possible Values
Stimulus	The stimulus is an attack.	<p>An unauthorized attempt to:</p> <ul style="list-style-type: none"> ▪ Display data ▪ Capture data ▪ Change or delete data ▪ Access system services ▪ Change the system's behavior ▪ Reduce availability
Artifact	What is the target of the attack?	<ul style="list-style-type: none"> ▪ System services ▪ Data within the system ▪ A component or resources of the system ▪ Data produced or consumed by the system
Environment	What is the state of the system when the attack occurs?	<p>The system is:</p> <ul style="list-style-type: none"> ▪ Online or offline ▪ Connected to or disconnected from a network ▪ Behind a firewall or open to a network ▪ Fully operational ▪ Partially operational ▪ Not operational
Response	The system ensures that confidentiality, integrity, and availability are maintained.	<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	Measures of a system's response are related to the frequency of successful attacks, the time and cost to resist and repair attacks, and the consequential damage of those attacks.	<p>One or more of the following:</p> <ul style="list-style-type: none"> ▪ How much of a resource is compromised or ensured ▪ Accuracy of attack detection ▪ How much time passed before an attack was detected ▪ How many attacks were resisted ▪ How long it takes to recover from a successful attack ▪ How much data is vulnerable to a particular attack

Figure 11.1 shows a sample concrete scenario derived from the general scenario: *A disgruntled employee at a remote location attempts to improperly modify the pay rate table during normal operations. The unauthorized access is detected, the system maintains an audit trail, and the correct data is restored within one day.*

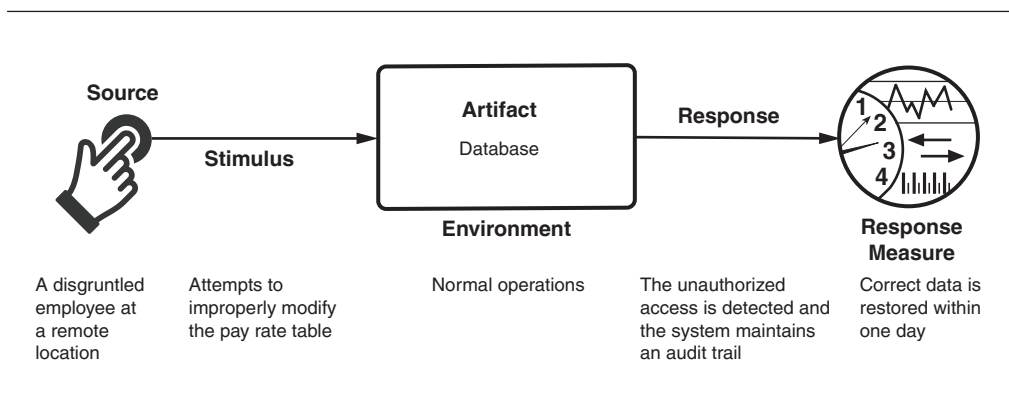


FIGURE 11.1 Sample scenario for security

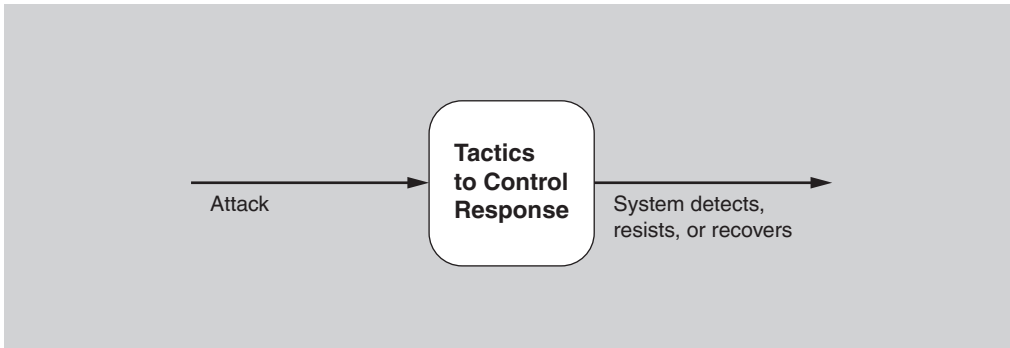
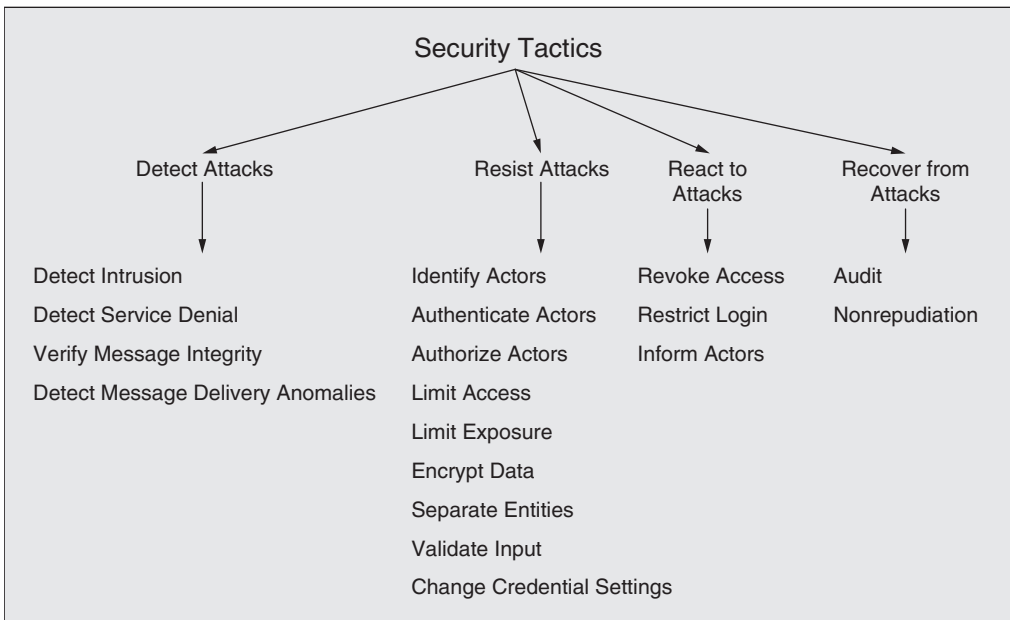
11.2 Tactics for Security

One method for thinking about how to achieve security in a system is to focus on physical security. Secure installations permit only limited access to them (e.g., by using fences and security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms (e.g., by having armed guards), have reaction mechanisms (e.g., automatic locking of doors), and have recovery mechanisms (e.g., off-site backup). These lead to our four categories of tactics: detect, resist, react, and recover. The goal of security tactics is shown in Figure 11.2, and Figure 11.3 outlines these categories of 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.* This tactic compares 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 characteristics, request characteristics, payload sizes, applications, source or destination address, or port number.
- *Detect service denial.* This tactic compares the pattern or signature of network traffic coming into a system to historical profiles of known denial-of-service (DoS) attacks.

**FIGURE 11.2** Goal of security tactics**FIGURE 11.3** Security tactics

- *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 separately maintains redundant information for files and messages, and uses this redundant information to verify the file or message. A hash value is a unique string generated by a hashing function, whose input could be files or messages. Even a slight change in the original files or messages results in a significant change in the hash value.
- *Detect message delivery anomalies.* This tactic seeks to detect potential man-in-the-middle-attacks, in which a malicious party is intercepting (and possibly modifying) messages. If message delivery times are normally stable, then by checking the time that it takes to deliver or receive a message, it becomes possible to detect suspicious timing behavior. Similarly, abnormal numbers of connections and disconnections may indicate such an attack.

Resist Attacks

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

- *Identify actors.* Identifying actors (users or remote computers) focuses on 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, or some other means.
- *Authenticate actors.* Authentication means ensuring that an actor is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, two-factor authentication, and biometric identification provide a means for authentication. Another example is CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart), a type of challenge–response test that is used to determine whether the user is human. Systems may require periodic reauthentication, such as when your smartphone automatically locks after a period of inactivity.
- *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 assigned per actor, per actor class, or per role.
- *Limit access.* This tactic involves limiting access to computer resources. Limiting access might mean restricting the number of access points to the resources, or restricting the type of traffic that can go through the access points. Both kinds of limits minimize the attack surface of a system. For example, a demilitarized zone (DMZ) is used when an organization wants to let external users access certain services but not access other services. The DMZ sits between the Internet and an intranet, and is protected by a pair of firewalls, one on either side. The internal firewall is a single point of access to the intranet; it functions as a limit to the number of access points as well as controls the type of traffic allowed through to the intranet.

- *Limit exposure.* This tactic focuses on minimizing the effects of damage caused by a hostile action. It is a passive defense since it does not proactively prevent attackers from doing harm. Limiting exposure is typically realized by reducing the amount of data or services that can be accessed through a single access point, and hence compromised in a single attack.
- *Encrypt data.* 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, by comparison, may not have authorization controls. In such cases, encryption is the only protection for passing data over publicly accessible communication links. Encryption can be symmetric (readers and writers use the same key) or asymmetric (with readers and writers use paired public and private keys).
- *Separate entities.* Separating different entities limits the scope of an attack. Separation within the system can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”—that is, by having no electronic connection between different portions of a system. Finally, sensitive data is frequently separated from nonsensitive data to reduce the possibility of attack by users who have access to nonsensitive data.
- *Validate input.* Cleaning and checking input as it is received by a system, or portion of a system, is an important early line of defense in resisting attacks. This is often implemented by using a security framework or validation class to perform actions such as filtering, canonicalization, and sanitization of input. Data validation is the main form of defense against attacks such as SQL injection, in which malicious code is inserted into SQL statements, and cross-site scripting (XSS), in which malicious code from a server runs on a client.
- *Change credential settings.* Many systems have default security 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 may be publicly available. Similarly, many systems require users to choose a new password after some maximum time period.

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 under way, 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.
- *Restrict login.* 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. Of course, legitimate users may make mistakes

in attempting to log in, so the limited access may last for only a certain time period. In some cases, systems double the lockout time period after each unsuccessful login attempt.

- *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 4) 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 for recovery, the audit and nonrepudiation tactics can be used:

- *Audit.* We audit systems—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.
- *Nonrepudiation.* This tactic 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, and the merchant cannot disclaim getting your order. This could be achieved with some combination of digital signatures and authentication by trusted third parties.

11.3 Tactics-Based Questionnaire for Security

Based on the tactics described in Section 11.2, we can create a set of security tactics—inspired questions, as presented in Table 11.2. To gain an overview of the architectural choices made to support security, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 11.2 Tactics-Based Questionnaire for Security

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detecting Attacks	Does the system support the detection of intrusions by, for example, comparing network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database?				
	Does the system support the detection of denial-of-service attacks by, for example, comparing the pattern or signature of network traffic coming into a system to historical profiles of known DoS attacks?				
	Does the system support the verification of message integrity via techniques such as checksums or hash values?				
	Does the system support the detection of message delays by, for example, checking the time that it takes to deliver a message?				
Resisting Attacks	Does the system support the identification of actors through user IDs, access codes, IP addresses, protocols, ports, etc.?				
	Does the system support the authentication of actors via, for example, passwords, digital certificates, two-factor authentication, or biometrics?				
	Does the system support the authorization of actors , ensuring that an authenticated actor has the rights to access and modify either data or services?				
	Does the system support limiting access to computer resources via restricting the number of access points to the resources, or restricting the type of traffic that can go through the access points?				
	Does the system support limiting exposure by reducing the amount of data or services that can be accessed through a single access point?				

continues

TABLE 11.2 Tactics-Based Questionnaire for Security *continued*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Resisting Attacks	<p>Does the system support data encryption, for data in transit or data at rest?</p> <p>Does the system design consider the separation of entities via physical separation on different servers attached to different networks, virtual machines, or an “air gap”?</p> <p>Does the system support changing credential settings, forcing the user to change those settings periodically or at critical events?</p> <p>Does the system validate input in a consistent, system-wide way—for example, using a security framework or validation class to perform actions such as filtering, canonicalization, and sanitization of external input?</p>				
Reacting to Attacks	<p>Does the system support revoking access by limiting access to sensitive resources, even for normally legitimate users and uses if an attack is under way?</p> <p>Does the system support restricting login in instances such as multiple failed login attempts?</p> <p>Does the system support informing actors such as operators, other personnel, or cooperating systems when the system has detected an attack?</p>				
Recovering from Attacks	<p>Does the system support maintaining an audit trail to help trace the actions of, and to identify, an attacker?</p> <p>Does the system guarantee the property of nonrepudiation, which guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message?</p> <p>Have you checked the “recover from faults” category of tactics from Chapter 4?</p>				

11.4 Patterns for Security

Two of the more well-known patterns for security are intercepting validator and intrusion prevention system.

Intercepting Validator

This pattern inserts a software element—a wrapper—between the source and the destination of messages. This approach assumes greater importance when the source of the messages is outside the system. The most common responsibility of this pattern is to implement the verify message integrity tactic, but it can also incorporate tactics such as detect intrusion and detect service denial (by comparing messages to known intrusion patterns), or detect message delivery anomalies.

Benefits:

- Depending on the specific validator that you create and deploy, this pattern can cover most of the waterfront of the “detect attack” category of tactics, all in one package.

Tradeoffs:

- As always, introducing an intermediary exacts a performance price.
- Intrusion patterns change and evolve over time, so this component must be kept up-to-date so that it maintains its effectiveness. This imposes a maintenance obligation on the organization responsible for the system.

Intrusion Prevention System

An intrusion prevention system (IPS) is a standalone element whose main purpose is to identify and analyze any suspicious activity. If the activity is deemed acceptable, it is allowed. Conversely, if it is suspicious, the activity is prevented and reported. These systems look for suspicious patterns of overall usage, not just anomalous messages.

Benefits:

- These systems can encompass most of the “detect attacks” and “react to attacks” tactics.

Tradeoffs:

- The patterns of activity that an IPS looks for change and evolve over time, so the patterns database must be constantly updated.
- Systems employing an IPS incur a performance cost.
- IPSs are available as commercial off-the-shelf components, which makes them unnecessary to develop but perhaps not entirely suited to a specific application.

Other notable security patterns include compartmentalization and distributed responsibility. Both of these combine the “limit access” and “limit exposure” tactics—the former with respect to information, the latter with respect to activities.

Just as we included (by reference) tactics for availability in our list of security tactics, patterns for availability also apply to security by counteracting attacks that seek to stop the system from operating. Consider the availability patterns discussed in Chapter 4 here as well.

11.5 For Further Reading

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

- Coding. *Secure Coding in C and C++* [Seacord 13] describes how to code securely.
- Organizational processes. Organizations must have processes that take responsibility for various aspects of security, including ensuring that systems are upgraded to put into place the latest protections. NIST 800-53 provides an enumeration of organizational processes [NIST 09]. Organizational processes must account for insider threats, which account for 15–20 percent of attacks. [Cappelli 12] discusses insider threats.
- Technical processes. Microsoft’s Security Development Lifecycle includes modeling of threats: microsoft.com/download/en/details.aspx?id=16420.

The Common Weakness Enumeration is a list of the most common categories of vulnerabilities discovered in systems, including SQL injection and XSS: <https://cwe.mitre.org/>.

NIST has published 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, or technical in nature.

Good books on engineering systems for security include Ross Anderson’s *Security Engineering: A Guide to Building Dependable Distributed Systems*, third edition [Anderson 20], and the series of books by Bruce Schneier.

Different domains have different sets of security practices that are relevant to their domain. The Payment Card Industry (PCI), for example, has established a set of standards intended for those involved in credit card processing (pcisecuritystandards.org).

The Wikipedia page on “Security Patterns” contains brief definitions of a large number of security patterns.

Access control is commonly performed using a standard called OAuth. You can read about OAuth at <https://en.wikipedia.org/wiki/OAuth>.

11.6 Discussion Questions

1. Write a set of concrete scenarios for security for an automobile. Consider in particular how you would specify scenarios regarding control of the vehicle.

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 described in this chapter.
3. 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. Nevertheless, 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 users. Discuss.
4. List some examples of critical resources for security, which a DoS attack might target and try to exhaust. Which architectural mechanisms could be employed to prevent this kind of attack?
5. Which of the tactics detailed in this chapter will protect against an insider threat? Can you think of any that should be added?
6. In the United States, Netflix typically accounts for more than 10 percent of all Internet traffic. How would you recognize a DoS attack on Netflix.com? Can you create a scenario to characterize this situation?
7. The public disclosure of vulnerabilities in an organization's production systems is a matter of controversy. Discuss why this is so, and identify the pros and cons of public disclosure of vulnerabilities. How could this issue affect your role as an architect?
8. Similarly, the public disclosure of an organization's security measures and the software to achieve them (via open source software, for example) is a matter of controversy. Discuss why this is so, identify the pros and cons of public disclosure of security measures, and describe how this could affect your role as an architect.

This page intentionally left blank

12



Testability

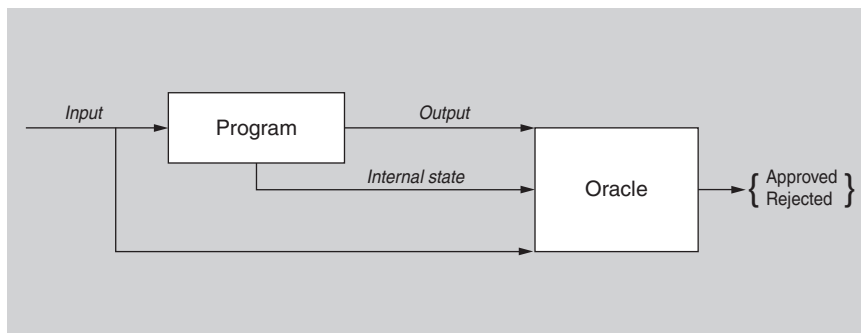
Testing leads to failure, and failure leads to understanding.
—Burt Rutan

A substantial portion of the cost of developing well-engineered systems is taken up by testing. If a carefully thought-out software architecture 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 “reveals” 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. In addition, an architecture can enhance testability by making it easier both to replicate a bug and to narrow down the possible root causes of the bug. We do not typically think of these activities as part of testability per se, but in the end just revealing a bug isn’t enough: You also need to find and fix the bug!

Figure 12.1 shows a simple model of testing in which a program processes input and produces output. An oracle is an agent (human or computational) that decides whether the output is correct by comparing the output to the expected result. Output is not just the functionally produced value, but can also include derived measures of quality attributes such as how long it took to produce the output. Figure 12.1 also indicates that the program’s internal state can be shown to the oracle, and an oracle can decide whether that state is correct—that is, it can detect whether the program has entered an erroneous state and render a judgment as to the correctness of the program. 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, control and observation are accomplished through the use of a *test harness*, a set of specialized software (or in some cases, hardware) designed to exercise the software under test. Test harnesses come in various forms, and may include capabilities such as a record-and-playback capability

**FIGURE 12.1** A model of testing

for data sent across interfaces, or a simulator for an external environment in which a piece of embedded software is tested, or even distinct software that runs during production (see the sidebar “Netflix’s Simian Army”). The test harness can provide assistance in executing the test procedures and recording the output. A test harness and its accompanying infrastructure can be substantial pieces of software in their own right, with their own architecture, stakeholders, and quality attribute requirements.

Netflix’s Simian Army

Netflix distributes movies and television shows via both DVD and streaming video. Its streaming video service has been extremely successful. In fact, in 2018, Netflix’s streaming video accounted for 15 percent of the global Internet traffic. Naturally, high availability is important to Netflix.

Netflix hosts its computer services in the Amazon EC2 cloud, and the company utilizes a set of services that were originally called the “Simian Army” as a portion of its testing process. Netflix began with a Chaos Monkey, which randomly killed processes in the running system. This allows the monitoring of the effect of failed processes and gives the ability to ensure that the system will not fail or suffer serious degradation as a result of a process failure.

The Chaos Monkey acquired some friends to assist in the testing. The Netflix Simian Army included these, in addition to the Chaos Monkey:

- The Latency Monkey induced artificial delays in network communication to simulate service degradation and measured whether upstream services responded appropriately.
- The Conformity Monkey identified instances that did not adhere to best practices and shut them down. For example, if an instance did not belong to an auto-scaling group, it would not appropriately scale when demand went up.

- The Doctor Monkey tapped into health checks that ran on each instance as well as monitoring other external signs of health (e.g., CPU load) to detect unhealthy instances.
- The Janitor Monkey ensured that the Netflix cloud environment was running free of clutter and waste. It searched for unused resources and disposed of them.
- The Security Monkey was an extension of Conformity Monkey. It found security violations or vulnerabilities, such as improperly configured security groups, and terminated the offending instances. It also ensured that all SSL and digital rights management (DRM) certificates were valid and not coming up for renewal.
- The 10-18 Monkey (localization-internationalization) detected configuration and run-time 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 members of the Simian Army used fault injection to place faults into the running system in a controlled and monitored fashion. Other members monitored various specialized aspects of the system and its environment. Both of these techniques have broader applicability than just for Netflix.

Given that 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 reflected a determination by Netflix that the targeted faults were the most serious in terms of their impacts.

Netflix’s strategy illustrates that some systems are too complex and adaptive to be tested fully, because some of their behaviors are emergent. One 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.

—LB

Testing is carried out by various developers, users, or quality assurance personnel. Either 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. In some cases, testing actually drives development, as is the case with test-driven development.

Testing of code is a special case of validation, which entails making sure that an engineered artifact meets its stakeholders’ needs or is suitable for use. In Chapter 21, we will discuss architectural design reviews—another kind of validation, in which the artifact being tested is the architecture.

12.1 Testability General Scenario

Table 12.1 enumerates the elements of the general scenario that characterize testability.

TABLE 12.1 Testability General Scenario

Portion of Scenario	Description	Possible Values
Source	The test cases can be executed by a human or an automated test tool.	<p>One or more of the following:</p> <ul style="list-style-type: none"> ▪ Unit testers ▪ Integration testers ▪ System testers ▪ Acceptance testers ▪ End users <p>Either run tests manually or use automated testing tools</p>
Stimulus	A test or set of tests is initiated.	<p>These tests serve to:</p> <ul style="list-style-type: none"> ▪ Validate system functions ▪ Validate qualities ▪ Discover emerging threats to quality
Environment	Testing occurs at various events or life-cycle milestones.	<p>The set of tests is executed due to:</p> <ul style="list-style-type: none"> ▪ 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 ▪ The deployment of the system into a production environment ▪ The delivery of the system to a customer ▪ A testing schedule
Artifacts	The artifact is the portion of the system being tested and any required test infrastructure.	<p>The portion being tested:</p> <ul style="list-style-type: none"> ▪ A unit of code (corresponding to a module in the architecture) ▪ Components ▪ Services ▪ Subsystems ▪ The entire system ▪ The test infrastructure
Response	The system and its test infrastructure can be controlled to perform the desired tests, and the results from the test can be observed.	<p>One or more of the following:</p> <ul style="list-style-type: none"> ▪ Execute test suite and capture results ▪ Capture activity that resulted in the fault ▪ Control and monitor the state of the system

Portion of Scenario	Description	Possible Values
Response measure	Response measures are aimed at representing how easily a system under test “gives up” its faults or defects.	<p>One or more of the following:</p> <ul style="list-style-type: none"> ▪ Effort to find a fault or class of faults ▪ Effort to achieve a given percentage of state space coverage ▪ Probability of a fault being revealed by the next test ▪ Time to perform tests ▪ Effort to detect faults ▪ Length of time to prepare test infrastructure ▪ Effort required to bring the system into a specific state ▪ Reduction in risk exposure: $\text{size}(\text{loss}) \times \text{probability}(\text{loss})$

Figure 12.2 shows a concrete scenario for testability: *The developer completes a code unit during development and performs a test sequence whose results are captured and that gives 85 percent path coverage within 30 minutes.*

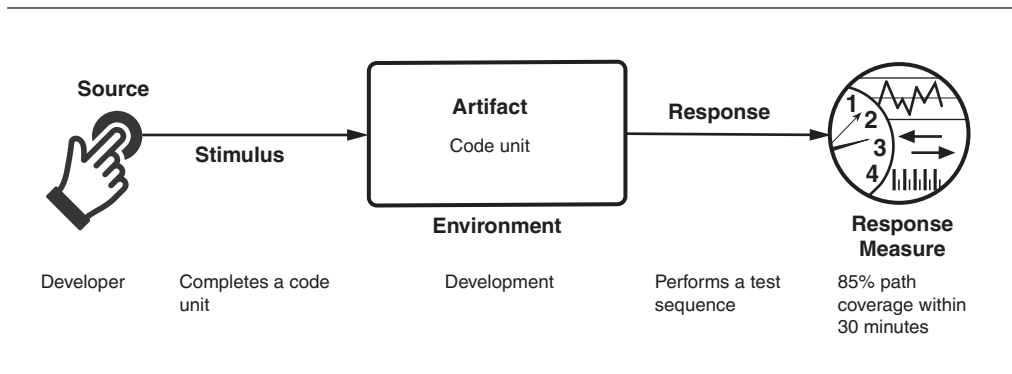


FIGURE 12.2 Sample testability scenario

12.2 Tactics for Testability

Tactics for testability are intended to promote easier, more efficient, and more capable testing. Figure 12.3 illustrates the goal of the testability tactics. Architectural techniques for enhancing the software testability have not received as much attention as other quality attribute disciplines such as modifiability, performance, and availability, but as we stated earlier, anything the architect can do to reduce the high cost of testing will yield a significant benefit.

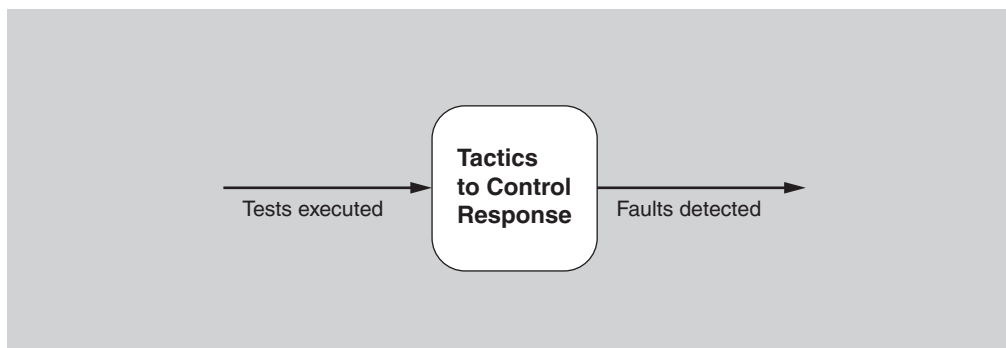


FIGURE 12.3 The goal of testability tactics

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 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 category of testability tactics provides insights into software that go 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 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 the application of a test harness or through normal execution. Examples of specialized test routines, some of which might otherwise not be available except for testing purposes, include these:
 - A *set* and *get* method for important variables, modes, or attributes
 - A *report* method that returns the full state of the object
 - A *reset* method to set the internal state (e.g., 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. Note, however, that 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 released has the same behavior, particularly the same timing behavior, as the code you tested? Thus this strategy is more effective for other kinds of systems.

- *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* refers to capturing information crossing an interface and *playback* refers to using it as input for further testing.
- *Localize state storage.* To start a system, subsystem, or component 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 approach 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 (i.e., to make it amenable to manipulation 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 the case when controlling a program’s state, the ability to control 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 you can readily 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 any worries about having to undo the consequences of the experiment. Testing is facilitated 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. The sandbox tactic can be used for scenario analysis, training, and simulation. Simulation, in particular, is a commonly employed strategy for testing and training in contexts where failure in the real world might lead to severe consequences.

One common form of sandboxing is to virtualize resources. Testing a system often involves interacting with resources whose behavior is outside the system’s control. 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. Thus, 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. When we can abstract system time from clock time, we can allow the system (or components) to run at faster than wall-clock time, and test the system (or components) at critical time boundaries such as the next transition to or from Daylight Savings Time. Similar virtualizations could be done for other resources, such as the memory, battery, network, and so on. Stubs, mocks, and dependency injection are simple but effective forms of virtualization.

- *Executable assertions.* With 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 increases the system's observability, as an assertion can be 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. Anytime an object of that type is modified, the checking code executes automatically, with warnings being 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 some capability or abstraction to the software that (were we not interested in testing) otherwise would not be there. They can be seen as augmenting bare-bones, get-the-job-done software with more elaborate software that has some special capabilities designed to enhance the efficiency and effectiveness of testing.

In addition to the testability tactics, a number of techniques are available for replacing one component with a different version of itself that facilitates testing:

- *Component replacement* 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*, when activated, can 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) can handle the cross-cutting concern of how the state is reported.

Limit Complexity

Complex software is much harder to test. Its operating state space is 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 also about finding the fault that caused the failure so that it can be removed, we are often concerned with making behavior repeatable. This category includes two 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 (typically realized by lowering the coupling between components). For example, 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 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. In addition, architecture-level coupling metrics, such as propagation cost and decoupling level, can be used to measure and track the overall level of coupling in a system's architecture.

Ensuring that the system has high cohesion, loose coupling, and separation of concerns—all modifiability tactics (see Chapter 8)—can also help with testability. These characteristics limit the complexity of the architectural elements by giving each element a focused task such that it has limited interactions with other elements. Separation of concerns can help achieve controllability and observability, as well as reduce the size of the overall program's state space.

Finally, some architectural patterns lend themselves to testability. In a layered pattern, 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. When it comes to testing, nondeterminism is a pernicious form of complex behavior, and nondeterministic systems are more difficult to test than deterministic systems. This tactic involves finding all the sources of nondeterminism, such as unconstrained parallelism, and weeding them out to the extent 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 to help manage this complexity.

Figure 12.4 summarizes the tactics used for testability.

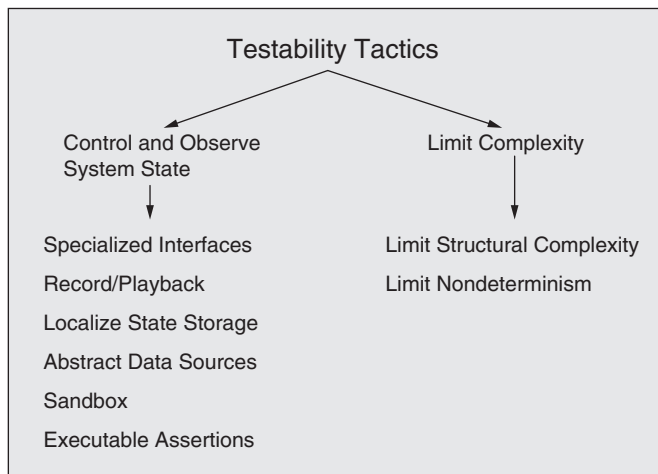


FIGURE 12.4 Testability tactics

12.3 Tactics-Based Questionnaire for Testability

Based on the tactics described in Section 12.2, we can create a set of tactics-inspired questions, as presented in Table 12.2. To gain an overview of the architectural choices made to support testability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 12.2 Tactics-Based Questionnaire for Testability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Control and Observe System State	Does your system have specialized interfaces for getting and setting values?				
	Does your system have a record/playback mechanism?				
	Is your system's state storage localized ?				
	Does your system abstract its data sources ?				
	Can some or all of your system operate in a sandbox ?				
Limit Complexity	Is there a role for executable assertions in your system?				
	Does your system limit structural complexity in a systematic way?				
	Is there nondeterminism in your system, and is there a way to control or limit this nondeterminism ?				

12.4 Patterns for Testability

Patterns for testability all make it easier to decouple test-specific code from the actual functionality of a system. We discuss three patterns here: dependency injection, strategy, and intercepting filter.

Dependency Injection Pattern

In the dependency injection pattern, a client's dependencies are separated from its behavior. This pattern makes use of inversion of control. Unlike in traditional declarative programming, where control and dependencies reside explicitly in the code, inversion of control dependencies means that control and dependencies are provided from, and injected into the code, by some external source.

In this pattern, there are four roles:

- A service (that you want to make broadly available)
- A client of the service
- An interface (used by the client, implemented by the service)
- An injector (that creates an instance of the service and injects it into the client)

When an interface creates the service and injects it into the client, a client is written with no knowledge of a concrete implementation. In other words, all of the implementation specifics are injected, typically at runtime.

Benefits:

- Test instances can be injected (rather than production instances), and these test instances can manage and monitor the state of the service. Thus the client can be written with no knowledge of how it is to be tested. This is, in fact, how many modern testing frameworks are implemented.

Tradeoffs:

- Dependency injection makes runtime performance less predictable, because it might change the behavior being tested.
- Adding this pattern adds a small amount of up-front complexity and may require retraining of developers to think in terms of inversion of control.

Strategy Pattern

In the strategy pattern, a class's behavior can be changed at runtime. This pattern is often employed when multiple algorithms can be employed to perform a given task, and the specific algorithm to be used can be chosen dynamically. The class simply contains an abstract method for the desired functionality, with the concrete version of this method being selected based on contextual factors. This pattern is often used to replace non-test versions of some functionality with test versions that provide additional outputs, additional internal sanity checking, and so forth.

Benefits:

- This pattern makes classes simpler, by not combining multiple concerns (such as different algorithms for the same function) into a single class.

Tradeoffs:

- The strategy pattern, like all design patterns, adds a small amount of up-front complexity. If the class is simple or if there are few runtime choices, this added complexity is likely wasted.
- For small classes, the strategy pattern can make code slightly less readable. However, as complexity grows, breaking up the class in this way can enhance readability.

Intercepting Filter Pattern

The intercepting filter pattern is used to inject pre- and post-processing to a request or a response between a client and a service. Any number of filters can be defined and applied, in an arbitrary order, to the request before passing the request to the eventual service. For example, logging and authentication services are filters that are often useful to implement once and apply universally. Testing filters can be inserted in this way, without disturbing any of the other processing in the system.

Benefits:

- This pattern, like the strategy pattern, makes classes simpler, by not placing all of the pre- and post-processing logic in the class.
- Using an intercepting filter can be a strong motivator for reuse and can dramatically reduce the size of the code base.

Tradeoffs:

- If a large amount of data is being passed to the service, this pattern can be highly inefficient and can add a nontrivial amount of latency, as each filter makes a complete pass over the entire input.

12.5 For Further Reading

The literature on software testing would sink a battleship, but the writing about how to make your system more testable from an architectural standpoint is less voluminous. For a good overview of testing, see [Binder 00]. Jeff Voas's foundational work on testability and the relationship between testability and reliability is worth investigating, too. There are several papers to choose from, but [Voas 95] is a good start that will point you to others.

Bertolino and Strigini [Bertolino 96a, 96b] are the developers of the model of testing shown in Figure 12.1.

“Uncle Bob” Martin has written extensively on test-driven development and the relationship between architecture and testing. The best book on this is Robert C. Martin’s *Clean Architecture: A Craftsman’s Guide to Software Structure and Design* [Martin 17]. An early and authoritative reference for test-driven development was written by Kent Beck: *Test-Driven Development by Example* [Beck 02].

The propagation cost coupling metric was first described in [MacCormack 06]. The decoupling level metric was described in [Mo 16].

Model checking is a technique that symbolically executes all possible code paths. The size of a system that can be validated using model checking is limited, but device drivers and microkernels have successfully been model checked. See https://en.wikipedia.org/wiki/Model_checking for a list of model checking tools.

12.6 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. In contrast, 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. 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?
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 a GPS-based navigation app. What tactics would you employ in a design to respond to these scenarios?
5. One of our tactics is to limit nondeterminism, and one method is to use locking to enforce synchronization. What impact does the use of locks have on other quality attributes?
6. 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, 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 network system.
7. 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.

This page intentionally left blank

13



Usability

People ignore design that ignores people.
—Frank Chimero

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support that 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) and hence end-user satisfaction.

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 enabling 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 user errors.* What can the system do to ensure that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly or undo its effects.
- *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, along with the completion percentage so far, will increase the user's confidence in the system.

Researchers focusing on human–computer interactions 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 display a progress indicator (system initiative). Thus the cancel operation may comprise a mixed initiative. In this chapter, we will use this distinction between user initiative and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

There is a strong connection between the achievement of usability and modifiability. The user interface design process consists of generating and then testing a user interface design. It is highly unlikely that you will get this right the first time, so you should plan to iterate this process—and hence you should design your architecture to make that iteration less painful. This is why usability is strongly connected to modifiability. As you iterate, deficiencies in the design are—one hopes—corrected and the process repeats.

This connection has resulted in standard patterns to support user interface design. Indeed, one of the most helpful things you can do to achieve usability is to modify your system, over and over, to make it better as you learn from your users and discover improvements to be made.

13.1 Usability General Scenario

Table 13.1 enumerates the elements of the general scenario that characterize usability.

TABLE 13.1 Usability General Scenario

Portion of Scenario	Description	Possible Values
Source	Where does the stimulus come from?	The end user (who may be in a specialized role, such as a system or network administrator) is the primary source of the stimulus for usability. An external event arriving at a system (to which the user may react) may also be a stimulus source.
Stimulus	What does the end user want?	End user wants to: <ul style="list-style-type: none"> ▪ Use a system efficiently ▪ Learn to use the system ▪ Minimize the impact of errors ▪ Adapt the system ▪ Configure the system
Environment	When does the stimulus reach the system?	The user actions with which usability is concerned always occur at runtime or at system configuration time.
Artifacts	What portion of the system is being stimulated?	Common examples include: <ul style="list-style-type: none"> ▪ A GUI ▪ A command-line interface ▪ A voice interface ▪ A touch screen

Portion of Scenario	Description	Possible Values
Response	How should the system respond?	<p>The system should:</p> <ul style="list-style-type: none"> ▪ Provide the user with the features needed ▪ Anticipate the user's needs ▪ Provide appropriate feedback to the user
Response measure	How is the response measured?	<p>One or more of the following:</p> <ul style="list-style-type: none"> ▪ Task time ▪ Number of errors ▪ Learning time ▪ Ratio of learning time to task time ▪ Number of tasks accomplished ▪ User satisfaction ▪ Gain of user knowledge ▪ Ratio of successful operations to total operations ▪ Amount of time or data lost when an error occurs

Figure 13.1 gives an example of a concrete usability scenario that you could generate using Table 13.1: *The user downloads a new application and is using it productively after 2 minutes of experimentation.*

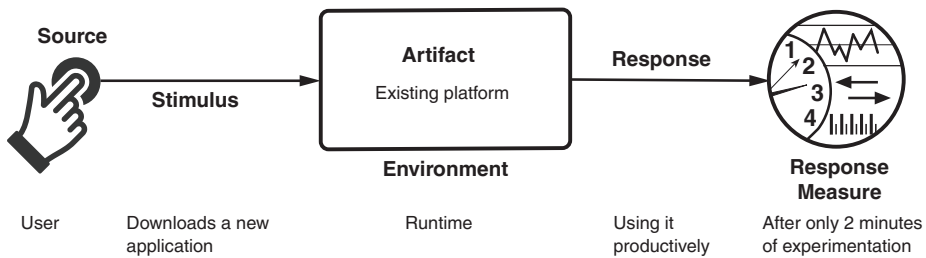


FIGURE 13.1 Sample usability scenario

13.2 Tactics for Usability

Figure 13.2 shows the goal of the set of usability tactics.

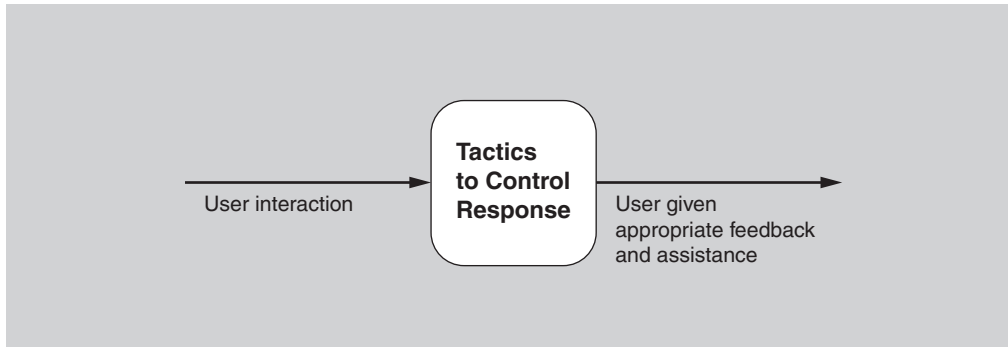


FIGURE 13.2 The goal of usability tactics

Support User Initiative

Once a system is executing, usability is enhanced by giving the user feedback about 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 tactics to support 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 activity being canceled must be terminated; any resources being used by the canceled activity must be freed; and components that are collaborating with the canceled activity 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 take the form of state "snapshots"—for example, checkpoints—or 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 cannot be undone at all: You can't unship a package or unfire a missile, for example.

Undo comes in flavors. Some systems allow a single undo (where invoking undo again reverts you to the state in which you commanded the first undo, essentially undoing the undo). In other systems, commanding multiple undo operations steps you back through many previous states, either up to some limit or all the way back to the time when the application was last opened.

- *Pause/resume.* When a user has initiated a long-running operation—say, downloading a large file or a set of files from a server—it is often useful to provide the ability to pause and resume the operation. Pausing a long-running operation may be done 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. An example is aggregating all of the objects in a slide and changing the text to 14-point font.

Support System Initiative

When the system takes the initiative, it must rely on a model of the user, a model of the task being undertaken by the user, or a model of the system state. Each model requires various types of input to accomplish its initiative. The support system initiative tactics identify the models the system uses to predict either its own behavior or the user's intention. Encapsulating this information will make it easier to tailor or modify it. Tailoring and modification can either be dynamically based on past user behavior or happen offline during development. The relevant tactics are described here:

- *Maintain task model.* The task model is used to determine context so the system can have some idea of what the user is attempting to do and provide assistance. For example, many search engines provide predictive type-ahead capabilities, and many mail clients provide spell-correction. Both of these functions are based on task models.
- *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, language-learning apps are constantly monitoring areas where a user makes mistakes and then providing additional exercises to correct those behaviors. 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.* 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 13.3 summarizes the tactics to achieve usability.

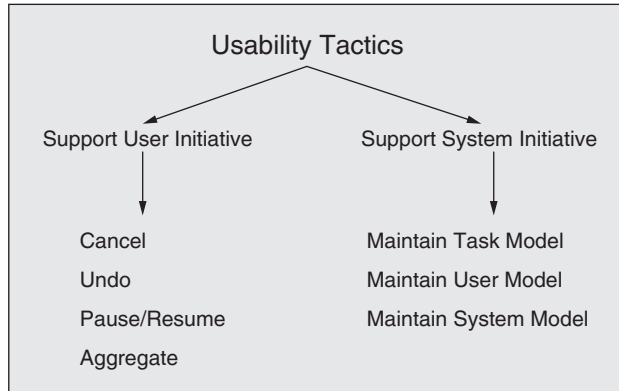


FIGURE 13.3 Usability tactics

13.3 Tactics-Based Questionnaire for Usability

Based on the tactics described in Section 13.2, we can create a set of usability tactics–inspired questions, as presented in Table 13.2. To gain an overview of the architectural choices made to support usability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

TABLE 13.2 Tactics-Based Questionnaire for Usability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Support User Initiative	Is the system able to listen to and respond to a cancel command?				
	Is it possible to undo the last command, or the last several commands?				
	Is it possible to pause and then resume long-running operations?				
	Is it possible to aggregate UI objects into a group and apply operations on the group?				

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Support System Initiative	Does the system maintain a model of the task ?				
	Does the system maintain a model of the user ?				
	Does the system maintain a model of itself ?				

13.4 Patterns for Usability

We will briefly discuss three usability patterns: model-view-controller (MVC) and its variants, observer, and memento. These patterns primarily promote usability by promoting separation of concerns, which in turn makes it easy to iterate the design of a user interface. Other kinds of patterns are also possible—including patterns used in the design of the user interface itself, such as breadcrumbs, shopping cart, or progressive disclosure—but we will not discuss them here.

Model-View-Controller

MVC is likely the most widely known pattern for usability. It comes in many variants, such as MVP (model-view-presenter), MVVM (model-view-view-model), MVA (model-view-adapter), and so forth. Essentially all of these patterns are focused on separating the model—the underlying “business” logic of the system—from its realization in one or more UI views. In the original MVC model, the model would send updates to a view, which a user would see and interact with. User interactions—key presses, button clicks, mouse motions, and so forth—are transmitted to the controller, which interprets them as operations on the model and then sends those operations to the model, which changes its state in response. The reverse path was also a portion of the original MVC pattern. That is, the model might be changed and the controller would send updates to the view.

The sending of updates depends on whether the MVC is in one process or is distributed across processes (and potentially across the network). If the MVC is in one process, then the updates are sent using the observer pattern (discussed in the next subsection). If the MVC is distributed across processes, then the publish-subscribe pattern is often used to send updates (see Chapter 8).

Benefits:

- Because MVC promotes clear separation of concerns, changes to one aspect of the system, such as the layout of the UI (the view), often have no consequences for the model or the controller.

- Additionally, because MVC promotes separation of concerns, developers can be working on all aspects of the pattern—model, view, and controller—relatively independently and in parallel. These separate aspects can also be tested in parallel.
- A model can be used in systems with different views, or a view might be used in systems with different models.

Tradeoffs:

- MVC can become burdensome for complex UIs, as information is often sprinkled throughout several components. For example, if there are multiple views of the same model, a change to the model may require changes to several otherwise unrelated components.
- For simple UIs, MVC adds up-front complexity that may not pay off in downstream savings.
- MVC adds a small amount of latency to user interactions. While this is generally acceptable, it might be problematic for applications that require very low latency.

Observer

The observer pattern is a way to link some functionality with one or more views. This pattern has a *subject*—the entity being observed—and one or more *observers* of that subject. Observers need to register themselves with the subject; then, when the state of the subject changes, the observers are notified. This pattern is often used to implement MVC (and its variants)—for example, as a way to notify the various views of changes to the model.

Benefits:

- This pattern separates some underlying functionality from the concern of how, and how many times, this functionality is presented.
- The observer pattern makes it easy to change the bindings between the subject and the observers at runtime.

Tradeoffs:

- The observer pattern is overkill if multiple views of the subject are not required.
- The observer pattern requires that all observers register and de-register with the subject. If observers neglect to de-register, then their memory is never freed, which effectively results in a memory leak. In addition, this can negatively affect performance, since obsolete observers will continue to be invoked.
- Observers may need to do considerable work to determine if and how to reflect a state update, and this work may be repeated for each observer. For example, suppose the subject is changing its state at a fine granularity, such as a temperature sensor that reports 1/100th degree fluctuations, but the view updates changes only in full degrees. In such cases where there is an “impedance mismatch,” substantial processing resources may be wasted.

Memento

The memento pattern is a common way to implement the undo tactic. This pattern features three major components: the *originator*, the *caretaker*, and the *memento*. The originator is processing some stream of events that change its state (originating from user interaction). The caretaker is sending events to the originator that cause it to change its state. When the caretaker is about to change the state of the originator, it can request a memento—a snapshot of the existing state—and can use this artifact to restore that existing state if needed, by simply passing the memento back to the originator. In this way, the caretaker knows nothing about how state is managed; the memento is simply an abstraction that the caretaker employs.

Benefits:

- The obvious benefit of this pattern is that you delegate the complicated process of implementing undo, and figuring out what state to preserve, to the class that is actually creating and managing that state. In consequence, the originator's abstraction is preserved and the rest of the system does not need to know the details.

Tradeoffs:

- Depending on the nature of the state being preserved, the memento can consume arbitrarily large amounts of memory, which can affect performance. In a very large document, try cutting and pasting many large sections, and then undoing all of that. This is likely to result in your text processor noticeably slowing down.
- In some programming languages, it is difficult to enforce the memento as an opaque abstraction.

13.5 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 of the ROI of usability [Nielsen 08].

Bonnie John and Len Bass have investigated the relation between usability and software architecture. They have enumerated approximately two dozen usability scenarios that have architectural impact and given associated patterns for these scenarios [Bass 03].

Greg Hartman has defined attentiveness as the system's ability to support user initiative and allow cancel or pause/resume [Hartman 10].

13.6 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 (Table 13.1).

2. How might usability trade off against security? How might it trade off against performance?
3. Pick a few of your favorite websites 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 “anticipate the user’s need”) and an appropriate corresponding response measure. Using the response and response measure you chose, compare the websites’ usability.
4. Why is it that in so many systems, the cancel button in a dialog box appears to be unresponsive? Which architectural principles do you think were ignored in these systems?
5. 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?
6. Research the crash of Air France flight 296 into the forest at Habsheim, France, in 1988. The pilots said they were unable to read the digital display of the radio altimeter or hear its audible readout. In this context, discuss the relationship between usability and safety.

14



Working with Other Quality Attributes

Quality is not what happens when what you do matches your intentions. It is what happens when what you do matches your customers' expectations.
—Guaspari

Chapters 4–13 each dealt with a particular quality attribute (QA) that is important to software systems. Each of those chapters discussed how its particular QA is defined, gave a general scenario for that QA, and showed how to write specific scenarios to express precise shades of meaning concerning that QA. In addition, each provided a collection of techniques to achieve that QA in an architecture. In short, each chapter presented a kind of portfolio for specifying and designing to achieve a particular QA.

However, as you can no doubt infer, those ten chapters only begin to scratch the surface of the various QAs that you might need in a software system you're working on.

This chapter will show how to build the same kind of specification and design approach for a QA not covered in our “A list.”

14.1 Other Kinds of Quality Attributes

The quality attributes covered so far in Part II of this book all have something in common: They deal with either the system in operation, or the development project that creates and fields the system. Put another way, to measure one of those QAs, either you measure the system while it is running (availability, energy efficiency, performance, security, safety, usability), or you measure the people doing something to the system while it is not (modifiability, deployability, integrability, testability). While these certainly give you an “A list” of important QAs, there are other qualities that could be equally useful.

Quality Attributes of the Architecture

Another category of QAs focuses on measuring the architecture itself. Here are three examples:

- *Buildability*. This QA measures how well the architecture lends itself to rapid and efficient development. It is measured by the cost (typically in money or time) that it takes to turn the architecture into a working product that meets all of its requirements. In that sense, it resembles the other QAs that measure a development project, but it differs in that the knowledge targeted by the measurement relates to the architecture itself.
- *Conceptual integrity*. Conceptual integrity refers to consistency in the design of the architecture, and it contributes to the architecture’s understandability and leads to less confusion and more predictability in its implementation and maintenance. 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 a small number of ways, and provide alternatives only if there is 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, sanitize data in the same way, and so forth.
- *Marketability*. An architecture’s “marketability” is another QA of concern. Some systems are well known for their architectures, and these architectures sometimes carry a meaning all their own, independent of what other QAs they bring to the system. The current emphasis on building cloud-based and micro-service-based systems has taught us that the perception of an architecture can be at least as important as the actual qualities that the architecture brings. Many organizations, for example, have felt compelled to build cloud-based systems (or some other *technologie du jour*) whether or not that was the correct technical choice.

Development Distributability

Development distributability is the quality of designing the software to support distributed software development. Like modifiability, this quality is measured in terms of the activities of a development project. Many systems these days are developed using globally distributed teams. One problem that must be overcome when adopting this approach is coordinating the teams’ activities. The system should be designed so that coordination among teams is minimized—that is, the major subsystems should exhibit low coupling. 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. Thus the architectural structure and the social (and business) structure of the project need to be reasonably aligned. 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 utilized by the organizations doing the development.

System Quality Attributes

Physical systems, such as aircraft and automobiles and kitchen appliances, that rely on software embedded within them are designed to meet a whole litany of QAs: weight, size, electric consumption, power output, pollution output, weather resistance, battery life, and on and on. Often the software architecture can have a profound effect on the system's QAs. 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 (we dealt with the topic of energy efficiency as a QA in Chapter 6). Additional processors will add to a system's power consumption, of course, but also to its weight, its physical profile, and expense.

Conversely, the architecture or implementation of a system can enable or preclude software from meeting its QA requirements. For example:

1. The performance of a piece of software is fundamentally constrained by the performance of the processor that runs it. No matter how well you design the software, you just can't run the latest whole-earth weather forecasting models on Grandpa's laptop and expect to know if it's going to rain tomorrow.
2. 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, please don't do that. Consider this a thought experiment.)

The lesson here is that if you are the architect for software that resides in a physical system, you will need to understand the QAs that are important for the entire system to achieve, and work with the *system* architects and engineers to ensure that your software architecture contributes positively to achieving them.

The scenario techniques we introduced for software QAs work equally well for system QAs. If the system engineers and architects aren't already using them, try to introduce them.

14.2 Using Standard Lists of Quality Attributes—Or Not

Architects have no shortage of QA lists 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 (Figure 14.1). This standard divides QAs into those supporting a “quality in use” model and those supporting a “product quality”

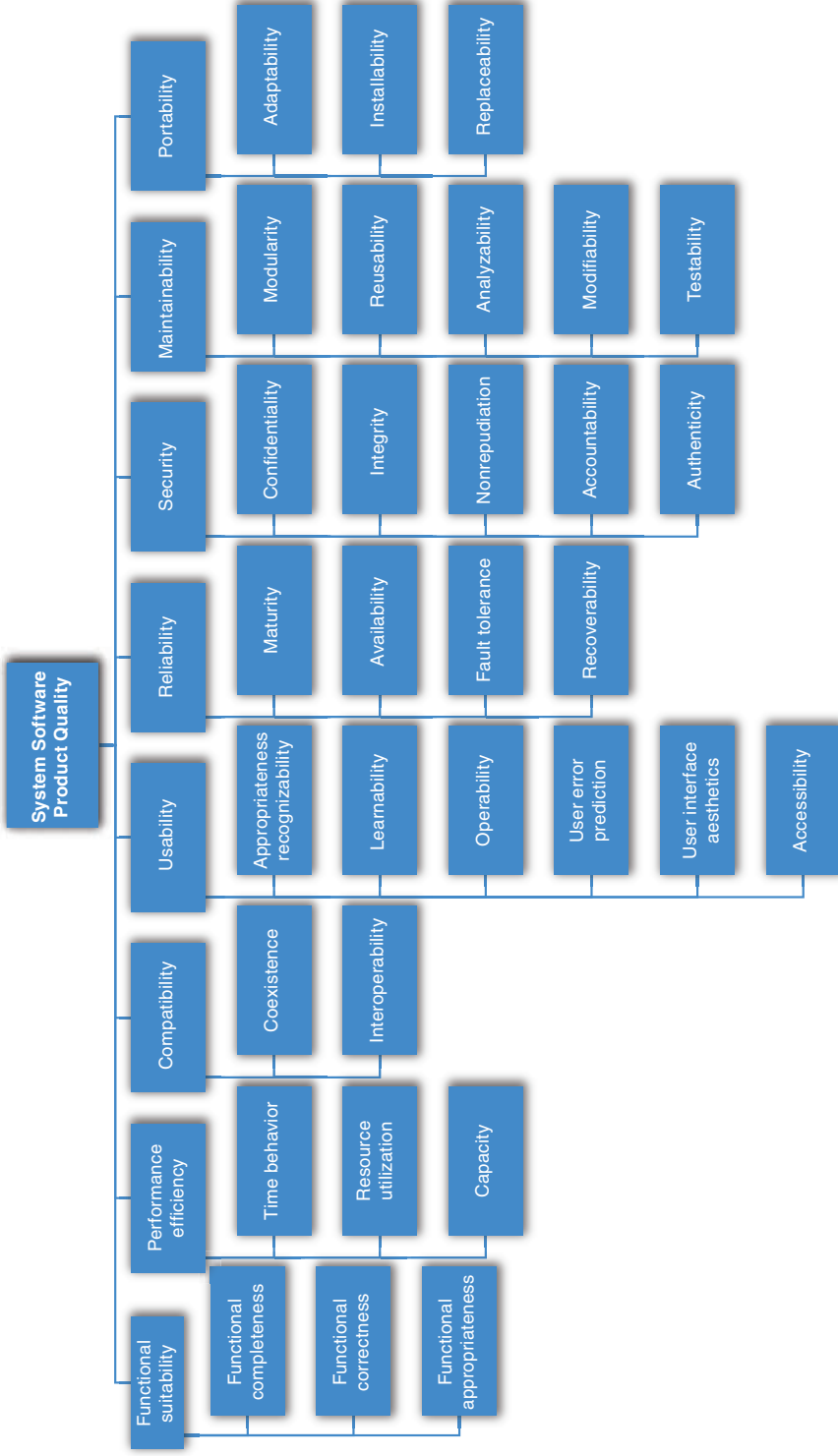


FIGURE 14.1 ISO/IEC FCD 25010 Product Quality Standard

model. That division is a bit of a stretch in some places, but it nevertheless begins a divide-and-conquer march through a breathtaking array of qualities.

ISO 25010 lists the following QAs that deal with product quality:

- *Functional suitability.* Degree to which a product or system provides functions that meet the stated and implied needs when used under the specified conditions.
- *Performance efficiency.* Performance relative to the amount of resources used under the stated conditions.
- *Compatibility.* 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.* 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.* Degree to which a system, product, or component performs the specified functions under the specified conditions for a specified period of time.
- *Security.* 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.* Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
- *Portability.* 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 sub-characteristics” (for example, nonrepudiation is a sub-characteristic of security). The standard slogs through almost five dozen separate descriptions of quality sub-characteristics in this way. It defines for us the qualities of “pleasure and “comfort.” It distinguishes between “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 of them floating around—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 QAs of concern in your domain, your industry, your organization, your products. QA lists can also serve as the basis for establishing measures, though the names themselves give little clue as to how to do this. If “fun” 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?

General lists like these also have some drawbacks. First, no list will ever be complete. As an architect, you will inevitably 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 operations 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 QAs, 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 sub-characteristic of functionality, which it was in a previous version. We strongly believe that effort in making these arguments could be better spent elsewhere.

Third, these lists often purport to be *taxonomies*—that is, lists with the special property that every member can be assigned to exactly one place. But QAs are notoriously squishy in this regard. For example, we discussed denial of service as being part of security, availability, performance, and usability in Chapter 3.

These observations reinforce the lesson introduced in Chapter 3: QA names, by themselves, are largely useless and are at best invitations to begin a conversation. Moreover, spending time worrying about which qualities are subqualities of which other qualities is almost useless. Instead, scenarios provide the best way for us to specify precisely what we mean when we speak of a QA.

Use standard lists of QAs to the extent that they are helpful as checklists, but don’t feel the need to slavishly adhere to their terminology or structure. And don’t fool yourself that such a checklist removes the need for deeper analysis.

14.3 Dealing with “X-Ability”: Bringing a New QA into the Fold

Suppose, as an architect, you had to deal with a QA for which there is no compact body of knowledge, no “portfolio” like Chapters 4–13 provided for those QAs. Suppose you find yourself having to deal with a QA like “development distributability” or “manageability” or even “Iowability”? What do you do?

Capture Scenarios for the New Quality Attribute

The first step is to interview the stakeholders whose concerns have led to the need for this QA. 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, you might decompose development distributability into the subattributes of software segmentation, software composition, and team coordination. After that refinement, you can work with the stakeholders to craft a set of specific scenarios that characterize what is meant by that QA. An example of this process can be found in Chapter 22, where we describe building a “utility tree.”

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.

Model the Quality Attribute

If you can build (or even better, find) a conceptual model of the QA, that foundation can be helpful in creating a set of design approaches for it. By “model,” we don’t mean anything more than an understanding of the set of parameters to which the QA is sensitive and the set of architectural characteristics that influence those parameters. 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.

Figure 14.2 shows a simple 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.

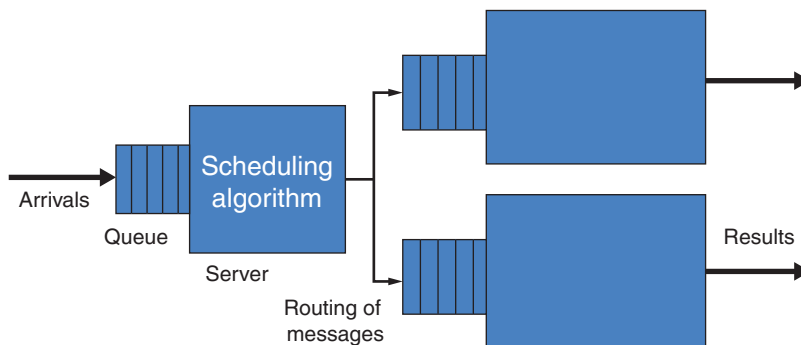


FIGURE 14.2 A generic queuing model

Within this model, seven parameters 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.

If you are creating your own model, your set of scenarios will inform your investigation. Its parameters can be derived from the stimuli (and its sources), the responses (and their measures), the artifacts (and their properties), and the environment (and its characteristics).

Assemble Design Approaches for the New Quality Attribute

The process of generating a set of mechanisms based on a model includes the following steps:

- Enumerate the model's parameters.
- For each parameter, enumerate the architectural characteristics (and the mechanisms to achieve those characteristics) that can affect this parameter. You can do this by:
 - Revisiting a body of mechanisms you're familiar with and asking yourself how each one affects the QA parameter.
 - Searching for designs that have successfully dealt 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 subattributes.
 - Searching for publications and blog posts on this QA and attempting to generalize their observations and findings.
 - Finding experts in this area and interviewing them or simply writing and asking them for advice.

What results is a list of mechanisms to, in the example case, control performance and, in the more general case, to control the QA that the model is concerned with. This makes the design problem much more tractable. This list of mechanisms 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.

14.4 For Further Reading

The mother of all QA lists may be the one on—where else?—Wikipedia. This list can be found, naturally enough, under “List of system quality attributes.” As this book went to publication, you could gorge yourself on definitions of more than 80 distinct QAs. Our favorite is “demonstrability,” which is helpfully defined as the quality of being demonstrable. Who says you can’t believe what you read on the Internet?

See Chapter 8 of [Bass 19] to get a list of qualities of a deployment pipeline. These include traceability, testability (of the deployment pipeline), tooling, and cycle time.

14.5 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). Read about how the GNH is measured (try grossnationalhappiness.com) and then sketch a general scenario for the QA of *happiness* that will let you express concrete happiness requirements for a software system.
2. Choose a QA not described in Chapters 4–13. For that QA, assemble a set of specific scenarios that describe what you mean by it. Use that set of scenarios to construct a general scenario for it.
3. For the QA you chose for question 2, assemble a set of design mechanisms (patterns and tactics) that help you achieve it.
4. Repeat questions 2 and 3 for the QA of *development cost*, and then for the QA of *operating cost*.
5. What might cause you to add a tactic or pattern to the sets of QAs already described in Chapters 4–13 (or any other QA, for that matter)?
6. Discuss how you think development distributability tends to trade off against the QAs of performance, availability, modifiability, and integrability.
7. Research some QA lists for things that are not software systems: qualities of a good car, for example, or a good person to be in a relationship with. Add qualities of your own choosing to the list or lists that you find.
8. Development-time tactics have to do with separating and encapsulating responsibilities. Performance tactics have to do with putting things together. That is why they are perpetually in conflict. Must it always be so? Is there a principled way of quantifying the tradeoffs?

9. Is there a taxonomy of tactics? Chemists have the periodic table and laws of molecular interaction, atomic physicists have their catalogs of subatomic particles and laws for what happens when they collide, pharmacologists have their catalogs of chemicals and laws for their interactions with receptors and metabolic systems, and so forth. What is the equivalent for tactics? And are there laws for their interaction?
10. Security is a QA that is especially sensitive to processes that take place in the physical world outside the computer: processes for applying patches, processes for choosing and safeguarding your passwords, processes for physically securing the installations where computers and data live, processes for deciding whether to trust a piece of imported software, processes for deciding whether to trust a human developer or user, and so forth. What are the corresponding processes that are important for performance? Or usability? Are there any? Why is security so process-sensitive? Should processes be a portion of the QA structure or are they orthogonal to it?
11. What is the relationship between each pair of QAs in the following list?
 - Performance and security
 - Security and buildability
 - Energy efficiency and time to market

15



Software Interfaces

With Cesare Pautasso

NASA lost its \$125-million Mars Climate Orbiter because spacecraft engineers failed to convert from English to metric measurements when exchanging vital data before the craft was launched. . . .

A navigation team at [NASA] used the metric system of millimeters and meters in its calculations, while [the company that] designed and built the spacecraft provided crucial acceleration data in the English system of inches, feet and pounds. . . .

In a sense, the spacecraft was lost in translation.

—Robert Lee Hotz, “Mars Probe Lost Due to Simple Math Error,” *Los Angeles Times*, October 1, 1999

This chapter describes the concepts surrounding interfaces, and discusses how to design and document them.

An interface, software or otherwise, is a boundary across which elements meet and interact, communicate, and coordinate. Elements have interfaces that control access to their internals. Elements may also be subdivided, with each sub-element having its own interface.

An element’s *actors* are the other elements, users, or systems with which it interacts. The collection of actors with which an element interacts is called the *environment* of the element. By “interacts,” we mean anything one element does that can impact the processing of another element. This interaction is part of the element’s interface. Interactions can take a variety of forms, though most involve the transfer of control and/or data. Some are supported by standard programming-language constructs, such as local or remote procedure calls (RPCs), data streams, shared memory, and message passing.

These constructs, which provide points of direct interaction with an element, are called *resources*. Other interactions are indirect. For example, the fact that using resource X on element A leaves element B in a particular state is something that other elements using the resource may need to know if it affects their processing, even though they never interact with element A directly. That fact about A is a part of the interface between A and the other elements in A’s environment. In this chapter, we focus only on the direct interactions.

Recall that, in Chapter 1, we defined architecture in terms of elements and their relationships. In this chapter, we focus on one type of relationship. Interfaces are a fundamental abstraction mechanism necessary to connect elements together. They have an outsized impact on a system's modifiability, usability, testability, performance, integrability, and more. Furthermore, asynchronous interfaces, which are commonly part of distributed systems, require event handlers—an architectural element.

For a given element's interface, there can be one or more implementations, each of which might have different performance, scalability, or availability guarantees. Likewise, different implementations for the same interface may be constructed for different platforms.

Three points are implied by the discussion thus far:

1. *All elements have interfaces.* All elements interact with some actors; otherwise, what is the point of the element's existence?
2. *Interfaces are two-way.* When considering interfaces, most software engineers first think of a summary of what an element provides. What methods does the element make available? What events does it process? But an element also interacts with its environment by making use of resources external to it or by assuming that its environment behaves in a certain way. If these resources are missing or if the environment doesn't behave as expected, the element can't function correctly. So an interface is more than what is *provided* by an element; an interface also includes what is *required* by an element.
3. *An element can interact with more than one actor through the same interface.* For example, web servers often restrict the number of HTTP connections that can be open simultaneously.

15.1 Interface Concepts

In this section, we discuss the concepts of multiple interfaces, resources, operations, properties, and events, as well as the evolution of interfaces.

Multiple Interfaces

It is possible to split a single interface into multiple interfaces. Each of these has a related logical purpose, and serves a different class of actors. Multiple interfaces provide a kind of separation of concerns. A specific class of actor might require only a subset of the functionality available; this functionality can be provided by one of the interfaces. Conversely, the provider of an element may want to grant actors different access rights, such as read or write, or to implement a security policy. Multiple interfaces support different levels of access. For example, an element might expose its functionality through its main interface and give access to debugging or performance monitoring data or administrative functions via separate interfaces. There may be public read-only interfaces for anonymous actors and private interfaces that allow authenticated and authorized actors to modify the state of an element.

Resources

Resources have syntax and semantics:

- *Resource syntax.* The syntax is the resource's signature, which includes any information that another program will need to write a syntactically correct program that uses the resource. The signature includes the name of the resource, the names and data types of arguments, if any, and so forth.
- *Resource semantics.* What is the result of invoking this resource? Semantics come in a variety of guises, including the following:
 - Assignment of values to data that the actor invoking the resource can access. The value assignment might be as simple as setting the value of a return argument or as far-reaching as updating a central database.
 - Assumptions about the values crossing the interface.
 - Changes in the element's state brought about by using the resource. This includes exceptional conditions, such as side effects from a partially completed operation.
 - Events that will be signaled or messages that will be sent as a result of using the resource.
 - How other resources will behave differently in the future as the result of using this resource. For example, if you ask a resource to destroy an object, trying to access that object in the future through other resources could produce an error as a result.
 - Humanly observable results. These are prevalent in embedded systems. For example, calling a program that turns on a display in a cockpit has a very observable effect—the display comes on. In addition, the statement of semantics should make it clear whether the execution of the resource will be atomic or may be suspended or interrupted.

Operations, Events, and Properties

The resources of provided interfaces consist of operations, events, and properties. These resources are complemented by an explicit description of the behavior caused or data exchanged when accessing each interface resource in terms of its syntax, structure, and semantics. (Without this description, how would the programmer or actor know whether or how to use the resources?)

Operations are invoked to transfer control and data to the element for processing. Most operations also return a result. Operations may fail, and as part of the interface it should be clear how actors can detect errors, either signaled as part of the output or through some dedicated exception-handling channel.

In addition, *events*—which are normally asynchronous—may be described in interfaces. Incoming events can represent the receipt of a message taken from a queue, or the arrival of a stream element that is to be consumed. Active elements—those that do not passively wait to be invoked by other elements—produce outgoing events used to notify listeners (or subscribers) about interesting things happening within the element.

In addition to the data transferred via operations and events, an important aspect of interfaces is metadata, such as access rights, units of measure, or formatting assumptions. Another

name for this interface metadata is *properties*. Property values can influence the behavior of operations, as highlighted in the quotation that began this chapter. Property values also affect the behavior of the element, depending on its state.

Complex interfaces of elements that are both stateful and active will feature a combination of operations, events, and properties.

Interface Evolution

All software evolves, including interfaces. Software that is encapsulated by an interface is free to evolve without impact to the elements that use this interface as long as the interface itself does not change. An interface, however, is a contract between an element and its actors. Just as a legal contract can be changed only within certain constraints, software interfaces should be changed with care. Three techniques can be used to change an interface: deprecation, versioning, and extension.

- *Deprecation.* Deprecation means removing an interface. Best practice when deprecating an interface is to give extensive notice to the actors of the element. This warning, in theory, allows the actors time to adjust to the interface's removal. In practice, many actors will not adjust in advance, but rather will discover the deprecation only when the interface is removed. One technique when deprecating an interface is to introduce an error code signifying that this interface is to be deprecated at (specific date) or that this interface has been deprecated.
- *Versioning.* Multiple interfaces support evolution by keeping the old interface and adding a new one. The old one can be deprecated when it is no longer needed or the decision has been made to no longer support it. This requires the actor to specify which version of an interface it is using.
- *Extension.* Extending an interface means leaving the original interface unchanged and adding new resources to the interface that embody the desired changes. Figure 15.1(a) shows the original interface. If the extension does not contain any incompatibilities with the original interface, then the element can implement the external interface directly, as shown in Figure 15.1(b). In contrast, if the extension introduces some incompatibilities, then it is necessary to have an internal interface for the element and to add a mediator to translate between the external interface and the internal interface, as shown in Figure 15.1(c). As an example of an incompatibility, suppose the original interface assumed that apartment numbers were included in the address but the extended interface broke out apartment numbers as a separate parameter. The internal interface would have the apartment number as a separate parameter. Then the mediator, if invoked from the original interface, would parse the address to determine any apartment number, whereas the mediator would pass the apartment number included in the separate parameter on to the internal interface unchanged.

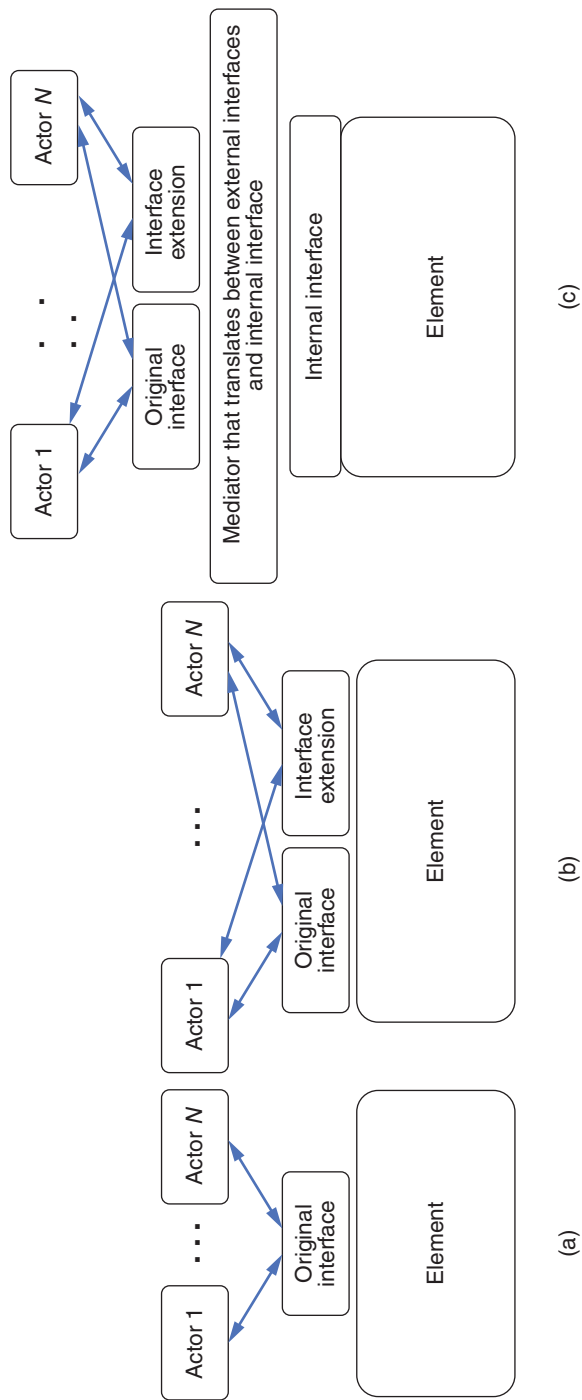


FIGURE 15.1 (a) The original interface. (b) Extending the interface. (c) Using an intermediary.

15.2 Designing an Interface

Decisions about which resources should be externally visible should be driven by the needs of actors that use the resources. Adding resources to an interface implies a commitment to maintain those resources as part of the interface for as long as the element will be in use. Once actors start to depend on a resource you provide, their elements will break if the resource is changed or removed. The reliability of your architecture is affected when the interface contract between elements is broken.

Some additional design principles for interfaces are highlighted here:

- *Principle of least surprise.* Interfaces should behave consistently with the actor's expectations. Names play a role here: An aptly named resource gives actors a good hint about what the resource can be used for.
- *Small interfaces principle.* If two elements need to interact, have them exchange as little information as possible.
- *Uniform access principle.* Avoid leaking implementation details through the interface. A resource should be accessible to its actors in the same way regardless of how they are implemented. An actor should be unaware, for example, whether a value is returned from a cache, from a computation, or from a fresh fetch of the value from some external source.
- *Don't repeat yourself principle.* Interfaces should offer a set of composable primitives as opposed to many redundant ways to achieve the same goal.

Consistency is an important aspect of designing clear interfaces. As an architect, you should establish and follow conventions on how resources are named, how API parameters are ordered, and how errors should be handled. Of course, not all interfaces are under the control of the architect, but insofar as possible the design of interfaces should be consistent throughout all elements of the same architecture. Developers will also appreciate it if interfaces follow the conventions of the underlying platform or the programming language idioms they expect. More than winning developers' goodwill, however, consistency will help minimize the number of development errors based on misunderstanding.

A successful interaction with an interface requires agreement on the following aspects:

1. Interface scope
2. Interaction style
3. Representation and structure of the exchanged data
4. Error handling

Each of these constitutes an important aspect of designing an interface. We'll cover each in turn.

Interface Scope

The scope of an interface defines the collection of resources directly available to the actors. You, as an interface designer, might want to reveal all resources; alternatively, you might wish to constrain the access to certain resources or to certain actors. For example, you might want to constrain access for reasons of security, performance management, and extensibility.

A common pattern for constraining and mediating access to resources of an element or a group of elements is to establish a *gateway* element. A gateway—often called a message gateway—translates actor requests into requests to the target element’s (or elements’) resources, and so becomes an actor for the target element or elements. Figure 15.2 provides an example of a gateway. Gateways are useful for the following reasons:

- The granularity of resources provided by an element may be different than an actor needs. A gateway can translate between elements and actors.
- Actors may need access to, or be restricted to, specific subsets of the resources.
- The specifics of the resources—their number, protocol, type, location, and properties—may change over time, and the gateway can provide a more stable interface.

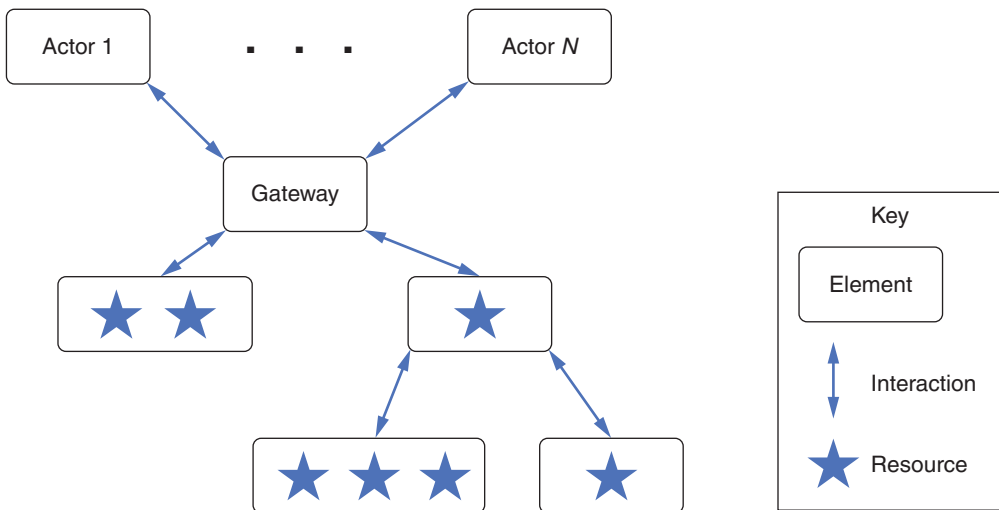


FIGURE 15.2 A gateway that provides access to a variety of different resources

We now turn to the specifics of designing particular interfaces. This means deciding which operations, events, and properties it should feature. Additionally, you must choose suitable data representation formats and data semantics to ensure the compatibility and interoperability of your architectural elements with each other. Our opening quotation gives one example of the importance of these decisions.

Interaction Styles

Interfaces are meant to be connected together so that different elements can communicate (transfer data) and coordinate (transfer control). There are many ways for such interactions to take place, depending on the mix between communication and coordination, and on whether the elements will be co-located or remotely deployed. For example:

- Interfaces of co-located elements may provide efficient access to large quantities of data via local shared memory buffers.
- Elements that are expected to be available at the same time can use synchronous calls to invoke the operations they require.
- Elements deployed in an unreliable distributed environment will need to rely on asynchronous interactions based on consuming and producing events, exchanged via message queues or data streams.

Many different interaction styles exist, but we will focus on two of the most widely used: RPC and REST.

- *Remote Procedure Call (RPC)*. RPC is modeled on procedure calls in imperative languages, except that the called procedure is located elsewhere on a network. The programmer codes the procedure call as if a local procedure were being called (with some syntactic variation); the call is then translated into a message sent to a remote element where the actual procedure is invoked. Finally, the results are sent back as a message to the calling element.

RPC dates from the 1980s and has undergone many modifications since its inception. The early versions of this protocol were synchronous, with the parameters of the message being sent as text. The most recent RPC version, called gRPC, transfers parameters in binary, is asynchronous, and supports authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts. gRPC uses HTTP 2.0 for transport.

- *Representational State Transfer (REST)*. REST is a protocol for web services. It grew out of the original protocol used when the World Wide Web was introduced. REST comprises a set of six constraints imposed on the interactions between elements:
 - *Uniform interface*. All interactions use the same form (typically HTTP). Resources on the providing side of the interface are specified via URIs (Uniform Resource Identifiers). Naming conventions should be consistent and, in general, the principle of least surprise should be followed.
 - *Client-server*. The actors are clients and the resource providers are servers using the client-server pattern.
 - *Stateless*. All client-server interactions are stateless. That is, the client should not assume that the server has retained any information about the client's last request. In consequence, interactions such as authorization are encoded into a token and the token is passed with each request.
 - *Cacheable*. Caching is applied to resources when applicable. Caching can be implemented on the server side or the client side.

- *Tiered system architecture.* The “server” can be broken into multiple independent elements, which may be deployed independently. For example, the business logic and the database can be deployed independently.
- *Code on demand (optional).* It is possible for the server to provide code to the client to be executed. JavaScript is an example.

Although not the only protocol that can be used with REST, HTTP is the most common choice. HTTP, which has been standardized by the World Wide Web Consortium (W3C), has the basic form of <command><URI>. Other parameters can be included, but the heart of the protocol is the command and the URI. Table 15.1 lists the five most important commands in HTTP and describes their relationship to the traditional CRUD (create, read, update, delete) database operations.

TABLE 15.1 Most Important Commands in HTTP and Their Relationship to CRUD Database Operations

HTTP Command	CRUD Operation Equivalent
post	create
get	read
put	update/replace
patch	update/modify
delete	delete

Representation and Structure of Exchanged Data

Every interface provides the opportunity to abstract the internal data representation, which is typically built using programming language data types (e.g., objects, arrays, collections), into a different one—that is, a representation more suitable for being exchanged across different programming language implementations and sent across the network. Converting from the internal to the external representation is termed “serialization,” “marshaling,” or “translation.”

In the following discussion, we focus on the selection of a general-purpose data interchange format or representation for sending information over a network. This decision is based on the following concerns:

- *Expressiveness.* Can the representation serialize arbitrary data structures? Is it optimized for trees of objects? Does it need to carry text written in different languages?
- *Interoperability.* Does the representation used by the interface match what its actors expect and know how to parse? A standard representation (such as JSON, described later in this section) will make it easy for actors to transform the bits transmitted across the network into internal data structures. Does the interface implement a standard?
- *Performance.* Does the chosen representation allow efficient usage of the available communication bandwidth? What is the algorithmic complexity of parsing the representation to read its content into the internal element representation? How much time is spent preparing the messages before they can be sent out? What is the monetary cost of the required bandwidth?

- *Implicit coupling.* What are the assumptions shared by the actors and elements that could lead to errors and data loss when decoding messages?
- *Transparency.* Is it possible to intercept the exchanged messages and easily observe their content? This is a double-edged sword. On the one hand, if self-describing messages help developers more easily debug message payloads and eavesdroppers more readily intercept and interpret their content. On the other hand, binary representations, particularly encrypted ones, require special debugging tools, but are more secure.

The most common programming-language-independent data representation styles can be divided between textual (e.g., XML or JSON) and binary (e.g., protocol buffers) options.

Extensible Markup Language (XML)

XML was standardized by the World Wide Web Consortium (W3C) in 1998. XML annotations to a textual document, called *tags*, are used to specify how to interpret the information in the document by breaking the information into chunks or fields and identifying the data type of each field. Tags can be annotated with attributes.

XML is a meta-language: Out of the box, it does nothing except allow you to define a customized language to describe your data. Your customized language is defined by an *XML schema*, which is itself an XML document that specifies the tags you will use, the data type that should be used to interpret fields enclosed by each tag, and the constraints that apply to the structure of your document. XML schemas enable you as an architect to specify a rich information structure.

XML documents are used as representations of structured data for many purposes: for messages exchanged in a distributed system (SOAP), the content of web pages (XHTML), vector images (SVG), business documents (DOCX), web service interface description (WSDL), and static configuration files (e.g., MacOS property lists).

One strength of XML is that a document annotated using this language can be checked to validate that it conforms to a schema. This prevents faults caused by malformed documents and eliminates the need for some kinds of error checking by the code that reads and processes the document. The tradeoff is that parsing the document and validating it are relatively expensive in terms of processing and memory. A document must be read completely before it can be validated and may require multiple read passes to unmarshal. This requirement, coupled with XML's verbosity, can result in unacceptable runtime performance and bandwidth consumption. While during XML's heyday the argument was often made that "XML is human readable," today this benefit is cited far less often.

JavaScript Object Notation (JSON)

JSON structures data as nested name/value pairs and array data types. The JSON notation grew out of the JavaScript language and was first standardized in 2013; today, however, it is independent of any programming language. Like XML, JSON is a textual representation featuring its own schema language. Compared to XML, however, JSON is significantly less verbose, as field names occur only once. Using a name/value representation instead of start and end tags, JSON documents can be parsed as they are read.

JSON data types are derived from JavaScript data types, and resemble those of any modern programming language. This makes JSON serialization and deserialization much more efficient than XML. The notation's original use case was to send JavaScript objects between a browser and web server—for example, to transfer a lightweight data representation to be rendered as HTML in the browser, as opposed to performing the rendering on the server side and having to download more verbose views represented using HTML.

Protocol Buffers

The Protocol Buffer technology originated at Google and was used internally for several years before being released as open source in 2008. Like JSON, Protocol Buffers use data types that are close to programming-language data types, making serialization and deserialization efficient. As with XML, Protocol Buffer messages have a schema that defines a valid structure, and that schema can specify both required and optional elements and nested elements. However, unlike both XML and JSON, Protocol Buffers are a binary format, so they are extremely compact and use memory and network bandwidth resources quite efficiently. In this respect, Protocol Buffers harken back to a much earlier binary representation called Abstract Syntax Notation One (ASN.1), which originated in the early 1980s when network bandwidth was a precious resource and no bit could be wasted.

The Protocol Buffers open source project provides code generators to allow easy use of Protocol Buffers with many programming languages. You specify your message schema in a *proto* file, which is then compiled by a language-specific protocol buffer compiler. The procedures generated by the compilers will be used by an actor to serialize and by an element to deserialize the data.

As when using XML and JSON, the interacting elements may be written in different languages. Each element then uses the Protocol Buffer compiler specific to its language. Although Protocol Buffers can be used for any data-structuring purpose, they are mostly employed as part of the gRPC protocol.

Protocol Buffers are specified using an interface description language. Since they are compiled by language-specific compilers, the specification is necessary to ensure correct behavior of the interface. It also acts as documentation for the interfaces. Placing the interface specification in a database allows for searching it to see how values propagate through the various elements.

Error Handling

When designing an interface, architects naturally concentrate on how it is supposed to be used in the nominal case, when everything works according to plan. The real world, of course, is far from the nominal case, and a well-designed system must know how to take appropriate action in the face of undesired circumstances. What happens when an operation is called with invalid parameters? What happens when a resource requires more memory than is available? What happens when a call to an operation never returns, because it has failed? What happens when the interface is supposed to trigger a notification event based on the value of a sensor, but the sensor isn't responding or is responding with gibberish?

Actors need to know whether the element is working correctly, whether their interaction is successful and whether an error has occurred. Strategies to do so include the following:

- Failed operations may throw an exception.
- Operations may return a status indicator with predefined codes, which would need to be tested to detect erroneous outcomes.
- Properties may be used to store data indicating whether the latest operation was successful or not, or whether stateful elements are in an erroneous state.
- Error events such as a timeout may be triggered for failed asynchronous interactions.
- The error log may be read by connecting to a specific output data stream.

The specification of which exceptions, which status codes, which events, and which information are used to describe erroneous outcomes becomes part of the interface of an element. Common sources of errors (which the interface should handle gracefully) include the following:

- Incorrect, invalid, or illegal information was sent to the interface—for example, calling an operation with a null value parameter that should not be null. Associating an error condition with the resource is the prudent thing to do.
- The element is in the wrong state for handling the request. The element may have entered the improper state as a result of a previous action or the lack of a previous action on the part of the same or another actor. Examples of the latter include invoking an operation or reading a property before the element's initialization has completed, and writing to a storage device that has been taken offline by the system's human operator.
- A hardware or software error occurred that prevented the element from successfully executing. Processor failures, failure of the network to respond, and inability to allocate more memory are examples of this kind of error condition.
- The element is not configured correctly. For example, its database connection string refers to the wrong database server.

Indicating the source of the error helps the system choose the appropriate correction and recovery strategy. Temporary errors with idempotent operations can be dealt with by waiting and retrying. Errors due to invalid input require fixing the bad requests and resending them. Missing dependencies should be reinstalled before reattempting to use the interface. Implementation bugs should be fixed by adding the usage failure scenario as an additional test case to avoid regressions.

15.3 Documenting the Interface

Although an interface comprises all aspects of the interaction that an element has with its environment, what we choose to disclose about an interface—that is, what we put in an interface's documentation—is more limited. Writing down every aspect of every possible interaction is not practical and almost never desirable. Rather, you should expose only what the actors

on an interface *need* to know to interact with it. Put another way, you choose what information is permissible and appropriate for people to assume about the element.

The interface documentation indicates what other developers need to know about an interface to use it in combination with other elements. A developer might subsequently observe properties that are a manifestation of how the element is implemented, but that are not detailed in the interface documentation. Because these are not part of the interface documentation, they are subject to change, and developers use them at their own risk.

Also recognize that different people need to know different kinds of information about the interface. You may have to include separate sections in the interface documentation that accommodate different stakeholders of the interface. As you document an element's interface, keep the following stakeholder roles in mind:

- *Developer of the element.* Needs to be aware of the contract that their interface must fulfill. Developers can test only the information embodied in the interface description.
- *Maintainer.* A special kind of developer who makes assigned changes to the element and its interface while minimizing disruption of existing actors.
- *Developer of an element using the interface.* Needs to understand the interface's contract and how to use it. Such developers can provide input to the interface design and documentation process in terms of use cases that the interface should support.
- *Systems integrator and tester.* Puts the system together from its constituent elements and has a strong interest in the behavior of the resulting assembly. This role needs detailed information about all the resources and functionality provided by and required by an element.
- *Analyst.* This role depends on the types of analyses conducted. For a performance analyst, for example, the interface documentation should include a service level agreement (SLA) guarantee, so that actors can adjust their requests appropriately.
- *Architect looking for assets to reuse in a new system.* Often starts by examining the interfaces of elements from a previous system. The architect may also look in the commercial marketplace to find off-the-shelf elements that can be purchased and do the job. To see whether an element is a candidate, the architect is interested in the capabilities of the interface resources, their quality attributes, and any variability that the element provides.

Describing an element's interface means making statements about the element that other elements can depend on. Documenting an interface means that you have to describe which services and properties are parts of the contract—a step that represents a promise to actors that the element will, indeed, fulfill this contract. Every implementation of the element that does not violate the contract is a valid implementation.

A distinction must be drawn between the interface of an element and the documentation of that interface. What you can observe about an element is part of its interface—how long an operation takes, for example. The documentation of the interface covers a subset of that behavior: It lays out what we *want* our actors to be able to depend on.

“Hyrum’s law” (www.hyrumslaw.com) states: “With a sufficient number of users of an interface, it does not matter what you promise in the contract: All observable behaviors of your system will be depended on by somebody.” True enough. But, as we said earlier, an actor that depends on what you do not publish about an element's interface does so at its own risk.

15.4 Summary

Architectural elements have interfaces, which are boundaries over which elements interact with each other. Interface design is an architectural duty, because compatible interfaces allow architectures with many elements to do something productive and useful together. A primary use of an interface is to encapsulate an element's implementation, so that this implementation may change without affecting other elements.

Elements may have multiple interfaces, providing different types of access and privileges to different classes of actors. Interfaces state which resources the element provides to its actors as well as what the element needs from its environment to function correctly. Like architectures themselves, interfaces should be as simple as possible, but no simpler.

Interfaces have operations, events, and properties; these are the parts of an interface that the architect can design. To do so, the architect must decide the element's

- Interface scope
- Interaction style
- Representation, structure, and semantics of the exchanged data
- Error handling

Some of these issues can be addressed by standardized means. For example, data exchange can use mechanisms such as XML, JSON, or Protocol Buffers.

All software evolves, including interfaces. Three techniques that can be used to change an interface are deprecation, versioning, and extension.

The interface documentation indicates what other developers need to know about an interface to use it in combination with other elements. Documenting an interface involves deciding which element operations, events, and properties to expose to the element's actors, and detailing the interface's syntax and semantics.

15.5 For Further Reading

To see the difference between an XML representation, a JSON representation, and a Protocol Buffer representation of a postal address, see <https://schema.org/PostalAddress>, <https://schema.org/PostalAddress>, and https://github.com/mgravell/protobuf-net/blob/master/src/protogen.site/wwwroot/protoc/google/type/postal_address.proto.

You can read more about gRPC at <https://grpc.io/>.

REST was defined by Roy Fielding in his PhD thesis: ics.uci.edu/~fielding/pubs/dissertation/top.htm.

15.6 Discussion Questions

1. Describe the interface to a dog, or another kind of animal with which you are familiar. Describe its operations, events, and properties. Does a dog have multiple interfaces (e.g., one for a known human and another for a stranger)?
2. Document the interface to a light bulb. Document its operations, events, and properties. Document its performance and resource utilization. Document any error states it may enter and what the result will be. Can you think of multiple implementations that have the same interface you just described?
3. Under what circumstances should performance (e.g., how long an operation takes) be a part of an element's published interface? Under what circumstances should it not?
4. Suppose an architectural element will be used in a high-availability system. How might that affect its interface documentation? Suppose the same element will now be used in a high-security system. What might you document differently?
5. The section "Error Handling" listed a number of different error-handling strategies. For each, when is its use appropriate? Inappropriate? What quality attributes will each enhance or diminish?
6. What would you have done to prevent the interface error that led to the loss of the Mars Climate Orbiter, as described at the beginning of this chapter?
7. On June 4, 1996, an Ariane 5 rocket failed quite spectacularly, only 37 seconds after launch. Research this failure, and discuss what better interface discipline could have done to prevent it.
8. A database schema represents an interface between an element and a database; it provides the metadata for accessing the database. Given this view, schema evolution is a form of interface evolution. Discuss ways in which a schema can evolve and not break the existing interface, and ways in which it does break it. Describe how deprecation, versioning, and extension apply to schema evolution.

This page intentionally left blank

16



Virtualization

Virtual means never knowing where your next byte is coming from.
—Unknown

In the 1960s, the computing community was frustrated by the problem of sharing resources such as memory, disk, I/O channels, and user input devices on one physical machine among several independent applications. The inability to share resources meant that only one application could be run at a time. Computers at that time cost millions of dollars—real money in those days—and most applications used only a fraction, typically around 10%, of the available resources, so this situation had a significant effect on computing costs.

Virtual machines and, later, containers emerged to deal with sharing. The goal of these virtual machines and containers is to isolate one application from another, while still sharing resources. Isolation allows developers to write applications as if they are the only ones using the computer, while sharing resources allows multiple applications to run on the computer at the same time. Because the applications are sharing one physical computer with a fixed set of resources, there are limits to the illusion that isolation creates. If, for example, one application consumes all of the CPU resources, then the other applications cannot execute. For most purposes, however, these mechanisms have changed the face of systems and software architecture. They fundamentally change how we conceive of, deploy, and pay for computing resources.

Why is this topic of interest and concern to architects? As an architect, you may be inclined—or indeed required—to use some form of virtualization to deploy the software that you create. For an increasingly large set of applications, you'll be deploying to the cloud (coming up in Chapter 17) and using containers to do it. Furthermore, in cases where you will deploy to specialized hardware, virtualization allows you to perform testing in an environment that is much more accessible than the specialized hardware.

The purpose of this chapter is to introduce some of the most important terms, considerations, and tradeoffs in employing virtual resources.

16.1 Shared Resources

For economic reasons, many organizations have adopted some forms of shared resources. These can dramatically lower the costs of deploying a system. There are four resources that we typically care about sharing:

1. *Central processor unit (CPU)*. Modern computers have multiple CPUs (and each CPU can have multiple processing cores). They may also have one or more graphics processing units (GPUs), or other special-purpose processors, such as a tensor processing unit (TPU).
2. *Memory*. A physical computer has a fixed amount of physical memory.
3. *Disk storage*. Disks provide persistent storage for instructions and data, across reboots and shutdowns of the computer. A physical computer typically has one or more attached disks, each with a fixed amount of storage capacity. Disk storage can refer to either a rotating magnetic or optical hard disk drive device, or a solid-state disk drive device; the latter has neither disks nor any moving parts to drive.
4. *Network connection*. Today, every nontrivial physical computer has one or more network connections through which all messages pass.

Now that we have enumerated the resources that we want to share, we need to think about *how* to share them, and how to do this in a sufficiently “isolated” way so that different applications are unaware of each other’s existence.

Processor sharing is achieved through a thread-scheduling mechanism. The scheduler selects and assigns an execution thread to an available processor, and that thread maintains control until the processor is rescheduled. No application thread can gain control of a processor without going through the scheduler. Rescheduling occurs when the thread yields control of the processor, when a fixed time interval expires, or when an interrupt occurs.

Historically, as applications grew, all the code and data would not fit into physical memory. Virtual memory technology was developed to deal with this challenge. Memory management hardware partitions a process’s address space into pages, and swaps pages between physical memory and secondary storage as needed. The pages that are in physical memory can be accessed immediately, and other pages are stored on the secondary memory until they are needed. The hardware supports the isolation of one address space from another.

Disk sharing and isolation are achieved using several mechanisms. First, the physical disks can be accessed only through a disk controller that ensures the data streams to and from each thread are delivered in sequence. Also, the operating system may tag executing threads and disk content such as files and directories with information such as a user ID and group, and restrict visibility or access by comparing the tags of the thread requesting access and the disk content.

Network isolation is achieved through the identification of messages. Every virtual machine (VM) or container has an Internet Protocol (IP) address, which is used to identify messages to or from that VM or container. In essence, the IP address is used to route responses to the correct VM or container. Another network mechanism for sending and receiving messages relies on the use of ports. Every message intended for a service has a port number associated with it. A service listens on a port and receives messages that arrive at the device on which the service is executing designated for the port on which the service is listening.

16.2 Virtual Machines

Now that we have seen how the resource usage of one application can be isolated from the resource usage of another application, we can employ and combine these mechanisms. *Virtual machines* allow the execution of multiple simulated, or virtual, computers in a single physical computer.

Figure 16.1 depicts several VMs residing in a physical computer. The physical computer is called the “host computer” and the VMs are called “guest computers.” Figure 16.1 also shows a *hypervisor*, which is an operating system for the VMs. This hypervisor runs directly on the physical computer hardware and is often called a *bare-metal* or *Type 1* hypervisor. The VMs that it hosts implement applications and services. Bare-metal hypervisors typically run in a data center or cloud.

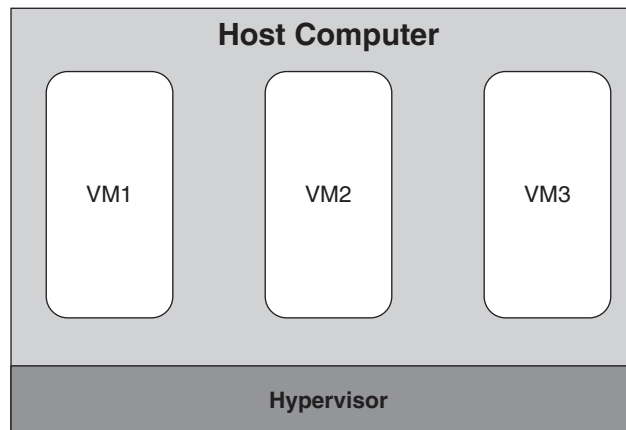
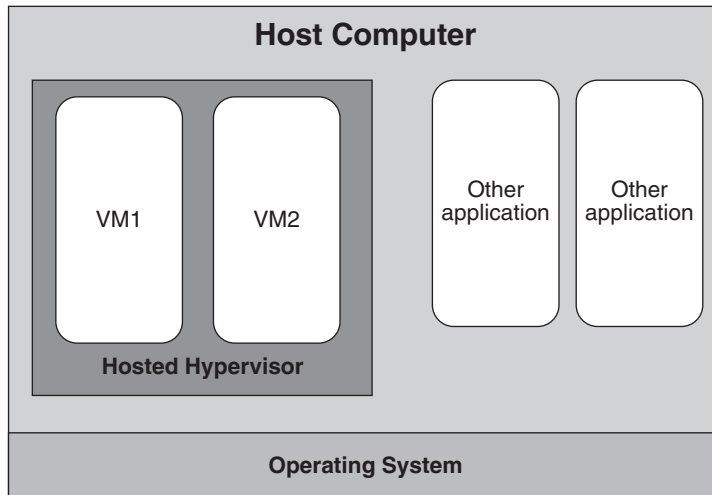


FIGURE 16.1 Bare-metal hypervisor and VMs

Figure 16.2 depicts another type of hypervisor, called a *hosted* or *Type 2* hypervisor. In this case, the hypervisor runs as a service on top of a host operating system, and the hypervisor in turn hosts one or more VMs. Hosted hypervisors are typically used on desktop or laptop computers. They allow developers to run and test applications that are not compatible with the computer’s host operating system (e.g., to run Linux applications on a Windows computer or to run Windows applications on an Apple computer). They can also be used to replicate a production environment on a development computer, even if the operating system is the same on both. This approach ensures that the development and production environments match each other.

**FIGURE 16.2** Hosted hypervisor

A hypervisor requires that its guest VMs use the same instruction set as the underlying physical CPU—the hypervisor does not translate or simulate instruction execution. For example, if you have a VM for a mobile or embedded device that uses an ARM processor, you cannot run that virtual machine on a hypervisor that uses an x86 processor. Another technology, related to hypervisors, supports cross-processor execution; it is called an *emulator*. An emulator reads the binary code for the target or guest processor and simulates the execution of guest instructions on the host processor. The emulator often also simulates guest I/O hardware devices. For example, the open source QEMU emulator¹ can emulate a full PC system, including BIOS, x86 processor and memory, sound card, graphics card, and even a floppy disk drive.

Hosted/Type 2 hypervisors and emulators allow a user to interact with the applications running inside the VM through the host machine's on-screen display, keyboard, and mouse/touchpad. Developers working on desktop applications or working on specialized devices, such as mobile platforms or devices for the Internet of Things, may use a hosted/Type 2 hypervisor and/or an emulator as part of their build/test/integrate toolchain.

A hypervisor performs two main functions: (1) It manages the code running in each VM, and (2) it manages the VMs themselves. To elaborate:

1. Code that communicates outside the VM by accessing a virtualized disk or network interface is intercepted by the hypervisor and executed by the hypervisor on behalf of the VM. This allows the hypervisor to tag these external requests so that the response to these requests can be routed to the correct VM.

1. qemu.org

The response to an external request to an I/O device or the network is an asynchronous interrupt. This interrupt is initially handled by the hypervisor. Since multiple VMs are operating on a single physical host machine and each VM may have I/O requests outstanding, the hypervisor must have a method for forwarding the interrupt to the correct VM. This is the purpose of the tagging mentioned earlier.

2. VMs must be managed. For example, they must be created and destroyed, among other things. Managing VMs is a function of the hypervisor. The hypervisor does not decide on its own to create or destroy a VM, but rather acts on instructions from a user or, more frequently, from a cloud infrastructure (you'll read more about this in Chapter 17). The process of creating a VM involves loading a *VM image* (discussed in the next section).

In addition to creating and destroying VMs, the hypervisor monitors them. Health checks and resource usage are part of the monitoring. The hypervisor is also located inside the defensive security perimeter of the VMs, as a defense against attacks.

Finally, the hypervisor is responsible for ensuring that a VM does not exceed its resource utilization limits. Each VM has limits on CPU utilization, memory, and disk and network I/O bandwidth. Before starting a VM, the hypervisor first ensures that sufficient physical resources are available to satisfy that VM's needs, and then the hypervisor enforces those limits while the VM is running.

A VM is booted just as a bare-metal physical machine is booted. When the machine begins executing, it automatically reads a special program called the boot loader from disk storage, either internal to the computer or connected through a network. The boot loader reads the operating system code from disk into memory, and then transfers execution to the operating system. In the case of a physical computer, the connection to the disk drive is made during the power-up process. In the case of the VM, the connection to the disk drive is established by the hypervisor when it starts the VM. The "VM Images" section discusses this process in more detail.

From the perspective of the operating system and software services inside a VM, it appears as if the software is executing inside of a bare-metal physical machine. The VM provides a CPU, memory, I/O devices, and a network connection.

Given the many concerns that it must address, the hypervisor is a complicated piece of software. One concern with VMs is the overhead introduced by the sharing and isolation needed for virtualization. That is, how much slower does a service run on a virtual machine, compared to running directly in a bare-metal physical machine? The answer to this question is complicated: It depends on the characteristics of the service and on the virtualization technology used. For example, services that perform more disk and network I/O incur more overhead than services that do not share these host resources. Virtualization technology is improving all the time, but overheads of approximately 10% have been reported by Microsoft on its Hyper-V hypervisor.²

There are two major implications of VMs for an architect:

1. *Performance.* Virtualization incurs a performance cost. While Type 1 hypervisors carry only a modest performance penalty, Type 2 hypervisors may impose a significantly larger overhead.

2. <https://docs.microsoft.com/en-us/biztalk/technical-guides/system-resource-costs-on-hyper-v>

2. *Separation of concerns.* Virtualization allows an architect to treat runtime resources as commodities, deferring provisioning and deployment decisions to another person or organization.

16.3 VM Images

We call the contents of the disk storage that we boot a VM from a *VM image*. This image contains the bits that represent the instructions and data that make up the software that we will run (i.e., the operating system and services). The bits are organized into files and directories according to the file system used by your operating system. The image also contains the boot load program, stored in its predetermined location.

There are three approaches you can follow to create a new VM image:

1. You can find a machine that is already running the software you want and make a snapshot copy of the bits in that machine's memory.
2. You can start from an existing image and add additional software.
3. You can create an image from scratch. Here, you start by obtaining installation media for your chosen operating system. You boot your new machine from the install media, and it formats the machine's disk drive, copies the operating system onto the drive, and adds the boot loader in the predetermined location.

For the first two approaches, repositories of machine images (usually containing open-source software) are available that provide a variety of minimal images with just OS kernels, other images that include complete applications, and everything in between. These efficient starting points can support you in quickly trying out a new package or program.

However, some issues may arise when you are pulling down and running an image that you (or your organization) did not create:

- You cannot control the versions of the OS and software.
- The image may have software that contains vulnerabilities or that is not configured securely; even worse, the image may include malware.

Other important aspects of VM images are:

- These images are very large, so transferring them over a network can be very slow.
- An image is bundled with all of its dependencies.
- You can build a VM image on your development computer and then deploy it to the cloud.
- You may wish to add your own services to the VM.

While you could easily install services when creating an image, this would lead to a unique image for every version of every service. Aside from the storage cost, this proliferation of images becomes difficult to keep track of and manage. Thus it is customary to create images that contain only the operating system and other essential programs, and then add services to these images after the VM is booted, in a process called configuration.

16.4 Containers

VMs solve the problem of sharing resources and maintaining isolation. However, VM images can be large, and transferring VM images around the network is time-consuming. Suppose you have an 8 GB(yte) VM image. You wish to move this from one location on the network to another. In theory, on a 1 Gb(it) per second network, this will take 64 seconds. However, in practice a 1 Gbps network operates at around 35% efficiency. Thus transferring an 8 GB VM image will take more than 3 minutes in the real world. Although you can adopt some techniques to reduce this transfer time, the result will still be a duration measured in minutes. After the image is transferred, the VM must boot the operating system and start your services, which takes still more time.

Containers are a mechanism to maintain most of the advantages of virtualization while reducing the image transfer time and startup time. Like VMs and VM images, containers are packaged into executable container images for transfer. (However, this terminology is not always followed in practice.)

Reexamining Figure 16.1, we see that a VM executes on virtualized hardware under the control of the hypervisor. In Figure 16.3, we see several containers operating under the control of a *container runtime engine*, which in turn is running on top of a fixed operating system. The container runtime engine acts as a virtualized operating system. Just as all VMs on a physical host share the same underlying physical hardware, all containers within a host share the same operating system kernel through the runtime engine (and through the operating system, they share the same underlying physical hardware). The operating system can be loaded either onto a bare-metal physical machine or a virtual machine.

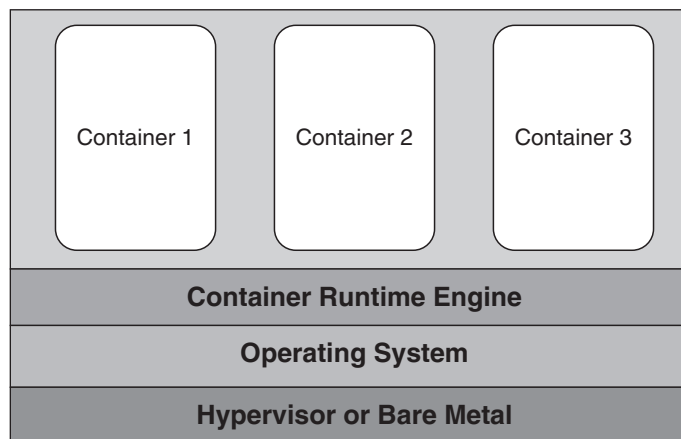


FIGURE 16.3 Containers on top of a container runtime engine on top of an operating system on top of a hypervisor (or bare metal)

VMs are allocated by locating a physical machine that has sufficient unused resources to support an additional VM. This is done, conceptually, by querying the hypervisors to find one with spare capacity. Containers are allocated by finding a container runtime engine that has sufficient unused resources to support an additional container. This may, in turn, require the creation of an additional VM to support an additional container runtime engine. Figure 16.3 depicts containers running on a container runtime engine running on an operating system running in a VM under the control of a hypervisor.

This sharing of the operating system represents a source of performance improvement when transferring images. As long as the target machine has a standard container runtime engine running on it (and these days all container runtime engines are built to standards), there is no need to transfer the operating system as part of the container image.

The second source of performance improvement is the use of “layers” in the container images. (Note that container layers are different from the notion of layers in module structures that we introduced in Chapter 1.) To better understand container layers, we will describe how a container image is constructed. In this case, we will illustrate the construction of a container to run the *LAMP stack*, and we will build the image in layers. (LAMP—which stands for Linux, Apache, MySQL, and PHP—is a widely used stack for constructing web applications.)

The process of building an image using the LAMP stack is as follows:

1. Create a container image containing a Linux distribution. (This image can be downloaded from a library using a container management system.)
2. Once you create the image and identify it as an image, execute it (i.e., instantiate it).
3. Use that container to load services—Apache, in our example, using features of Linux.
4. Exit the container and inform the container management system that this is a second image.
5. Execute this second image and load MySQL.
6. Exit the container and give this third image a name.
7. Repeat this process one more time and load PHP. Now you have a fourth container image; this one holds the entire LAMP stack.

Because this image was created in steps and you told the container management system to make each step an image, the container management system considers the final image to be made up of “layers.”

Now you can move the LAMP stack container image to a different location for production use. The initial move requires moving all the elements of the stack. Suppose, however, you update PHP to a newer version and move this revised stack into production (Step 7 in the preceding process). The container management system knows that only PHP was revised and moves only the PHP layer of the image. This saves the effort involved in moving the rest of the stack. Since changing a software component within an image happens much more frequently than initial image creation, placing a new version of the container into production becomes a much faster process than it would be using a VM. Whereas loading a VM takes on the order of minutes, loading a new version of a container takes on the order of microseconds or milliseconds. Note that this process works only with the uppermost layer of the stack. If, for example, you wanted to update MySQL with a newer version, you would need to execute Steps 5 through 7 in the earlier list.

You can create a script with the steps for the creation of a container image and store it in a file. This file is specific to the tool you are using to create the container image. Such a file allows you to specify which pieces of software are to be loaded into the container and saved as an image. Using version control on the specification file ensures that each member of your team can create an identical container image and modify the specification file as needed. Treating these scripts as code brings a wealth of advantages: These scripts can be consciously designed, tested, configuration controlled, reviewed, documented, and shared.

16.5 Containers and VMs

What are the tradeoffs between delivering your service in a VM and delivering your service in a container?

As we noted earlier, a VM virtualizes the physical hardware: CPU, disk, memory, and network. The software that you run on the VM includes an entire operating system, and you can run almost any operating system in a VM. You can also run almost any program in a VM (unless it must interact directly with the physical hardware), which is important when working with legacy or purchased software. Having the entire operating system also allows you to run multiple services in the same VM—a desirable outcome when the services are tightly coupled or share large data sets, or if you want to take advantage of the efficient interservice communication and coordination that are available when the services run within the context of the same VM. The hypervisor ensures that the operating system starts, monitors its execution, and restarts the operating system if it crashes.

Container instances share an operating system. The operating system must be compatible with the container runtime engine, which limits the software that can run on a container. The container runtime engine starts, monitors, and restarts the service running in a container. This engine typically starts and monitors just one program in a container instance. If that one program completes and exits normally, execution of that container ends. For this reason, containers generally run a single service (although that service can be multi-threaded). Furthermore, one benefit of using containers is that the size of the container image is small, including only those programs and libraries necessary to support the service we want to run. Multiple services in a container could bloat the image size, increasing the container startup time and runtime memory footprint. As we will see shortly, we can group container instances running related services so that they will execute on the same physical machine and can communicate efficiently. Some container runtime engines even allow containers within a group to share memory and coordination mechanisms such as semaphores.

Other differences between VMs and containers are as follows:

- Whereas a VM can run any operating system, containers are currently limited to Linux, Windows, or IOS.
- Services within the VM are started, stopped, and paused through operating system functions, whereas services within containers are started and stopped through container runtime engine functions.

- VMs persist beyond the termination of services running within them; containers do not.
- Some restrictions on port usage exist when using containers that do not exist when using VMs.

16.6 Container Portability

We have introduced the concept of a container runtime manager with which the container interacts. Several vendors provide container runtime engines, most notably Docker, containerd, and Mesos. Each of these providers has a container runtime engine that provides capabilities to create container images and to allocate and execute container instances. The interface between the container runtime engine and the container has been standardized by the Open Container Initiative, allowing a container created by one vendor's package (say, Docker) to be executed on a container runtime engine provided by another vendor (say, containerd).

This means that you can develop a container on your development computer, deploy it to a production computer, and have it execute there. Of course, the resources available will be different in each case, so deployment is still not trivial. If you specify all the resources as configuration parameters, the movement of your container into production is simplified.

16.7 Pods

Kubernetes is open source orchestration software for deploying, managing, and scaling containers. It has one more element in its hierarchy: Pods. A *Pod* is a group of related containers. In Kubernetes, nodes (hardware or VMs) contain Pods, and Pods contain containers, as shown in Figure 16.4. The containers in a Pod share an IP address and port space to receive requests from other services. They can communicate with each other using interprocess communication (IPC) mechanisms such as semaphores or shared memory, and they can share ephemeral storage volumes that exist for the lifetime of the Pod. They have the same lifetime—the containers in Pods are allocated and deallocated together. For example, service meshes, discussed in Chapter 9, are often packaged as a Pod.

The purpose of a Pod is to reduce communication costs between closely related containers. In Figure 16.4, if container 1 and container 2 communicate frequently, the fact they are deployed as a Pod, and thus allocated onto the same VM, allows the use of faster communication mechanisms than message passing.

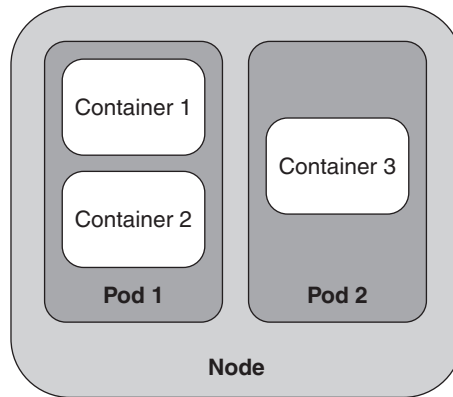


FIGURE 16.4 Node with Pods that in turn have containers

16.8 Serverless Architecture

Recall that allocating a VM starts by locating a physical machine with enough free capacity and then loading a VM image into that physical machine. The physical computers, therefore, constitute a pool from which you can allocate resources. Suppose now that instead of allocating VMs into physical machines, you wish to allocate containers into container runtime engines. That is, you have a pool of container runtime engines, into which containers are allocated.

Load times for a container are very short—taking just a few seconds for a cold start and a few milliseconds to reallocate. Now let’s carry this one step further. Since VM allocation and loading are relatively time-consuming, potentially taking minutes to load and start the instance, you typically leave a VM instance running even if there is idle time between requests. In comparison, since the allocation of a container into a container runtime engine is fast, it is not necessary to leave the container running. We can afford to reallocate a new container instance for every request. When your service completes the processing of a request, instead of looping back to take another request, it exits, and the container stops running and is deallocated.

This approach to system design is called *serverless architecture*—though it is not, in fact, serverless. There are servers, which host container runtime engines, but since they are allocated dynamically with each request, the servers and container runtime engines are embodied in the infrastructure. You, as a developer, are not responsible for allocating or deallocating them. The cloud service provider features that support this capability are called function-as-a-service (FaaS).

A consequence of the dynamic allocation and deallocation in response to individual requests is that these short-lived containers cannot maintain any state: The containers must be stateless. In a serverless architecture, any state needed for coordination must be stored in an infrastructure service delivered by the cloud provider or passed as parameters.

Cloud providers impose some practical limitations on FaaS features. The first is that the providers have a limited selection of base container images, which restricts your programming language options and library dependencies. This is done to reduce the container load time—your service is constrained to be a thin image layer on top of the provider’s base image layer. The next limitation is that the “cold start” time, when your container is allocated and loaded the first time, can be several seconds. Subsequent requests are handled nearly instantaneously, as your container image is cached on a node. Finally, the execution time for a request is limited—your service must process the request and exit within the provider’s time limit or it will be terminated. Cloud providers do this for economic reasons, so that they can tailor the pricing of FaaS compared to other ways of running containers, and to ensure that no FaaS user consumes too much of the resource pool. Some designers of serverless systems devote considerable energy to working around or defeating these limitations—for example, prestarting services to avoid cold-start latency, making dummy requests to keep services in cache, and forking or chaining requests from one service to another to extend the effective execution time.

16.9 Summary

Virtualization has been a boon for software and system architects, as it provides efficient, cost-effective allocation platforms for networked (typically web-based) services. Hardware virtualization allows for the creation of several virtual machines that share the same physical machine. It does this while enforcing isolation of the CPU, memory, disk storage, and network. Consequently, the resources of the physical machine can be shared among several VMs, while the number of physical machines that an organization must purchase or rent is minimized.

A VM image is the set of bits that are loaded into a VM to enable its execution. VM images can be created by various techniques for provisioning, including using operating system functions or loading a pre-created image.

Containers are a packaging mechanism that virtualizes the operating system. A container can be moved from one environment to another if a compatible container runtime engine is available. The interface to container runtime engines has been standardized.

Placing several containers into a Pod means that they are all allocated together and any communication between the containers can be done quickly.

Serverless architecture allows for containers to be rapidly instantiated and moves the responsibility for allocation and deallocation to the cloud provider infrastructure.

16.10 For Further Reading

The material in this chapter is taken from *Deployment and Operations for Software Engineers* [Bass 19], where you can find more detailed discussions.

Wikipedia is always a good place to find current details of protocols, container runtime engines, and serverless architectures.

16.11 Discussion Questions

1. Create a LAMP container using Docker. Compare the size of your container image to one you find on the Internet. What is the source of the difference? Under what circumstances is this a cause of concern for you as an architect?
2. How does the container management system know that only one layer has been changed so that it needs to transport only one layer?
3. We have focused on isolation among VMs that are running at the same time on a hypervisor. VMs may shut down and stop executing, and new VMs may start up. What does a hypervisor do to maintain isolation, or prevent leakage, between VMs running at different times? Hint: Think about the management of memory, disk, virtual MAC, and IP addresses.
4. What set of services would it make sense to group into a Pod (as was done with service meshes) and why?
5. What are the security issues associated with containers? How would you mitigate them?
6. What are the concerns associated with employing virtualization technologies in embedded systems?
7. What class of integration and deployment errors can be avoided with VMs, containers, and Pods? What class cannot?

This page intentionally left blank



The Cloud and Distributed Computing

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

—Leslie Lamport

Cloud computing is about the on-demand availability of resources. This term is used to refer to a wide range of computing capabilities. For example, you might say, “All my photos are backed up to the cloud.” But what does that mean? It means:

- My photos are stored on someone else’s computers. They worry about the capital investment and maintenance and upkeep and backups.
- My photos are accessible by me over the Internet.
- I pay only for the space that I use, or that I requisition.
- The storage service is elastic, meaning that it can grow or shrink as my needs change.
- My use of the cloud is self-provisioned: I create an account and can immediately begin using it to store my materials.

The computing capabilities delivered from the cloud range from applications such as photo (or other kinds of digital artifact) storage, to fine-grained services exposed through APIs (e.g., text translation or currency conversion), to low-level infrastructure services such as processors, network, and storage virtualization.

In this chapter, we will focus on how a software architect can use infrastructure services from the cloud to deliver the services that the architect is designing and developing. Along the way, we will take a journey into some of the most important principles and techniques of distributed computing. This means using multiple (real or virtual) computers to work cooperatively together, thereby producing faster performance and a more robust system than a single computer doing all the work. We included this subject matter in this chapter because nowhere is distributed computing more ingrained than in cloud-based systems. The treatment we give here is a brief overview of the principles most relevant to architecture.

We first discuss how the cloud provides and manages virtual machines.

17.1 Cloud Basics

Public clouds are owned and provided by cloud service providers. These organizations provide infrastructure services to anyone who agrees to the terms of service and can pay for use of the services. In general, the services you build using this infrastructure are accessible on the public Internet, although you can provision mechanisms such as firewalls to restrict visibility and access.

Some organizations operate a *private cloud*. A private cloud is owned and operated by an organization for the use of members of that organization. An organization might choose to operate a private cloud because of concerns such as control, security, and cost. In this case, the cloud infrastructure and the services developed on it are visible and accessible only within the organization's network.

The *hybrid cloud* approach is a mixed model, in which some workloads are run in a private cloud and other workloads are run in a public cloud. A hybrid cloud might be used during a migration from a private cloud to a public cloud (or vice versa), or it might be used because some data are legally required to be subject to greater control and scrutiny than is possible with a public cloud.

For an architect designing software using cloud services, there is not much difference, from a technical perspective, between private clouds and public clouds. Thus we will focus our discussion here on infrastructure-as-a-service public clouds.

A typical public cloud data center has tens of thousands of physical devices—closer to 100,000 than to 50,000. The limiting factor on the size of a data center is the electric power it consumes and the amount of heat that the equipment produces: There are practical limits to bringing electrical power into the buildings, distributing it to the equipment, and removing the heat that the equipment generates. Figure 17.1 shows a typical cloud data center. Each rack consists of more than 25 computers (each with multiple CPUs), with the exact number depending on the power and cooling available. The data center consists of rows and rows of such racks, with high-speed network switches connecting the racks. Cloud data centers are one reason why energy efficiency (a topic discussed in Chapter 6) has become a critical quality attribute in some applications.

When you access a cloud via a public cloud provider, you are actually accessing data centers scattered around the globe. The cloud provider organizes its data centers into *regions*. A cloud region is both a logical and a physical construct. Since the services you develop and deploy to the cloud are accessed over the Internet, cloud regions can help you be sure that the service is physically close to its users, thereby reducing the network delay to access the service. Also, some regulatory constraints, such as the General Data Protection Regulation (GDPR), may restrict the transmission of certain types of data across national borders, so cloud regions help cloud providers comply with these regulations.

A cloud region has many data centers that are physically distributed and have different sources for electrical power and Internet connectivity. The data centers within a region are grouped into *availability zones*, such that the probability of all data centers in two different availability zones failing at the same time is extremely low.

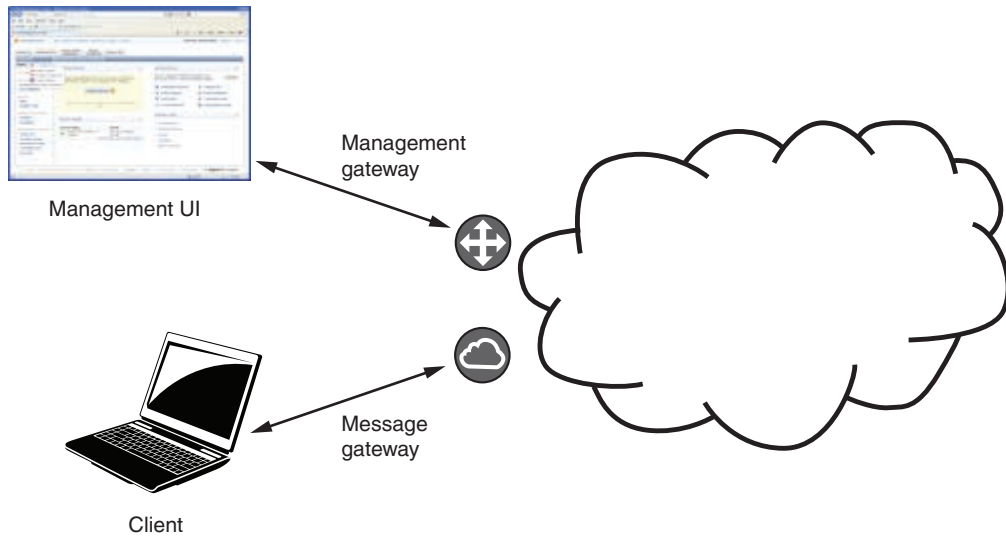


FIGURE 17.1 A cloud data center

Choosing the cloud region that your service will run on is an important design decision. When you ask to be provided with a new virtual machine (VM) that runs in the cloud, you may specify which region the VM will run on. Sometimes the availability zone may be chosen automatically, but you often will want to choose the zone yourself, for availability and business continuity reasons.

All access to a public cloud occurs over the Internet. There are two main gateways into a cloud: a management gateway and a message gateway (Figure 17.2). Here we will focus on the management gateway; we discussed message gateways in Chapter 15.

Suppose you wish to have a VM allocated for you in the cloud. You send a request to the management gateway asking for a new VM instance. This request has many parameters, but three essential parameters are the cloud region where the new instance will run, the instance type (e.g., CPU and memory size), and the ID of a VM image. The management gateway is responsible for tens of thousands of physical computers, and each physical computer has a hypervisor that manages the VMs on it. So, the management gateway will identify a hypervisor that can manage an additional VM of the type you have selected by asking, Is there enough unallocated CPU and memory capacity available on that physical machine to meet

**FIGURE 17.2** Gateways into a public cloud

your needs? If so, it will ask that hypervisor to create an additional VM; the hypervisor will perform this task and return the new VM's IP address to the management gateway. The management gateway then sends that IP address to you. The cloud provider ensures that enough physical hardware resources are available in its data centers so that your request will never fail due to insufficient resources.

The management gateway returns not only the IP address for the newly allocated VM, but also a hostname. The hostname returned after allocating a VM reflects the fact that the IP address has been added to the cloud Domain Name System (DNS). Any VM image can be used to create the new VM instance; that is, the VM image may comprise a simple service or be just one step in the deployment process to create a complex system.

The management gateway performs other functions in addition to allocating new VMs. It supports collecting billing information about the VM, and it provides the capability to monitor and destroy the VM.

The management gateway is accessed through messages over the Internet to its API. These messages can come from another service, such as a deployment service, or they can be generated from a command-line program on your computer (allowing you to script operations). The management gateway can also be accessed through a web-based application operated by the cloud service provider, although this kind of interactive interface is not efficient for more than the most trivial operations.

17.2 Failure in the Cloud

When a data center contains tens of thousands of physical computers, it is almost a certainty that one or more will fail every day. Amazon reports that in a data center with around 64,000 computers, each with two spinning disk drives, approximately 5 computers and 17 disks will fail every day. Google reports similar statistics. In addition to computer and disk failures, network switches can fail; the data center can overheat, causing all the computers to fail; or some natural disaster may bring the entire data center down. Although your cloud provider will have relatively few total outages, the physical computer on which your specific VM is running may fail. If availability is important to your service, you need to think carefully about what level of availability you wish to achieve and how to achieve it.

We'll discuss two concepts especially relevant to failure in the cloud: timeouts and long tail latency.

Timeouts

Recall from Chapter 4 that timeout is a tactic for availability. In a distributed system, timeouts are used to detect failure. There are several consequences of using timeouts:

- Timeouts can't distinguish between a failed computer or broken network connection and a slow reply to a message that exceeds the timeout period. This will cause you to label some slow responses as failures.
- A timeout will not tell you where the failure or slowness occurs.
- Many times, a request to a service triggers that service to make requests to other services, which make more requests. Even if each of the responses in this chain has a latency that is close to (but slower than) the expected average response time, the overall latency may (falsely) suggest a failure.

A timeout—a decision that a response has taken too long—is commonly used to detect a failure. A timeout cannot isolate whether the failure is due to a failure in the software of the requested service, the virtual or physical machine that the service is running on, or the network connection to the service. In most cases, the cause is not important: You made a request, or you were expecting a periodic keep-alive or heartbeat message, and did not receive a timely response, and now you need to take action to remedy this.

This seems simple, but in real systems it can be complicated. There is usually a cost, such as a latency penalty, for a recovery action. You may need to start a new VM, which could take minutes before it is ready to accept new requests. You may need to establish a new session with a different service instance, which may affect the usability of your system. The response times in cloud systems can show considerable variations. Jumping to a conclusion that there was a failure, when there was actually just a temporary delay, may add a recovery cost when it isn't necessary.

Distributed system designers generally parameterize the timeout detection mechanism so that it can be tuned for a system or infrastructure. One parameter is the timeout interval—how

long the system should wait before deciding that a response has failed. Most systems do not trigger failure recovery after a single missed response. Instead, the typical approach is to look for some number of missed responses over a longer time interval. The number of missed responses is a second parameter for the timeout mechanism. For example, a timeout might be set to 200 milliseconds, and failure recovery is triggered after 3 missed messages over a 1-second interval.

For systems running with a single data center, timeouts and thresholds can be set aggressively, since network delays are minimal and missed responses are likely due to software crashes or hardware failures. In contrast, for systems operating over a wide area network, a cellular radio network, or even a satellite link, more thought should be put into setting the parameters, as these systems may experience intermittent but longer network delays. In such cases, the parameters may be relaxed to reflect this possibility and avoid triggering unnecessary recovery actions.

Long Tail Latency

Regardless of whether the cause is an actual failure or just a slow response, the response to your original request may exhibit what is called long tail latency. Figure 17.3 shows a histogram of the latency of 1,000 “launch instance” requests to Amazon Web Services (AWS). Notice that some requests took a very long time to satisfy. When evaluating measurement sets such as this one, you must be careful which statistic you use to characterize the data set. In this case, the histogram peaks at a latency of 22 seconds; however, the average latency over all the measurements is 28 seconds, and the median latency (half the requests are completed with latency less than this value) is 23 seconds. Even after a latency of 57 seconds, 5 percent of the requests have still not been completed (i.e., the 95th percentile is 57 seconds). So, although the mean latency for each service-to-service request to a cloud-based service may be within tolerable limits, a reasonable number of these requests can have much greater latency—in this case, from 2 to 10 times longer than the average. These are the measurements in the long tail on the right side of the histogram.

Long tail latencies are a result of congestion or failure somewhere in the path of the service request. Many factors may contribute to congestion—server queues, hypervisor scheduling, or others—but the cause of the congestion is out of your control as a service developer. Your monitoring techniques and your strategies to achieve your required performance and availability must reflect the reality of a long tail distribution.

Two techniques to handle long tail problems are hedged requests and alternative requests.

- *Hedged requests.* Make more requests than are needed and then cancel the requests (or ignore responses) after sufficient responses have been received. For example, suppose 10 instances of a microservice (see Chapter 5) are to be launched. Issue 11 requests and after 10 have completed, terminate the request that has not responded yet.
- *Alternative requests.* A variant of the hedged request technique is called alternative request. In the just-described scenario, issue 10 requests. When 8 requests have completed, issue 2 more, and when a total of 10 responses have been received, cancel the 2 requests that are still remaining.

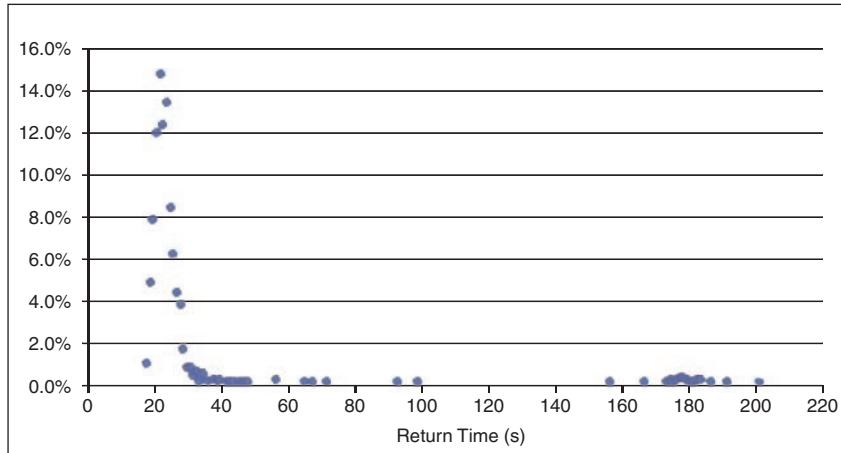


FIGURE 17.3 Long tail distribution of 1,000 “launch instance” requests to AWS

17.3 Using Multiple Instances to Improve Performance and Availability

If a service hosted in a cloud receives more requests than it can process within the required latency, the service becomes overloaded. This can occur because there is an insufficient I/O bandwidth, CPU cycles, memory, or some other resource. In some cases, you can resolve a service overload issue by running the service in a different instance type that provides more of the resource that is needed. This approach is simple: The design of the service does not change; instead, the service just runs on a larger virtual machine. Called vertical scaling or scaling up, this approach corresponds to the increased resources performance tactic from Chapter 9.

There are limits to what can be achieved with vertical scaling. In particular, there may not be a large enough VM instance type to support the workload. In this case, horizontal scaling or scaling out provides more resources of the type needed. Horizontal scaling involves having multiple copies of the same service and using a load balancer to distribute requests among them—equivalent to the maintain multiple copies of computations tactic and the load balancer pattern, respectively, from Chapter 9.

Distributed Computing and Load Balancers

Load balancers can be standalone systems, or they can be bundled with other functions. A load balancer must be very efficient because it sits in the path of every message from a client to a

service, and even when it is packaged with other functions, it is logically isolated. Here, we divide our discussion into two main aspects: how load balancers work and how services that sit behind a load balancer must be designed to manage the service state. Once we understand these processes, we can explore the management of the system's health and how load balancers can improve its availability.

A load balancer solves the following problem: There is a single instance of a service running on a VM or in a container, and too many requests are arriving at this instance for it to provide acceptable latency. One solution is to have multiple instances of the service and distribute the requests among them. The distribution mechanism in such a case is a separate service—the load balancer. Figure 17.4 shows a load balancer distributing requests between two VM (service) instances. The same discussion would apply if there were two container instances. (Containers were discussed in Chapter 16.)

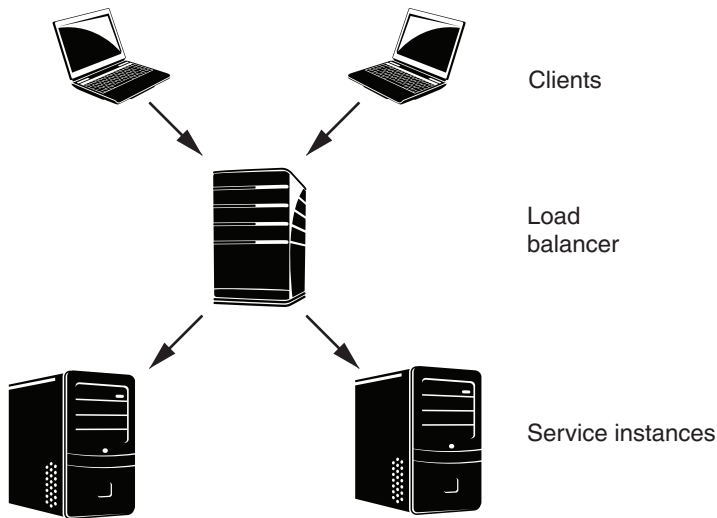


FIGURE 17.4 A load balancer distributing requests from two clients to two service instances

You may be wondering what constitutes “too many requests” and “reasonable response time.” We’ll come back to these questions later in this chapter when we discuss autoscaling. For now, let’s focus on how a load balancer works.

In Figure 17.4, each request is sent to a load balancer. For the purposes of our discussion, suppose the load balancer sends the first request to instance 1, the second request to instance 2, the third request back to instance 1, and so forth. This sends half of the requests to each instance, *balancing the load* between the two instances—hence the name.

Some observations about this simple example of a load balancer:

- The algorithm we provided—alternate the messages between the two instances—is called “round-robin.” This algorithm balances the load uniformly across the service instances only if every request consumes roughly the same resources in its response. Other algorithms for distributing the messages exist for cases where the resource consumption needed to process requests varies.
- From a client’s perspective, the service’s IP address is actually the address of the load balancer. This address may be associated with a hostname in the DNS. The client does not know, or need to know, how many instances of the service exist or the IP address of any of those service instances. This makes the client resilient to changing this information—an example of using an intermediary, as discussed in Chapter 8.
- Multiple clients may coexist. Each client sends its messages to the load balancer, which does not care about the message source. The load balancer distributes the messages as they arrive. (We’ll ignore the concept called “sticky sessions” or “session affinity” for the moment.)
- Load balancers may get overloaded. In this case, the solution is to balance the load of the load balancer, sometimes referred to as global load balancing. That is, a message goes through a hierarchy of load balancers before arriving at the service instance.

So far, our discussion of load balancers has focused on increasing the amount of work that can be handled. Here, we will consider how load balancers also serve to increase the availability of services.

Figure 17.4 shows messages from clients passing through the load balancer, but does not show the return messages. Return messages go directly from the service instances to the clients (determined by the “from” field in the IP message header), bypassing the load balancer. As a consequence, the load balancer has no information about whether a message was processed by a service instance, or how long it took to process a message. Without additional mechanisms, the load balancer would not know whether any service instance was alive and processing, or if any instance or all instances had failed.

Health checks are a mechanism that allow the load balancer to determine whether an instance is performing properly. This is the purpose of the “fault detection” category of availability tactics from Chapter 4. The load balancer will periodically check the health of the instances assigned to it. If an instance fails to respond to a health check, it is marked as unhealthy and no further messages are sent to it. Health checks can consist of pings from the load balancer to the instance, opening a TCP connection to the instance or even sending a message for processing. In the latter case, the return IP address is the address of the load balancer.

It is possible for an instance to move from healthy to unhealthy, and back again. Suppose, for example, that the instance has an overloaded queue. When initially contacted, it may not respond to the load balancer’s health check, but once the queue has been drained, it may be ready to respond again. For this reason, the load balancer checks multiple times before moving an instance to an unhealthy list, and then periodically checks the unhealthy list to determine whether an instance is again responding. In other cases, a hard failure or crash may cause the failed instance to restart and re-register with the load balancer, or a new replacement

instance may be started and registered with the load balancer, so as to maintain overall service delivery capacity.

A load balancer with health checking improves availability by hiding the failure of a service instance from clients. The pool of service instances can be sized to accommodate some number of simultaneous service instance failures while still providing enough overall service capacity to handle the required volume of client requests within the desired latency. However, even when using health checking, a service instance might sometimes start processing a client request but never return a response. Clients must be designed so that they resend a request if they do not receive a timely response, allowing the load balancer to distribute the request to a different service instance. Services must correspondingly be designed such that multiple identical requests can be accommodated.

State Management in Distributed Systems

State refers to information internal to a service that affects the computation of a response to a client request. State—or, more precisely, the collection of the values of the variables or data structures that store the state—depends on the history of requests to the service.

Management of state becomes important when a service can process more than one client request at the same time, either because a service instance is multi-threaded, because there are multiple service instances behind a load balancer, or both. The key issue is where the state is stored. The three options are:

1. The history maintained in each service instance, in which case the services are described as “stateful.”
2. The history maintained in each client, in which case the services are described as “stateless.”
3. The history persists outside the services and clients, in a database, in which case the services are described as “stateless.”

Common practice is to design and implement services to be stateless. Stateful services lose their history if they fail, and recovering that state can be difficult. Also, as we will see in the next section, new service instances may be created, and designing services to be stateless allows a new service instance to process a client request and produce the same response as any other service instance.

In some cases, it may be difficult or inefficient to design a service to be stateless, so we might want a series of messages from a client to be processed by the same service instance. We can accomplish this by having the first request in the series be handled by the load balancer and distributed to a service instance, and then allowing the client to establish a session directly with that service instance and subsequent requests to bypass the Load balancer. Alternatively, some load balancers can be configured to treat certain types of requests as sticky, which causes the load balancer to send subsequent requests from a client to the same service instance that handled the last message from this client. These approaches—direct sessions and sticky messages—should be used only under special circumstances because of the possibility of failure of the instance and the risk that the instance to which the messages are sticking may become overloaded.

Frequently, there is a need to share information across all instances of a service. This information may consist of state information, as discussed earlier, or it may be other information that is needed for the service instances to work together efficiently—for example, the IP address of the load balancer for the service. A solution exists to manage relatively small amounts of information shared among all instances of a service, as discussed next.

Time Coordination in a Distributed System

Determining exactly what time it is might seem to be a trivial task, but it is actually not easy. Hardware clocks found in computers will gain or lose one second about every 12 days. If your computing device is out in the world, so to speak, it may have access to a time signal from a Global Positioning System (GPS) satellite, which provides a time accurate to within 100 nanoseconds or less.

Having two or more devices agree on what time it is can be even more challenging. The clock readings from two different devices on a network *will* be different. The Network Time Protocol (NTP) is used to synchronize time across different devices that are connected over a local or wide area network. It involves exchanging messages between a time server and client devices to estimate the network latency, and then applying algorithms to synchronize a client device's clock to the time server. NTP is accurate to around 1 millisecond on local area networks and around 10 milliseconds on public networks. Congestion can cause errors of 100 milliseconds or more.

Cloud service providers provide very precise time references for their time servers. For example, Amazon and Google use atomic clocks, which have virtually unmeasurable drift. Both can therefore provide an extremely accurate answer to the question, “What time is it?” Of course, what time it is when you get the answer is another matter.

Happily, for many purposes, almost-accurate time is good enough. However, as a practical matter, you should assume some level of error exists between the clock readings on two different devices. For this reason, most distributed systems are designed so that time synchronization among devices is not required for applications to function correctly. You can use device time to trigger periodic actions, to timestamp log entries, and for a few other purposes where accurate coordination with other devices is not necessary.

Also happily, for many purposes, it is more important to know the order of events rather than the time at which those events occurred. Trading decisions on the stock market fall into this category, as do online auctions of any form. Both rely on processing packets in the same order in which they were transmitted.

For critical coordination across devices, most distributed systems use mechanisms such as vector clocks (which are not really clocks, but rather counters that trace actions as they propagate through the services in an application) to determine whether one event happened before another event, rather than comparing times. This ensures that the application can apply the actions in the correct order. Most of the data coordination mechanisms that we discuss in the next section rely on this kind of ordering of actions.

For an architect, successful time coordination involves knowing whether you really need to rely on actual clock times, or whether ensuring correct sequencing suffices. If the former is important, then know your accuracy requirements and choose a solution accordingly.

Data Coordination in a Distributed System

Consider the problem of creating a resource lock to be shared across distributed machines. Suppose some critical resource is being accessed by service instances on two distinct VMs running on two distinct physical computers. We assume this critical resource is a data item—for example, your bank account balance. Changing the account balance requires reading the current balance, adding or subtracting the transaction amount, and then writing back the new balance. If we allow both service instances to operate independently on this data item, there is the possibility of a race condition, such as two simultaneous deposits overwriting each other. The standard solution in this situation is to lock the data item, so that a service cannot access your account balance until it gets the lock. We avoid a race condition because service instance 1 is granted a lock on your bank account and can work in isolation to make its deposit until it yields the lock. Then service instance 2, which has been waiting for the lock to become available, can lock the bank account and make the second deposit.

This solution using a shared lock is easy to implement when the services are processes running on a single machine, and requesting and releasing a lock are simple memory access operations that are very fast and atomic. However, in a distributed system, two problems arise with this scheme. First, the two-phase commit protocol traditionally used to acquire a lock requires multiple messages to be transmitted across the network. In the best case, this just adds delay to the actions, but in the worst case, any of these messages may fail to be delivered. Second, service instance 1 may fail after it has acquired the lock, preventing service instance 2 from proceeding.

The solution to these problems involves complicated distributed coordination algorithms. Leslie Lamport, quoted at the beginning of the chapter, developed one of the first such algorithms, which he named “Paxos.” Paxos and other distributed coordination algorithms rely on a consensus mechanism to allow participants to reach agreement even when computer or network failures occur. These algorithms are notoriously complicated to design correctly, and even implementing a proven algorithm is difficult due to subtleties in programming language and network interface semantics. In fact, distributed coordination is one of those problems that you should *not* try to solve yourself. Using one of the existing solution packages, such as Apache Zookeeper, Consul, and etcd, is almost always a better idea than rolling your own. When service instances need to share information, they store it in a service that uses a distributed coordination mechanism to ensure that all services see the same values.

Our last distributed computing topic is the automatic creation and destruction of instances.

Autoscaling: Automatic Creation and Destruction of Instances

Consider a traditional data center, where your organization owns all the physical resources. In this environment, your organization needs to allocate enough physical hardware to a system to handle the peak of the largest workload that it has committed to process. When the workload is less than the peak, some (or much) of the hardware capacity allocated to the system is idle. Now compare this to a cloud environment. Two of the defining features of the cloud are that you pay only for the resources you requisition and that you can easily and quickly add and

release resources (elasticity). Together, these features allow you to create systems that have the capacity to handle your workload, and you don't pay for any excess capacity.

Elasticity applies at different time scales. Some systems see relatively stable workloads, in which case you might consider manually reviewing and changing resource allocation on a monthly or quarterly time scale to match this slowly changing workload. Other systems see more dynamic workloads with rapid increases and decreases in the rate of requests, and so need a way to automate adding and releasing service instances.

Autoscaling is an infrastructure service that automatically creates new instances when needed and releases surplus instances when they are no longer needed. It usually works in conjunction with load balancing to grow and shrink the pool of service instances behind a load balancer. Autoscaling containers is slightly different from autoscaling VMs. We discuss autoscaling VMs first and then discuss the differences when containers are being autoscaled.

Autoscaling VMs

Returning to Figure 17.4, suppose that the two clients generate more requests than can be handled by the two service instances shown. Autoscaling creates a third instance, based on the same virtual machine image that was used for the first two instances. The new instance is registered with the load balancer so that subsequent requests are distributed among three instances rather than two. Figure 17.5 shows a new component, the autoscaler, that monitors and autoscales the utilization of the server instances. Once the autoscaler creates a new service instance, it notifies the load balancer of the new IP address so that the load balancer can distribute requests to the new instance, in addition to the requests it distributes to the other instances.

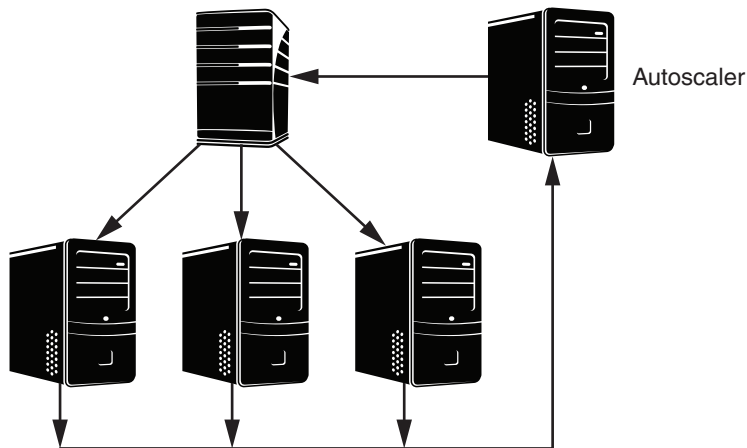


FIGURE 17.5 An autoscaler monitoring the utilization

Because the clients do not know how many instances exist or which instance is serving their requests, autoscaling activities are invisible to service clients. Furthermore, if the client request rate decreases, an instance can be removed from the load balancer pool, halted, and deallocated, again without the client's knowledge.

As an architect of a cloud-based service, you can set up a collection of rules for the autoscaler that govern its behavior. The configuration information you provide to the autoscaler includes the following items:

- The VM image to be launched when a new instance is created, and any instance configuration parameters required by the cloud provider, such as security settings
- The CPU utilization threshold (measured over time) for any instance above which a new instance is launched
- The CPU utilization threshold (measured over time) for any instance below which an existing instance is shut down
- The network I/O bandwidth thresholds (measured over time) for creating and deleting instances
- The minimum and maximum number of instances you want in this group

The autoscaler does not create or remove instances based on instantaneous values of the CPU utilization or network I/O bandwidth metrics, for two reasons. First, these metrics have spikes and valleys and are meaningful only when averaged over a reasonable time interval. Second, allocating and starting a new VM takes a relatively long time, on the order of minutes. The VM image must be loaded and connected to the network, and the operating system must boot before it will be ready to process messages. Consequently, autoscaler rules typically are of the form, "Create a new VM when CPU utilization is above 80 percent for 5 minutes."

In addition to creating and destroying VMs based on utilization metrics, you can set rules to provide a minimum or maximum number of VMs or to create VMs based on a time schedule. During a typical week, for example, load may be heavier during work hours; based on this knowledge, you can allocate more VMs before the beginning of a workday and remove some after the workday is over. These scheduled allocations should be based on historical data about the pattern of usage of your services.

When the autoscaler removes an instance, it cannot just shut down the VM. First, it must notify the load balancer to stop sending requests to the service instance. Next, because the instance may be in the process of servicing a request, the autoscaler must notify the instance that it should terminate its activities and shut down, after which it can be destroyed. This process is called "draining" the instance. As a service developer, you are responsible for implementing the appropriate interface to receive instructions to terminate and drain an instance of your service.

Autoscaling Containers

Because containers are executing on runtime engines that are hosted on VMs, scaling containers involves two different types of decisions. When scaling VMs, an autoscaler decides that additional VMs are required, and then allocates a new VM and loads it with the appropriate software. Scaling containers means making a two-level decision. First, decide that an

additional container (or Pod) is required for the current workload. Second, decide whether the new container (or Pod) can be allocated on an existing runtime engine instance or whether a new instance must be allocated. If a new instance must be allocated, you need to check whether a VM with sufficient capacity is available or if an additional VM needs to be allocated.

The software that controls the scaling of containers is independent of the software that controls the scaling of VMs. This allows the scaling of containers to be portable across different cloud providers. It is possible that the evolution of containers will integrate the two types of scaling. In such a case, you should be aware that you may be creating a dependency between your software and the cloud provider that could be difficult to break.

17.4 Summary

The cloud is composed of distributed data centers, with each data center containing tens of thousands of computers. It is managed through a management gateway that is accessible over the Internet and is responsible for allocating, deallocating, and monitoring VMs, as well as measuring resource usage and computing billing.

Because of the large number of computers in a data center, failure of a computer in such a center happens quite frequently. You, as an architect of a service, should assume that at some point, the VMs on which your service is executing will fail. You should also assume that your requests for other services will exhibit a long tail distribution, such that as many as 5 percent of your requests will take 5 to 10 times longer than the average request. Thus you must be concerned about the availability of your service.

Because single instances of your service may not be able to satisfy all requests in a timely manner, you may decide to run multiple VMs or containers containing instances of your service. These multiple instances sit behind a load balancer. The load balancer receives requests from clients and distributes the requests to the various instances.

The existence of multiple instances of your service and multiple clients has a significant impact on how you handle state. Different decisions on where to keep the state will lead to different results. The most common practice is to keep services stateless, because stateless services allow for easier recovery from failure and easier addition of new instances. Small amounts of data can be shared among service instances by using a distributed coordination service. Distributed coordination services are complicated to implement, but several proven open source implementations are available for your use.

The cloud infrastructure can automatically scale your service by creating new instances when demand grows and removing instances when demand shrinks. You specify the behavior of the autoscaler through a set of rules giving the conditions for the creation or deletion of instances.

17.5 For Further Reading

More details about how networks and virtualization work can be found in [Bass 19].

The long tail latency phenomenon in the context of the cloud was first identified in [Dean 13].

Paxos was first presented by [Lamport 98]. People found the original article difficult to understand, but a very thorough description of Paxos can be found in Wikipedia—[https://en.wikipedia.org/wiki/Paxos_\(computer_science\)](https://en.wikipedia.org/wiki/Paxos_(computer_science)). Around the same time, Brian Oki and Barbara Liskov independently developed and published an algorithm called Viewstamped Replication that was later shown to be equivalent to Lamport’s Paxos [Oki 88].

A description of Apache Zookeeper can be found at <https://zookeeper.apache.org/>. Consul can be found at <https://www.consul.io/>, and etcd can be found at <https://etcd.io/>

A discussion of different types of load balancers can be found at <https://docs.aws.amazon.com/ECS/latest/developerguide/load-balancer-types.html>.

Time in a distributed system is discussed in <https://medium.com/coinmonks/time-and-clocks-and-ordering-of-events-in-a-distributed-system-cdd3f6075e73>.

Managing state in a distributed system is discussed in <https://conferences.oreilly.com/software-architecture/sa-ny-2018/public/schedule/detail/64127>.

17.6 Discussion Questions

1. A load balancer is a type of intermediary. Intermediaries enhance modifiability but detract from performance, yet a load balancer exists to increase performance. Explain this apparent paradox.
2. A context diagram displays an entity and other entities with which it communicates. It separates the responsibilities allocated to the chosen entity from those responsibilities allocated to other entities, and shows the interactions needed to accomplish the chosen entity’s responsibilities. Draw a context diagram for a load balancer.
3. Sketch the set of steps to allocate a VM within a cloud and display its IP address.
4. Research the offerings of a major cloud provider. Write a set of rules that would govern the autoscaling for a service that you would implement on this cloud.
5. Some load balancers use a technique called message queues. Research message queues and describe the differences between load balancers with and without message queues.

18



Mobile Systems

With Yazid Hamdi and Greg Hartman

*The telephone will be used to inform people that
a telegram has been sent.*
—Alexander Graham Bell

So, what did Alexander Graham Bell know, anyway? Mobile systems, including and especially phones, are ubiquitous in our world today. Besides phones, they include trains, planes, and automobiles; they include ships and satellites, entertainment and personal computing devices, and robotic systems (autonomous or not); they include essentially any system or device that has no permanent connection to a continuous abundant power source.

A mobile system has the ability to be in movement while continuing to deliver some or all of its functionality. This makes dealing with some of its characteristics a different matter from dealing with fixed systems. In this chapter we focus on five of those characteristics:

1. *Energy.* Mobile systems have limited sources of power and must be concerned with using power efficiently.
2. *Network connectivity.* Mobile systems tend to deliver much of their functionality by exchanging information with other devices while they are in motion. They must therefore connect to those devices, but their mobility makes these connections tricky.
3. *Sensors and actuators.* Mobile systems tend to gain more information from sensors than fixed systems do, and they often use actuators to interact with their environment.
4. *Resources.* Mobile systems tend to be more resource-constrained than fixed systems. For one thing, they are often quite small, such that physical packaging becomes a limiting factor. For another, their mobility often makes weight a factor. Mobile devices that must be small and lightweight have limits on the resources they can provide.
5. *Life cycle.* Testing mobile systems differs from the testing of other systems. Deploying new versions also introduces some special issues.

When designing a system for a mobile platform, you must deal with a large number of domain-specific requirements. Self-driving automobiles and autonomous drones must be safe;

smartphones must provide an open platform for a variety of vastly different applications; entertainment systems must work with a wide range of content formats and service providers. In this chapter, we'll focus on the characteristics shared by many (if not all) mobile systems that an architect must consider when designing a system.

18.1 Energy

In this section, we focus on the architectural concerns most relevant to managing the energy of mobile systems. For many mobile devices, their source of energy is a battery with a very finite capacity for delivering that energy. Other mobile devices, such as cars and planes, run on the power produced by generators, which in turn may be powered by engines that run on fuel—again, a finite resource.

The Architect's Concerns

The architect must be concerned with monitoring the power source, throttling energy usage, and tolerating loss of power. We elaborate on these concerns in the next three subsections.

Monitoring the Power Source

In Chapter 6 on energy efficiency, we introduced a category of tactics called “resource monitoring” for monitoring the usage of computational resources, which are consumers of energy. In mobile systems, we need to monitor the energy source, so that we can initiate appropriate behavior when the energy available becomes low. Specifically, in a mobile device powered by a battery, we may need to inform a user that the battery level is low, put the device into battery-saving mode, alert applications to the imminent shutdown of the device so they can prepare for a restart, and determine the power usage of each application.

All of these uses depend on monitoring the current state of the battery. Most laptops or smartphones use a smart battery as a power source. A smart battery is a rechargeable battery pack with a built-in battery management system (BMS). The BMS can be queried to get the current state of the battery. Other mobile systems might use a different battery technology, but all have some equivalent capability. For the purposes of this section, we will assume that the reading identifies the percentage of capacity left.

Battery-powered mobile systems include a component, often in the kernel of the operating system, that knows how to interact with the BMS and can return the current battery capacity on request. A battery manager is responsible for periodically querying that component to retrieve the state of the battery. This enables the system to inform the user of the energy status and trigger the battery-saving mode, if necessary. To inform the applications that the device is about to shut down, the applications must register with the battery manager.

Two characteristics of batteries change as they age: the maximum battery capacity and the maximum sustained current. An architect must allow for managing consumption within

the changing envelope of available power so that the device still performs at an acceptable level. Monitoring plays a role in generator-equipped systems as well, since some applications may need to be shut down or put on standby when generator output is low. The battery manager can also determine which applications are currently active and what their energy consumption is. The overall percentage of the change in battery capacity can then be estimated based on this information.

Of course, the battery manager itself utilizes resources—memory and CPU time. The amount of CPU time consumed by the battery manager can be managed by adjusting the query interval.

Throttling Energy Usage

Energy usage can be reduced by either terminating or degrading portions of the system that consume energy; this is the throttle usage tactic described in Chapter 6. The specifics of how this is done depend on the individual elements of the system, but a common example is reducing the brightness or the refresh rate of the display on a smartphone. Other techniques for throttling energy usage include reducing the number of active cores of the processor, reducing the clock rate of the cores, and reducing the frequency of sensor readings. For example, instead of asking for GPS location data every few seconds, ask for it every minute or so. Instead of relying on different location data sources such as GPS and cell towers, use just one of those.

Tolerating a Loss of Power

Mobile systems should gracefully tolerate power failures and restarts. For example, a requirement of such a system could be that following restoration of power, the system is back on and working in the nominal mode within 30 seconds. This requirement implies different requirements apply to different portions of the system, such as the following:

- Example hardware requirements:
 - The system's computer does not suffer permanent damage if power is cut at any time.
 - The system's computer (re)starts the OS robustly whenever sufficient power is provided.
 - The system's OS has the software scheduled to launch as soon as the OS is ready.
- Example software requirements:
 - The runtime environment can be killed at any moment without affecting the integrity of the binaries, configurations, and operational data in permanent storage, and while keeping the state consistent after a restart (whether that is a reset or a resume).
 - Applications need a strategy to deal with data that arrives while the application is inoperative.
 - The runtime can start after a failure so that the startup time, from system power on to the software being in a ready state, is less than a specified period.

18.2 Network Connectivity

In this section, we focus on the architectural concerns most relevant to network connectivity of mobile systems. We will focus on wireless communication between the mobile platform and the outside world. The network might be used to control the device or to send and receive information.

Wireless networks are categorized based on the distance over which they operate.

- *Within 4 centimeters.* Near Field Communication (NFC) is used for keycards and contactless payment systems. Standards in this area are being developed by the GSM Alliance.
- *Within 10 meters.* The IEEE 802.15 family of standards covers this distance. Bluetooth and Zigbee are common protocols within this category
- *Within 100 meters.* The IEEE 802.11 family of standards (Wi-Fi) is used within this distance.
- *Within several kilometers.* The IEEE 802.16 standards cover this distance. WiMAX is the commercial name for the IEEE 802.16 standards.
- *More than several kilometers.* This is achieved by cellular or satellite communication.

Within all of these categories, the technologies and the standards are evolving rapidly.

The Architect's Concerns

Designing for communication and network connectivity requires the architect to balance a large number of concerns, including the following:

- *Number of communication interfaces to support.* With all of the different protocols and their rapid evolution, it is tempting for an architect to include all possible kinds of network interfaces. The goal when designing a mobile system is just the opposite: Only the strictly required interfaces should be included to optimize power consumption, heat generation, and space allocation.
- *Movement from one protocol to another.* Despite the need to take a minimalist approach to interfaces, the architect must account for the possibility that during the course of a session, the mobile system may move from an environment that supports one protocol to an environment that supports another protocol. For example, a video may be streaming on Wi-Fi, but then the system may move to an environment without Wi-Fi and the video will be received over a cellular network. Such transitions should be seamless to the user.
- *Choosing the appropriate protocol dynamically.* In the event that multiple protocols are simultaneously available, the system should choose a protocol dynamically based on factors such as cost, bandwidth, and power consumption.
- *Modifiability.* Given the large number of protocols and their rapid evolution, it is likely that over the lifetime of a mobile system, new or alternative protocols will need to be supported. The system should be designed to support changes or replacements in the elements of the system involved in communication.

- *Bandwidth.* The information to be communicated to other systems should be analyzed for distance, volume, and latency requirements so that appropriate architectural choices can be made. The protocols all vary in terms of those qualities.
- *Intermittent/limited/no connectivity.* Communication may be lost while the device is in motion (e.g., a smartphone going through a tunnel). The system should be designed so that data integrity is maintained in case of a loss of connectivity, and computation can be resumed without loss of consistency when connectivity returns. The system should be designed to deal gracefully with limited connectivity or even no connectivity. Degraded and fallback modes should be dynamically available to deal with such situations.
- *Security.* Mobile devices are particularly vulnerable to spoofing, eavesdropping, and man-in-the-middle attacks, so responding to such attacks should be part of the architect's concerns.

18.3 Sensors and Actuators

A *sensor* is a device that detects the physical characteristics of its environment and translates those characteristics into an electronic representation. A mobile device gathers environmental data either to guide its own operation (such as the altimeter in a drone), or to report that data back to a user (such as the magnetic compass in your smartphone).

A *transducer* senses external electronic impulses and converts them into a more usable internal form. In this section, we will use the term “sensor” to encompass transducers as well, and assume the electronic representation is digital.

A *sensor hub* is a coprocessor that helps integrate data from different sensors and process it. A sensor hub can help offload these jobs from a product's main CPU, thereby saving battery consumption and improving performance.

Inside the mobile system, software will abstract some characteristics of the environment. This abstraction may map directly to a sensor, such as with measurement of temperature or pressure, or it may integrate the input of several sensors, such as pedestrians identified in a self-driving automobile controller.

An *actuator* is the reverse of a sensor: It takes a digital representation as input and causes some action in the environment. The lane keep assist feature in an automobile utilizes actuators, as does an audio alert from your smartphone.

The Architect's Concerns

An architect has several concerns with respect to sensors:

- How to create an accurate representation of the environment based on the sensor inputs.
- How the system should respond to that representation of the environment.
- Security and privacy of the sensor data and actuator commands.
- Degraded operation. If sensors fail or become unreadable, the system should enter a degraded mode. For example, if GPS readings are not available in tunnels, the system can use dead reckoning techniques to estimate location.

The representation of the environment that is created and acted upon by a system is domain specific, as is the appropriate approach to degraded operation. We discussed security and privacy in detail in Chapter 8, but here we will focus on only the first concern: creating an accurate representation of the environment based on the data returned by the sensors. This is performed using the sensor stack—a confederation of devices and software drivers that help turn raw data into interpreted information about the environment.

Different platforms and domains tend to have their own sensor stacks, and sensor stacks often come with their own frameworks to help deal with the devices more easily. Over time, sensors are likely to encompass more and more functionality; in turn, the functions of a particular stack will change over time. Here, we enumerate some of the functions that must be achieved in the stack regardless of where a particular decomposition may have placed them:

- *Reading raw data.* The lowest level of the stack is a software driver to read the raw data. The driver reads the sensor either directly or, in the case where the sensor is a portion of a sensor hub, through the hub. The driver gets a reading from the sensor periodically. The period frequency is a parameter that will influence both the processor load from reading and processing the sensor and the accuracy of the created representation.
- *Smoothing data.* Raw data usually has a great deal of noise or variation. Voltage variations, dirt or grime on a sensor, and a myriad of other causes can make two successive readings of a sensor differ. Smoothing is a process that uses a series of measurements over time to produce an estimate that tends to be more accurate than single readings. Calculating a moving average and using a Kalman filter are two of the many techniques for smoothing data.
- *Converting data.* Sensors can report data in many formats—from voltage readings in millivolts to altitude above sea level in feet to temperature in degrees Celsius. It is possible, however, that two different sensors measuring the same phenomenon might report their data in different formats. The converter is responsible for converting readings from whatever form is reported by the sensor into a common form meaningful to the application. As you might imagine, this function may need to deal with a wide variety of sensors.
- *Sensor fusion.* Sensor fusion combines data from multiple sensors to build a more accurate or more complete or more dependable representation of the environment than would be possible from any individual sensor. For example, how does an automobile recognize pedestrians in its path or likely to be in its path by the time it gets there, day or night, in all kinds of weather? No single sensor can accomplish this feat. Instead, the automobile must intelligently combine inputs from sensors such as thermal imagers, radar, lidar, and cameras.

18.4 Resources

In this section, we discuss computing resources from the perspective of their physical characteristics. For example, in devices where energy comes from batteries, we need to be concerned

with battery volume, weight, and thermal properties. The same holds true for resources such as networks, processors, and sensors.

The tradeoff in the choice of resources is between the contribution of the particular resource under consideration and its volume, weight, and cost. Cost is always a factor. Costs include both the manufacturing costs and nonrecurring engineering costs. Many mobile systems are manufactured by the millions and are highly price-sensitive. Thus a small difference in the price of a processor multiplied by the millions of copies of the system in which that processor is embedded can make a significant difference to the profitability of the organization producing the system. Volume discounts and reuse of hardware across different products are techniques that device vendors use to reduce costs.

Volume, weight, and cost are constraints given both by the marketing department of an organization and by the physical considerations of its use. The marketing department is concerned with customers' reactions. The physical considerations for the device's use depend on both human and usage factors. Smartphone displays must be large enough for a human to read; automobiles are constrained by weight limits on roads; trains are constrained by track width; and so forth.

Other constraints on mobile system resources (and therefore on software architects) reflect the following factors:

- *Safety considerations.* Physical resources that have safety consequences must not fail or must have backups. Backup processors, networks, or sensors add cost and weight, as well as consume space. For example, many aircraft have an emergency source of power that can be used in case of engine failure.
- *Thermal limits.* Heat can be generated by the system itself (think of your lap on which your laptop sits), which can have a detrimental effect on the system's performance, even to the point of inducing failure. The environment's ambient temperature—too high or too low—can have an impact as well. There should be an understanding of the environment in which the system will be operated prior to making hardware choices.
- *Other environmental concerns.* Other concerns include exposure to adverse conditions such as moisture or dust, or being dropped.

The Architect's Concerns

An architect must make a number of important decisions surrounding resources and their usage:

- *Assigning tasks to electronic control units (ECUs).* Larger mobile systems, such as cars or airplanes, have multiple ECUs of differing power and capacity. A software architect must decide which subsystems will be assigned to which ECUs. This decision can be based on a number of factors:
- *Fit of the ECU to the function.* Functions must be allocated to ECUs with sufficient power to perform the function. Some ECUs may have specialized processors; for example, an ECU with a graphics processor is a better fit for graphics functions.

- *Criticality.* More powerful ECUs may be reserved for critical functions. For example, engine controllers are more critical and more reliable than the comfort features subsystem.
- *Location in the vehicle.* First-class passengers may have better Wi-Fi connectivity than second-class passengers.
- *Connectivity.* Some functions may be split among several ECUs. If so, they must be on the same internal network and able to communicate with each other.
- *Locality of communication.* Putting components that intensely communicate with each other on the same ECU will improve their performance and reduce network traffic.
- *Cost.* Typically a manufacturer wants to minimize the number of ECUs deployed.
- *Offloading functionality to the cloud.* Applications such as route determination and pattern recognition can be performed partly by the mobile system itself—where the sensors are located—and partly from portions of the application that are resident on the cloud—where more data storage and more powerful processors are available. The architect must determine whether the mobile system has sufficient power for specific functions, whether there is adequate connectivity to offload some functions, and how to satisfy performance requirements when the functions are split between the mobile system and the cloud. The architect should also take into consideration data storage available locally, data update intervals, and privacy concerns.
- *Shutting down functions depending on the mode of operations.* Subsystems that are not being used can scale down their footprint, allowing competing subsystems to access more resources, and thereby deliver better performance. In sports cars, an example is switching on a “race mode,” which disables the processes responsible for calculating comfortable suspension parameters based on the road profile and activates calculations of torque distribution, braking power, suspension hardening, and centrifugal forces.
- *Strategy for displaying information.* This issue is tied to available display resolution. It’s possible to do GPS style mapping on a 320×320 pixel display, but a lot of effort has to go into minimizing the information on the display. At a resolution of $1,280 \times 720$, there are more pixels, so the information display can be richer. (Having the ability to change the information on the display is a strong motivator for a pattern such as MVC [see Chapter 13] so that the view can be swapped out based on the specific display characteristics.)

18.5 Life Cycle

The life cycle of mobile systems tends to feature some idiosyncrasies that an architect needs to take into account, and these differ from the choices made for traditional (nonmobile) systems. We’ll dive right in.

The Architect's Concerns

The architect must be concerned with the hardware choices, testing, deploying updates, and logging. We elaborate on these concerns in the next four subsections.

Hardware First

For many mobile systems, the hardware is chosen before the software is designed. Consequently, the software architecture must live with the constraints imposed by the chosen hardware.

The main stakeholders in early hardware choices are management, sales, and regulators. Their concerns typically focus on ways to reduce risks rather than ways to promote quality attributes. The best approach for a software architect is to actively drive these early discussions, emphasizing the tradeoffs involved, instead of passively awaiting their outcomes.

Testing

Mobile devices present some unique considerations for testing:

- *Test display layouts.* Smartphones and tablets come in a wide variety of shapes, sizes, and aspect ratios. Verifying the correctness of the layout on all of these devices is complicated. Some operating system frameworks allow the user interface to be operated from unit tests, but may miss some unpleasant edge cases. For example, suppose you display control buttons on your screen, with the layout specified in HTML and CSS, and suppose it's automatically generated for all display devices you anticipate using. A naive generation for a tiny display could produce a control on a 1×1 pixel, or controls right at the edge of the display, or controls that overlap. These may easily escape detection during testing.
- *Test operational edge cases.*
 - An application should survive battery exhaustion and shutdown of the system. The preservation of state in such cases needs to be ensured and tested.
 - The user interface typically operates asynchronously from the software that provides the functionality. When the user interface does not react correctly, re-creating the sequence of events that caused the problem is difficult because the problem may depend on the timing, or on a specific set of operations in progress at the time.
- *Test resource usage.* Some vendors will make simulators of their devices available to software architects. That's helpful, but testing battery usage with a simulator is problematic.
- *Test for network transitions.* Ensuring that the system makes the best choice when multiple communication networks are available is also difficult. As a device moves from one network to another (e.g., from a Wi-Fi network to a cellular network and then to a different Wi-Fi network), the user should be unaware of these transitions.

Testing for transportation or industrial systems tends to happen on four levels: the individual software component level, the function level, the device level, and the system level. The levels and boundaries between them may vary depending on the system, but they are implied in several reference processes and standards such as Automotive SPICE.

For example, suppose we are testing a car's lane keep assist function, where the vehicle stays in the lane defined by markings on the road and does so without driver input. Testing of this system may address the following levels:

1. *Software component.* A lane detection software component will be tested through the usual techniques for unit and end-to-end testing, with the aim of validating the software's stability and correctness.
2. *Function.* The next step is to run the software component together with other components of the lane keep assist function, such as a mapping component to identify highway exits, in a simulated environment. The aim is to validate the interfacing and safe concurrency when all components of the function are working together. Here, simulators are used to provide the software function with inputs that correspond to a vehicle driving down a marked road.
3. *Device.* The bundled lane keep assist function, even if it passes the tests in the simulated environment and on the development computers, needs to be deployed on its target ECU and tested there for performance and stability. In this device test phase, the environment would still be simulated, but this time through simulated external inputs (messages from other ECUs, sensor inputs, and so forth) connected to the ECU's ports.
4. *System.* In the final system integration testing phase, all devices with all functions and all components are built into full-size configurations, first in a test lab and then in a test prototype. For example, the lane keep assist function could be subjected to testing, along with its actions on the steering and acceleration/braking functions, while being fed a projected image or a video of the road. The role of these tests is to confirm that the integrated subsystems work together and deliver the desired functionality and system quality attributes.

An important point here is test traceability: If an issue is found in step 4, it needs to be reproducible and traceable through all test setups, since a fix will have to go through all four test levels again.

Deploying Updates

In a mobile device, updates to the system either fix issues, provide new functionality, or install features that are unfinished but perhaps were partially installed at the time of an earlier release. Such an update may target the software, the data, or (less often) the hardware. Modern cars, for example, require software updates, which are fetched over networks or downloaded via USB interfaces. Beyond providing for the capability of updates during operation, the following specific issues relate to deploying updates:

- *Maintaining data consistency.* For consumer devices, upgrades tend to be automatic and one-way (there's no way to roll back to an earlier version). This suggests keeping data on the cloud is a good idea—but then all the interactions between the cloud and the application need to be tested.
- *Safety.* The architect needs to determine which states of the system can safely support an update. For example, updating a car's engine control software while the vehicle is

driving down the highway is a bad idea. This, in turn, implies that the system needs to be aware of safety-relevant states with respect to updates.

- *Partial system deployment.* Re-deploying a total application or large subsystem will consume both bandwidth and time. The application or subsystem should be architected so that the portions that change frequently can be easily updated. This calls for a specific type of modifiability (see Chapter 8) and an attention to deployability (see Chapter 5). In addition, updates should be easy and automated. Accessing physical portions of a device to update them may be awkward. Returning to the engine controller example, updating the controller software should not require access to the engine.
- *Extendability.* Mobile vehicle systems tend to have relatively long lifetimes. Retrofitting cars, trains, airplanes, satellites, and so forth will likely become necessary at some point. Retrofitting means adding new technology to old systems, either by replacement or addition. This could occur for the following reasons:
 - The component reaches the end of its life before the overall system reaches its end. The end of life means support will be discontinued, which creates high risks in case of failures: There will be no trusted source from which to get answers or support with reasonable costs—that is, without having to dissect and reverse-engineer the component in question.
 - Newer better technology has come out, prompting a hardware/software upgrade. An example is retrofitting a 2000s car with a smartphone-connected infotainment system instead of an old radio/CD player.
 - Newer technology is available that adds functionality without replacing existing functionality. For example, suppose the 2000s-era car never had a radio/CD player at all, or lacked a backup camera.

Logging

Logs are critical when investigating and resolving incidents that have occurred or may occur. In mobile systems, the logs should be offloaded to a location where they are accessible regardless of the accessibility of the mobile system itself. This is useful not only for incident handling, but also for performing various types of analyses on the usage of the system. Many software applications do something similar when they encounter a problem and ask for permission to send the details to the vendor. For mobile systems, this logging capability is particularly important, and they may very well not ask permission to obtain the data.

18.6 Summary

Mobile systems span a broad range of forms and applications, from smartphones and tablets to vehicles such as automobiles and aircraft. We have categorized the differences between mobile systems and fixed systems as being based on five characteristics: energy, connectivity, sensors, resources, and life cycle.

The energy in many mobile systems comes from batteries. Batteries are monitored to determine both the remaining time on the battery and the usage of individual applications. Energy usage can be controlled by throttling individual applications. Applications should be constructed to survive power failures and restart seamlessly when power is restored.

Connectivity means connecting to other systems and the Internet through wireless means. Wireless communication can be via short-distance protocols such as Bluetooth, medium-range protocols such as Wi-Fi protocols, and long-distance cellular protocols. Communication should be seamless when moving from one protocol class to another, and considerations such as bandwidth and cost help the architect decide which protocols to support.

Mobile systems utilize a variety of sensors. Sensors provide readings of the external environment, which the architect then uses to develop a representation within the system of the external environment. Sensor readings are processed by a sensor stack specific to each operating system; these stacks will deliver readings meaningful to the representation. It may take multiple sensors to develop a meaningful representation, with the readings from these sensors then being fused (integrated). Sensors may also become degraded over time, so multiple sensors may be needed to get an accurate representation of the phenomenon being measured.

Resources have physical characteristics such as size and weight, have processing capabilities, and carry a cost. The design choices involve tradeoffs among these factors. Critical functions may require more powerful and reliable resources. Some functions may be shared between the mobile system and the cloud, and some functions may be shut down in certain modes to free up resources for other functions.

Life-cycle issues include choice of hardware, testing, deploying updates, and logging. Testing of the user interface may be more complicated with mobile systems than with fixed systems. Likewise, deployment is more complicated because of bandwidth, safety considerations, and other issues.

18.7 For Further Reading

The Battery University (<https://batteryuniversity.com/>) has more materials than you care about on batteries of various types and their measurement.

You can read more about various network protocols at the following sites:

link-labs.com/blog/complete-list-iot-network-protocols

https://en.wikipedia.org/wiki/Wireless_ad_hoc_network

<https://searchnetworking.techtarget.com/tutorial/Wireless-protocols-learning-guide>

https://en.wikipedia.org/wiki/IEEE_802

You can find out more about sensors in [Gajjarby 17].

Some test tools for mobile applications can be found at these two sites:

<https://codelabs.developers.google.com/codelabs/firebase-test-lab/index.html#0>

<https://firebase.google.com/products/test-lab>

Some of the difficulties involved in making self-driving cars safe are discussed in “Adventures in Self Driving Car Safety,” Philip Koopman’s presentation on Slideshare: slideshare.net/PhilipKoopman1/adventures-in-self-driving-car-safety?qid=eb5f5305-45fb-419e-83a5-998a0b667004&v=&b=&from_search=3.

You can find out about Automotive SPICE at automotivespice.com.

ISO 26262, “Road Vehicles: Functional Safety,” is an international standard for functional safety of automotive electrical and/or electronic systems (iso.org/standard/68383.html).

18.8 Discussion Questions

1. Which architectural choices would you make to design a system that could tolerate complete loss of power and have the ability to restart where it left off without compromising the integrity of its data?
2. What are the architectural issues involved in network transitions, such as starting a file transfer over Bluetooth and then moving out of Bluetooth range and switching over to Wi-Fi, all the while keeping the transfer seamlessly proceeding?
3. Determine the weight and size of the battery in one of your mobile systems. What compromises do you think the architect made because of the size and weight?
4. Which types of problems can a CSS testing tool find? Which does it miss? How do these considerations affect the testing of mobile devices?
5. Consider an interplanetary probe such as those used in NASA’s Mars exploration program. Does it meet the criteria of a mobile device? Characterize its energy characteristics, network connectivity issues (obviously, none of the network types discussed in Section 18.2 are up to the task), sensors, resource issues, and special life-cycle considerations.
6. Consider mobility not as a class of computing system, but rather as a quality attribute, like security or modifiability. Write a general scenario for mobility. Write a specific mobility scenario for a mobile device of your choosing. Describe a set of tactics to achieve the quality attribute of “mobility.”
7. Section 18.5 discussed several aspects of testing that are more challenging in mobile systems. What testability tactics from Chapter 12 can help with these issues?

This page intentionally left blank

19



Architecturally Significant Requirements

The most important single aspect of software development is to be clear about what you are trying to build.

—Bjarne Stroustrup, creator of C++

Architectures exist to build systems that satisfy requirements. By “requirements,” we do not necessarily mean a documented catalog produced using the best techniques that requirements engineering has to offer. Instead, we mean the set of properties that, if not satisfied by your system, will cause the system to be a failure. Requirements exist in as many forms as there are software development projects—from polished specifications to verbal shared understanding (real or imagined) among principal stakeholders. The technical, economic, and philosophical justifications for your project’s requirements practices are beyond the scope of this book. What is in scope is that, regardless of how they are captured, they establish the criteria for success or failure, and architects need to know them.

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 (QA) requirements—the performance, security, modifiability, availability, usability, and so forth, that the architecture must provide to the system. In Chapters 4–14, we introduced patterns and tactics to achieve QAs. Each time you select a pattern or tactic to use in your architecture, you are doing so because of the need to meet QA 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 must identify ASRs, usually after doing a significant bit of work to uncover candidate ASRs. Competent architects know this. Indeed, 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 been previously identified.

This chapter provides some systematic techniques for identifying the ASRs and other factors that will shape the architecture.

19.1 Gathering ASRs from Requirements Documents

An obvious location to look for candidate ASRs is in the requirements document or in user stories. After all, we are looking for requirements, and requirements should be (duh) in requirements documents. Unfortunately, this is not usually the case, although information in the requirements documents can certainly be useful.

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. 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 uncertain when the architect starts work. Even where they exist and are stable, requirements documents often fail an architect in two ways:

- Most of the information found in a requirements specification does not affect the architecture. As we've seen over and over, architectures are mostly driven or "shaped" by QA requirements, which determine and constrain the most important architectural decisions. Even so, the vast bulk of most requirements specifications focus on the required features and functionality of a system, which shape the architecture the least. The best software engineering practices do prescribe capturing QA requirements. For example, the Software Engineering Body of Knowledge (SWEBOK) says that QA 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 QA 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 useful requirements because they are not testable; they are not falsifiable. But, looking on the bright side, they can be viewed as invitations for the architect to begin a conversation about what the requirements in these areas really are.

- Much of what is useful to an architect won't be found 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 thus are not the subject of requirements specifications. ASRs often derive from business goals in the development organization itself; we'll explore this connection in Section 19.3. 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 those of the developer. Stakeholders, the technical environment, and the organization itself all play a role in influencing architectures. When we discuss architecture design, in Chapter 20, we will explore these requirements in more detail.

Sniffing out ASRs from a Requirements Document

While requirements documents won't tell an architect the whole story, they are still an important source of ASRs. Of course, ASRs will not be conveniently labeled as such; the architect should expect to perform a bit of investigation and archaeology to ferret them out.

Some specific things to look for are the following categories of information:

- *Usage.* User roles versus system modes, internationalization, language distinctions.
- *Time.* Timeliness and element coordination.
- *External elements.* External systems, protocols, sensors or actuators (devices), middleware.
- *Networking.* Network properties and configurations (including their security properties).
- *Orchestration.* Processing steps, information flows.
- *Security properties.* User roles, permissions, authentication.
- *Data.* Persistence and currency.
- *Resources.* Time, concurrency, memory footprint, scheduling, multiple users, multiple activities, devices, energy usage, soft resources (e.g., buffers, queues), and scalability requirements.
- *Project management.* Plans for teaming, skill sets, training, team coordination.
- *Hardware choices.* Processors, families of processors, evolution of processors.
- *Flexibility of functionality, portability, calibrations, configurations.*
- *Named technologies, commercial packages.*

Anything that is known about their planned or anticipated evolution will be useful information, too.

Not only are these categories architecturally significant in their own right, but the possible change and evolution of each are also likely to be architecturally significant. Even if the requirements document you're mining doesn't mention evolution, consider which of the items in the preceding list are likely to change over time, and design the system accordingly.

19.2 Gathering ASRs by Interviewing Stakeholders

Suppose your project isn't producing a comprehensive requirements document. Or maybe it is, but it won't have the QAs nailed down by the time you need to start your design work. What do you do?

First, stakeholders often don't know what their QA requirements actually are. In that case, architects are called upon to help set the QA requirements for a system. Projects that recognize this need for collaboration and encourage it are much more likely to be successful than those that don't. Relish the opportunity! No amount of nagging your stakeholders will suddenly instill in them the necessary insights. If you insist on quantitative QA requirements, you may get numbers that are arbitrary and at least some of those requirements will be difficult to satisfy and, in the end, actually detract from system success.

Experienced architects often have deep insights into which QA responses have been exhibited by similar systems, and which QA responses are reasonable to expect and to provide in the current context. Architects can also usually give quick feedback as to which QA responses will be straightforward to achieve and which will likely be problematic or even prohibitive.

For example, a stakeholder may ask for 24/7 availability—who wouldn't want that? However, the architect can explain how much that requirement is likely to cost, which will give the stakeholders information to make a tradeoff between availability and affordability. Also, architects are the only people in the conversation 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 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. It emphasizes 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.

After introductions and an overview of the workshop steps, the QAW involves the following elements:

- *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 QA requirements. The QAs that will be refined in later steps will be derived largely from the business/mission needs presented in this step.
- *Architectural plan presentation.* While 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.

- *Identification of architectural drivers.* The facilitators will share their list of key architectural drivers that they assembled in the prior two steps, 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 include overall requirements, business drivers, constraints, and quality attributes.
 - *Scenario brainstorming.* Each stakeholder expresses a scenario representing his or her concerns with respect to the system. Facilitators ensure that each scenario addresses a QA concern, by specifying an explicit stimulus and response.
 - *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.
 - *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.
 - *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 3. As the scenarios are refined, issues surrounding their satisfaction will emerge and should be recorded. This step lasts as long as time and resources allow.
-

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 for the following purposes:

- Refine system and software requirements.
 - Understand and clarify the system’s architectural drivers.
 - Provide a 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.
-

I Don’t Know What That Requirement Should Be

It is not uncommon when interviewing stakeholders and probing for ASRs that they will complain, “I don’t know what that requirement should be.” While it is true that this is the way that they *feel*, it is also frequently the case that they know *something* about the requirement, particularly if the stakeholders are experienced in the domain. In this case, eliciting this “something” is far better than simply making up the requirement on your own. For example, you might ask, “How quickly should the system respond to this

transaction request?” If the answer is “I don’t know,” my advice here is to play dumb. You can say, “So . . . 24 hours would be OK?” The response is often an indignant and astonished “No!” “Well, how about 1 hour?” “No!” “Five minutes?” “No!” “How about 10 seconds?” “Well, <grumble, mumble> I suppose I could live with something like that. . . .”

By playing dumb, you can often get people to at least give you a range of acceptable values, even if they do not know precisely what the requirement should be. And this range is typically enough for you to choose architectural mechanisms. A response time of 24 hours versus 10 minutes versus 10 seconds versus 100 milliseconds means, to an architect, choosing very different architectural approaches. Armed with this information, you can now make informed design decisions.

—RK

19.3 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 people involved 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. In still other organizations (e.g., nonprofits, charities, governments), profit is the furthest thing from anyone’s mind.

Business goals are of interest to architects because they 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.* Every quality attribute requirement—such as user-visible response time or platform flexibility or iron-clad security or any of a dozen other needs—originates from some higher purpose that can be described in terms of added value. A desire to differentiate a product from its competition and let the developing organization capture market share may lead to a requirement for what might seem like an unusually fast response time. Also, knowing the business goal behind a particularly stringent requirement enables the architect to question the requirement in a meaningful way—or marshal the resources to meet it.
2. *Business goals may affect the architecture without inducing a quality attribute requirement at all.* A software architect related to us that some years ago he delivered an early draft of the architecture to his manager. The manager remarked that a database was missing from the architecture. The architect, pleased that the manager had noticed, explained how he (the architect) had devised a design approach that obviated the need for a bulky, expensive database. The manager, however, pressed for the design to include a database, *because the organization had a database unit employing a number of highly paid technical staff who were currently unassigned and needed work.* No requirements specification would capture such a requirement, nor would any manager allow such a motivation to be captured. And yet that architecture, had it been delivered without a database, would have been just as deficient—from the manager’s point of view—as if it had failed to deliver an important function or QA.

3. *No influence of a business goal on the architecture.* Not all business goals lead to quality attributes. For example, a business goal to “reduce cost” might be realized by lowering the facility’s thermostats in the winter or reducing employees’ salaries or pensions.

Figure 19.1 illustrates the major points from this discussion. In the figure, the arrows mean “leads to.” The solid arrows highlight the relationships of greatest interest to architects.

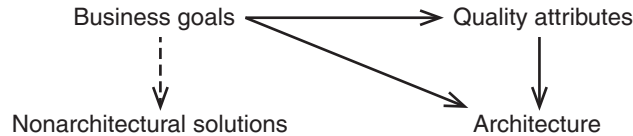


FIGURE 19.1 Some business goals may lead to quality attribute requirements, or lead directly to architectural decisions, or lead to non-architectural solutions.

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 of determining such goals are both possible and desirable. Moreover, it is worthwhile to capture business goals explicitly, because they often imply ASRs that would otherwise go undetected until it is too late or too expensive to address them.

One way to do this is to employ the PALM method, which entails holding a workshop with the architect and key business stakeholders. The heart of PALM consists of these steps:

- *Business goals elicitation.* Using the categories given later in this section to guide the discussion, capture from stakeholders the set of important business goals for this system. Elaborate the business goals and express them as business goal scenarios.¹ Consolidate almost-alike business goals to eliminate duplication. Have the participants prioritize the resulting set to identify the most important goals.
- *Identify potential QAs from business goals.* For each important business goal scenario, have the participants describe a QA and response measure value that (if architected into the system) would help achieve the goal.

The process of 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 motivation to engage the right stakeholders in your organization: “What are our ambitions about market share for this product, and how could the architecture contribute to meeting them?”

1. A business goal scenario is a structured seven-part expression that captures a business goal, similar in intent and usage to a QA scenario. This chapter’s “For Further Reading” section contains a reference that describes PALM, and business goal scenarios, in full detail.

Our research in business goals has led us to adopt the categories shown in the list that follows. 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.

1. Growth and continuity of the organization
2. Meeting financial objectives
3. Meeting personal objectives
4. Meeting responsibility to the employees
5. Meeting responsibility to society
6. Meeting responsibility to the state
7. Meeting responsibility to the shareholders
8. Managing market position
9. Improving business processes
10. Managing the quality and reputation of products
11. Managing change in the environment over time

19.4 Capturing ASRs in a Utility Tree

In a perfect world, the techniques described in Sections 19.2 and 19.3 would be applied early on in your development process: You would interview the key stakeholders, elicit their business goals and driving architectural requirements, and have them prioritize all of these inputs for you. Of course, the real world, lamentably, is less than perfect. It is often the case that you do not have access to these stakeholders when you need them, for organizational or business reasons. So what do you do?

Architects can use a construct called a *utility tree* when the “primary sources” of requirements are not available. A utility tree is a top-down representation of what you, as an architect, believe to be the QA-related ASRs that are critical to the success of the system.

A utility tree begins with the word “Utility” as the root node. Utility is an expression of the overall “goodness” of the system. You then elaborate on this root node by listing the major QAs that the system is required to exhibit. (You might recall that we said in Chapter 3 that QA names by themselves were not very useful. Never fear—they are only being used as intermediate placeholders for subsequent elaboration and refinement!)

Under each QA, record specific refinements of that QA. For example, performance might be decomposed into “data latency” and “transaction throughput” or, alternatively, “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, you can then record the specific ASRs, expressed as QA scenarios.

Once the ASRs are recorded as scenarios and placed at the leaves of the tree, you can evaluate these scenarios against two criteria: the business value of the candidate scenario and the technical risk of achieving it. You can use any scale you like, but we find that a simple “H” (high), “M” (medium), and “L” (low) scoring system suffices for each criterion. For business

value, “high” designates a must-have requirement, “medium” identifies a requirement that is important but would not lead to project failure were it omitted, and “low” describes a nice requirement to meet but not something worth much effort. For technical risk, “high” means that meeting this ASR is keeping you awake at night, “medium” means meeting this ASR is concerning but does not carry a high risk, and “low” means that you have confidence in your ability to meet this ASR.

Table 19.1 shows a portion of an example utility tree. Each ASR is labeled with an indicator of its business value and its technical risk.

TABLE 19.1 Tabular Form of the Utility Tree for a System in the Healthcare Space

Quality Attribute	Attribute Refinement	ASR Scenario
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 seconds. (H, H)
	Throughput	At peak load, the system is able to complete 150 normalized transactions per second. (M, M)
Usability	Proficiency training	A new hire with two or more years’ experience in the business can learn, with 1 week of training, to execute any of the system’s core functions in less than 5 seconds. (M, L)
	Efficiency of operations	A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process with no input errors. (M, M)
Configurability	Data configurability	A hospital increases the fee for a particular service. The configuration team makes and tests the change in 1 working day; no source code needs to change. (H, L)
Maintainability	Routine changes	A maintainer encounters response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H, M)
		A reporting requirement requires a change to the report-generating metadata. Change is made and tested in 4 person-hours of effort (M, L)
	Upgrades to commercial components	The database vendor releases a new major version that is successfully tested and installed in less than 3 person-weeks. (H, M)
Security	Adding new feature	A feature that tracks blood bank donors is created and successfully integrated within 2 person-months. (M, M)
	Confidentiality	A physical therapist is allowed to see that part of a patient’s record dealing with orthopedic treatment, but not other parts or any financial information. (H, M)
	Resisting attacks	The system repels an unauthorized intrusion attempt and reports the attempt to authorities within 90 seconds. (H, M)
Availability	No down time	The database vendor releases new software, which is hot-swapped into place, with no downtime. (H, L)
		The system supports 24/7/365 web-based account access by patients. (M, M)

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 scenario is not necessarily an error or omission that needs to be rectified, but rather an indication you should investigate whether there are unrecorded ASR scenarios in that area.
- ASR scenarios that receive 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 scenarios might be a cause for concern regarding whether the system is, in fact, achievable.

19.5 Change Happens

Edward Berard said, “Walking on water and developing software from a specification are both easy if both are frozen.” Nothing in this chapter should be taken to assume that such a miraculous state of affairs is likely to exist. Requirements—whether captured or not—change all the time. Architects have to adapt and keep up, to ensure that their architectures are still the right ones that will bring success to the project. In Chapter 25, where we discuss architecture competence, we’ll advise that architects need to be great communicators, and this means great bidirectional communicators, taking in as well as supplying information. Always keep a channel open to the key stakeholders who determine the ASRs so you can keep up with changing requirements. The methods offered in this chapter can be applied repetitively to accommodate change.

Even better than keeping up with change is staying one step ahead of it. If you get wind of a change to the ASRs, you can take preliminary steps to design for it, as an exercise to understand the implications. If the change will be prohibitively expensive, sharing that information with the stakeholders will be a valuable contribution, and the earlier they know it, the better. Even more valuable might be suggestions about changes that would do (almost) as well in meeting the goals but without breaking the budget.

19.6 Summary

Architectures are driven by architecturally significant requirements. An ASR must have:

- A profound impact on the architecture. Including this requirement will 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.

ASRs can be extracted from a requirements document, captured from stakeholders during a workshop (e.g., a QAW), captured from the architect in a utility tree, 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.

In gathering these requirements, you should be mindful of the organization's business goals. Business goals can be expressed in a common, structured form and represented as business goal scenarios. Such goals may be elicited and documented using PALM, a structured facilitation method.

A useful representation of QA requirements is a utility tree. Such a graphical depiction helps to capture these requirements in a structured form, starting from coarse, abstract notions of QAs and gradually refining them to the point where they are captured as scenarios. These scenarios are then prioritized, with this prioritized set defining your “marching orders” as an architect.

19.7 For Further Reading

The Open Group Architecture Framework, available at opengroup.org/togaf/, provides a 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 SARA group (Software Architecture Review and Assessment), as part of a document that can be retrieved at <http://pkruchten.wordpress.com/architecture/SARAv1.pdf>.

The Software Engineering Body of Knowledge (SWEBOK), third edition, can be downloaded here: computer.org/education/bodies-of-knowledge/software-engineering/v3. As we go to press, a fourth edition is being developed.

A full description of PALM [Clements 10b] can be found here: https://resources.sei.cmu.edu/asset_files/TechnicalNote/2010_004_001_15179.pdf.

19.8 Discussion Questions

1. Interview representative stakeholders for a business system in use at your company or your university and capture at least three business goals for it. To do so, use PALM's seven-part business goal scenario outline, referenced in the “For Further Reading” section.
2. Based on the business goals you uncovered for question 1, propose a set of corresponding ASRs.

3. Create a utility tree for an ATM. (Interview some of your friends and colleagues if you would like to have them contribute QA considerations and scenarios.) Consider a minimum of four different QAs. Ensure that the scenarios that you create at the leaf nodes have explicit responses and response measures.
4. 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?

20



Designing an Architecture

With Humberto Cervantes

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.
—Antoine de Saint-Exupéry

Design—including architectural design—is a complex activity to perform. It involves making a myriad of decisions that take into account many aspects of a system. In the past, this task was only entrusted to senior software engineers—gurus—with decades of hard-won experience. A systematic method provides guidance in performing this complex activity so that it can be learned and capably performed by mere mortals.

In this chapter, we provide a detailed discussion of a method—Attribute-Driven Design (ADD)—that allows an architecture to be designed in a systematic, repeatable, and cost-effective way. Repeatability and teachability are the hallmarks of an engineering discipline. To make a method repeatable and teachable, we need a set of steps that any suitably trained engineer can follow.

We begin by providing an overview of ADD and its steps. This overview is followed by more detailed discussions of some of the key steps.

20.1 Attribute-Driven Design

Architectural design for software systems is no different than design in general: It involves making decisions, and working with the available materials and skills, to satisfy requirements and constraints. In architectural design, we turn decisions about architectural drivers into structures, as shown in Figure 20.1. Architectural drivers comprise architecturally significant requirements (ASRs—the topic of Chapter 19), but also include functionality, constraints, architectural concerns, and design purpose. The resulting structures are then used to guide the project in the many ways we laid out in Chapter 2: They guide analysis and construction. They

serve as the foundation for educating a new project member. They guide cost and schedule estimations, team formation, risk analysis and mitigation, and, of course, implementation.

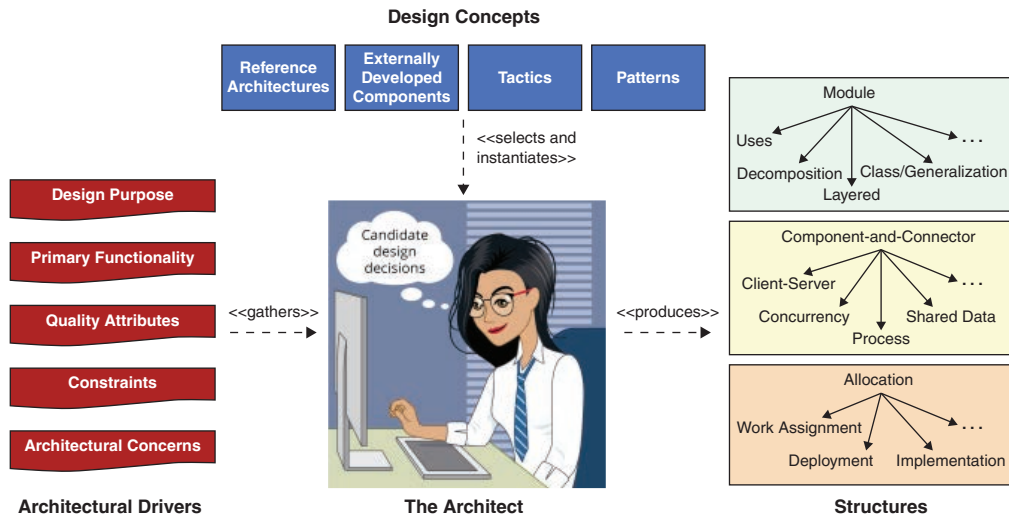


FIGURE 20.1 Overview of the architecture design activity

Prior to starting architecture design, it is important to determine the scope of the system—what is inside and what is outside of the system you are creating, and which external entities the system will interact with. This context can be represented using a system context diagram, like that shown in Figure 20.2. Context diagrams are discussed in more detail in Chapter 22.

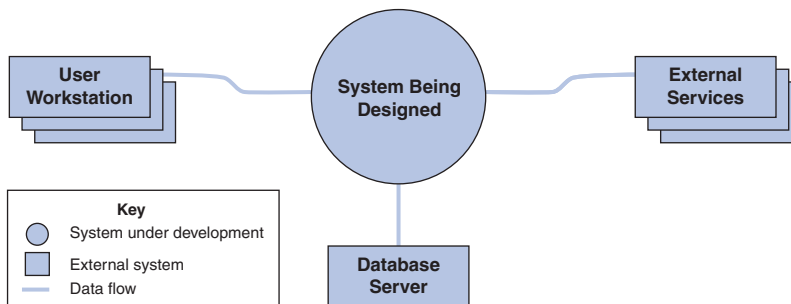


FIGURE 20.2 Example of a system context diagram

In ADD, architecture design is performed in rounds, each of which may consist of a series of design iterations. A round comprises the architecture design activities performed within a development cycle. Through one or more iterations, you produce an architecture that suits the established design purpose for this round.

Within each iteration, a series of design steps is performed. ADD provides detailed guidance on the steps that need to be performed inside each iteration. Figure 20.3 shows the steps and artifacts associated with ADD. In the figure, steps 1–7 constitute a round. Within a round, steps 2–7 constitute one or more iterations within a round. In the following subsections, we provide an overview of each of these steps.

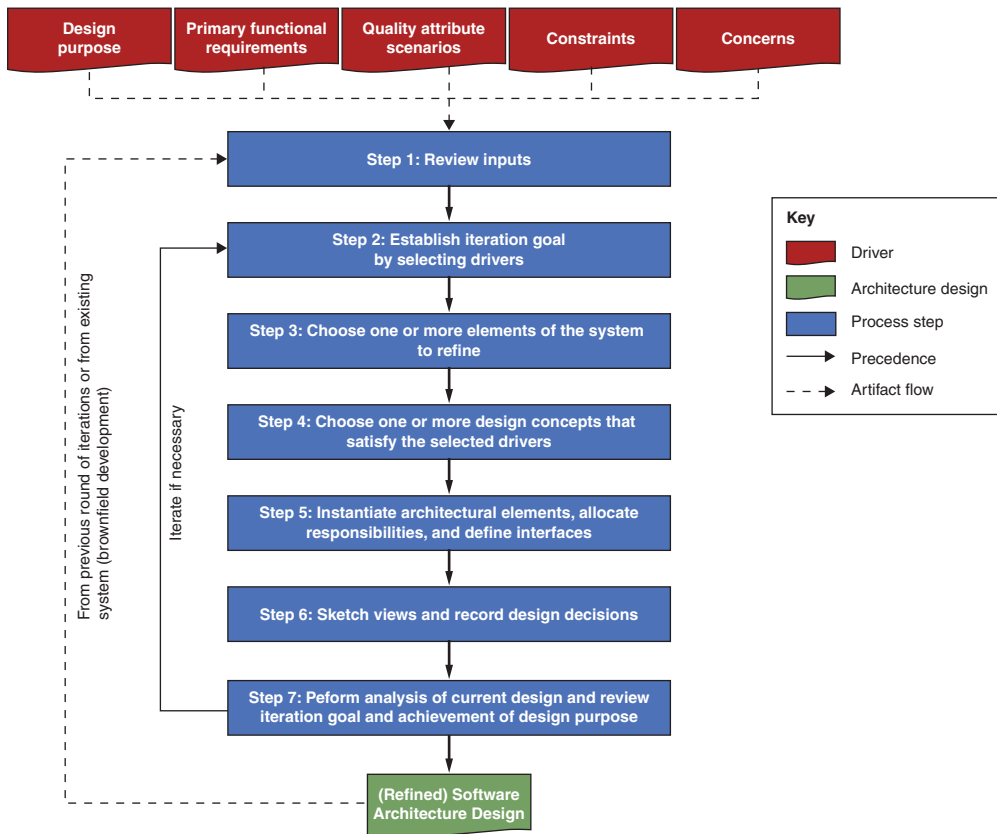


FIGURE 20.3 Steps and artifacts of ADD

20.2 The Steps of ADD

The sections that follow describe the steps for ADD.

Step 1: Review Inputs

Before starting a design round, you need to ensure that the architectural drivers (the inputs to the design process) are available and correct. These include:

- The purpose of the design round
- The primary functional requirements
- The primary quality attribute (QA) scenarios
- Any constraints
- Any concerns

Why do we explicitly capture the design purpose? You need to make sure that you are clear about your goals for a round. In an incremental design context comprising multiple rounds, the purpose for a design round may be, for example, to produce a design for early estimation, to refine an existing design to build a new increment of the system, or to design and generate a prototype to mitigate certain technical risks. In addition, you need to know the existing architecture's design, if this is not greenfield development.

At this point, the primary functionality—typically captured as a set of use cases or user stories—and QA scenarios should have been prioritized, ideally by your most important project stakeholders. (You can employ several different techniques to elicit and prioritize them, as discussed in Chapter 19). You, the architect, must now “own” these. For example, you need to check whether any important stakeholders were overlooked in the original requirements elicitation process, and whether any business conditions have changed since the prioritization was performed. These inputs really do “drive” design, so getting them right and getting their priority right are crucial. We cannot stress this point strongly enough. Software architecture design, like most activities in software engineering, is a “garbage-in-garbage-out” process. The results of ADD cannot be good if the inputs are poorly formed.

The drivers become part of an architectural design backlog that you should use to perform the different design iterations. When you have made design decisions that account for all of the items in the backlog, you've completed this round. (We discuss the idea of a backlog in more depth in Section 20.8.)

Steps 2–7 make up the activities for each design iteration carried out within this design round.

Step 2: Establish Iteration Goal by Selecting Drivers

Each design iteration focuses on achieving a particular goal. Such a goal typically involves designing to satisfy a subset of the drivers. For example, an iteration goal could be to create structures from elements that will allow a particular performance scenario, or a use case to

be achieved. For this reason, when performing design activities, you need to establish a goal before you start a particular design iteration.

Step 3: Choose One or More Elements of the System to Refine

Satisfying drivers requires you to make architectural design decisions, which then manifest themselves in one or more architectural structures. These structures are composed of inter-related elements—modules and/or components, as defined in Chapter 1—and these elements are generally obtained by refining other elements that you previously identified in an earlier iteration. Refinement can mean decomposition into finer-grained elements (top-down approach), combination of elements into coarser-grained elements (bottom-up approach) or the improvement of previously identified elements. For greenfield development, you can start by establishing the system context and then selecting the only available element—that is, the system itself—for refinement by decomposition. For existing systems or for later design iterations in greenfield systems, you normally choose to refine elements that were identified in prior iterations.

The elements that you will select are the ones involved in the satisfaction of specific drivers. For this reason, when the design addresses an existing system, you need to have a good understanding of the elements that are part of the as-built architecture of the system. Obtaining this information might involve some “detective work,” reverse engineering, or discussions with developers.

In some cases, you may need to reverse the order of steps 2 and 3. For example, when designing a greenfield system or when fleshing out certain types of reference architectures, you will, at least in the early stages of design, focus on elements of the system and start the iteration by selecting a particular element and then considering the drivers that you want to address.

Step 4: Choose One or More Design Concepts That Satisfy the Selected Drivers

Choosing the design concept(s) is probably the most difficult decision you will face in the design process, because it requires you to identify the various design concepts that might plausibly be used to achieve your iteration goal, and to then make a selection from these alternatives. Many different types of design concepts are available—for example, tactics, patterns, reference architectures, and externally developed components—and, for each type, many options may exist. This can result in a considerable number of alternatives that need to be analyzed to before making the final choice. In Section 20.3, we discuss the identification and selection of design concepts in more detail.

Step 5: Instantiate Architectural Elements, Allocate Responsibilities, and Define Interfaces

Once you have selected one or more design concepts, you must make another type of design decision: how to *instantiate* elements out of the design concepts that you just selected. For example, if you selected the layers pattern as a design concept, you must decide how many layers will be used, and their allowed relationships, since the pattern itself does not prescribe these.

After instantiating the elements, you then need to allocate responsibilities to each of them. For example, in an app, at least three layers are usually present: presentation, business, and data. The responsibilities of these layers differ: The responsibilities of the presentation layer include managing all of the user interactions, the business layer manages application logic and enforces business rules, and the data layer manages the persistence and consistency of data.

Instantiating elements is only one part of creating structures that satisfy a driver or a concern. The elements that have been instantiated also need to be connected, thereby allowing them to collaborate with each other. This requires the existence of *relationships* between the elements and the exchange of information through some kind of interface. The interface is a contractual specification indicating how information should flow between the elements. In Section 20.4, we present more details on how the different types of design concepts are instantiated, how structures are created, and how interfaces are defined.

Step 6: Sketch Views and Record Design Decisions

At this point, you have finished performing the design activities for the iteration. However, you may have not taken any actions to ensure that the views—the representations of the structures you created—are preserved. For instance, if you performed step 5 in a conference room, you probably ended up with a series of diagrams on a whiteboard. This information is essential to the rest of the process, and you must capture it so that you can later analyze and communicate it to other stakeholders. Capturing the views may be as simple as taking a picture of the whiteboard.

The views that you have created are almost certainly not complete; thus, these diagrams may need to be revisited and refined in a subsequent iteration. This is typically done to accommodate elements resulting from other design decisions that you will make to support additional drivers. This is why we speak of “sketching” the views in ADD, where a “sketch” refers to a preliminary type of documentation. The more formal, more fully fleshed-out documentation of these views—should you choose to produce it (see Chapter 22)—occurs only after the design iterations have been finished (as part of the architectural documentation activity).

In addition to capturing the sketches of the views, you should record the significant decisions made in the design iteration, as well as the reasons that motivated these decisions (i.e., the rationale), to facilitate later analysis and understanding of the decisions. For example, decisions about important tradeoffs should be recorded at this time. During a design iteration, decisions are primarily made in steps 4 and 5. In Section 20.5, we explain how to create

preliminary documentation *during* the design process, including recording design decisions and their rationale.

Step 7: Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

By step 7, you should have created a partial design that addresses the goal established for the iteration. Making sure that this is actually the case is a good idea, to avoid unhappy stakeholders and later rework. You can perform the analysis yourself by reviewing the sketches of the views and design decisions that you captured, but an even better idea is to have someone else help you review this design. We do this for the same reason that organizations frequently have a separate testing/quality assurance group: Another person will not share your assumptions, and will have a different experience base and a different perspective. This diversity helps to find “bugs,” in both code and architecture. We discuss architectural analysis in more depth in Chapter 21.

Once the design performed in the iteration has been analyzed, you should review the state of your architecture in terms of your established design purpose. This means considering if, at this point, you have performed enough design iterations to satisfy the drivers that are associated with the design round. It also means considering whether the design purpose has been achieved or if additional design rounds are needed in future project increments. In Section 20.6, we discuss simple techniques that allow you to keep track of design progress.

Iterate If Necessary

You should perform additional iterations and repeat steps 2–7 for every driver that was considered. More often than not, however, this kind of repetition will not be possible because of time or resource constraints that force you to stop the design activities and move on to implementation.

What are the criteria for evaluating if more design iterations are necessary? Let *risk* be your guide. You should at least have addressed the drivers with the highest priority. Ideally, you should have certainty that critical drivers are satisfied or, at least, that the design is “good enough” to satisfy them.

20.3 More on ADD Step 4: Choose One or More Design Concepts

Most of the time you, as an architect, don’t need to, and should not, reinvent the wheel. Rather, your major design activity is to identify and select design concepts to meet the most important challenges and address the key drivers across the design iterations. Design is still an original and creative endeavor, but the creativity resides in the appropriate identification of these existing solutions, followed by combining and adapting them to the problem at hand. Even with

an existing corpus of solutions to choose from—and we are not always blessed with a rich corpus—this is still the hardest part of design.

Identification of Design Concepts

The identification of design concepts might appear daunting, because of the vast number of options available. There are likely dozens of design patterns and externally developed components that you could use to address any particular issue. To make things worse, these design concepts are scattered across many different sources: in practitioner blogs and websites, in research literature, and in books. Moreover, in many cases, there is no canonical definition of a concept. Different sites, for example, will define the broker pattern in different, largely informal ways. Finally, once you have identified the alternatives that can potentially help you achieve the design goals of the iteration, you need to select the best one(s) for your purposes.

To address a specific design problem, you can and often will use and combine different types of design concepts. For example, to build a security driver, you might employ a security pattern, a security tactic, a security framework, or some combination of these.

Once you have more clarity regarding the types of design concepts that you wish to use, you still need to identify alternatives—that is, design candidates. You can achieve this in several ways, although you will probably use a combination of these techniques rather than a single method:

- *Leverage existing best practices.* You can identify alternatives by making use of existing catalogs. Some design concepts, such as patterns, are extensively documented; others, such as externally developed components, are documented in a less thorough way. The benefits of this approach are that you can identify many alternatives and leverage the considerable knowledge and experience of others. The downsides are that searching and studying the information can require a considerable amount of time, the quality of the documented knowledge is often unknown, and the assumptions and biases of the authors are also unknown.
- *Leverage your own knowledge and experience.* If the system you are designing is similar to other systems you have designed in the past, you will probably want to begin with some of the design concepts that you have used before. The benefit of this approach is that the identification of alternatives can be performed rapidly and confidently. The downside is that you may end up using the same ideas repeatedly, even if they are not the most appropriate for all the design problems that you are facing, or if they have been superseded by newer, better approaches. As the saying goes: If all you have is a hammer, all the world looks like a nail.
- *Leverage the knowledge and experience of others.* As an architect, you have a background and knowledge that you have gained through the years. This background and knowledge will vary from person to person, especially if the types of design problems they have addressed in the past differ. You can leverage this information by performing the identification and selection of design concepts with some of your peers through brainstorming.

Selection of Design Concepts

Once you have identified a list of alternative design concepts, you need to select which one of the alternatives is the most appropriate to solve the design problem at hand. You can achieve this in a relatively simple way, by creating a table that lists the pros and cons associated with each alternative and selecting one of the alternatives based on those criteria and your drivers. The table can also contain other criteria, such as the cost associated with the use of the alternative. Methods such as SWOT (strengths, weaknesses, opportunities, threats) analysis can help you make this decision.

When identifying and selecting design concepts, keep in mind the constraints that are part of the architectural drivers, because some constraints will restrict you from selecting particular alternatives. For example, a constraint might be that all libraries and frameworks must employ an approved license. In that case, even if you have found a framework that could be useful for your needs, you may need to discard it if it does not carry an approved license.

You also need to keep in mind that the decisions regarding the selection of design concepts that you made in previous iterations may restrict the design concepts that you can now select due to incompatibilities. An example would be selecting a web architecture in an initial iteration and then selecting a user interface framework for local applications in a subsequent iteration.

Creation of Prototypes

In case the previously mentioned analysis techniques do not guide you to make an appropriate selection of design concepts, you may need to create prototypes and collect measurements from them. Creating early “throwaway” prototypes is a useful technique to help in the selection of externally developed components. This type of prototype is usually created without consideration for maintainability, reuse, or allowance for achieving other important goals. Such a prototype should not be used as a basis for further development.

Although the creation of prototypes can be costly, certain scenarios strongly motivate them. When thinking about whether you should create a prototype, ask these questions:

- Does the project incorporate emerging technologies?
- Is the technology new in the company?
- Are there certain drivers, particularly QAs, whose satisfaction using the selected technology presents risks (i.e., it is not understood whether they can be satisfied)?
- Is there a lack of trusted information, internal or external, that would provide some degree of certainty that the selected technology will be useful to satisfy the project drivers?
- Are there configuration options associated with the technology that need to be tested or understood?
- Is it unclear whether the selected technology can be easily integrated with other technologies that are used in the project?

If most of your answers to these questions are “yes,” then you should strongly consider the creation of a throwaway prototype.

To Prototype or Not to Prototype?

Architectural decisions must often be made with imperfect knowledge. To decide which way to go, a team could run a series of experiments (such as building prototypes) to try to reduce their uncertainty about which path to follow. The problem is that such experiments could carry a substantial cost, and the conclusions drawn from them might not be definitive.

For example, suppose a team needs to decide whether the system they are designing should be based on a traditional three-tier architecture or should be composed of microservices. Since it is the team's first project with microservices, they are not confident about that approach. They do a cost estimation for the two alternatives, and project that the cost of developing the three-tier architecture would be \$500,000 and that of developing the microservices would be \$650,000. If, having developed the three-tier architecture, the team later concluded that the wrong architecture was chosen, the estimated refactoring cost would be \$300,000. If the microservices architecture was the first one developed, and a later refactoring was needed, its estimated additional cost would be \$100,000.

What should the team do?

To decide whether it is worth it to conduct the experiments, or how much we should be willing to spend on experimentation in relation to the confidence to be gained and the cost of being wrong, the team could use a technique known as Value of Information (Vol) to settle the questions. The Vol technique is used to calculate the expected gain from a reduction in the uncertainty surrounding a decision through some form of data collection exercise—in this case, the construction of prototypes. To use Vol, the team will need to assess the following parameters: the cost of making the wrong design choice, the cost of performing the experiments, the team's level of confidence in each design choice, and their level of confidence in the results of the experiments. Using these estimates, Vol then applies Bayes's Theorem to calculate two quantities: the expected value of perfect information (EVPI) and the expected value of sample or imperfect information (EVSPI). EVPI denotes the maximum one should be willing to pay for the experiments, were they to provide definitive results (e.g., no false positives or false negatives). EVSPI represents how much one should be willing to spend knowing that the results of the experiment might not identify the right solution with 100 percent certainty.

As these results represent expected values, they should be evaluated in the context of the team's appetite for risk.

—Eduardo Miranda

20.4 More on ADD Step 5: Producing Structures

Design concepts per se won't help you satisfy your drivers unless you produce *structures*; that is, you need to identify and connect elements that are derived from the selected design concepts. This is the “instantiation” phase for architectural elements in ADD: creating elements and relationships between them, and associating responsibilities with these elements. Recall

that the architecture of a software system is composed of a set of structures. As we saw in Chapter 1, these structures can be grouped into three major categories:

- Module structures, which are composed of elements that exist at development time, such as files, modules, and classes
- Component and connector (C&C) structures, which are composed of elements that exist at runtime, such as processes and threads
- Allocation structures, which are composed of both software elements (from a module or C&C structure) and non-software elements that may exist both at development and at runtime, such as file systems, hardware, and development teams

When you instantiate a design concept, you may actually affect more than one structure. For example, in a particular iteration, you might instantiate the passive redundancy (warm spare) pattern, introduced in Chapter 4. This will result in both a C&C structure and an allocation structure. As part of applying this pattern, you will need to choose the number of spares, the degree to which the state of the spares is kept consistent with that of the active node, a mechanism for managing and transferring state, and a mechanism for detecting the failure of a node. These decisions are responsibilities that must live somewhere in the elements of a module structure.

Instantiating Elements

Here's how instantiation might look for each of the design concept categories:

- *Reference architectures.* In the case of reference architectures, instantiation typically means that you perform some sort of customization. This will require you to add or remove elements that are part of the structure that is defined by the reference architecture. For example, if you are designing a web application that needs to communicate with an external application to handle payments, you will probably need to add an integration component alongside the traditional presentation, business, and data tiers.
- *Patterns.* Patterns provide a generic structure composed of elements, along with their relationships and their responsibilities. As this structure is generic, you will need to adapt it to your specific problem. Instantiation usually involves transforming the generic structure defined by the pattern into a specific one that is adapted to the needs of the problem you are solving. For example, consider the client-server architectural pattern. It establishes the basic elements of computation (i.e., clients and servers) and their relationships (i.e., connection and communication), but does not specify how many clients or servers you should use for your problem, or what the functionality of each should be, or which clients should talk to which servers, or which communication protocol they should use. Instantiation fills in these blanks.
- *Tactics.* This design concept does not prescribe a particular structure. Thus, to instantiate a tactic, you may adapt a different type of design concept (that you're already using) to realize the tactic. Alternatively, you may utilize a design concept that, without any need for adaptation, already realizes the tactic. For example, you might (1) select a security tactic of *authenticating actors* and instantiate it through a custom-coded solution that you weave into your preexisting login process; or (2) adopt a security pattern that

includes actor authentication; or (3) integrate an externally developed component such as a security framework that authenticates actors.

- *Externally developed components.* The instantiation of these components may or not imply the creation of new elements. For example, in the case of object-oriented frameworks, instantiation may require you to create new classes that inherit from the base classes defined in the framework. This will result in new elements. An example that does not involve the creation of new elements is specifying configuration options for a chosen technology, such as the number of threads in a thread pool.

Associating Responsibilities and Identifying Properties

When you are creating elements by instantiating design concepts, you need to consider the responsibilities that are allocated to these elements. For example, if you instantiate the microservices architecture pattern (Chapter 5), you need to decide what the microservices will do, how many of each you will deploy, and what the properties of those microservices will be. When instantiating elements and allocating responsibilities, you should keep in mind the design principle that elements should have high cohesion (internally), be defined by a narrow set of responsibilities, and demonstrate low coupling (externally).

An important aspect that you need to consider when instantiating design concepts is the properties of the elements. This may involve aspects such as the configuration options, statefulness, resource management, priority, or even hardware characteristics (if the elements that you created are physical nodes) of the chosen technologies. Identifying these properties supports analysis and the documentation of your design rationale.

Establishing Relationships between the Elements

The creation of structures also requires making decisions with respect to the relationships that exist between the elements and their properties. Consider again the client-server pattern. In instantiating this pattern, you need to decide which clients will talk to which servers, via which ports and protocols. You also need to decide whether communication will be synchronous or asynchronous. Who initiates interactions? How much information is transferred and at what rate?

These design decisions can have a significant impact with respect to achieving QAs such as performance.

Defining Interfaces

Interfaces establish a contractual specification that allows elements to collaborate and exchange information. They may be either external or internal.

External interfaces are interfaces of other systems with which your system must interact. These may form constraints for your system, since you usually cannot influence their specification. As we noted earlier, establishing a system context at the beginning of the design

process is useful to identify external interfaces. Since external entities and the system under development interact via interfaces, there should be at least one external interface per external system (as shown in Figure 20.2).

Internal interfaces are interfaces between the elements that result from the instantiation of design concepts. To identify the relationships and the interface details, you need to understand how the elements interact with each other to support use cases or QA scenarios. As we said in Chapter 15 in our discussion of software Interfaces, “interacts” means anything one element does that can impact the processing of another element. A particularly common type of interaction is the runtime exchange of information.

Behavioral representations such as UML sequence diagrams, statecharts, and activity diagrams (see Chapter 22) allow you to model the information that is exchanged between elements during execution. This type of analysis is also useful to identify relationships between elements: If two elements need to exchange information directly or otherwise depend on each other, then a relationship between these elements exists. Any information that is exchanged becomes part of the specification of the interface.

The identification of interfaces is usually not performed equally across all design iterations. When you are starting the design of a greenfield system, for example, your first iterations will produce only abstract elements such as layers; these elements will then be refined in later iterations. The interfaces of abstract elements such as layers are typically underspecified. For example, in an early iteration you might simply specify that the UI tier sends “commands” to the business logic tier, and the business logic tier sends “results” back. As the design process proceeds, and particularly when you create structures to address specific use cases and QA scenarios, you will need to refine the interfaces of the elements that participate in these interactions.

In some special cases, identifying the appropriate interfaces may be greatly simplified. For example, if you choose a complete technology stack or a set of components that have been designed to interoperate, then the interfaces will already be defined by those technologies. In such a case, the specification of interfaces is a relatively trivial task, as the chosen technologies have “baked in” many interface assumptions and decisions.

Finally, be aware that not all of the internal interfaces need to be identified in any given ADD iteration. Some may be delegated to later design activities.

20.5 More on ADD Step 6: Creating Preliminary Documentation during the Design

As we will see in Chapter 22, software architecture is documented as a set of *views*, which represent the different structures that compose the architecture. The formal documentation of these views is not part of ADD. Structures, however, are produced as part of design. Capturing them, even if they are represented informally (as sketches), along with the design decisions that led you to create these structures, is a task that should be performed as part of normal ADD activities.

Recording Sketches of the Views

When you produce structures by instantiating the design concepts that you have selected to address a particular design problem, you will typically not only produce these structures in your mind but also create some *sketches* of them. In the simplest case, you will produce these sketches on a whiteboard, a flipchart, a drawing tool, or even just a piece of paper. Additionally, you may use a modeling tool to draw the structures in a more rigorous way. The sketches that you produce are an initial documentation for your architecture that you should capture and that you may flesh out later, if necessary. When you create sketches, you don't necessarily need to use a more formal language such as UML—although if you're fluent and comfortable with this process, please do so. If you use some informal notation, you should be careful in maintaining consistency in the use of symbols. Eventually, you will need to add a legend to your diagrams to provide clarity and avoid ambiguity.

You should develop a discipline of writing down the responsibilities that you allocate to the elements as you create the structures. The reasons for this are simple: As you identify an element, you are determining some responsibilities for that element in your mind. Writing them down *at that moment* ensures that you won't have to remember the intended responsibilities later. Also, it is easier to write down the responsibilities associated with your elements gradually, rather than documenting all of them together at a later time.

Creating this preliminary documentation as you design the architecture requires some discipline. The benefits are worth the effort, though, as you will be able to later produce the more detailed architecture documentation relatively easily and quickly. One simple way to document responsibilities, if you are using a whiteboard or a flipchart, is to take a photo of the sketch that you have produced and paste it in a document, along with a table that summarizes the responsibilities of every element depicted in the diagram (see an example in Figure 20.4). If you use a design tool, you can select an element to create and use the text area that usually appears in the properties sheet of the element to document its responsibilities, and then generate the documentation automatically.

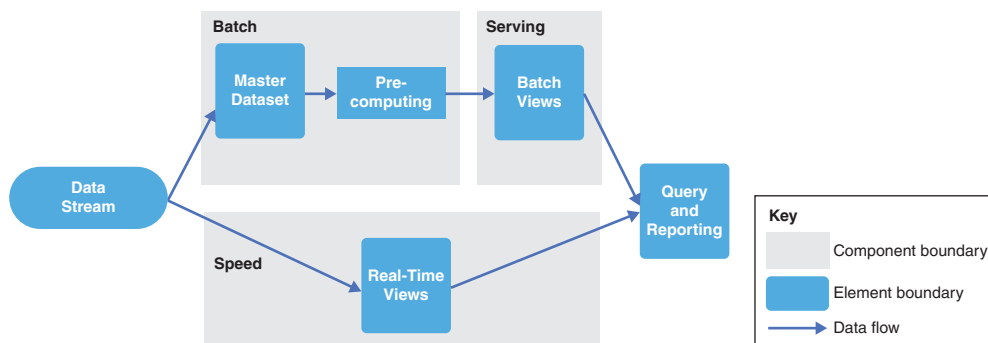


FIGURE 20.4 Example preliminary documentation

The diagram is complemented by a table that describes the element's responsibilities. Table 20.1 serves this purpose for some of the elements identified in Figure 20.4.

TABLE 20.1 Elements and Responsibilities

Element	Responsibility
Data Stream	This element collects data from all data sources in real time, and dispatches it to both the Batch Component and the Speed Component for processing.
Batch	This is responsible for storing raw data and pre-computing the Batch Views to be stored in the Serving Component.
...	...

Of course, it's not necessary to document *everything* at this stage. The three purposes of documentation are analysis, construction, and education. At the moment you are designing, you should choose a documentation purpose and then document to fulfill that purpose, based on your risk mitigation concerns. For example, if you have a critical QA scenario that your architecture design needs to meet, and if you will need to prove the proposed design satisfies this criterion in an analysis, then you must take care to document the information that is relevant for the analysis to be satisfactory. Likewise, if you anticipate having to train new team members, then you should sketch a C&C view of the system, showing how it operates and how the elements interact at runtime, and perhaps a module view of the system, showing at least the major layers or subsystems.

Finally, remember as you are documenting that your design may eventually be analyzed. Consequently, you need to think about which information should be documented to support this analysis.

Recording Design Decisions

In each design iteration, you will make important design decisions to achieve your iteration goal. When you study a diagram that represents an architecture, you might see the end product of a thought process but can't always easily understand the decisions that were made to achieve this result. Recording design decisions *beyond* the representation of the chosen elements, relationships, and properties is fundamental to help clarify how you arrived at the result—that is, the design rationale. We delve into this topic in detail in Chapter 22.

20.6 More on ADD Step 7: Perform Analysis of the Current Design and Review the Iteration Goal and Achievement of the Design Purpose

At the end of an iteration, it is prudent to do some analysis to reflect on the design decisions that you just made. We describe several techniques to do so in Chapter 21. One kind of analysis that you need to perform at this point is to assess whether you have done enough design work. In particular:

- How much design do you need to do?
- How much design have you done so far?
- Are you finished?

Practices such as the use of backlogs and Kanban boards can help you track the design progress and answer these questions.

Use of an Architectural Backlog

An architectural backlog is a to-do list of the pending actions that still need to be performed as part of the architecture design process. Initially, you should populate the design backlog with your drivers, but other activities that support the design of the architecture can also be included—for example:

- Creation of a prototype to test a particular technology or to address a specific QA risk
- Exploration and understanding of existing assets (possibly requiring reverse engineering)
- Issues uncovered in a review of the design decisions made to this point

Also, you may add more items to the backlog as decisions are made. As a case in point, if you choose a reference architecture, you will probably need to add specific concerns, or QA scenarios derived from them, to the architectural design backlog. For example, if we choose a web application reference architecture and discover that it does not provide session management, then that becomes a concern that needs to be added to the backlog.

Use of a Design Kanban Board

Another tool that can be used to track design progress is a Kanban board, such as the one shown in Figure 20.5. This board establishes three categories of backlog items: “Not Yet Addressed,” “Partially Addressed,” and “Completely Addressed.”

At the beginning of an iteration, the inputs to the design process become entries in the backlog. Initially (in step 1), the entries in your backlog for this design round should be located in the “Not Yet Addressed” column of the board. When you begin a design iteration, in step 2, the backlog entries that correspond to the drivers that you address in the design iteration goal should be moved to the “Partially Addressed” column. Finally, once you finish an iteration and the analysis of your design decisions reveals that a particular driver has been addressed (step 7), the entry should be moved to the “Completely Addressed” column of the board.

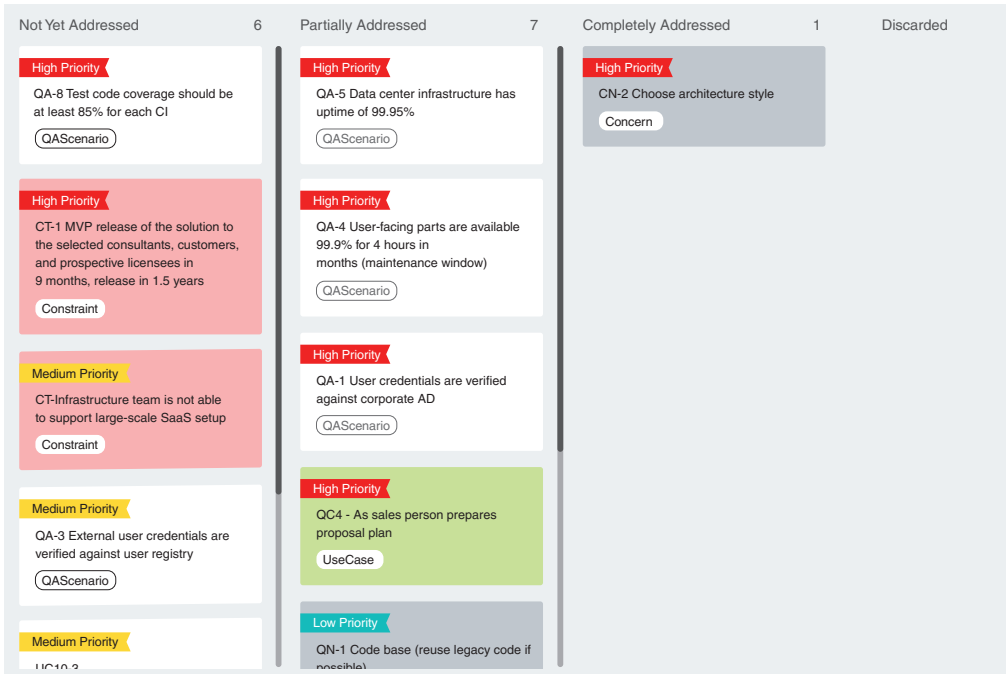


FIGURE 20.5 A Kanban board used to track design progress

It is important to establish clear criteria that will allow a driver to be moved to the “Partially Addressed” or “Completely Addressed” columns. A criterion for “Completely Addressed” may be, for example, that the driver has been analyzed or that it has been implemented in a prototype, and you determine that the requirements for that driver have been satisfied. Drivers that are selected for a particular iteration may not be completely addressed in that iteration. In that case, they should remain in the “Partially Addressed” column.

It can be useful to select a technique that will allow you to differentiate the entries in the board according to their priority. For example, you might use different colors for entries, depending on the priority.

A Kanban board makes it easy to visually track the advancement of design, as you can quickly see how many of the (most important) drivers are being or have been addressed in the iteration. This technique also helps you decide whether you need to perform additional iterations. Ideally, the design round is terminated when a majority of your drivers (or at least the ones with the highest priority) are located under the “Completely Addressed” column.

20.7 Summary

Design is hard. Methods are needed to make it more tractable (and repeatable). In this chapter, we discussed the attribute-driven design (ADD) method in detail; it allows an architecture to be designed in a systematic and cost-effective way.

We also discussed several important aspects that need to be considered in the steps of the design process. These aspects include the identification and selection of design concepts, their use in producing structures, the definition of interfaces, the production of preliminary documentation, and ways to track design progress.

20.8 For Further Reading

The first version of ADD, initially called “Architecture-Based Design,” was documented in [Bachmann 00b].

A description of ADD 2.0 was subsequently published in 2006. It was the first method to focus specifically on QAs and their achievement through the selection of different types of structures and their representation through views. Version 2.0 of ADD was first documented in an SEI Technical Report [Wojcik 06].

The version of ADD described in this chapter is ADD 3.0. Some important improvements over the original version include giving more consideration to the selection of implementation technologies as primary design concepts, considering additional drivers such as design purpose and architectural concerns, making initial documentation and analysis be explicit steps of the design process, and providing guidance in how to begin the design process and how to use it in Agile settings. An entire book [Cervantes 16] is devoted to architecture design using ADD 3.0. Some of the concepts of ADD 3.0 were first introduced in an *IEEE Software* article [Cervantes 13].

George Fairbanks wrote an engaging book that describes a risk-driven process of architecture design, entitled *Just Enough Software Architecture: A Risk-Driven Approach* [Fairbanks 10].

The Value of Information technique dates from the 1960s [Raiffa 00]. A more modern treatment can be found in [Hubbard 14].

For a general approach on systems design, you can read the classic tome by Butler Lampson [Lampson 11].

Using concepts of lean manufacturing, Kanban is a method for scheduling the production of a system, as described by Corey Ladas [Ladas 09].

20.9 Discussion Questions

1. What are the advantages of following an established method for design? What are the disadvantages?
2. Is performing architectural design compatible with an agile development methodology? Choose an agile method and discuss ADD in that context.
3. What is the relationship between design and analysis? Are there some kinds of knowledge that you need for one but not the other?
4. If you had to argue for the value of creating and maintaining architectural documentation to your manager during the design process, what arguments would you put forward?
5. How would your realization of the steps of ADD differ if you were doing greenfield development versus brownfield development?

This page intentionally left blank

21



Evaluating an Architecture

A doctor can bury his mistakes, but an architect can only advise his clients to plant vines.

—Frank Lloyd Wright

In Chapter 2, we said that one major reason architecture is important is that you can predict the quality attributes of any system derived from it, *before you build the system*, by examining its architecture. That's a pretty good deal, if you think about it. And this is the chapter where that capability comes home.

Architecture evaluation is the process of determining the degree to which 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're designing will be able to provide all that's expected of it. That's the role of evaluation, which is based on analyzing the alternatives. Fortunately, there are mature methods to analyze architectures that use many of the concepts and techniques you've already learned in this book.

To be useful, the cost of evaluation needs to be less than the value it provides. Given this relationship, an important question is "How much time and money is the evaluation going to cost?" Different evaluation techniques come with different costs, but all of them can be measured in terms of the time spent by the people involved in the preparation, execution, and follow-up of the evaluation activities.

21.1 Evaluation as a Risk Reduction Activity

Every architecture comes with risks. The output of an architecture evaluation includes an identification of risky portions of the architecture. A risk is an event that has both an impact and a probability. The estimated cost of a risk is the probability of that event occurring multiplied by the cost of the impact. Fixing those risks is not an output of the evaluation. Once the risks have been identified, then fixing them is, like the evaluation itself, a cost/benefit issue.

Applying this concept to architecture evaluation, you can see that if the system being constructed costs millions or billions of dollars or has large safety-critical implications, then the impact of a risk event will be large. By comparison, if the system is a console-based game costing tens or hundreds of thousands of dollars to create, then the impact of a risk event will be considerably smaller.

The probability of a risk event is related to, among other things, howprecedented or unprecedented the system under development and its architecture are. If you and your organization have long and deep experience in this domain, then the probability of producing a bad architecture is less than if this project is your first go.

Thus evaluations act like an insurance policy. How much insurance you need depends on how exposed you are to the risk of an unsuitable architecture and your risk tolerance.

Evaluations can be done throughout the development process at different phases, with different evaluators, and with differences in how the evaluation is performed—we'll cover some of the options in this chapter. Regardless of their precise details, evaluations build on the concepts you have already learned: Systems are constructed to satisfy business goals, business goals are exemplified by quality attribute scenarios, and quality attribute goals are achieved through the application of tactics and patterns.

21.2 What Are the Key Evaluation Activities?

Regardless of who performs the evaluation and when it is performed, an evaluation is based on architectural drivers—primarily architecturally significant requirements (ASRs) expressed as quality attribute scenarios. Chapter 19 describes how to determine ASRs. The number of ASRs that enter into the evaluation is a function of the contextual factors and the cost of the evaluation. We next describe the possible contextual factors for architecture evaluation.

An evaluation can be carried out at any point in the design process where a candidate architecture, or at least a coherent reviewable part of one, exists.

Every evaluation should include (at least) these steps:

1. *The reviewers individually ensure that they understand the current state of the architecture.* This can be done through shared documentation, through a presentation by the architect, or through some combination of these.
2. *The reviewers determine a number of drivers to guide the review.* These drivers may already be documented, or they can be developed by the review team or by additional stakeholders. Typically the most important drivers to review are the high-priority quality attribute scenarios (and not, say, purely functional use cases).
3. *For each scenario, each reviewer should determine whether the scenario is satisfied.* The reviewers pose questions to determine two types of information. First, they want to determine that the scenario is, in fact, satisfied. This could be done by having the architect walk through the architecture and explain how the scenario is satisfied. If the architecture is already documented, then the reviewers can use that documentation to make this assessment. 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. The reviewers may pose alternatives to any risky aspect of the current design that might better satisfy the scenario. These alternatives should be subjected to the same type of analysis. Time constraints play a role in determining how long this step is allowed to continue.

4. *The reviewers capture potential problems exposed during the prior step.* This list of potential problems forms the basis for the follow-up of the review. If the potential problem is a real problem, then either it must be fixed or a decision must be explicitly made by the designers and the project manager that they are willing to accept the risk.

How much analysis should you do? Decisions made to achieve one of the driving architectural requirements should be subject to more analysis than others, because they will shape critical portions of the architecture. 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.
- *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.

21.3 Who Can Perform the Evaluation?

Evaluators should be highly skilled in the domain and the various quality attributes for which the system is to be evaluated. Excellent organizational and facilitation skills are also a must for evaluators.

Evaluation by the Architect

Evaluation is done—implicitly or explicitly—every time the architect makes a key design decision to address an ASR or completes a design milestone. This evaluation involves deciding among the competing alternatives. Evaluation by the architect is an integral part of the process of architecture design, as we discussed in Chapter 20.

Evaluation by Peer Review

Architectural designs to address ASRs can be peer reviewed, just as code can be peer reviewed. There should be a fixed amount of time allocated for the peer review, typically several hours to half a day.

If the designers are using the Attribute-Driven Design (ADD) process described in Chapter 20, then a peer review can be done at the end of step 7 of each ADD iteration. Reviewers should also use the tactics-based questionnaires that we presented in Chapters 4–13.

Evaluation by Outsiders

Outside evaluators can cast a more 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 to participate in the evaluation because they possess specialized knowledge or experience, such as knowledge about a quality attribute that’s important to the system being examined, skill with a particular technology being employed, 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 than by team members within the organization. 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.

21.4 Contextual Factors

For peer reviews or outside analysis, a number of contextual factors must be considered when setting up an evaluation:

- *What artifacts are available?* To perform an architectural evaluation, there must be an artifact that both describes the architecture and is readily available. Some evaluations may take place after the system is operational. In this case, some architecture recovery and analysis tools may be used to assist in discovering the architecture, to find architecture design flaws, 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.
- *Which stakeholders will participate?* The evaluation process should include a method to elicit the important stakeholders’ goals and concerns regarding the system. At this stage, it is critical to identify the individuals who are needed and ensure their participation in the evaluation.
- *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 a portion of the evaluation should be dedicated to this task.

Evaluations by peers and by outside evaluators are common enough that we have formalized processes to guide the evaluation. These processes define who should participate and which activities should occur during the evaluation. Formalizing a process enables the organization to make the process more repeatable, help the stakeholders understand what will be required and delivered by the evaluation, train new evaluators to use the process, and understand the investment required to perform the evaluation.

We begin by describing a process for outside evaluators (Architecture Tradeoff Analysis Method); we then describe a process for peer review (Lightweight Architecture Evaluation).

21.5 The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) is the process we have formalized to perform architecture evaluations. The ATAM has been used for more than two decades to evaluate software architectures of large systems in domains ranging from automotive to financial to defense. The ATAM is designed so that evaluators do not need prior familiarity with the architecture or its business goals, and the system need not be constructed yet. An ATAM exercise may be held either in person or remotely.

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; a single person may adopt several roles in an ATAM exercise. (See Table 21.1 for a description of these roles.) 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.
- *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 an identifiable customer is footing the bill for the development, a representative of that customer may be present 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 people 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, and

builders of systems interacting with the one under consideration. Their job during an evaluation is to articulate the specific quality attribute goals that the architecture should meet 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 10 to 25 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.

TABLE 21.1 ATAM Evaluation Team Roles

Role	Responsibilities
Team Leader	Sets up the evaluation; coordinates with the client, making sure the client's needs are met; establishes the evaluation contract; forms the evaluation team; sees that the final report is produced and delivered.
Evaluation Leader	Runs the evaluation; facilitates elicitation of scenarios; administers the scenario prioritization process; facilitates the evaluation of scenarios against the architecture.
Scenario Scribe	Writes scenarios in a sharable, public form during scenario elicitation; captures the agreed-on wording of each scenario, halting discussion until the exact wording is captured.
E-Scribe	Captures the proceedings in electronic form: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and the results of each scenario's analysis; also generates a list of adopted scenarios for distribution to all participants.
Questioner	Asks probing quality attribute–based questions.

Outputs of the ATAM

1. *A concise presentation of the architecture.* One requirement of the ATAM is that the architecture be presented in one hour or less, 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 exercise 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.
3. *Prioritized quality attribute requirements expressed as quality attribute scenarios.* These quality attribute scenarios take the form described in Chapter 3. The ATAM uses prioritized quality attribute scenarios as the basis for evaluating the architecture. Those scenarios may already exist (perhaps as a result of a prior requirements-capture exercise or ADD activity), but if not, they are generated by the participants as part of the ATAM exercise.
4. *A set of risks and non-risks.* An architectural risk is a decision that may lead to undesirable consequences in light of stated quality attribute requirements. Similarly, an architectural non-risk is a decision that, upon analysis, is deemed safe. The identified

risks form the basis for an architectural risk mitigation plan. These risks are the primary output of an ATAM exercise.

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.
6. *Mapping of architectural decisions to quality requirements.* Architectural decisions can be interpreted in terms of the drivers that they support or hinder. For each quality attribute scenario examined during an ATAM exercise, those architectural decisions that help to achieve it are determined and captured. They can serve as a statement of the rationales for those decisions.
7. *A set of identified sensitivity points and tradeoff points.* Sensitivity points are architectural decisions that have a marked effect on a quality attribute response. Tradeoffs occur when two or more quality attribute responses are sensitive to the same architectural decision, but one of them improves while the other degrades—hence the tradeoff.

The outputs of the ATAM exercise can be used to build a final report that recaps the method, summarizes the proceedings, captures the scenarios and their analysis, and catalogs the findings.

An ATAM-based evaluation also produces intangible results that should not be ignored. These include a sense of community on the part of the stakeholders, open communication channels between the architect and the stakeholders, and a better overall understanding among 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.

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 work out the details of the exercise. The project representatives brief the evaluators about the project so that the evaluation team can be supplemented by people who possess the appropriate expertise. Together, the two groups agree on logistics, such as the time when the evaluation will take place and technology used to support the meetings. They also agree on a preliminary list of stakeholders (by name, not just role), and negotiate when the final report will 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.
- During Phases 1 and 2, collectively known as “Evaluation,” 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 major architectural

approaches taken, and the quality attributes that are of paramount importance. During phase 1, the evaluation team meets with the project decision makers to begin information gathering and analysis. In phase 2, the architecture’s stakeholders add their input to the proceedings and analysis continues.

- In Phase 3, “Follow-up,” the evaluation team produces and delivers its final report. This report—which may be a formal document or simply a set of slides—is first circulated to key stakeholders to ensure that it contains no errors of understanding. After this review is complete, it is delivered to the client.

Table 21.2 shows the four phases of the ATAM, who participates in each phase, and the typical cumulative time spent on the activity—possibly in several segments.

TABLE 21.2 ATAM Phases and Their Characteristics

Phase	Activity	Participants	Typical Cumulative Time
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
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 (phases 1 and 2) consist of nine steps. Steps 1–6 are carried out in phase 1 with the evaluation team and the project’s decision makers—typically, the architecture team, project manager, and client. In phase 2, with all stakeholders involved, steps 1–6 are summarized and steps 7–9 are carried out.

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 Goals

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 goals motivating its development. In this step, a project decision maker (ideally the project manager

or customer representative) presents a system overview from a business perspective. This presentation should describe the following aspects of the project:

- 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 (emphasizing architecturally significant requirements)

Step 3: Present the Architecture

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 the operating system, platforms prescribed for use, and other systems with which this system must interact. Most importantly, the architect describes the architectural approaches (or patterns, or tactics, if the architect is fluent in that vocabulary) used to meet the requirements.

We expect architectural views, as introduced in Chapter 1 and described in detail in Chapter 22, to be the primary vehicle by which the architect conveys 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 satisfying important quality attribute requirements.

Step 4: Identify the Architectural Approaches

The ATAM focuses on analyzing an architecture by understanding its architectural approaches. Architectural patterns and tactics are useful for (among other reasons) the known ways in which each one affects particular quality attributes. For example, 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, whereas the active redundancy pattern promotes high availability.

Step 5: Generate a Quality Attribute Utility Tree

The quality attribute goals are articulated in detail via a quality attribute utility tree, which we introduced in Section 19.4. Utility trees 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 or implied in step 2, when the business goals were presented, but not with a 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 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 to achieve those aims. Modifiable in what way? Throughput that is how high? Ported to what platforms and in how much time? The answers to these kinds of questions are expressed as quality attribute scenarios representing architecturally significant requirements.

Recall that the utility tree is constructed by the architect and the project decision makers. Together, they determine the importance of each scenario: The architect rates the technical difficulty or risk of the scenario (on a H, M, L scale), and the project decision makers rate its business importance.

Step 6: Analyze the Architectural Approaches

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, non-risks, 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 and non-risks. 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 frequency of heartbeats determines the time for detection of a fault.
- Higher frequency leads to improved availability but also consumes more processing time and communication bandwidth (potentially leading to reduced performance). This is a tradeoff.

These issues, 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 proceeding to any 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 during step 6 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 in the figure, based on the results of this step, the evaluation team can identify and record a set of risks and non-risks, sensitivity points, and tradeoffs.

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; In(()) --> P[Primary CPU OS1]; In --> B[Backup CPU with Watchdog OS2]; P --> S[Switch CPU OS1]; B --> S; P -.-> Heartbeat 1 sec B; S --> 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, non-risks, 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 with the architect during a hiatus of a week or so. More scenarios might be analyzed during this period, if desired, or answers to questions posed in phase 1 may be clarified.

Attendees at the phase 2 meeting include an expanded list of participants, with additional stakeholders joining the discussion. 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 2–6, and shares the current list of risks, non-risks, sensitivity points, and tradeoffs. After bringing the stakeholders up to speed with the evaluation results so far, the remaining three steps can be carried out.

Step 7: Brainstorm and Prioritize Scenarios

The evaluation team asks the stakeholders to brainstorm quality attribute 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 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 analysis time. First, stakeholders are asked to merge scenarios they feel represent the same behavior or quality concern. Next, they vote for those scenarios 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. Thus, if 40 scenarios were 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.

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. Such discoveries indicate some level of disagreement about the system's important goals between the stakeholders and the architect.

1. This is a common facilitated brainstorming technique.

Step 8: Analyze the Architectural Approaches

After the scenarios have been collected and prioritized in step 7, the evaluation team guides the architect in the process of analyzing the highest-ranked scenarios. The architect explains how architectural decisions contribute to realizing each scenario. 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 the Results

In step 9, the evaluation team convenes and 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 goals 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, thereby providing a satisfying closure to the exercise. Equally 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. The following outputs are presented:

- The architectural approaches documented
- The set of scenarios and their prioritization from the brainstorming
- The utility tree
- The risks and non-risks discovered
- The sensitivity points and tradeoffs found
- Risk themes and the business goals threatened by each one

Going Off Script

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 have been 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 is a result of experiences like these.) But it was okay. 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 for the architect. In all cases, the client felt that the detailed scenarios, the analysis we were able to perform on the elicited architecture, and 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 organization was in turmoil, and the architect simply got a better offer in a calmer environment elsewhere. Usually, we don't proceed without the architect, but it was okay, because 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 no one 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 goals, utility tree, scenarios—remained 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 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 he said could give us more architectural insights. It turned out that the executives were busy at the time of our scheduled meeting. So the manager asked if we could come back and meet with them later on.

By now, phase 2 was thrown off schedule to such an extent 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 other's 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 okay.

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 making 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 okay. It wasn't the right architecture. But it was okay. There was no architect. But it was okay. The client really wanted to effect a team reorganization. In every instance, we reacted as reasonably as we could, and each time it was okay.

Why? Why, time after time, does it turn out okay? I think there are three reasons.

First, the people who commission 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 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 false note 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 okay.

—PCC (Adapted from [Clements 01b])

21.6 Lightweight Architecture Evaluation

The Lightweight Architecture Evaluation (LAE) method is intended to be used in a project-internal context where the reviewing is carried out by peers on a regular basis. It uses the same concepts as the ATAM and is meant to be performed regularly. An LAE session may be convened to focus on what has changed since the prior review—in the architecture or in the architecture drivers—or to examine a previously unexamined portion of the architecture. Because of this limited scope, many of the ATAM's steps can be omitted or shortened.

The duration of an LAE exercise depends on the number of quality attribute scenarios generated and examined, which is in turn based on the scope of the review. The number of scenarios examined depends on the importance of the system being reviewed. Thus an LAE exercise can be as short as a couple of hours or as long as a full day. It is carried out entirely by members internal to the organization.

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. In addition, an LAE exercise, because it is a lightweight process, can be done regularly; in turn, many of the steps of the method can be omitted or only briefly touched upon. The potential steps in an LAE exercise, along with our experiences with how these play out in practice, are shown in Table 21.3. The LAE exercise is typically convened by and led by the project architect.

TABLE 21.3 A Typical Agenda for Lightweight Architecture Evaluation

Step	Notes
1: Present the method steps	Assuming the participants are familiar with the process, this step may be omitted.
2: Review the business goals	The participants are expected to understand the system and its business goals and their priorities. A brief review may be done to ensure that these are fresh in everyone's mind and that there are no surprises.
3: Review the architecture	All participants are expected to be familiar with the system, so a brief overview of the architecture is presented, using at least the module and C&C views, highlighting any changes since the last review, and one or two scenarios are traced through these views.
4: Review the architectural approaches	The architect highlights the architectural approaches used for specific quality attribute concerns. This is typically done as a portion of step 3.
5: Review the quality attribute utility tree	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: Brainstorm and prioritize scenarios	A brief brainstorming activity can occur at this time to establish whether any new scenarios merit analysis.
7: Analyze the architectural approaches	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and should focus on the most recent changes to the architecture, or on a part of the architecture that the team has not previously analyzed. If the architecture has changed, the high-priority scenarios should be reanalyzed in light of these changes.
8: Capture the results	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.

There is no final report, but (as in the ATAM) a scribe is responsible for capturing results, which can then be shared and serve as the basis for risk remediation.

An entire LAE 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 less objective than an external evaluation team, and this may compromise the value of its results: One tends to hear fewer new ideas and fewer dissenting opinions. Nevertheless, this version of evaluation is inexpensive, is easy to convene, and involves relatively low ceremony, so it can be quickly deployed whenever a project wants an architecture quality assurance sanity check.

Tactics-Based Questionnaires

Another (even lighter) lightweight evaluation method that we discussed in Chapter 3 is the tactics-based questionnaire. A tactics-based questionnaire focuses on a single quality attribute at a time. It can be used by the architect to aid in reflection and introspection, or it can be used to structure a question-and-answer session between an evaluator (or evaluation team) and an architect (or group of designers). This kind of session is typically short—around one hour per quality attribute—but can reveal a great deal about the design decisions taken, and those not taken, in pursuit of control of a quality attribute and the risks that are often buried within those decisions. We have provided quality attribute–specific questionnaires in Chapters 4–13 to help guide you in this process.

A tactics-based analysis can lead to surprising results in a very short time. For example, once I was analyzing a system that managed healthcare data. We had agreed to analyze the quality attribute of security. During the session, I dutifully walked through the security tactics–based questionnaire, asking each question in turn (as you may recall, in these questionnaires each tactic is transformed into a question). For example, I asked, “Does the system support the detection of intrusions?”, “Does the system support the verification of message integrity?”, and so forth. When I got to the question “Does the system support data encryption?”, the architect paused and smiled. Then he (sheepishly) admitted that the system had a requirement that no data could be passed over a network “in the clear”—that is, without encryption. So they XOR’ed all data before sending it over the network.

This is a great example of the kind of risk that a tactics-based questionnaire can uncover, very quickly and inexpensively. Yes, they had met the requirement in a strict sense—they were not sending any data in the clear. But the encryption algorithm that they chose could be cracked by a high school student with modest abilities!

—RK

21.7 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.

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 analyzing each high-priority scenario. The decisions can then be understood in terms of risks or non-risks to find any trouble spots in the architecture.

Lightweight evaluations can be performed regularly as part of a project's internal peer review activities. Lightweight Architecture Evaluation, based on the ATAM, provides an inexpensive, low-ceremony architecture evaluation that can be carried out in less than a day.

21.8 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 sei.cmu.edu/library and searching for "ATAM case study."

Several lighter-weight architecture evaluation methods have been developed. They can be found in [Bouwers 10], [Kanwal 10], and [Bachmann 11].

Analyses of the kinds of insights derived from an ATAM can be found in [Bass 07] and [Bellomo 15].

21.9 Discussion Questions

1. Think of a software system that you're working on. Prepare a 30-minute presentation on the business goals 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. Calculate the cost of an ATAM-based evaluation for a large enterprise-scale system's architecture. Assume a fully burdened labor rate of \$250,000 per year for the participants. Assuming that an evaluation uncovers an architectural risk and mitigating this risk saves 10 percent of project costs, under what circumstances would this ATAM be a sensible choice for a project?
4. Research a costly system failure that could be attributed to one or more poor architectural decisions. Do you think an architecture evaluation might have caught the risks? If so, compare the cost of the failure with the cost of the evaluation.
5. 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?
6. 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?
7. Under what circumstances would you want to employ a full-strength ATAM, and under what circumstances would you want to employ an LAE?

This page intentionally left blank



Documenting an Architecture

Documentation is a love letter that you write to your future self.
—Damian Conway

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 organized 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, who has forgotten the details of what the architecture includes, or when that person has left the project and someone else is now the architect.

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, 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.

Note that "documentation" doesn't necessarily mean producing a physical, printed, book-like artifact. Online documentation such as a wiki, hosted in ways that can engender discussion, stakeholder feedback, and searching, is an ideal forum for architecture documentation. Also, don't think of documentation as a step that is distinct from and follows design. The language you use to explain the architecture to others can be used by you as you carry out your design work. Design and documentation are, ideally, the same piece of work.

22.1 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 or forensics. It should have enough information to serve as a basis for analysis.

Architecture documentation can be seen as 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.

Many different kinds of people will have an interest in architecture documentation. They hope and expect that this documentation will help them do their respective jobs. Understanding the uses of architecture documentation is essential, as those uses determine the important information to capture.

Fundamentally, architecture documentation has four 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.* Its 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 project's future architect. That may be the same person (as noted in the quotation that opened this chapter) or it may be a replacement, but in either case the future architect 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.

- We enumerate the stakeholders for architecture, and its documentation, in Section 22.8.
3. *Architecture documentation serves as the basis for system analysis and construction.* Architecture tells implementers which modules to implement and how those modules are wired together. These dependencies determine the other teams with which the development team for the module must communicate.

For those interested in the design's ability to meet the system's quality objectives, the architecture documentation serves as fodder for evaluation. It must contain the information necessary to evaluate a variety of attributes, such as security, performance, usability, availability, and modifiability.

4. *Architecture documentation serves as the basis for forensics when an incident occurs.* When an incident occurs, someone is responsible for tracking down both the immediate

cause of the incident and the underlying cause. Information about the flow of control immediately prior to the incident will provide the “as executed” architecture. For example, a database of interface specifications will provide context for the flow of control, and component descriptions will indicate what should have happened in each component on the trace of events.

For the documentation to continue to provide value over time, it needs to be kept up to date.

22.2 Notations

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

- *Informal notations.* Views may be depicted (often graphically) using general-purpose diagramming and editing tools and visual conventions chosen for the system at hand. Most box-and-line drawings you’ve probably seen fall into this category—think PowerPoint or something similar, or hand-drawn sketches on a whiteboard. The semantics of the description are characterized in natural language, and cannot be formally analyzed.
- *Semiformal notations.* Views may be expressed in a standardized notation that prescribes graphical elements and rules of construction, but does not provide a complete semantic treatment of the meaning of those elements. Rudimentary analysis can be applied to determine if a description satisfies syntactic properties. UML and its system-engineering adjunct SysML are semiformal notations in this sense. Most widely used commercially available modeling tools employ notations in this category.
- *Formal notations.* Views may be described in a notation that has a precise (usually mathematically based) semantics. Formal analysis of both syntax and semantics is possible. A variety of formal notations for software architecture are 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 other cases, 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 formal notations is rare.

Typically, more formal notations take more time and effort to create and understand, but repay this effort with reduced ambiguity and more opportunities for analysis. Conversely, more informal notations are easier to create, but 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 diagram tell you very much about the system’s likelihood of being delivered on time. You should choose your

notations and representation languages while keeping in mind the important issues you need to capture and reason about.

22.3 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, along with 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* leads to a basic 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.

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 reverse engineering, or the blueprint for project estimation and planning.

Different views also expose different quality attributes to different degrees. In turn, the quality attributes that are of most concern to you and the other stakeholders in the system’s development will affect which views you choose to document. For instance, a *module view* will let you reason about your system’s maintainability, a *deployment view* will let you reason about your system’s performance and reliability, and so forth.

Because different views support different goals and uses, we do not advocate using any 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 expected benefits of creating and maintaining a particular view outweigh its costs.

The choice of views is driven by the need to document a particular pattern in your design. Some patterns are composed of modules, others consist of components and connectors, and still others have deployment considerations. Module views, component-and-connector (C&C) views, and allocation views are the appropriate mechanism for representing these considerations, respectively. These categories of views correspond, of course, to the three categories of architectural structures described in Chapter 1. (Recall from Chapter 1 that a structure is a

collection of elements, relations, and properties, whereas a view is a representation of one or more architectural structures.)

In this section, we explore these three categories of structure-based views and then introduce a new category: quality views.

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 view has a collection of properties assigned to it. These properties express important information associated with each module and the relationships among the modules, as well as constraints on the module. Example properties include responsibilities, visibility information (what other modules can use it), 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, it determines how a system’s source code is decomposed into units, what kinds of assumptions each unit can make about services provided by other units, and how those units are aggregated into larger ensembles. It also includes shared 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.

The documentation of any software architecture is unlikely to be complete without at least one module view. Table 22.1 summarizes the characteristics of module views.

TABLE 22.1 Summary of 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 ▪ <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 topological constraints, such as limitations on the visibility between modules.
Usage	<ul style="list-style-type: none"> ▪ Blueprint for construction of the code ▪ Analysis of the impact of changes ▪ 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 data model

Properties of modules that help to guide implementation or are input into 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, which is 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 that role with much more certainty. Responsibilities should be described in sufficient detail to make clear to the reader what each module does. A module's responsibilities are often captured by tracing to a project's requirements specification, if there is one.
- *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:
 - *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` (implementation of Account functionality), and perhaps even a unit test `AccountTest.java`.
 - *Test information.* The module's test plan, test cases, test harness, 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 its authors and particular changes, may help you when you're performing maintenance activities.

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 because they explain the structure of the code base to a new developer on the team.

Conversely, 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 purposes, we rely on component-and-connector and allocation views.

Component-and-Connector Views

C&C views show elements that have some runtime presence, such as processes, services, objects, clients, servers, and data stores. These elements are termed *components*. Additionally, C&C 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. Example C&C views include client-server, microservice, and communicating processes.

A component in a C&C view may represent a complex subsystem, which itself can be described as a C&C subarchitecture. A component's subarchitecture may employ a different pattern than the one in which the component appears.

Simple examples of connectors include 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 need not be binary; that is, they need not have exactly two components with which they interact. For example, a publish-subscribe connector might have an arbitrary number of publishers and subscribers. 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. Compatibility often is defined in terms of information type and protocol. For example, if a web server expects encrypted communication via HTTPS, then the client must perform the encryption.

An element (component or connector) of a C&C view will have various properties associated with it. Specifically, every element should have a name and type, with its additional properties depending on the type of component or connector. As an architect, you should define values for the properties that support the intended analyses for the particular C&C view. 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, or jitter 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? If relevant, how much energy does it consume? 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 the end-to-end 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 potential 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 and bottlenecks.
- *Runtime extensibility.* Does the messaging structure support evolving data exchanges? Can the connectors be adapted to process those new message types?

C&C views are commonly used to show 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 22.2 summarizes the characteristics of C&C views.

TABLE 22.2 Summary of C&C Views

Elements	<ul style="list-style-type: none"> ▪ <i>Components:</i> principal processing units and data stores. ▪ <i>Connectors:</i> pathways of interaction between components.
Relations	<ul style="list-style-type: none"> ▪ <i>Attachments:</i> Components are associated with connectors to yield a graph.
Constraints	<p>Components can only be attached to connectors, and connectors can only be attached to components.</p> <ul style="list-style-type: none"> ▪ Attachments can only be made between compatible components and connectors. ▪ Connectors cannot appear in isolation; a connector must be attached to a component.
Usage	<p>Show how the system works.</p> <ul style="list-style-type: none"> ▪ Guide development by specifying the 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 terms of the semantics that they can convey, 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 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 such as interfaces, properties, and behavioral

descriptions. UML components also distinguish between component types and component instances, which is useful when defining view-specific component types.

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 in such a view varies; it 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 22.3 summarizes the characteristics of allocation views. These 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 22.3 Summary of Allocation Views

Elements	Software element and environmental element. A software element has properties that are <i>required</i> of the environment. 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.
Constraints	Varies by view.
Usage	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 would also be relevant and potentially 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, during 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.

Software elements and environmental elements have properties in allocation views. One 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. For example, to ensure its *required* response time, a component has to execute on (be allocated to) a processor that *provides* sufficiently fast processing power. As 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 to work with that module.

Allocation views can depict either static or dynamic views. A static view illustrates a fixed allocation of resources in an environment. A dynamic view shows the conditions and the triggers for which allocation of resources changes. For example, some systems provision and utilize new resources as their loads increase. 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 view 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 allocation can also be changed, depending on the “load”—in this case, the load on development teams already at work.

Quality Views

Module, C&C, and allocation views are all structural views: They primarily show the structures that the architect has designed into the architecture to satisfy functional and quality attribute requirements.

These views are excellent choices for guiding and constraining downstream developers, whose primary job is to implement those structures. However, in systems in which certain quality attributes (or, for that matter, any 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 cumbersome to combine (e.g., because the element types shown in each structure are different).

Another kind of view, which we call a *quality view*, can be tailored for specific stakeholders or to address specific concerns. 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 depict 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 properties would include other security measures (e.g., physical security) in the system’s environment. The security view would also show the operation of security protocols and where and how humans interact with the security elements. Finally, it would capture how the system responds 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, various network channels, quality-of-service parameter values, and areas of concurrency. Such a view can be used to analyze certain kinds of performance and reliability, such as deadlock or race condition detection. In addition, it 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 the architect identify the sources of errors and specify appropriate corrective actions for each. Finally, it would facilitate root-cause analysis in those cases.
- A *reliability* view would model reliability mechanisms such as replication and switch-over. 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 the concerns of the architecture's stakeholders.

22.4 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. Of course, if those views were irrevocably different, with no association with one another, no one would be able to understand the system as a whole. However, because all structures in an architecture are part of the 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, independently 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*. A combined view contains elements and relations that come from two or more other views. Such 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 appeared in two separate views. This works well if the relationship between the two views is tight—that is, if there are strong associations between elements in one view and elements in the other view. In such a case, the structure described by the combined view will be easier to understand than the two views seen separately. In an overlay, the elements and the relations keep the types as defined in their constituent views.

The following combinations of views often occur quite naturally:

- *C&C views with each other*. Because all C&C views 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 any C&C view that shows processes.* Processes are the components that are deployed onto processors, virtual machines, or containers. Thus there is a strong association between the elements in these views.
- *Decomposition view and any work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses. In addition, these modules populate layers.

Figure 22.1 shows an example of a combined view that is an overlay of client-server, multi-tier, and deployment views.

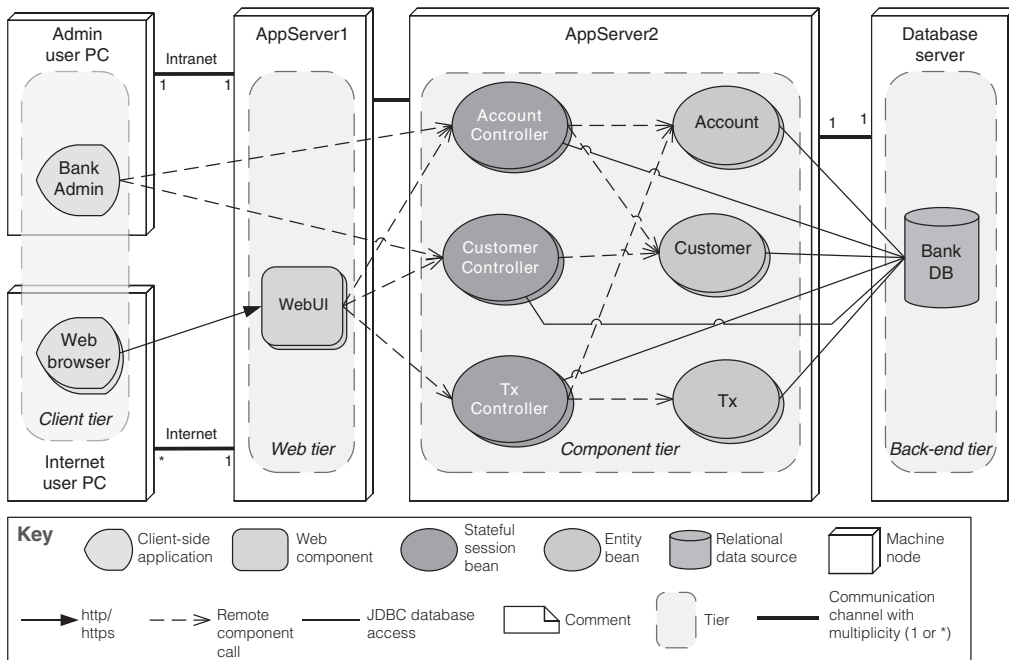


FIGURE 22.1 A combined view

22.5 Documenting Behavior

Documenting an architecture requires behavior documentation that complements the 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 provide information about the characteristics of individual elements and their resource consumption, as well as patterns of interaction among them—that is, how they behave in relation to each other. In this section, we provide guidance as to what types of things you will want to document to reap these benefits.

Two kinds of notations are available for documenting behavior: trace-oriented and comprehensive.

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 one might conceivably describe all possible traces to generate the equivalent of a comprehensive behavioral model, trace-oriented documentation does not really seek to do so. Here 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 notations.

- *Use cases* describe how actors can use a system to accomplish their goals; they are frequently used to capture the functional requirements for a system. UML provides a graphical notation for use case diagrams but does not specify how the text of a use case should be written. The UML use case diagram is a good way to provide an overview of the actors and the behavior of a system. Its description, which is textual, should include the following items: the use case name and a brief description, the actor or actors who initiate the use case (primary actors), other actors who participate in the use case (secondary actors), the flow of events, alternative flows, and non-success cases.
- A UML *sequence diagram* shows a sequence of interactions among instances of elements pulled from the structural documentation. It is useful, when designing a system, for identifying where interfaces need to be defined. The sequence diagram shows only the instances participating in the scenario being documented. It has two dimensions: vertical, representing time, and horizontal, representing the various instances. The interactions are arranged in time sequence from top to bottom. Figure 22.2 is an example of a sequence diagram that illustrates the basic UML notation. Sequence diagrams are not explicit about showing concurrency. If that is your goal, use activity diagrams instead.

As shown in Figure 22.2, objects (i.e., element instances) have a *lifeline*, drawn as a vertical dashed line down 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 message sent over a network, a function call, or an event sent through a queue. 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 an 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.

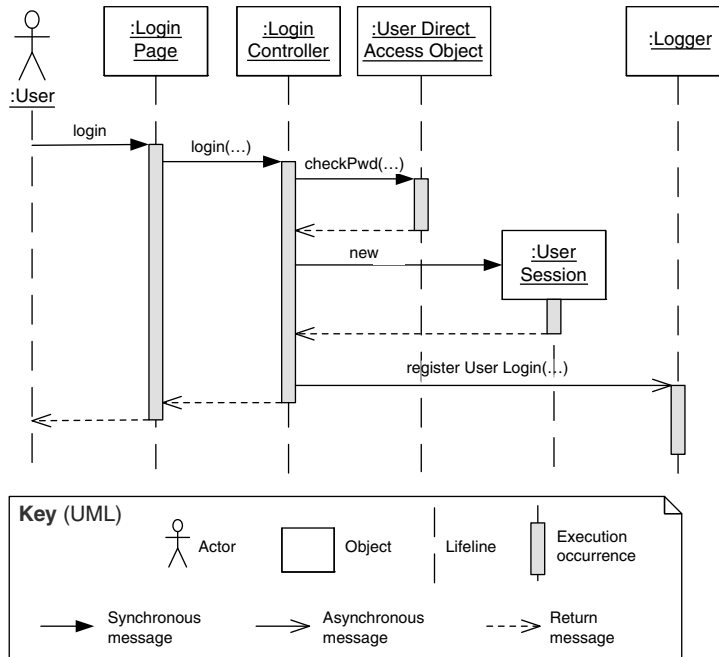


FIGURE 22.2 A simple example of a UML sequence diagram

- A UML *communication diagram* shows a graph of interacting elements and annotates each interaction with a number denoting its order. Similar 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. Such diagrams are not useful when understanding of concurrent actions is important, as when conducting a performance analysis.
- UML *activity diagrams* are similar to flowcharts. 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. Notably, 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. These 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.

Unlike sequence and communication diagrams, activity diagrams don't show the actual operations being performed on specific objects. Thus these diagrams are useful to broadly describe the steps in a specific workflow. Conditional branching (shown by a diamond symbol) allows a single diagram to represent multiple traces, although an activity diagram usually does not attempt to show all possible traces or the complete behavior for the system (or part of it). Figure 22.3 shows an activity diagram.

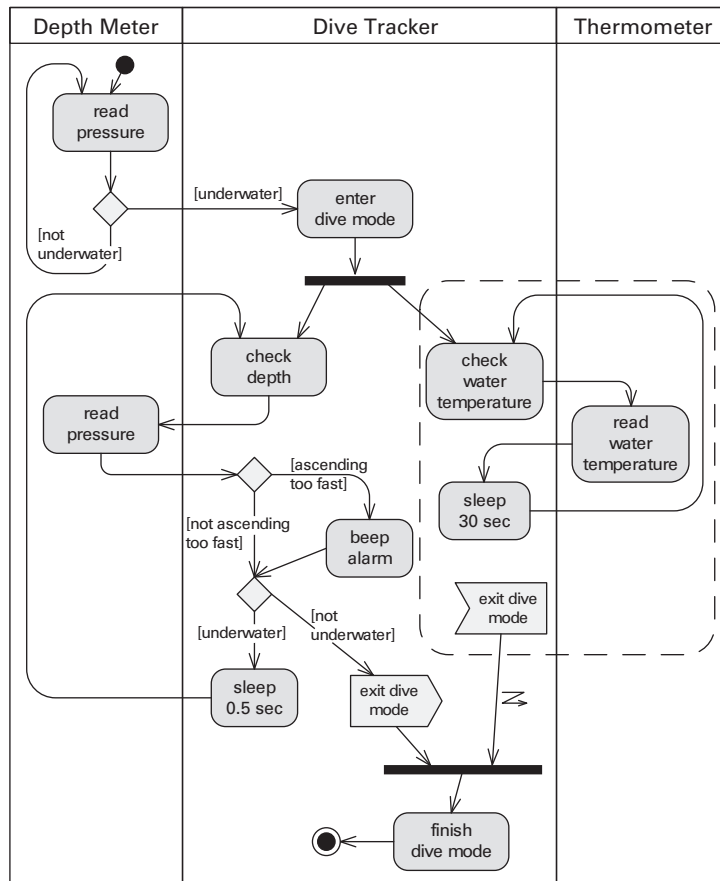


FIGURE 22.3 Activity diagram

In contrast to trace notations, *comprehensive* notations show the complete behavior of structural elements. Given this type of documentation, it is possible to infer all possible paths from the initial state to the final state. State machines are a kind of formalism used by many

comprehensive notations. This 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 diagrams allow you to trace the behavior of your system, given specific inputs. Such a diagram represents states using boxes and transitions between states using arrows. Thus it models elements of the architecture and helps illustrate their runtime interactions. Figure 22.4 is an example of a state machine diagram showing the states of a car stereo.

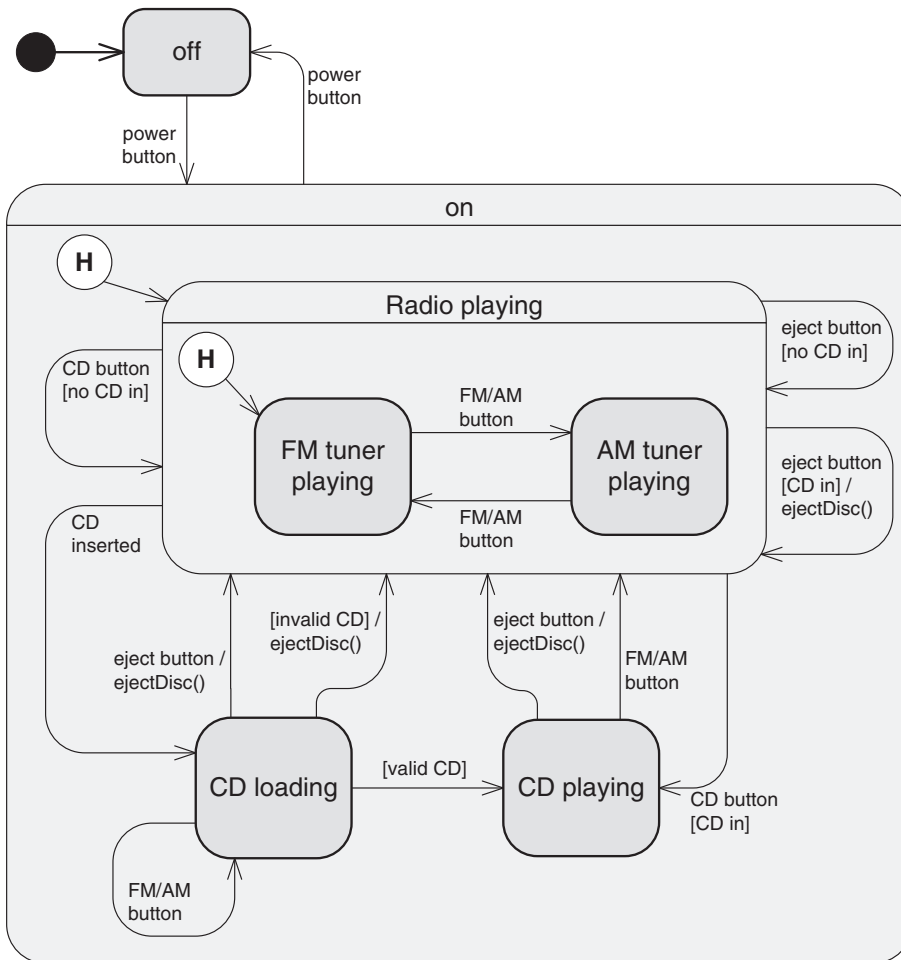


FIGURE 22.4 UML state machine diagram for a car stereo system

Each transition in a state machine diagram is labeled with the event causing the transition. For example, in Figure 22.4, 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 enabled only if the guard is true at that time. Transitions can also have consequences, called actions or effects, which are indicated by a slash. When an action is present, it indicates that the behavior following the slash will be performed when the transition occurs. The states may also specify entry and exit actions.

22.6 Beyond Views

In addition to views and behavior, comprehensive information about an architecture will include the following items:

- *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. Combining views (as described in Section 22.4) produces a set of views. Illuminating the associations among those views can then 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. To create such a table 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, the correspondence 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.

- *Documenting patterns.* If you employ patterns in your design, as recommended in Chapter 20, these patterns should be identified in the documentation. First, record the fact that the given pattern is being used. Then say why this solution approach was chosen—why the pattern is appropriate for the problem at hand. Using a pattern involves making successive design decisions that eventually result in that pattern’s instantiation. These design decisions may manifest themselves as newly instantiated elements and the relations among them, which in turn should be documented in structural views.
- *One or more context diagrams.* A *context diagram* shows how the system or portion of the system relates to its environment. The purpose of this 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. A context diagram may be created for each view, with each diagram showing how different types of elements interact with the

system's environment. Context diagrams are useful for presenting an initial picture of how a system or subsystem interacts with its environment.

- *Variability guide.* A *variability guide* shows how to exercise any variation points that are part of the architecture shown in this view.
- *Rationale.* The *rationale* explains why the design reflected in the view came to be. The goal of this section is to explain why the design has its present form and to provide a convincing argument that it is sound. Documenting the rationale is described in more detail in Section 22.7.
- *Glossary and acronym list.* Likely your architecture will contain many specialized terms and acronyms. Decoding these for your readers will ensure that all your stakeholders are speaking the same language, as it were.
- *Document control information.* List the issuing organization, the current version number, the date of issue and status, a change history, and the procedure for submitting change requests to the document. Usually this information is captured in the front matter. Change control tools can provide much of this information.

22.7 Documenting the Rationale

When designing, you make important design decisions to achieve the goals of each iteration. These design decisions include:

- Selecting a design concept from several alternatives
- Creating structures by instantiating the selected design concept
- Establishing relationships between elements and defining interfaces
- Allocating resources (e.g., people, hardware, computation)

When you study a diagram that represents an architecture, you see the end product of a thought process but can't always easily understand the decisions that were made to achieve this result. Recording design decisions *beyond* the representation of the chosen elements, relationships, and properties is fundamental to help in understanding how you arrived at the result; in other words, it lays out the design rationale.

When your iteration goal involves satisfying an important quality attribute scenario, some of the decisions that you make will play a significant role in achieving the scenario response measure. Consequently, you should take the greatest care in recording these decisions: They are essential to facilitate analysis of the design you created, to facilitate implementation, and, still later, to aid in understanding the architecture (e.g., during maintenance). Given that most design decisions are “good enough,” and seldom optimal, you also need to justify the decisions made, and to record the risks associated with your decisions so that they may be reviewed and possibly revisited.

You may perceive recording design decisions as a tedious task. However, depending on the criticality of the system being developed, you can adjust the amount of information that is recorded. For example, to record a minimum of information, you can use a simple table such

as Table 22.4. If you decide to record more than this minimum, the following information might prove useful:

- What evidence was produced to justify decisions?
- Who did what?
- Why were shortcuts taken?
- Why were tradeoffs made?
- What assumptions did you make?

In the same way that we suggest that you record responsibilities as you identify elements, you should record the design decisions as you make them. If you leave it until later, you will not remember why you did things.

TABLE 22.4 Example Table to Document Design Decisions

Design Decisions and Location	Rationale and Assumptions (Include Discarded Alternatives)
Introduce concurrency (tactic) in the TimeServerConnector and FaultDetectionService	Concurrency should be introduced to be able to receive and process several events (traps) simultaneously.
Use of the messaging pattern through the introduction of a message queue in the communications layer	Although the use of a message queue imposes a performance penalty, a message queue was chosen because some implementations have high performance and, furthermore, this will be helpful to support quality attribute scenario QA-3.
...	...

22.8 Architecture Stakeholders

In Chapter 2, we said that one of the key purposes of architecture was to enable communication among stakeholders. In this chapter, we have said that architecture documentation is produced in service of architecture stakeholders. So who are they?

The set of stakeholders will vary, depending on the organization and the project. The list of stakeholders in this section is suggestive but is not intended to be complete. As an architect, one of your primary obligations is to identify the real stakeholders for your project. Similarly, the documentation needs we lay out here for each stakeholder are typical but not definitive. You'll need to take the following discussion as a starting point and adapt it according to the needs of your project.

Key stakeholders of an architecture include the following:

- *Project managers* care about schedule, resource assignments, and perhaps contingency plans to release a subset of the system for business reasons. To create a schedule, the

project manager needs information about the modules to be implemented and in what sequence, with some information about their complexity, such as the list of responsibilities, as well as their dependencies on other modules. The dependencies may suggest a certain sequence in the implementation. The project manager is not interested in the design specifics of any element or the exact interface beyond knowing whether those tasks have been completed. However, this person is interested in the system's overall purpose and constraints; its interaction with other systems, which may suggest an organization-to-organization interface that the manager will have to establish; and the hardware environment, which the manager may have to procure. The project manager might create or help create the work assignment view, in which case he or she will need a decomposition view to do it. A project manager, then, will likely be interested in the following views:

- *Module views.* Decomposition and uses and/or layered.
 - *Allocation views.* Deployment and work assignment.
 - *Other.* Top-level context diagrams showing interacting systems and system overview and purpose.
- *Members of the development team,* for whom the architecture provides marching orders, are given constraints on how they do their job. Sometimes developers are given responsibility for an element they did not implement, such as a commercial off-the-shelf product or a legacy element. Someone still has to be responsible for that element, to make sure that it performs as advertised and to tailor it as necessary. This person will want to know the following information:
- The general idea behind the system. Although that information lies in the realm of requirements rather than architecture, a top-level context diagram or system overview can go a long way toward providing the necessary information.
 - Which elements the developer has been assigned for implementation—that is, where functionality should be implemented.
 - The details of the assigned element, including the data model with which it must operate.
 - The elements with which the assigned part interfaces and what those interfaces are.
 - The code assets that the developer can utilize.
 - The constraints, such as quality attributes, legacy system interfaces, and budget (resource or fiscal), that must be met.

A developer, then, is likely to want to see

- *Module views.* Decomposition, uses and/or layered, and generalization.
- *Component-and-connector (C&C) views.* Various, showing the component(s) the developer was assigned and the components they interact with.
- *Allocation views.* Deployment, implementation, and installation.
- *Other.* System overview; a context diagram containing the module(s) the developer has been assigned; the interface documentation of the developer's element(s) and the

interface documentation of those elements with which they interact; a variability guide to implement required variability; and rationale and constraints.

- *Testers and integrators* are stakeholders for whom the architecture specifies the correct black-box behavior of the pieces that must fit together. A black-box tester will need to access the interface documentation for the element. Integrators and system testers need to see collections of interfaces, behavior specifications, and a uses view so they can work with incremental subsets. Testers and integrators, then, are likely to want to see the following views:
 - *Module views.* Decomposition, uses, and data model.
 - *C&C views.* All.
 - *Allocation views.* Deployment; install; and implementation, to find out where the assets to build the module are.
 - *Other.* Context diagrams showing the module(s) to be tested or integrated; the interface documentation and behavior specification(s) of the module(s) and the interface documentation of those elements with which they interact.

Testers and integrators deserve special attention because it is not unusual for a project to spend roughly half of its overall effort in testing. Ensuring a smooth, automated, and error-free testing process will have a major positive effect on the project's overall cost.

- *Designers of other systems* with which this one must interoperate are also stakeholders. For these people, the architecture defines the set of operations provided and required, as well as the protocols for their operation. These stakeholders will likely want to see the following artifacts:
 - Interface documentations for those elements with which their system will interact, as found in module and/or C&C views
 - The data model for the system with which their system will interact
 - Top-level context diagrams from various views showing the interactions
- *Maintainers* use architecture as a starting point for maintenance activities, revealing the areas a prospective change will affect. Maintainers will want to see the same information as developers, as both must make their changes within the same constraints. But maintainers will also want to see a decomposition view that allows them to pinpoint the locations where a change will need to be carried out, and perhaps a uses view to help them build an impact analysis to fully scope out the effects of the change. In addition, they will want to see the design rationale, which will allow them to benefit from the architect's original thinking and save them time by identifying already discarded design alternatives. A maintainer, then, is likely to want to see the same views as the developers of a system do.
- *End users* do not need to see the architecture, which is, after all, largely invisible to them. Nevertheless, they can often gain useful insights into the system, what it does, and how they can use it effectively by examining the architecture. If end users or their representatives review your architecture, you may be able to uncover design discrepancies that

would otherwise have gone unnoticed until deployment. To serve this purpose, an end user is likely to be interested in the following views:

- *C&C views.* Views emphasizing flow of control and transformation of data, to see how inputs are transformed into outputs; analysis results dealing with properties of interest, such as performance or reliability.
 - *Allocation views.* A deployment view to understand how functionality is allocated to the platforms with which the users interact.
 - *Other.* Context diagrams.
- *Analysts* are interested in whether the design meets the system's quality objectives. The architecture serves as fodder for architecture evaluation methods and must provide the information necessary to evaluate quality attributes. For example, architecture includes the model that drives such analytical tools as rate-monotonic real-time schedulability analysis, reliability block diagrams, simulations and simulation generators, theorem provers, and model checkers. These tools require information about resource consumption, scheduling policies, dependencies, component failure rates, and so forth. Because analysis can encompass almost any subject matter area, analysts may need access to information documented in any part of the architecture documentation.
 - *Infrastructure support personnel* set up and maintain the infrastructure that supports the development, integration, staging, and production environments of the system. A variability guide is particularly useful to help set up the software configuration management environment. Infrastructure support people likely want to see the following views:
 - *Module views.* Decomposition and uses.
 - *C&C views.* Various, to see what will run on the infrastructure.
 - *Allocation views.* Deployment and install, to see where the software (including the infrastructure) will run; implementation.
 - *Other.* Variability guides.
 - *Future architects* are the most avid readers of architecture documentation, with a vested interest in everything. You, after a period of time, or your replacement (when you get promoted and assigned to a more complex project) will want to know all the key design decisions and why they were made. Future architects are interested in it all, but they will be especially keen to have access to comprehensive and candid rationale and design information. And, remember, that future architect might be you! Do not expect to remember all of these minute design decisions that you're making now. Remember, architecture documentation is a love letter you write to your future self.

22.9 Practical Considerations

Up to now, this chapter has been concerned with the information that architecture documentation should contain. Over and above the contents of architecture documentation, however, are

issues dealing with its form, distribution, and evolution. In this section, we discuss some of these concerns.

Modeling Tools

Many commercially available modeling tools are available that support the specification of architectural constructs in a defined notation; SysML is a widely used choice. Many of these tools offer features aimed at practical large-scale use in industrial settings: interfaces that support multiple users, version control, syntactic and semantic consistency checking of the models, support for trace links between models and requirements or models and tests, and, in some cases, automatic generation of executable source code that implements the models. In many projects, these are must-have capabilities, so the purchase price of the tool—which is not insignificant in some cases—should be evaluated against what it would cost the project to achieve these capabilities on its own.

Online Documentation, Hypertext, and Wikis

Documentation for a system can be structured as linked web pages. 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.

Using tools such as wikis, it's possible to create a *shared* document to which many stakeholders can contribute. The hosting organization needs to decide what permissions it wants to give to various stakeholders; the tool used has to support the chosen permissions policy. In the case of architecture documentation, we want selected stakeholders to comment on and add clarifying information to the architecture, but we would want only selected team personnel to be able to actually change it.

Follow a Release Strategy

Your project's development plan should specify the process for keeping the important documentation, including the architecture documentation, current. Document artifacts should be subject to version control, as with any other important project artifact. 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, revised documentation could be provided to the development team at the end of each iteration or sprint or with each incremental release.

Documenting Architectures That Change Dynamically

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 they change at runtime or as a result of high-frequency release-and-deploy cycles, all dynamic architectures share something in common with respect to documentation: They change much faster than the documentation cycle. In either case, no one is going to hold up things until a new architecture document is produced, reviewed, and released.

Even so, knowing the architecture of these ever-changing systems is every bit as important, and arguably more so, than for systems that follow 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 that new piece of software doesn’t just plug in anywhere, but rather in a predetermined location in the architecture. Record those invariants. This process 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 examples mentioned earlier, this will usually mean adding new components and replacing components with new implementations. The place to do this is the variability guide discussed in Section 22.6
- *Generate interface documentation automatically.* If you use explicit interface mechanisms such as protocol buffers (described in Chapter 15), then there are always up-to-date definitions of component interfaces; otherwise, the system would not work. Incorporate those interface definitions into a database so that revision histories are available and the interfaces can be searched to determine what information is used in which components.

Traceability

Architecture, of course, does not live in a bubble, but in a milieu of information about the system under development that includes requirements, code, tests, budgets and schedules, and more. The purveyors of each of these areas must ask themselves, “Is my part right? How do I know?” This question takes on different specific forms in different areas; for example, the tester asks, “Am I testing the right things?” As we saw in Chapter 19, architecture is a response to requirements and business goals, and its version of the “Is my part right?” question is to ensure

that those have been satisfied. Traceability means linking specific design decisions to the specific requirements or business goals that led to them, and those links should be captured in the documentation. If, at the end of the day, all ASRs are accounted for (“covered”) in the architecture’s trace links, then we have assurance that the architecture part is right. Trace links may be represented informally—a table, for instance—or may be supported technologically in the project’s tool environment. In either case, trace links should be part of the architecture documentation.

22.10 Summary

Writing architectural documentation is much like other types of writing. The golden rule is: Know your reader. You must understand the uses to which the writing will be put and the audience for the writing. Architectural documentation serves as a means for communication among various stakeholders: up the management chain, down into the developers, and across to peers.

An architecture is a complicated artifact, best expressed by focusing on particular perspectives, called views, which depend on the message to be communicated. You must choose the views to document and choose the notation to document these views. This may involve combining various views that have a large overlap. You must not only document the structure of the architecture but also the behavior.

In addition, you should document the relations among the views in your documentation, the patterns you use, the system’s context, any variability mechanisms built into the architecture, and the rationale for your major design decisions.

There are other practical considerations for creating, maintaining, and distributing the documentation, such as choosing a release strategy, choosing a dissemination tool such as a wiki, and creating documentation for architectures that change dynamically.

22.11 For Further Reading

Documenting Software Architectures: Views and Beyond [Clements 10a] is a comprehensive treatment of the architecture documentation approach described in this chapter. It details a multitude of different views and notations for them. It also describes how to package the documentation into a coherent whole. Appendix A covers using the Unified Modeling Language (UML) to document architecture and architectural information.

ISO/IEC/IEEE 42010:2011 (“eye-so-forty-two-oh-ten” for short) is the ISO (and IEEE) standard, *Systems and Software Engineering: Architecture Description*. This standard centers on two key ideas: a conceptual framework for architecture description and a statement of which information must be found in any ISO/IEC/IEEE 42010-compliant architecture description, using multiple viewpoints driven by stakeholders’ concerns.

AADL (addl.info) is an architecture description language that has become an SAE standard for documenting architectures. The SAE is an organization for engineering professionals in the aerospace, automotive, and commercial vehicle industries.

SysML is a general-purpose systems modeling language intended to support a broad range of analysis and design activities for systems engineering applications. It is defined so that sufficient detail can be specified to support a variety of automated analysis and design tools. The SysML standard is maintained by the Object Management Group (OMG); this language was developed by OMG in cooperation with the International Council on Systems Engineering (INCOSE). SysML was developed as a profile of UML, which means that it reuses much of UML, but also provides the extensions necessary to meet the needs of systems engineers. Copious information about SysML is available online, but Appendix C of [Clements 10a] discusses how SysML can be used to document architectures. As this book went to press, SysML 2.0 was under development.

An extended example of documenting architectural decisions while designing can be found in [Cervantes 16].

22.12 Discussion Questions

1. Go to the website of your favorite open source system and look for its architectural documentation. 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 ATM to reason about its security architecture.
3. If you are designing a microservice-based architecture, what elements, relations, and properties would you need to document to be able to reason about end-to-end latency or throughput?
4. Suppose your company has just purchased another company and 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?
5. When would you choose to document behavior using trace notations and when would you use a comprehensive notation? What value do you get and what effort is required for each of them?
6. How much of a project's budget would you devote to software architecture documentation? Why? How would you measure the cost and the benefit? How would this change if your project was a safety-critical system or a high-security system?

23



Managing Architecture Debt

With Yuanfang Cai

Some debts are fun when you are acquiring them, but none are fun when you set about retiring them.

—Ogden Nash

Without careful attention and the input of effort, designs become harder to maintain and evolve over time. We call this form of entropy “architecture debt,” and it is an important and highly costly form of technical debt. The broad field of technical debt has been intensively studied for more than a decade—primarily focusing on code debt. Architecture debt is typically more difficult to detect and more difficult to eradicate than code debt because it involves nonlocal concerns. The tools and methods that work well for discovering code debt—code inspections, code quality checkers, and so forth—typically do not work well for detecting architecture debt.

Of course, not all debt is burdensome and not all debt is bad debt. Sometimes a principle is violated when there is a worthy tradeoff—for example, sacrificing low coupling or high cohesion to improve runtime performance or time to market.

This chapter introduces a process to analyze existing systems for architecture debt. This process gives the architect both the knowledge and the tools to identify and manage such debt. It works by identifying architecturally connected elements—with problematic design relations—and analyzing a model of their maintenance costs. If that model indicates the existence of a problem, typically signaled by an unusually high amount of changes and bugs, this signifies an area of architecture debt.

Once architecture debt has been identified, if it is bad enough, it should be removed through refactoring. Without quantitative evidence of payoff, typically it is difficult to get project stakeholders to agree to this step. The business case (without architecture debt analysis) goes like this: “I will take three months to refactor this system and give you no new functionality.” What manager would agree to that? However, armed with the kinds of analyses we present here, you can make a very different pitch to your manager, one couched in terms of ROI and increased productivity that pays the refactoring effort back, and more, in a short time.

The process that we advocate requires three types of information:

- *Source code.* This is used to determine structural dependencies.
- *Revision history, as extracted from a project's version control system.* This is used to determine the co-evolution of code units.
- *Issue information, as extracted from an issue control system.* This is used to determine the reason for changes.

The model for analyzing debt identifies areas of the architecture that are experiencing unusually high rates of bugs and churn (committed lines of code) and attempts to associate these symptoms with design flaws.

23.1 Determining Whether You Have an Architecture Debt Problem

In our process for managing architecture debt, we will focus on the physical manifestation of architectural elements, which means the files in which their source code is stored. How do we determine if a group of files is architecturally connected? One way is to identify the static dependencies between the files in your project—this method calls that method, for example. You can find these by employing a static code analysis tool. A second approach is to capture the evolutionary dependencies between files in a project. An *evolutionary dependency* occurs when two files change together, and you can extract this information from your revision control system.

We can represent the file dependencies using a special kind of adjacency matrix called a design structure matrix (DSM). While other representations are certainly possible, DSMs have been used in engineering design for decades and are currently supported by a number of industrial tools. In a DSM, entities of interest (in our case, files) are placed both on the rows of the matrix and, in the same order, on the columns. The cells of the matrix are annotated to indicate the type of dependency.

We can annotate a DSM cell with information showing that the file on the row inherits from the file on the column, or that it calls the file on the column, or that it co-changes with the file on the column. The first two annotations are structural, whereas the third is an evolutionary (or history) dependency.

To repeat: Each row in the DSM represents a file. Entries on a row show the dependencies that this file has on other files in the system. If the system has low coupling, you would expect the DSM to be sparse; that is, any given file will be dependent on a small number of other files. Furthermore, you would hope that the DSM is lower diagonal; that is, all entries appear below the diagonal. This means that a file depends only on lower-level files, not on higher-level ones, and that you have no cyclic dependencies in your system.

Figure 23.1 shows 11 of the files from the Apache Camel project—an open source integration framework—and their structural dependencies (indicated by the labels “dp,” “im,” and “ex” for dependency, implementation, and extension, respectively). For example, the file on row 9 of Figure 23.1, `MethodCallExpression.java`, depends on and extends the file on column 1,

`ExpressionDefinition.java`, and the file on row 11, `AssertionClause.java`, depends on the file on column 10, `MockEndpoint.java`. These static dependencies are extracted by reverse-engineering the source code.

The matrix shown in Figure 23.1 is quite sparse. It means that these files are not heavily structurally coupled to each other and, as a consequence, you might expect that it would be relatively easy to change these files independently. In other words, this system seems to have relatively little architecture debt.

Now consider Figure 23.2, which overlays historical co-change information on Figure 23.1. Historical co-change information is extracted from the version control system. This indicates how often two files change together in commits.

Figure 23.2 shows a very different picture of the Camel project. For example, the cell at row 8, column 3 is marked with “4”: This means that there is no structural relation between `BeanExpression.java` and `MethodNotFoundException.java`, but they were found to have changed together four times in the revision history. A cell with both a number and text indicates that this pair of files has both structural and evolutionary coupling relations. For example, the cell at row 22, column 1 is marked with “dp, 3”: This means that `XMLTokenizerExpression.java` depends on `ExpressionDefinition.java`, and they were changed together three times.

The matrix in Figure 23.2 is rather dense. Although these files are generally not structurally coupled to each other, they are strongly evolutionarily coupled. Furthermore, we see many annotations in cells above the diagonal in the matrix. Thus the coupling is not just from higher-level to lower-level files, but rather goes in all directions.

This project, in fact, suffers from high architecture debt. The architects confirm this. They report that almost every change in the project is costly and complex, and predicting when new features will be ready or when bugs will be fixed is challenging.

While this kind of qualitative analysis can, by itself, be of value to an architect or analyst, we can do better: We can actually quantify the costs and impact of the debt that our code base is already carrying, and we can do this fully automatically. To do so, we use the concept of “hotspots”—areas of the architecture with design flaws, sometimes called architecture anti-patterns or architecture flaws.

23.2 Discovering Hotspots

If you suspect that your code base has architecture debt—perhaps bug rates are going up and feature velocity is going down—you need to identify the specific files and their flawed relationships that are creating that debt.

Compared to code-based technical debt, architecture debt is often harder to identify because its root causes are distributed among several files and their interrelationships. If you have a cyclic dependency where the cycle of dependencies passes through six files, it is unlikely that anyone in your organization completely understands this cycle and it is not easily observable. For these kinds of complex cases, we need help, in the form of automation, to identify the architecture debt.

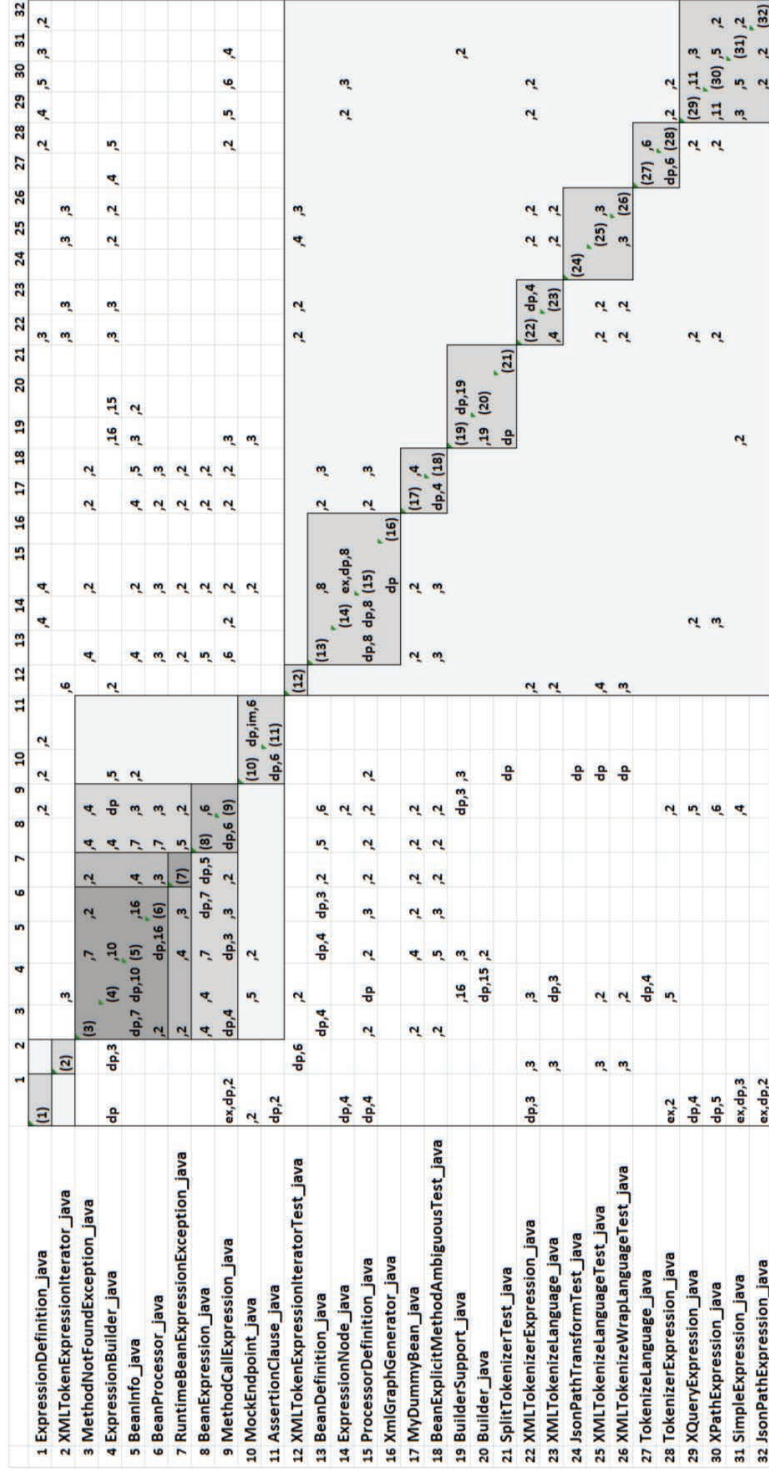


FIGURE 23.2 A DSM Apache Camel overlaying evolutionary dependencies

We call the sets of elements that make outsized contributions to the maintenance costs of a system *hotspots*. Architecture debt leads to high maintenance costs due to high coupling and low cohesion. So, to identify hotspots, we look for anti-patterns that contribute to high coupling and low cohesion. Six common anti-patterns—which occur in virtually every system—are highlighted here:

- *Unstable interface*. An influential file—one representing an important service, resource, or abstraction in the system—changes frequently with its dependents, as recorded in the revision history. The “interface” file is the entry point for other system elements to use the service or resource. It is frequently modified due to internal reasons, changes to its API, or both. To identify this anti-pattern, search for a file with a large number of dependents that is modified frequently with other files.
- *Modularity violation*. Structurally decoupled modules frequently change together. To identify this anti-pattern, search for two or more structurally independent files—that is, files that have no structural dependency on each other—that change together frequently.
- *Unhealthy inheritance*. A base class depends on its subclasses or a client class depends on both the base class and one or more of its subclasses. To determine unhealthy inheritance instances, search for either of the following two sets of relationships in a DSM:
 - In an inheritance hierarchy, a parent depends on its child class.
 - In an inheritance hierarchy, a client of the class hierarchy depends on both the parent and one or more of its children.
- *Cyclic dependency or clique*. A group of files is tightly connected. To identify this anti-pattern, search for sets of files that form a strongly connected graph, where there is a structural dependency path between any two elements of the graph.
- *Package cycle*. Two or more packages depend on each other, rather than forming a hierarchical structure, as they should. Detecting this anti-pattern is similar to detecting a clique: A package cycle is determined by discovering packages that form a strongly connected graph.
- *Crossing*. A file has both a high number of dependent files and a high number of files on which it depends, and it changes frequently with its dependents and the files it depends on. To determine the file at the center of a crossing, search for a file that has both high fan-in and fan-out with other files and that has substantial co-change relations with these other files.

Not every file in a hotspot will be tightly coupled to every other file. Instead, a collection of files may be tightly coupled to each other and decoupled from other files. Each such collection is a potential hotspot and is a potential candidate for debt removal, through refactoring.

Figure 23.3 is a DSM based on files in Apache Cassandra—a widely used NoSQL database. It shows an example of a clique (a cycle of dependencies). In this DSM, you can see that the file on row 8 (`locator.AbstractReplicationStrategy`) depends on file 4 (`service.WriteResponseHandler`) and aggregates file 5 (`locator.TokenMetadata`). Files 4 and 5, in turn, depend on file 8, thus forming a clique.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 config.DatabaseDescriptor	(1)	dp,44	14	10	10	6	14	36	118	12	16	12	42	52	4	18	30	
2 utils.FBUtilities	dp,44	(2)	40	4	6	10	6	12	38	28	12	8	14	24	46	6	18	28
3 utils.ByteBufferUtil	14	dp,40	(3)					4	10	20	4	4	10	26		12	4	
4 service.WriteResponseHandler	10	dp,4	2	(4)	4	6	18	dp,22						6				
5 locator.TokenMetadata	10	6		4	(5)	4	10	dp,24	8					4	6	4		
6 locator.NetworkTopologyStrategy	6	dp,10	2	6	dp,4	(6)	10	ih,22	4					16			8	
7 service.DatacenterWriteResponseHandler	dp,14	dp,6	2	ih,18	10	dp,10	(7)	20						6	6			
8 locator.AbstractReplicationStrategy	36	dp,12	4	dp,22	ag,24	22	dp,20	(8)	6					16	10		10	
9 config.CFMetaData	118	dp,38	dp,10		4		6	(9)			16		36	46			56	
10 dht.RandomPartitioner	12	dp,28	dp,20		8				(10)	dp,4			4	16		50		
11 utils.GuidGenerator		dp,12	4						4	(11)				4				
12 io.sstable.SSTable	16	8	dp,4					ag,16			(12)	4	dp,68	10				
13 utils.CLibary	12	dp,14									4	(13)	12					
14 io.sstable.SSTableReader	dp,42	24	dp,10					36	4		ih,68	dp,12	(14)	22	4		10	
15 cli.CIClient	52	dp,46	dp,26	6	4	16	6	16	46	16	4	10		22	(15)	6	14	48
16 locator.PropertyFileSnitch	4	dp,6		dp,6		6	10						4	6	(16)		4	
17 dht.OrderPreservingPartitioner	dp,18	dp,18	dp,12	4					50					14		(17)		
18 thrift.ThriftValidation	dp,30	28	dp,4		8		dp,10	dp,56					10	48	4		(18)	

FIGURE 23.3 An example of a clique

A second example from Cassandra demonstrates the unhealthy inheritance anti-pattern. The DSM in Figure 23.4 shows the `io.sstable.SSTableReader` class (row 14) inheriting from `io.sstable.SSTable` (row 12). The inheritance relationship is indicated in the DSM by the “ih” notation. Note, however, that `io.sstable.SSTable` depends on `io.sstable.SSTableReader`, as indicated by the “dp” annotation in cell (12, 14). This dependency is a calling relation, which means that the parent class calls the child class. Note that the cells (12, 14) and (14, 12) are both annotated with the number 68. This represents the number of times that `io.sstable.SSTable` and `io.sstable.SSTableReader` were co-committed in changes, according to the project’s revision history. This excessively high number of co-changes is a form of debt. This debt can be removed by refactoring—that is, by moving some functionality from the child class to the parent.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 config.DatabaseDescriptor	(1)	dp,44	14	10	10	6	14	36	118	12	16	12	42	52	4	18	30	
2 utils.FBUtilities	dp,44	(2)	40	4	6	10	6	12	38	28	12	8	14	24	46	6	18	28
3 utils.ByteBufferUtil	14	dp,40	(3)					4	10	20	4	4	10	26		12	4	
4 service.WriteResponseHandler	10	dp,4	2	(4)	4	6	18	dp,22						6				
5 locator.TokenMetadata	10	6		4	(5)	4	10	dp,24	8					4	6	4		
6 locator.NetworkTopologyStrategy	6	dp,10	2	6	dp,4	(6)	10	ih,22	4					16			8	
7 service.DatacenterWriteResponseHandler	dp,14	dp,6	2	ih,18	10	dp,10	(7)	20						6	6			
8 locator.AbstractReplicationStrategy	36	dp,12	4	dp,22	ag,24	22	dp,20	(8)	6					16	10		10	
9 config.CFMetaData	118	dp,38	dp,10		4		6	(9)			16		36	46			56	
10 dht.RandomPartitioner	12	dp,28	dp,20		8				(10)	dp,4			4	16		50		
11 utils.GuidGenerator		dp,12	4						4	(11)				4				
12 io.sstable.SSTable	16	8	dp,4					ag,16			(12)	4	dp,68	10				
13 utils.CLibary	12	dp,14									4	(13)	12					
14 io.sstable.SSTableReader	dp,42	24	dp,10					36	4		ih,68	dp,12	(14)	22	4		10	
15 cli.CIClient	52	dp,46	dp,26	6	4	16	6	16	46	16	4	10		22	(15)	6	14	48
16 locator.PropertyFileSnitch	4	dp,6		dp,6		6	10						4	6	(16)		4	
17 dht.OrderPreservingPartitioner	dp,18	dp,18	dp,12	4					50					14		(17)		
18 thrift.ThriftValidation	dp,30	28	dp,4		8		dp,10	dp,56					10	48	4		(18)	

FIGURE 23.4 Architecture anti-patterns in Apache Cassandra

The majority of issues in an issue tracking system can be divided into two broad categories: bug fixes and feature enhancements. Bug fixes and both bug-related and change-related churn are highly correlated with anti-patterns and hotspots. In other words, those files that participate in anti-patterns and require frequent bug fixes or frequent changes are likely hotspots.

For each file, we determine the total number of bug fixes and changes, as well as the total amount of churn that file has experienced. Next, we sum the bug fixes, changes, and churn experienced by the files in each anti-pattern. This gives us a weighting for each anti-pattern in terms of its contribution to architecture debt. In this way, all of the debt-laden files, along with all of their relationships, can be identified and their debt quantified.

Based on this process, a debt-reduction strategy (typically achieved through refactoring) is straightforward. Knowing the files implicated in the debt, along with their flawed relationships (as determined by the identified anti-patterns), allows the architect to fashion and justify a refactoring plan. If a clique exists, for example, a dependency needs to be removed or reversed, so as to break the cycle of dependencies. If unhealthy inheritance is present, some functionality needs to be moved, typically from a child class to a parent class. If a modularity violation is identified, the unencapsulated “secret” shared among files needs to be encapsulated as its own abstraction. And so forth.

23.3 Example

We illustrate this process with a case study, which we call SS1, done with SoftServe, a multinational software outsourcing company. At the time of the analysis, the SS1 system contained 797 source files, and we captured its revision history and issues over a two-year period. SS1 was maintained by six full-time developers and many more occasional contributors.

Identifying Hotspots

During the period that we studied SS1, 2,756 issues were recorded in its Jira issue-tracker (1,079 of which were bugs) and 3,262 commits were recorded in the Git version control repository.

We identified hotspots using the process just described. In the end, three clusters of architecturally related files were identified as containing the most harmful anti-patterns and hence the most debt in the project. The debt from these three clusters represented a total of 291 files, out of 797 files in the entire project, or a bit more than one-third of the project’s files. The number of defects associated with these three clusters covered 89 percent of the project’s total defects (265).

The chief architect of the project agreed that these clusters were problematic but had difficulty explaining why. When presented with this analysis, he acknowledged that these were true design problems, violating multiple design rules. The architect then crafted a number of refactorings, focusing on remedying the flawed relations among the files identified in the

hotspots. These refactorings were based on removing the anti-patterns in the hotspots, so the architect had a great deal of guidance in how to do this.

But does it *pay* to do these kinds of refactorings? After all, not all debts are worth paying off. This is the topic of the next section.

Quantifying Architecture Debt

Because the remediations suggested by the analysis are very specific, the architect can easily estimate the number of person-months required for each of the refactorings identified on the basis of the anti-patterns in the hotspots. The other side of the cost/benefit equation is the benefit from the refactorings. To estimate the savings, we make one assumption: The refactored files will have roughly the same number of bug fixes in the future as the average file had in the past. This is actually a very conservative assumption since the average number of bug fixes in the past was inflated by those files in the identified hotspots. Moreover, this calculation does not consider other significant costs of bugs, such as lost reputation, lost sales, and additional quality assurance and debugging effort.

We calculate the cost of these debts in terms of the lines of code committed for bug fixes. This information can be retrieved from a project's revision control and issue-tracking systems.

For SS1, the debt calculations we made were as follows:

1. The architect estimated the effort required to refactor the three hotspots as 14 person-months.
2. We calculated the average bug fixes per file annually for the total project as 0.33.
3. We calculated the average number of annual bug fixes for files in hotspots as 237.8.
4. Based on these results, we estimated that the annual number of bug fixes for the files in the hotspots, after refactoring, would be 96.
5. The difference between the actual churn associated with the hotspot files and the expected amount of churn after refactoring is the expected savings.

The estimated annual savings for the refactored files (using company average productivity numbers) was 41.35 person-months. Considering the calculations in steps 1–5, we see that for a cost of 14 person-months, the project can expect to save more than 41 person-months annually.

In case after case, we have seen these kinds of returns on investment. Once the architecture debts have been identified, they can be paid down and life becomes measurably better for the project, in terms of its feature velocity and bug-fixing time, in a way that more than pays for the effort involved.

23.4 Automation

This form of architectural analysis can be fully automated. Each of the anti-patterns introduced in Section 23.2 can be identified in an automated fashion and the tooling can be built

into a continuous integration tool suite so that architecture debt is continuously monitored. This analysis process requires the following tools:

- A tool to extract a set of issues from an issue tracker
- A tool to extract a log from a revision control system
- A tool to reverse-engineer the code base, to determine the syntactic dependencies among files
- A tool to build DSMs from the extracted information and walk through the DSM looking for the anti-patterns
- A tool that calculates the debt associated with each hotspot

The only specialized tools needed for this process are the ones to build the DSM and analyze the DSM. Projects likely already have issue tracking systems and revision histories, and plenty of reverse-engineering tools are available, including open source options.

23.5 Summary

This chapter has presented a process for identifying and quantifying architecture debt in a project. Architecture debt is an important and highly costly form of technical debt. Compared to code-based technical debt, architecture debt is often harder to identify because its root causes are distributed among several files and their interrelationships.

The process outlined in this chapter involves gathering information from the project's issue tracker, its revision control system, and the source code itself. Using this information, architecture anti-patterns can be identified and grouped into hotspots, and the impact of these hotspots can be quantified.

This architecture debt monitoring process can be automated and built into a system's continuous integration tool suite. Once architecture debt has been identified, if it is bad enough, it should be removed through refactoring. The output of this process provides the quantitative data necessary to make the business case for refactoring to project management.

23.6 For Further Reading

The field of technical debt has, at this point, a rich research literature. The term *technical debt* was coined by Ward Cunningham in 1992 (although, at the time, he simply called it "debt" [Cunningham 92]). This idea was refined and elaborated by many others, most prominent among them Martin Fowler [Fowler 09] and Steve McConnell [McConnell 07]. George Fairbanks describes the iterative nature of debt in his *IEEE Software* article, "Ur-Technical Debt" [Fairbanks 20]. A comprehensive look at the problem of managing technical debt can be found in [Kruchten 19].

The definition of architecture debt used in this chapter was borrowed from [Xiao 16]. The SoftServe case study was published in [Kazman 15].

Some of the tools used to create and analyze DSMs are described in [Xiao 14]. The tools to detect architectural flaws are introduced in [Mo 15].

The impacts of architecture flaws have been discussed and empirically investigated in several papers, including [Feng 16] and [Mo 18].

23.7 Discussion Questions

1. How would you distinguish a project with architecture debt from a “busy” project where lots of features are being implemented?
2. Find examples of projects that have undergone major refactorings. What evidence was used to motivate or justify these refactorings?
3. Under what circumstances is accumulating debt a reasonable strategy? How would you know that you had reached the point of too much debt?
4. Is architecture debt more or less detrimental than other kinds of debt, such as code debt, documentation debt, or testing debt?
5. Discuss the strengths and weaknesses of doing this kind of architecture analysis as compared with the methods discussed in Chapter 21.

This page intentionally left blank

24



The Role of Architects in Projects

I don't know why people hire architects and then tell them what to do.
—Frank Gehry

Any practice of architecture performed outside of a classroom takes place in the larger context of a development project, which is planned and carried out by people working in one or more organizations. Architecture, for all its importance, is only the means toward a larger end. In this chapter, we deal with the aspects of architecture and the architect's responsibilities that derive from the realities of development projects.

We begin by discussing a key project role with whom you as an architect are likely to have a close working relationship: the project manager.

24.1 The Architect and the Project Manager

One of the most important relations within a team is between the software architect and the project manager. The project manager is responsible for the overall performance of the project—typically for keeping it on budget, on schedule, and staffed with the right people doing the right jobs. To carry out these responsibilities, the project manager will often turn to the project architect for support.

Think of the project manager as primarily responsible for the external-facing aspects of the project and the software architect as responsible for the internal technical aspects of the project. The external view needs to accurately reflect the internal situation, and the internal activities need to accurately reflect the expectations of the external stakeholders. That is, the project manager should know, and reflect to upper management, the progress and the risks within the project, whereas the software architect should know, and reflect to developers, external stakeholder concerns. The relationship between the project manager and the software architect can have a large impact on the success of a project. They should have a good working relationship and be mindful of the roles they are filling and the boundaries of those roles.

The Project Management Body of Knowledge (PMBOK) lists a number of knowledge areas for project managers. These are the areas for which the project manager will likely turn to the architect for input. Table 24.1 identifies the knowledge area described by the PMBOK and the software architect's role in that area.

TABLE 24.1 Architect's Role in Supporting Project Management Knowledge Areas

PMBOK Knowledge Area	Description	Software Architect Role
Project Integration Management	Ensuring that the various elements of the project are properly coordinated	Create design and organize team around design; manage dependencies. Implement the capture of metrics. Orchestrate requests for changes.
Project Scope Management	Ensuring that the project includes all of the work required and only the work required	Elicit, negotiate, and review runtime requirements and generate development requirements. Estimate cost, schedule, and risk associated with meeting requirements.
Project Time Management	Ensuring that the project completes in a timely fashion	Help define the 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	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	Design for quality and track the system against the design. Define quality metrics.
Project Human Resource Management	Ensuring that the project makes the most effective use of the people involved with the project	Define the 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	Ensure communication and coordination among developers. Solicit feedback as to progress, problems, and risks. Oversee documentation.
Project Risk Management	Identifying, analyzing, and responding to project risk	Identify and quantify risks; adjust the architecture and processes to mitigate risk.
Project Procurement Management	Acquiring goods and services from outside the organization	Determine technology requirements; recommend technology, training, and tools.

Recommendations to the Architect

Maintain a good working relationship with the project manager. Be aware of the project manager's tasks and concerns, and how you as an architect may be asked to support those tasks and concerns.

24.2 Incremental Architecture and Stakeholders

Agile methodologies are built on the pillar of incremental development, with each increment delivering value to the customer or user. We'll discuss Agile and architecture in its own section, but even if your project is not an Agile one, you should still expect to develop and release your architecture in increments following a tempo that supports the project's own test and release schedule.

Incremental architecture, then, is about releasing the architecture in increments. Specifically, this means releasing architecture documentation (as described in Chapter 22) in increments. This, in turn, entails deciding which views to release (out of your planned set) and at which depth. Using the structures we outlined in Chapter 1, consider these as candidates for your first increment:

- A module decomposition structure. This will inform the team structure for the development project, allowing the project organization to emerge. Teams can be defined, staffed, budgeted, and trained. The team structure will be the basis of project planning and budgeting, so this technical structure defines the project's management structure.
- A module "uses" structure. This will allow increments to be planned, which is critical in any project that hopes to release its software incrementally. As we said in Chapter 1, the uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted. Trying to create a system that purposefully supports incremental development is problematic if you don't plan what exactly the increments will be.
- Whichever component-and-connector (C&C) structure(s) best convey the overall solution approach.
- A broad-brush deployment structure that at least addresses major questions such as whether the system will be deployed on mobile devices, on a cloud infrastructure, and so forth.

After that, use the needs of the architecture's stakeholders as a guide when crafting the contents of subsequent releases.

Recommendations to the Architect

First and foremost, make sure you know who your stakeholders are and what their needs are, so that you can design appropriate solutions and documentation. Moreover:

- Work with the project's stakeholders to determine the release tempo and the contents of each project increment.
- Your first architectural increment should include module decomposition and uses views, as well as a preliminary C&C view.
- Use your influence to ensure that early releases deal with the system's most challenging quality attribute requirements, thereby ensuring that no unpleasant architectural surprises appear late in the development cycle.

- Stage your architecture releases to support those project increments and to support the needs of the development stakeholders as they work on each increment.

24.3 Architecture and Agile Development

Agile development began as a rebellion against—among other things—development approaches that were rigid and heavyweight with respect to process, overbearing with respect to required documentation, focused on up-front planning and design, and culminating in a single delivery that everyone hoped would resemble what it was that the customer wanted in the first place. Agilistas advocate allocating resources that might otherwise be spent on process and documentation to figuring out what the customer really wants and providing it in small, testable delivery increments, starting very early on.

The key question is this: How much up-front work, in terms of requirements analysis, risk mitigation, and architecture design, should a project undertake? There is no single right answer to this question, but you can find a “sweet spot” for any given project. The “right” amount of project work depends on several factors, with the most dominant being project size, but other important factors include complex functional requirements, highly demanding quality attribute requirements, volatile requirements (related to the “precedentedness” or novelty of the domain), and degree of distribution of development.

So how do architects achieve the right amount of agility? Figure 24.1 shows your options. You can opt for waterfall-style “Big Design Up Front” (BDUF), shown in Figure 24.1(a). Or you can throw architectural caution to the wind and trust in what Agilistas call the “emergent” approach, wherein the final architecture emerges as coders deliver their increments, shown in Figure 24.1(b). That approach may work for small, simple projects that can turn on a dime and simply refactor on demand, but we have never seen it work for large, complex projects.

Not surprisingly, the approach we recommend lies in between these two extremes: It’s the “Iteration 0” approach, shown in Figure 24.1(c). In projects where you have some understanding of the requirements, you should consider beginning by performing a few iterations of attribute-driven design (ADD; described in Chapter 20). These design iterations can focus on choosing the major architectural patterns (including a reference architecture, if one is appropriate), frameworks, and components. Aim for support of the project’s increments in a way that helps the architecture’s stakeholders, as recommended in Section 24.2. Early on, this will help you structure the project, define work assignments and team formation, and address the most critical quality attributes. If and when requirements change—particularly if these are driving quality attribute requirements—adopt a practice of Agile experimentation, where spikes are used to address new requirements. A *spike* is a time-boxed task that is created to answer a technical question or gather information; it is not intended to lead to a finished product. Spikes are developed in a separate code branch and, if successful, merged into the main branch of the code. In this way, emerging requirements can be taken in stride and managed without being too disruptive to the overall process of development.

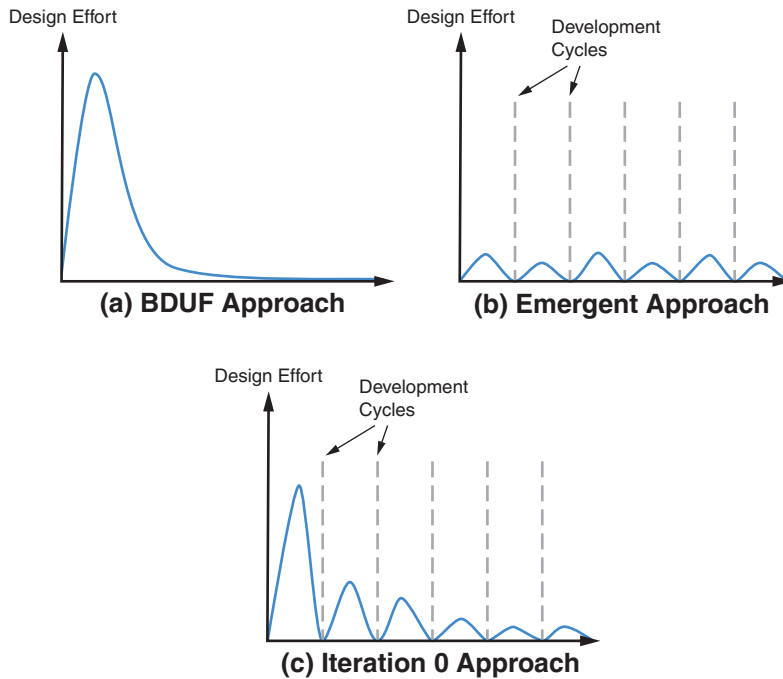


FIGURE 24.1 Three approaches to architectural design

Agile programming and architecture have not always been on the best of terms. The Agile Manifesto of 2001, the “Prime Directive” of the Agile movement, implies that architecture is emergent and does not need to be planned or designed up-front.

It was (and still is) easy to find published treatments of Agile that declare that if you aren’t delivering working software, then you aren’t doing anything of value. It follows that if you’re working on an architecture, then you’re taking resources away from programming and, therefore, you’re doing *nothing of value*—architecture, schmarchitecture! Write the code, and the architecture will emerge organically.

For medium to large systems, this view has inevitably collapsed under the harsh weight of experience. Solutions to quality attribute requirements cannot simply be “bolted on” to an existing system in an arbitrarily late stage of development. Solutions for security, high performance, safety, and many more concerns must be designed into the system’s architecture from the beginning, even if the first 20 planned incremental deliveries don’t exercise those capabilities. Yes, you can begin coding and yes, the architecture will emerge—but it will be the wrong one.

In short, the Agile Manifesto makes a pretty lousy prenup agreement for any marriage between Agile and architecture. However, accompanying the Manifesto are 12 Agile principles

that, if read charitably, hint at a middle ground between the two camps. Table 24.2 lists these principles and provides architecture-centric commentary on each one.

TABLE 24.2 Agile Principles and Architecture-centric Perspective

Agile Principle	Architecture-centric View
Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	Absolutely.
Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	Absolutely. This principle is served by architectures that provide high degrees of modifiability (Chapter 8) and deployability (Chapter 5).
Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter time scale.	Absolutely, as long as this principle is not seen as precluding a thoughtful architecture. DevOps has a large role to play here, and we have seen, in Chapter 5, how architectures can support DevOps.
Business people and developers must work together daily throughout the project.	Business goals lead to quality attribute requirements, which the architecture's primary duty is to fulfill, as we discussed in Chapter 19.
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	While we agree in principle, many developers are inexperienced. So make sure to include a skilled, experienced, and motivated architect to help guide these individuals.
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	This is nonsense for nontrivial systems. Humans invented writing because our brains can't remember everything we need to remember. Interfaces, protocols, architectural structures, and more need to be written down, and the inefficiencies and ineffectiveness of repeated instruction and resulting errors from misunderstanding belie this principle. According to this argument, nobody should produce user manuals, but should just publish the developers' phone numbers with an open invitation to call them anytime. This is also nonsense for any system that has a maintenance phase (that's pretty much every system) in which the original team is nowhere to be found. With whom are you going to have that face-to-face conversation to learn important details? See Chapter 22 for our guidance in this matter.
Working software is the primary measure of progress.	Yes, as long as "primary" is not taken to mean "only," and as long as this principle is not used as an excuse to eliminate all work except coding.
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	Absolutely.
Continuous attention to technical excellence and good design enhances agility.	Absolutely.
Simplicity—the art of maximizing the amount of work not done—is essential.	Yes, of course, as long as it is understood that the work we are not doing can actually be jettisoned safely without detriment to the system being delivered.

Agile Principle	Architecture-centric View
The best architectures, requirements, and designs emerge from self-organizing teams.	No, they don't. The best architectures are consciously designed by skilled, talented, trained, and experienced architects, as we describe in Chapter 20
At regular intervals, the team reflects on how to become more effective, and then tunes and adjusts its behavior accordingly.	Absolutely.

So that's six "Absolutely" agreements, four general agreements, and two strong disagreements.

Agile, as it was first codified, seemed to work best in small organizations building small products. Organizations of medium to large size wishing to apply Agile to large projects quickly found that coordinating the large number of small Agile teams was a formidable challenge. In Agile, small teams do small pieces of work over small intervals. One challenge is ensuring that these many (dozens to hundreds) small teams have divided the work suitably so that no work is overlooked and no work is done twice. Another challenge is sequencing the teams' many tasks so that their results can be amalgamated, frequently and quickly, to produce the next small increment of a sensibly working system.

One example of an approach to apply Agile at enterprise scale is the Scaled Agile Framework (SAFe), which emerged around 2007 and has been refined continuously since then. SAFe provides a reference model of workflows, roles, and processes under which large organizations can coordinate the activities of many teams, each operating in classic Agile fashion, to systematically and successfully produce a large-scale system.

SAFe acknowledges the role of architecture. It admits "intentional architecture," the definition of which will strike a chord with readers of this book. Intentional architecture "defines a set of purposeful, planned architectural strategies and initiatives, which enhance solution design, performance, and usability and provide guidance for inter-team design and implementation synchronization." But SAFe also strongly counsels a counterbalancing force called "emergent design," which "provides the technical basis for a fully evolutionary and incremental implementation approach" (scaledagileframework.com). We would argue that those qualities would emerge from an intentional architecture as well, since the ability to rapidly evolve and the ability to support incremental implementations do not happen without careful up-front thought. Ways to achieve these are, in fact, covered throughout this book.

24.4 Architecture and Distributed Development

Most substantial projects today are developed by distributed teams, where "distributed" may mean spread across floors in a building, across buildings on an industrial campus, across campuses in one or two different time zones, or among different divisions or subcontractors scattered around the globe.

Distributed development comes with both benefits and challenges:

- *Cost.* Labor costs vary depending on location, and there is a perception that moving some development to a low-cost venue will inevitably decrease the overall cost of the project. Indeed, experience has shown that, for software development, savings may be reaped in the long term. However, 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 may be high, the size of the developer pool may be small, or the skill sets needed may be 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, albeit at the cost of additional communication and coordination.
- *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 appropriate and the types of cultural issues that may arise.

How does distributed development play out on a project? Assume Module A uses an interface from Module B. In time, as circumstances change, this interface may need to be modified. In consequence, the team responsible for Module B must coordinate with the team responsible for Module A, as indicated in Figure 24.2. This kind of coordination is easy if it involves a short conversation at the shared vending machines, but it's not so easy if it involves a preplanned web conference at a time when it is the middle of the night for one of the teams.

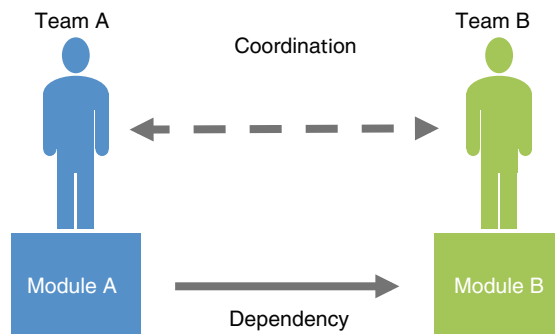


FIGURE 24.2 Coordination between teams and modules

More broadly, methods for coordination include the following options:

- *Informal contacts.* Informal contacts, such as meeting at the coffee room or in the hallway, are possible only 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 they are 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.
- *Asynchronous electronic communication.* Various forms of asynchronous electronic communication can be used as a coordination mechanism, such as email, news groups, blogs, and wikis.

The choice of coordination method depends on many factors, including the organization's infrastructure, corporate culture, language skills, time zones involved, and 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 will likely cause delays and, in some cases, serious defects in a project.

What does this mean for architecture and the architect? It means that allocation of responsibilities to teams is more important in distributed development than in co-located development, where all of the developers are in a single office, or at least in close proximity. It also means that attention to module dependencies takes on added importance over and above their usual role in quality attributes such as modifiability and performance: Dependencies among modules owned by globally distributed teams are more likely to be problematic and should be minimized to the extent possible.

In addition, 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 available, so they must rely on more formal mechanisms such as documentation, and team members must take the initiative to talk to each other when doubts arise.

As this book was being prepared for publication, companies around the world were learning to cope with remote participation and work-from-home practices due to the COVID-19 crisis. It is too soon to definitively state the long-term effects of this pandemic on the business world, but it seems likely to lead to distributed development becoming the norm. People working together are now all doing so via teleconference; there are no more hallway conversations or meetings at the vending machines. For work to continue at all, everyone is learning to adapt to the distributed development paradigm. It will be fascinating to see if this leads to any new architectural trends.

24.5 Summary

Software architects do their work in the context of a development project of some sort. As such, they need to understand their role and responsibilities from that perspective.

The project manager and the software architect may be seen as occupying complementary roles: The manager runs the project from an administrative perspective, and the architect runs the project from a technical solution perspective. These two roles intersect in various ways, and the architect can support the manager to enhance the project's chance of success.

In a project, architectures do not spring fully formed from Zeus's forehead, but rather are released in increments that are useful to stakeholders. Thus the architect needs to have a good understanding of the architecture's stakeholders and their information needs.

Agile methodologies focus on incremental development. Over time, architecture and Agile (although they got off to a rough start together) have become indispensable partners.

Global development creates a need for an explicit coordination strategy that is based on more formal strategies than are needed for co-located development.

24.6 For Further Reading

Dan Paulish has written an excellent book on managing in an architecture-centric environment—*Architecture-centric Software Project Management: A Practical Guide*—and the material in this chapter about distributed development is adapted from his book [Paulish 02].

You can read about SAFe at scaledagileframework.com. Before SAFe, some members of the Agile community had independently arrived at a medium-weight management process that advocates up-front architecture. See [Coplein 10] for a description of the role of architecture in agile projects.

Basic concepts of project management are covered in the IEEE Guide, *Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge*, sixth edition [IEEE 17].

Software architecture metrics often fall within an architect's purview on a project. A paper by Coulin et al. provides a helpful overview of the literature on this subject and, along the way, categorizes the metrics themselves [Coulin 19].

Architects occupy a unique position within an organization. They are expected to be fluent in all phases of the system's life cycle, from the cradle to the grave. Of all the members of a project, they are the ones most sensitive to the needs of all of the project's and the system's stakeholders. They usually are chosen to be architects in part because of their above-average communication skills. *The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise* [Hohpe 20] describes this unique ability of architects to interact with people at all levels inside and outside an organization.

24.7 Discussion Questions

1. Consider “amenable to globally distributed development” as a quality attribute that can be increased or decreased by architectural design decisions, just like the other quality attributes we outlined in Part II of this book. Construct a general scenario for it, and a list of tactics to help achieve it. Oh, and figure out a good name for it.
2. Generic project management practices often 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, which architectural documentation artifacts would you want to create first?
4. If you were managing a globally distributed team, which aspects of project management would have to change to account for cultural differences?
5. How could architectural evaluation be used to help guide and manage the project?
6. In Chapter 1, we described a work assignment structure for software architecture, which can be documented as a work assignment 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?

This page intentionally left blank

25



Architecture Competence

The lyf so short, the craft so long to lerne.
—Geoffrey Chaucer

If software architecture is worth doing, then 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. But architectures are created by *architects* working in *organizations* that are full of actual human beings. Dealing with these humans is a decidedly nontechnical undertaking. 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 organizations do a better job of encouraging their architects to produce their best work?

This chapter is about the competence of individual architects and the organizations that wish to produce high-quality architectures.

Since 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. Understanding only one or the other won't do.

25.1 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. 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, technologies, standards, quality attributes, and a host of other topics.

Duties, skills, and knowledge form a triad upon which architecture competence for individuals rests. The relationship among these three is shown in Figure 25.1—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.

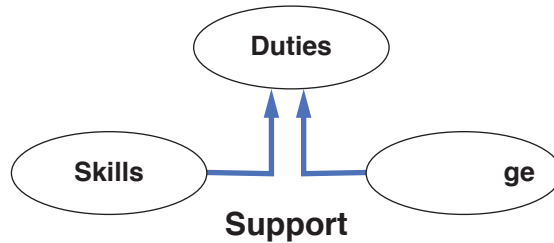


FIGURE 25.1 Skills and knowledge support the execution of duties.

To give examples of these concepts:

- “Design the architecture” is a duty.
- “Ability to think abstractly” is a skill.
- “Patterns and tactics” constitute knowledge.

These examples purposely illustrate 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 take the following steps:

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 keeping current with the field, consider the advances in knowledge required for architects that have emerged in just the last few years. For example, architectures to support computing in the cloud (Chapter 17) were not important several years

ago. Taking courses, becoming certified, reading books and journals, visiting websites, 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 architects' duties. Not every architect in every organization will perform every one of these duties on every project. However, competent architects should not be surprised to find themselves engaged in any of the activities listed here. We divide these duties into technical duties (Table 25.1) and nontechnical duties (Table 25.2). One immediate observation you should make is the large number of many nontechnical duties. 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 25.1 Technical Duties of a Software Architect

General Duty Area	Specific Duty Area	Example Duties
Architecting	Creating an architecture	Design or select an architecture. Create a software architecture design plan. Build a product line or product architecture. Make design decisions. Expand details and refine the design to converge on a 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. Create prototypes.
	Evaluating and analyzing an architecture	Evaluate an architecture (for your current system or for other systems) to determine the satisfaction of use cases and quality attribute scenarios. Create prototypes. Participate in design reviews. Review the designs of the components designed by junior engineers. Review designs for compliance with the architecture. Compare software architecture evaluation techniques. Model alternatives. Perform tradeoff analysis.
	Documenting an architecture	Prepare architectural documents and presentations useful to stakeholders. Document or automate the documentation of software interfaces. Produce documentation standards or guidelines. Document variability and dynamic behavior.
	Working with and transforming existing system(s)	Maintain and evolve an existing system and its architecture. Measure architecture debt. Migrate existing system to new technology and platforms. Refactor existing architectures to mitigate risks. Examine bugs, incident reports, and other issues to determine revisions to existing architecture.
	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. Provide architecture oversight of software development activities.

continues

TABLE 25.1 Technical Duties of a Software Architect *continued*

General Duty Area	Specific Duty Area	Example Duties
Duties concerned with life-cycle activities other than architecting	Managing the requirements	Analyze functional and quality attribute software requirements. Understand business, organizational, and customer needs, and ensure that the requirements meet these needs. Listen to and understand the scope of the project. Understand the client's key design needs and expectations. Advise on the tradeoffs between software design choices and requirements choices.
	Evaluating future technologies	Analyze the current IT environment and recommend solutions for deficiencies. Work with vendors to represent the organization's requirements and influence future products. Develop and present technical white papers.
	Selecting tools and technology	Manage the introduction of new software solutions. 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 25.2 Nontechnical Duties of a Software Architect

General Duty Area	Specific Duty Area	Example Duties
Management and business-related duties	Supporting project management	Provide feedback on the appropriateness and difficulty of 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 metrics and improve both personal results and teams' productivity. Identify and schedule architectural releases. Serve as a "bridge" between the technical team and the project manager.
	Managing the people on the architect's team	Build "trusted advisor" relationships. Coordinate. Motivate. Advocate. Train. Act as a supervisor. Allocate responsibilities.
	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 cost-effective and appropriate software architecture design reviews. Help develop intellectual property.
	Supporting the business	Understand and evaluate business processes. Translate business strategy into technical strategy. Influence the business strategy. Understand and communicate the business value of software architecture. Help the organization meet its business goals. Understand customer and market trends.

General Duty Area	Specific Duty Area	Example Duties
Leadership and team building	Providing technical leadership	Be a thought leader. Produce technology trend analysis or roadmaps. Mentor other architects.
	Building a team	Build the development team and align them with the architecture vision. Mentor developers and junior architects. Educate the team on the use of the architecture. 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. Maintain morale, both within and outside the architecture group. Monitor and manage team dynamics.

Architects also routinely perform many other duties, such as leading code reviews or getting involved in test planning. In many projects, architects pitch in to help with the actual implementation and testing, in critical areas. While important, these are not strictly speaking architectural duties.

Skills

Given the wide range of duties enumerated in the previous section, which skills 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 assessment areas of leadership, organization dynamics, and communication.

Table 25.3 enumerates the set of skills most useful to an architect.

TABLE 25.3 Skills of a Software Architect

General Skill Area	Specific Skill Area	Example Skills
Communication skills	Outward communication (beyond the team)	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 communication (within the team)	Ability to listen, interview, consult, and negotiate. Ability to understand and express complex topics.
Interpersonal skills	Team relationships	Ability to be a team player. Ability to work effectively with superiors, subordinates, 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. Ability to be diplomatic and respect others. Ability to mentor others. Ability to handle and resolve conflict.

continues

TABLE 25.3 Skills of a Software Architect *continued*

General Skill Area	Specific Skill Area	Example Skills
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, and estimate. Ability to support a wide range of issues and work on multiple complex tasks concurrently. Ability to effectively prioritize and execute tasks in a high-pressure environment.
	Skills to excel in the corporate environment	Ability to think strategically. Ability to work under general supervision and under constraints. Ability to organize workflow. Ability to detect 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, to be assertive without being aggressive, and to receive constructive criticism.
	Skills for handling information	Ability to be detail-oriented while maintaining overall vision and focus. Ability to see the big picture.
	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 think abstractly	Ability to look at different things and find a way to see how they are, in fact, just different instances of the same thing. This may be one of the most important skills for an architect to have.

Knowledge

A competent architect has an intimate familiarity with an architectural body of knowledge. Table 25.4 gives a set of knowledge areas for an architect.

TABLE 25.4 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, structures and views, reference architectures, relationships to system and enterprise architecture, emerging technologies, architecture evaluation models and methods, and quality attributes.
	Knowledge of software engineering	Knowledge of software development knowledge areas, including requirements, design, construction, maintenance, configuration management, engineering management, and software engineering process. Knowledge of systems engineering.

General Knowledge Area	Specific Knowledge Area	Specific Knowledge Examples
Computer science knowledge	Design knowledge	Knowledge of tools and design and analysis techniques. Knowledge of how to design complex multi-product systems. Knowledge of object-oriented analysis and design, and UML and SysML diagrams.
	Programming knowledge	Knowledge of programming languages and programming language models. Knowledge of specialized programming techniques for security, real time, safety, etc.
Knowledge of technologies and platforms	Specific technologies and platforms	Knowledge of hardware/software interfaces, web-based applications, and Internet technologies. Knowledge of specific software/operating systems.
	General knowledge of technologies and platforms	Knowledge of the IT industry's 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 domains 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 environments.
	Business knowledge	Knowledge of the company's business practices, and its 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	Knowledge of how to coach, mentor, and train software team members. Knowledge of project management. Knowledge of project engineering.

What about Experience?

Albert Einstein said, “The only source of knowledge is experience,” and just about everybody says that experience is the best teacher. We agree. However, experience is not the *only* teacher—you can also acquire knowledge from *real* teachers. How lucky we are that we need not all burn ourselves to acquire the knowledge that touching a hot stove is a bad idea.

We consider experience as something that adds to an architect's store of knowledge, which is why we don't treat it separately. As your career advances, you'll accumulate your own wealth of experience, which you'll store as knowledge.

As the old joke goes, a pedestrian in New York stopped a passerby and asked, “Excuse me. Could you tell me how to get to Carnegie Hall?” The passerby, who happened to be a musician, replied with a heavy sigh, “Practice, practice, practice.”

Exactly.

25.2 Competence of a Software Architecture Organization

Organizations, by their practices and structure, can either 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. The absence of these practices and structures 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. Here is our definition:

The architectural 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.

Organizations have duties, skills, and knowledge for architecture, just like individual architects. For example, adequately funding the architecture effort is an organizational duty, as is effectively using the available architecture workforce (by appropriate teaming and other means). 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 could perform to help improve the success of its architecture efforts:

- *Personnel-related:*
 - Hire talented architects.
 - Establish a career track for architects.
 - Make the position of architect highly regarded through visibility, rewards, and prestige.
 - Have architects join professional organizations.
 - Establish an architect certification program.
 - Establish a mentoring program for architects.
 - Establish an architecture training and education program.
 - Measure architects' performance.
 - Have architects receive external architect certifications.
 - Reward or penalize architects based on project success or failure.

- *Process-related:*

- Establish organization-wide architecture practices.
- Establish a clear statement of responsibilities and authority for architects.
- Establish a forum for architects to communicate and share information and experience.
- Establish an architecture review board.
- Include architecture milestones in project plans.
- Have architects provide input into product definition.
- Hold an organization-wide architecture conference.
- Measure and track the quality of architectures produced.
- Bring in outside expert consultants on architecture.
- Have architects advise on the development team structure.
- Give architects influence throughout the entire project life cycle.

- *Technology-related:*

- Establish and maintain a repository of reusable architectures and architecture-based artifacts.
- Create and maintain a repository of design concepts.
- Provide a centralized resource to analyze and help with architecture tools.

If you are interviewing for the position of architect in an organization, you'll probably have a list of questions to determine if you want to work there. To that list, you can add questions drawn from the preceding list to help you ascertain the organization's level of architecture competence.

25.3 Become a Better Architect

How do architects become good architects, and how do good architects become great architects? We close this chapter with a proposal, which is this: Be mentored, and mentor others.

Be Mentored

While experience may be the best teacher, most of us will not have the luxury, in a single lifetime, to gain firsthand all the experience needed to make us great architects. But we can gain experience secondhand. Find a skilled architect whom you respect, and attach yourself to that person. Find out if your organization has a mentoring program that you can join. Or establish an informal mentoring relationship—find excuses to interact, ask questions, or offer to help (for instance, offer to be a reviewer).

Your mentor doesn't have to be a colleague. You can also join professional societies where you can establish mentor relationships with other members. There are meetups. There are professional social networks. Don't limit yourself to just your organization.

Mentor Others

You should also be willing to mentor others as a way of giving back or paying forward the kindnesses that have enriched your career. But there is a selfish reason to mentor as well: We find that teaching a concept is the litmus test of whether we deeply understand that concept. If we can't teach it, it's likely we don't really understand it—so that can be part of your goal in teaching and mentoring others in the profession. Good teachers almost always report their delight in how much they learn from their students, and how much their students' probing questions and surprising insights add to the teachers' deeper understanding of the subject.

25.4 Summary

When we think of software architects, we usually first think of the technical work that they produce. But, in the same way that an architecture is much more than a technical “blueprint” for a system, an architect is much more than a designer of an architecture. 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. An architect must carry out the duties, hone the skills, and continuously acquire the knowledge necessary to be successful.

The key to becoming a good and then a better architect is continuous learning, mentoring, and being mentored.

25.5 For Further Reading

Questions to probe an organization's competence can be found in the Technical Note, “Models for Evaluating and Improving Architecture Competence,” sei.cmu.edu/library/abstracts/reports/08tr006.cfm.

The Open Group has 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.

The Information Technology Architecture Body of Knowledge (ITABoK) is a “free public archive of IT architecture best practices, skills, and knowledge developed from the experience of individual and corporate members of Iasa, the world's largest IT architecture professional organization” (<https://itabok.iasaglobal.org/itabok/>).

Bredemeyer Consulting (bredemeyer.com) provides copious materials about IT, software, and enterprise architects and their role.

Joseph Ingeno, in *Software Architect's Handbook*, devotes a chapter to “The Soft Skills of Software Architects” and another one to “Becoming a Better Software Architect” [Ingeno 18].

25.6 Discussion Questions

1. In which skills and knowledge discussed in this chapter do you think you might be most 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. Add three duties, three skills, and three knowledge areas that were not on our lists.
4. How would you measure the value of specific architecture duties in a project? How would you distinguish the value added by these duties from the value added by other activities such as quality assurance or configuration management?
5. How would you measure someone's communication skills?
6. This chapter 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?
8. Suppose you are the architect being hired. What questions would you ask about the company with which you're interviewing, related to the areas listed in Section 25.2? Try to answer this question from the point of view of an architect early in their career, and then from the point of view of a highly skilled architect with many years of experience.
9. Search for certification programs for architects. For each one, try to characterize how much it deals (respectively) with duties, skills, and knowledge.

This page intentionally left blank

26



A Glimpse of the Future: Quantum Computing

[A quantum computer can be compared] to the airplane the Wright brothers flew at Kitty Hawk in 1903. The Wright Flyer barely got off the ground, but it foretold a revolution.

—wired.com/2015/12/for-google-quantum-computing-is-like-learning-to-fly/

What will the future bring in terms of developments that affect the practice of software architecture? Humans are notoriously bad at predicting the long-term future, but we keep trying because, well, it's fun. To close our book, we have chosen to focus on one particular aspect that is firmly rooted in the future but seems tantalizingly close to reality: quantum computing.

Quantum computers will likely become practical over the next five to ten years. Consider that the system you are currently working on may have a lifetime on the order of tens—plural—of years. Code written in the 1960s and 1970s is still being used today on a daily basis. If the systems you are working on have lifetimes on that order, you may need to convert them to take advantage of quantum computer capabilities when quantum computers become practical.

Quantum computers are generating high interest because of their potential to perform calculations at speeds that far outpace the most capable and powerful of their classical counterparts. In 2019, Google announced that its quantum computer completed a complex computation in 200 seconds. That same calculation, claimed Google, would take even the most powerful supercomputers approximately 10,000 years to finish. It isn't that quantum computers do what classical computers do, only extraordinarily faster; rather, they do what classical computers can't do using the otherworldly properties of quantum physics.

Quantum computers won't be better than classical computers at solving every problem. For example, for many of the most common transaction-oriented data-processing tasks, they are likely irrelevant. They will be good at problems that involve combinatorics and are computationally difficult for classic computers. However, it is unlikely that a quantum computer will ever power your phone or watch or sit on your office desk.

Understanding the theoretical basis of a quantum computer involves deep understanding of physics, including quantum physics, and that is far outside our scope. For context, the same was also true of classical computers when they were invented in the 1940s. Over time, the

requirement for understanding how CPUs and memory work has disappeared due to the introduction of useful abstractions, such as high-level programming languages. The same thing will happen in quantum computers. In this chapter, we introduce the essential concepts of quantum computing without reference to the underlying physics (which has been known to make heads actually explode).

26.1 Single Qubit

The fundamental unit of calculation in a quantum computer is a unit of quantum information called a *qubit* (more on that shortly). The simple definition of a quantum computer is a processor that manipulates qubits. At the time of this book's publication, the best quantum computer in existence contained several hundred qubits.

A “QPU” will interact with a classic CPU in the same fashion that a graphic processing unit interacts with a CPU today. In other words, the CPU will view the QPU as a service to be provided with some input and that will produce some output. The communications between the CPU and the QPU will be in terms of classic bits. What the QPU does with the input to produce the output is outside of the scope of the CPU.

A bit in a classic computer has a value of either 0 or 1 and, when functioning properly, there is no ambiguity about which value it assumes. Also, a bit in a classic computer has a non-destructive readout. That is, measuring the value will give you a 0 or a 1, and the bit will retain the value that it had when the read operation began.

A qubit differs in both characteristics. A qubit is characterized by three numbers. Two of these numbers are probabilities: the probability that a measurement will deliver 1 and the probability that a measurement will deliver 0. The third number, called the phase, describes a rotation of the qubit. A measurement of a qubit will return either a 0 or a 1 (with probabilities as designated) and will destroy the current value of the qubit and replace it with the value that it returned. A qubit with non-zero probabilities for both 0 and 1 is said to be in superposition.

Phases are managed by making the probabilities complex numbers. The amplitudes (probabilities) are designated as $|\alpha|^2$ and $|\beta|^2$. If $|\alpha|^2$ is 40 percent and $|\beta|^2$ is 60 percent, then 4 out of 10 measurements will be 0, and 6 out of those 10 measurements will be 1. These amplitudes are subject to some probability of measurement error, and reducing this error probability is one of the engineering challenges of building quantum computers.

There are two consequences of this definition:

1. $|\alpha|^2 + |\beta|^2 = 1$. Because $|\alpha|^2$ and $|\beta|^2$ are probabilities of a measurement delivering 0 or 1, respectively, and because a measurement will deliver one or the other, the sum of the probabilities must be 1.
2. There is no copying of a qubit. A copy from classical bit A to classical bit B is a read of bit A followed by a store of that value into B. The measurement (i.e., read) of qubit A will destroy A and deliver either a value of 0 or a value of 1. The store into qubit B will thus be either a 0 or a 1 and will not encompass the probabilities or phases that were embedded into A.

The phase value is an angle between 0 and 2π radians. It does not affect the probabilities of the superposition, but gives another lever to manipulate qubits. Some quantum algorithms mark certain qubits by manipulating their phase.

Operations on Qubits

Some single qubit operations are analogs of classical bit operations, whereas others are specific to qubits. One characteristic of most quantum operations is that they are invertible; that is, given the result of an operation, it is possible to recover the input into that operation. Invertibility is another distinction between classical bit operations and qubit operations. The one exception to invertibility is the READ operation: Since measurement is destructive, the result of a READ operation does not allow the recovery of the original qubit. Examples of qubit operations include the following:

1. A READ operation takes as input a single qubit and produces as output either a 0 or a 1 with probabilities determined by the amplitudes of the input qubit. The value of the input qubit collapses to either a 0 or a 1.
2. A NOT operation takes a qubit in superposition and flips the amplitudes. That is, the probability of the resulting qubit being 0 is the original probability of it being 1, and vice versa.
3. A Z operation adds π to the phase of the qubit (modulo 2π).
4. A HAD (short for Hadamard) operation creates an equal superposition, which means the amplitudes of qubits with value 0 and 1, respectively, are equal. A 0 input value generates a phase of 0 radians, and a 1 input value generates a phase of π radians.

It is possible to chain multiple operations together to produce more sophisticated units of functionality.

Some operators work on more than one qubit. The primary two-qubit operator is CNOT—a controlled not. The first qubit is the control bit. If it is 1, then the operation performs a NOT on the second qubit. If the first qubit is 0, then the second qubit remains unchanged.

Entanglement

Entanglement is one of the key elements of quantum computing. It has no analog in classical computing, and gives quantum computing some of its very strange and wondrous properties, allowing it to do what classical computers cannot.

Two qubits are said to be “entangled” if, when measured, the second qubit measurement matches the measurement of the first. Entanglement can occur no matter the amount of time between the two measurements, or the physical distance between the qubits. This leads us to what is called quantum teleportation. Buckle up.

26.2 Quantum Teleportation

Recall that it is not possible to copy one qubit to another directly. Thus, if we want to copy one qubit to another, we must use indirect means. Furthermore, we must accept the destruction of the state of the original qubit. The recipient qubit will have the same state as the original, destroyed qubit. *Quantum teleportation* is the name given to this copying of the state. There is no requirement that the original qubit and the recipient qubit have any physical relationship, nor are there constraints on the distance that separates them. In consequence, it is possible to transfer information over great distances, even hundreds or thousands of kilometers, between qubits that have been physically implemented.

The teleportation of the state of a qubit depends on entanglement. Recall that entanglement means that a measurement of one entangled qubit will guarantee that a measurement of the second qubit will have the same value. Teleportation utilizes three qubits. Qubit A and B are entangled, and then qubit ψ is entangled with qubit A. Qubit ψ is teleported to the location of qubit B, and its state becomes the state of qubit B. Roughly speaking, teleportation proceeds through these four steps:

1. Entangle qubits A and B. We discussed what this means in the prior section. The locations of A and B can be physically separate.
2. Prepare the “payload.” The payload qubit will have the state to be teleported. The payload, which is the qubit ψ , is prepared at the location of A.
3. Propagate the payload. The propagation involves two classical bits that are transferred to the location of B. The propagation also involves measuring A and ψ , which destroys the state of both of these qubits.
4. Re-create the state of ψ in B.

We have omitted many key details, but the point is this: Quantum teleportation is an essential ingredient of quantum communication. It relies on transmitting two bits over conventional communication channels. It is inherently secure, since all that an eavesdropper can determine are the two bits sent over conventional channels. Because A and B communicate through entanglement, they are not physically sent over a communication line. The U.S. National Institute of Science and Technology (NIST) is considering a variety of different quantum-based communication protocols to be the basis of a transport protocol called HTTPQ, which is intended to be a replacement for HTTPS. Given that it takes decades to replace one communication protocol with another, the goal is for HTTPQ to be adopted prior to the availability of quantum computers that can break HTTPS.

26.3 Quantum Computing and Encryption

Quantum computers are extremely proficient at calculating the inverse of a function—in particular, the inverse of a hash function. There are many cases where this kind of calculation

would be extremely useful, but particularly so in decrypting passwords. Passwords are almost never directly stored; instead, the hash of them is stored. The assumption behind storing only the hash is that computing the inverse of the hash function is computationally difficult and would take hundreds, if not thousands, of years to do—using conventional computers, that is. Quantum computers, however, change this calculation.

Grover’s algorithm is an example of a probabilistic algorithm that computes the inverse of a function. It takes on the order of 2^{128} iterations to calculate the inverse of a hash based on 256 bits. This represents a quadratic speedup over conventional computational algorithms, meaning that the quantum algorithm time is approximately the square root of the conventional algorithm time. This makes an enormous amount of password-protected material, previously thought to be secure, quite vulnerable.

Modern secure encryption algorithms are based on the difficulty of factoring the product of two large prime numbers. Let p and q be two distinct primes each greater than 128 bits in magnitude. The product of these two primes pq is roughly 256 bits in magnitude. This product is relatively easy to compute given p and q . However, factoring the product, pq , and recovering p and q is computationally very difficult on a classical computer: It is in the category *NP*-hard.

What this means is that given a message encrypted based on the primes p and q , decrypting this message is relatively easy if you know p and q but practically impossible if you don’t—at least on a classical computer. Quantum computers, however, can factor pq much more efficiently than classical computers. Shor’s algorithm is a quantum algorithm that can factor pq with running time on the order of \log (number of bits in p and q).

26.4 Other Algorithms

Quantum computing holds similar game-changing potential for many applications. Here, we begin our discussion by introducing a necessary but currently nonexistent piece of hardware—QRAM.

QRAM

Quantum random access memory (QRAM) is a critical element for implementing and applying many quantum algorithms. QRAM, or something similar, will be necessary to provide efficient access to large amounts of data such as that used in machine learning applications. Currently, no implementation of QRAM exists, but several research groups are exploring how such an implementation could work.

Conventional RAM comprises a hardware device that takes as input a memory location and returns as output the contents of that memory location. QRAM is conceptually similar: It takes as input a memory location (likely a superposition of memory locations) and returns as output the superpositioned contents of those memory locations. The memory locations whose contents are returned were written conventionally—that is, each bit has one value. The values are returned in superposition, and the amplitudes are determined by the specification of the

memory locations to be returned. Because the original values were conventionally written, they can be copied in a nondestructive fashion.

A problem with the proposed QRAM is that the number of physical resources required scales linearly with the number of bits retrieved. Thus it may not be practical to construct QRAM for very large retrievals. As with much of the discussion of quantum computers, QRAM is in the theoretical discussion stage rather than the engineering phase. Stay tuned.

The remaining algorithms we discuss assume the existence of a mechanism for efficiently accessing the data manipulated by an algorithm, such as with QRAM.

Matrix Inversion

Matrix inversion underlies many problems in science. Machine learning, for example, requires the ability to invert large matrices. Quantum computers hold promise to speed up matrix inversion in this context. The HHL algorithm by Harrow, Hassidim, and Lloyd will invert a linear matrix, subject to some constraints. The general problem is to solve the equation $Ax = b$, where A is an $N \times N$ matrix, x is a set of N unknowns, and b is a set of N known values. You learned about the simplest case ($N = 2$) in elementary algebra. As N grows, however, matrix inversion becomes the standard technique to solve the set of equations.

The following constraints apply when solving this problem with quantum computers:

1. The b 's must be quickly accessible. This is the problem that QRAM is supposed to solve.
2. The matrix A must satisfy certain conditions. If it is a sparse matrix, then it likely can be processed efficiently on a quantum computer. The matrix must also be well conditioned; that is, the determinant of the matrix must be non-zero or close to zero. A small determinant causes issues when inverting a matrix on a classical computer, so this is not a quantum unique problem.
3. The result of applying the HHL algorithm is that the x values appear in superposition. Thus a mechanism is needed for efficiently isolating the actual values from the superposition.

The actual algorithm is too complicated for us to present here. One noteworthy element, however, is that it relies on an amplitude magnification technique based on using phases.

26.5 Potential Applications

Quantum computers are expected to have an impact on a wide variety of application areas. IBM, for example, is focusing on cybersecurity, drug development, financial modeling, better batteries, cleaner fertilization, traffic optimization, weather forecasting and climate change, and artificial intelligence and machine learning, to name just a few.

To date, except for cybersecurity, this list of potential quantum computing applications remains mostly speculation. Several cybersecurity algorithms have been proven to provide

substantial improvements over classical algorithms, but the remainder of the application areas are, thus far, the subject of much and feverish research. As yet, however, none of these efforts has generated public results.

As the chapter-opening quotation suggested, quantum computers are at the stage that airplanes were at the time of the Wright brothers. The promise is great but a tremendous amount of work must be done to turn the promise into reality.

26.6 Final Thoughts

Quantum computers are currently in their infancy. Applications for such computers are primarily speculation at this point, especially applications that require large amounts of data. Nonetheless, progress is happening rapidly in terms of the number of qubits in actual physical existence. It seems reasonable that Moore's law will apply to quantum computers, much as it has in conventional computing. If so, then the number of qubits available will grow exponentially over time.

The qubit operations discussed in Section 26.2 lend themselves to a programming style where operations are chained together to perform useful functionality. This will likely follow the same arc as machine languages for classical computers. Machine languages still exist but have become a realm consigned to only a handful of programmers. Most programmers use a wide variety of higher-level languages. We should expect to see the same evolution in programming quantum computers. Efforts at quantum computing language design are under way but remain in a nascent state.

Programming languages are only the tip of the iceberg. What about the other topics we have covered in this book? Are there new quality attributes relevant to quantum computers, new architectural patterns, an additional architecture view? Almost certainly.

What will a network of quantum computers look like? Will hybrid networks of quantum and classical computers become widespread? All of these are potential areas into which quantum computing will almost certainly evolve—eventually.

What can architects do in the meantime? First, pay attention to breaking developments. If the systems you are working on today involve areas that quantum computing is likely to affect (or, more likely, completely turn on its head), isolate those parts of the system to minimize the disruption when quantum computing finally shows up. Especially for secure systems, follow the field to find out what to do when your conventional encryption algorithms become worthless.

But your preparation need not all be defensive. Imagine what you could do with a communication network that is able to transfer information instantly, no matter the physical distance between the nodes. If this sounds far-fetched—well, so did flying machines once upon a time.

As always, we await the future with eagerness.

26.7 For Further Reading

General overview:

- *Programming Quantum Computers* by Eric Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia discusses quantum computing without reference to physics or linear algebra [Johnston 19].
- *Quantum Computing: Progress and Prospects* [NASEM 19] provides an overview of the current state of quantum computing and the challenges to be overcome to make real quantum computers.
- Quantum computers not only provide faster solutions compared to classical computers, but also address some problems that can *only* be solved with quantum computers. This powerful theoretical result emerged in May 2018: quantamagazine.org/finally-a-problem-that-only-quantum-computers-will-ever-be-able-to-solve-20180621/.

References

- [Abrahamsson 10] P. Abrahamsson, M. A. Babar, and P. Kruchten. “Agility and Architecture: Can They Coexist?” *IEEE Software* 27, no. 2 (March–April 2010): 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 20] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3rd ed. Wiley, 2020.
- [Argote 07] L. Argote and G. Todorova. *International Review of Industrial and Organizational Psychology*. John Wiley & Sons, 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* 1, no. 1 (January 2004): 11–33.
- [Bachmann 00a] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord, and Reed Little. “Software Architecture Documentation in Practice: Documenting Architectural Layers,” CMU/SEI-2000-SR-004, 2000.
- [Bachmann 00b] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi. “The Architecture-Based Design Method,” CMU/SEI-2000-TR-001, 2000.
- [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): 8–10, 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, 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, no. 3 (2003): 187–197.
- [Bass 07] Len Bass, Robert Nord, William G. Wood, and David Zubrow. “Risk Themes Discovered through Architecture Evaluations,” in *Proceedings of WICSA 07*, 2007.

- [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, sei.cmu.edu/library/abstracts/reports/08tr006.cfm.
- [Bass 15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect’s Perspective*. Addison-Wesley, 2015.
- [Bass 19] Len Bass and John Klein. *Deployment and Operations for Software Engineers*. Amazon, 2019.
- [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, no. 13 (October 2005): 859–879.
- [Beck 02] Kent Beck. *Test-Driven Development by Example*. Addison-Wesley, 2002.
- [Beck 04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, 2004.
- [Beizer 90] B. Beizer. *Software Testing Techniques*, 2nd ed. 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.
- [Bellomo 15] S. Bellomo, I. Gorton, and R. Kazman. “Insights from 15 Years of ATAM Data: Towards Agile Architecture,” *IEEE Software* 32, no. 5 (September/October 2015): 38–45.
- [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* 22, no. 2 (February 1996): 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. Finkelstein, 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, no. 9 (1994): 87–101.
- [Binder 00] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [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* 8, no. 1 (January 1991): 32–41.

- [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* 11, no. 3 (2007): 221–234.
- [Boehm 10] B. Boehm, J. Lane, S. Koolmanojwong, and R. Turner. “Architected Agile Solutions for Software-Reliant Systems,” Technical Report USC-CSSE-2010-516, 2010.
- [Bondi 14] A. B. Bondi. *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. Addison-Wesley, 2014.
- [Booch 11] Grady Booch. “An Architectural Oxymoron,” podcast available at 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, no. 4 (July/August 2010): 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): 23–29.
- [Brown 10] N. Brown, R. Nord, and I. Ozkaya. “Enabling Agility through Architecture,” *Crosstalk* (November/December 2010): 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, 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, no. 9 (2006): 1219–1232.
- [Buschmann 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume I: 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.
- [Cervantes 13] H. Cervantes, P. Velasco, and R. Kazman. “A Principled Way of Using Frameworks in Architectural Design,” *IEEE Software* (March/April 2013): 46–53.
- [Cervantes 16] H. Cervantes and R. Kazman. *Designing Software Architectures: A Practical Approach*. Addison-Wesley, 2016.
- [Chandran 10] S. Chandran, A. Dimov, and S. Punnekkat. “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 of 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, no. 2 (April–June 2010): 145–160.
- [Chidamber 94] S. Chidamber and C. Kemerer. “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering* 20, no. 6 (June 1994).
- [Chowdury 19] S. Chowdhury, A. Hindle, R. Kazman, T. Shuto, K. Matsui, and Y. Kamei. “GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing,” *Proceedings of the International Conference on Software Engineering*, May 2019.
- [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*, 2nd ed. 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, no. 6 (November–December 2010): 38–45.
- [Clements 16] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2016.
- [Cockburn 04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.
- [Cockburn 06] Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, 2006.
- [Conway 68] Melvin E. Conway. “How Do Committees Invent?” *Datamation* 14, no. 4 (1968): 28–31.

- [Coplein 10] J. Coplein and G. Bjornvig. *Lean Architecture for Agile Software Development*. Wiley, 2010.
- [Coulin 19] T. Coulin, M. Detante, W. Mouchère, F. Petrillo. et al. “Software Architecture Metrics: A Literature Review,” January 25, 2019, <https://arxiv.org/abs/1901.09050>.
- [Cruz 19] L. Cruz and R. Abreu. “Catalog of Energy Patterns for Mobile Applications,” *Empirical Software Engineering* 24 (2019): 2209–2235.
- [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>.
- [Dean 13] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale,” *Communications of the ACM* 56, no. 2 (February 2013): 74–80.
- [Dijkstra 68] E. W. Dijkstra. “The Structure of the ‘THE’-Multiprogramming System,” *Communications of the ACM* 11, no. 5 (1968): 341–346.
- [Dijkstra 72] Edsger W. Dijkstra, Ole-Johan Dahl, and Tony Hoare, *Structured Programming*. Academic Press, 1972: 175–220.
- [Dix 04] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human–Computer Interaction*, 3rd ed. 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): 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,” eurocontrol.int/valfor/gallery/content/public/OATA-P2-D8.1.4-01%20DMVO%20Architecture%20Tactics%20Assessment%20Process.pdf.
- [FAA 00] “System Safety Handbook,” faa.gov/library/manuals/aviation/risk_management/ss_handbook/.
- [Fairbanks 10] G. Fairbanks. *Just Enough Software Architecture: A Risk-Driven Approach*. Marshall & Brainerd, 2010.
- [Fairbanks 20] George Fairbanks. “Ur-Technical Debt,” *IEEE Software* 37, no. 4 (April 2020): 95–98.
- [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*. sei.cmu.edu/library/assets/ULS_Book20062.pdf.

- [Feng 16] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao. “An Architecture-centric Approach to Security Analysis,” in *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016)*, 2016.
- [Fiol 85] C. M. Fiol and M. A. Lyles. “Organizational Learning,” *Academy of Management Review* 10, no. 4 (1985): 803.
- [Fonseca 19] A. Fonseca, R. Kazman, and P. Lago. “A Manifesto for Energy-Aware Software,” *IEEE Software* 36 (November/December 2019): 79–82.
- [Fowler 09] Martin Fowler. “TechnicalDebtQuadrant,” <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>, 2009.
- [Fowler 10] Martin Fowler. “Blue Green Deployment,” <https://martinfowler.com/bliki/BlueGreenDeployment.html>, 2010.
- [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* 22, no. 3 (March/April 2009): 12–15.
- [Gajjarby 17] Manish J. Gajjarby. *Mobile Sensors and Context-Aware Computing*. Morgan Kaufman, 2017.
- [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., 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,” 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*, 2nd ed. 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, businessweek.com/technology/content/aug2004/tc20040818_1593.htm.
- [Hamming 80] R. W. Hamming. *Coding and Information Theory*. Prentice Hall, 1980.
- [Hanmer 13] Robert S. Hanmer. *Patterns for Fault Tolerant Software*, Wiley Software Patterns Series, 2013.
- [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.
- [Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.
- [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* 80, no. 1 (January 2007): 106–126.
- [Hohpe 20] Gregor Hohpe. *The Software Architect Elevator: Redefining the Architect’s Role in the Digital Enterprise*. O’Reilly, 2020.
- [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>.
- [Hubbard 14] D. Hubbard. *How to Measure Anything: Finding the Value of Intangibles in Business*. Wiley, 2014.
- [Humble 10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.
- [IEEE 94] “IEEE Standard for Software Safety Plans,” STD-1228-1994, <http://standards.ieee.org/findstds/standard/1228-1994.html>.
- [IEEE 17] “IEEE Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide), Sixth Edition,” 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,” incose.org/ProductsPubs/products/competencies-framework.aspx.
- [INCOSE 19] International Council on Systems Engineering, “Feature-Based Systems and Software Product Line Engineering: A Primer,” Technical Product INCOSE-TP-2019-002-03-0404, https://connect.incose.org/Pages/Product-Details.aspx?ProductCode=PLE_Primer_2019.
- [Ingeno 18] Joseph Ingeno. *Software Architect’s Handbook*. Packt Publishing, 2018.
- [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.
- [Johnston 19] Eric Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia, *Programming Quantum Computers*. O’Reilly, 2019.
- [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): 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, 1994, pp. 81–90.
- [Kazman 99] R. Kazman and S. J. Carriere. “Playing Detective: Reconstructing Software Architecture from Available Evidence,” *Automated Software Engineering* 6, no 2 (April 1999): 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, 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): 76–84.
- [Kazman 15] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyeve, V. Fedak, and A. Shapochka. “A Case Study in Locating the Architectural Roots of Technical Debt,” in *Proceedings of the International Conference on Software Engineering (ICSE) 2015*, 2015.
- [Kazman 18] R. Kazman, S. Haziyeve, A. Yakuba, and D. Tamburri. “Managing Energy Consumption as an Architectural Quality Attribute,” *IEEE Software* 35, no. 5 (2018).
- [Kazman 20a] R. Kazman, P. Bianco, J. Ivers, and J. Klein. “Integrability,” CMU/SEI-2020-TR-001, 2020.
- [Kazman 20b] R. Kazman, P. Bianco, J. Ivers, and J. Klein. “Maintainability,” CMU/SEI-2020-TR-006, 2020.
- [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, 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.
- [Koopman 10] Phil Koopman. *Better Embedded System Software*. Drumnadrochit Education, 2010.
- [Koziollet 10] H. Koziollet. “Performance Evaluation of Component-Based Software Systems: A Survey,” *Performance Evaluation* 67, no. 8 (August 2010).
- [Kruchten 95] P. B. Kruchten. “The 4+1 View Model of Architecture,” *IEEE Software* 12, no. 6 (November 1995): 42–50.
- [Kruchten 03] Philippe Kruchten. *The Rational Unified Process: An Introduction*, 3rd ed. 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, Netherlands, December 3–4, 2004.
- [Kruchten 19] P. Kruchten, R. Nord, and I. Ozkaya. *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley, 2019.
- [Kumar 10a] K. Kumar and T. V. Prabhakar. “Pattern-Oriented Knowledge Model for Architecture Design,” in *Pattern Languages of Programs Conference 2010*, Reno/Tahoe, NV: October 15–18, 2010.
- [Kumar 10b] Kiran Kumar and T. V. Prabhakar. “Design Decision Topology Model for Pattern Relationship Analysis,” *Asian Conference on Pattern Languages of Programs 2010*, Tokyo, Japan, March 15–17, 2010.
- [Ladas 09] Corey Ladas. *Scrumban: Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, 2009.
- [Lamport 98] Leslie Lamport. “The Part-Time Parliament,” *ACM Transactions on Computer Systems* 16, no. 2 (May 1998): 133–169.

- [Lampson 11] Butler Lampson, “Hints and Principles for Computer System Design,” <https://arxiv.org/pdf/2011.02455.pdf>.
- [Lattanze 08] Tony Lattanze. *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)*. Washington, DC: November 1997, pp. 91–98.
- [Leveson 04] Nancy G. Leveson. “The Role of Software in Spacecraft Accidents,” *Journal of Spacecraft and Rockets* 41, no. 4 (July 2004): 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): 319–340.
- [Lewis 14] J. Lewis and M. Fowler. “Microservices,” <https://martinfowler.com/articles/microservices.html>, 2014.
- [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* 4, no. 14 (2000).
- [Lyons 62] R. E. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability,” *IBM Journal of Research and Development* 6, no. 2 (April 1962): 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, no 7 (July 2006): 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, hbs.edu/research/pdf/10-059.pdf.
- [Malan 00] Ruth Malan and Dana Bredemeyer. “Creating an Architectural Vision: Collecting Input,” July 25, 2000, bredemeyer.com/pdf_files/vision_input.pdf.
- [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): 34–43.
- [Martin 17] Robert C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Pearson, 2017.
- [Mavis 02] D. G. Mavis. “Soft Error Rate Mitigation Techniques for Modern Microcircuits,” in *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, NY: Rome Air Development Center Air Force Systems Command.

- [McConnell 07] Steve McConnell. “Technical Debt,” construx.com/10x_Software_Development/Technical_Debt/, 2007.
- [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*, National Academy of Engineering, 1991, pp. 213–216.
- [Mo 15] R. Mo, Y. Cai, R. Kazman, and L. Xiao. “Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells,” in *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)*, 2015.
- [Mo 16] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. “Decoupling Level: A New Metric for Architectural Maintenance Complexity,” *Proceedings of the International Conference on Software Engineering (ICSE) 2016*, Austin, TX, May 2016.
- [Mo 18] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele. “Experiences Applying Automated Architecture Analysis Tool Suites,” in *Proceedings of Automated Software Engineering (ASE) 2018*, 2018.
- [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*, 2nd ed. Wiley, 2006.
- [Muccini 03] H. Muccini, A. Bertolino, and P. Inverardi. “Using Software Architecture for Code Testing,” *IEEE Transactions on Software Engineering* 30, no. 3 (2003): 160–171.
- [Muccini 07] H. Muccini. “What Makes Software Architecture-Based Testing Distinguishable,” in *Proceedings of the 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* 27 (2001): 364–380.
- [NASEM 19] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. National Academies Press, 2019. <https://doi.org/10.17226/25196>.
- [Newman 15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.
- [Nielsen 08] Jakob Nielsen. “Usability ROI Declining, But Still Strong,” 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 18] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*, 2nd ed. Pragmatic Programmers, 2018.
- [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, acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf.
- [Oki 88] Brian Oki and Barbara Liskov. “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems,” *PODC ‘88: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, January 1988, pp. 8–17, <https://doi.org/10.1145/62546.62549>.
- [Palmer 02] Stephen Palmer and John Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.
- [Pang 16] C. Pang, A. Hindle, B. Adams, and A. Hassan. “What Do Programmers Know about Software Energy Consumption?,” *IEEE Software* 33, no. 3 (2016): 83–89.
- [Paradis 21] C. Paradis, R. Kazman, and D. Tamburri. “Architectural Tactics for Energy Efficiency: Review of the Literature and Research Roadmap,” *Proceedings of the Hawaii International Conference on System Sciences (HICSS)* 54 (2021).
- [Parnas 72] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules,” *Communications of the ACM* 15, no. 12 (December 1972).
- [Parnas 74] D. Parnas. “On a ‘Buzzword’: Hierarchical Structure,” in *Proceedings of 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): 1–9.
- [Parnas 79] D. Parnas. “Designing Software for Ease of Extension and Contraction,” *IEEE Transactions on Software Engineering*, SE-5, 2 (1979): 128–137.
- [Parnas 95] David Parnas and Jan Madey. “Functional Documents for Computer Systems,” 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 Software Engineering Notes* 17, no. 4 (October 1992): 40–52.
- [Pettichord 02] B. Pettichord. “Design for Testability,” Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002.
- [Procaccianti 14] G. Procaccianti, P. Lago, and G. Lewis. “A Catalogue of Green Architectural Tactics for the Cloud,” in *IEEE 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2014, pp. 29–36.
- [Powel Douglass 99] B. Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
- [Raiffa 00] H. Raiffa & R. Schlaifer. *Applied Statistical Decision Theory*. Wiley, 2000.
- [SAE 96] SAE International, “ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment,” December 1, 1996, sae.org/standards/content/arp4761/.
- [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* 81, no. 5 (May 2008): 727–746.
- [Sato 14] D. Sato. “Canary Deployment,” <https://martinfowler.com/bliki/CanaryRelease.html>, 2014.
- [Schaarschmidt 20] M. Schaarschmidt, M. Uelschen, E. Pulvermuellerm, and C. Westerkamp. “Framework of Software Design Patterns for Energy-Aware Embedded Systems,” *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, 2020.
- [Schmerl 06] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. “Discovering Architectures from Running Systems,” *IEEE Transactions on Software Engineering* 32, no. 7 (July 2006): 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 13] Robert Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2013.
- [SEI 12] Software Engineering Institute. “A Framework for Software Product Line Practice, Version 5.0,” 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* 20, no. 1 (January 1995): 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, no 4 (2010): 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 (2010): 6.
- [Stonebraker 11] M. Stonebraker. "Stonebraker on NoSQL and Enterprises," *Communications of the ACM* 54, no. 8 (2011): 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.
- [USDOD 12] U.S. Department of Defense, "Standard Practice: System Safety, MIL-STD-882E," May 11, 2012, dau.edu/cop/armyesoh/DAU%20Sponsored%20Documents/MIL-STD-882E.pdf.
- [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,” 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,” 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,” 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, no. 3 (May 1995): 17–28.
- [Von Neumann 56] J. Von Neumann. “Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components,” in *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, 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, 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*, 2nd ed. Addison-Wesley, 2011.
- [Wozniak 07] J. Wozniak, V. Baggiolini, D. Garcia Quintas, and J. Wenninger. “Software Interlocks System,” *Proceedings of ICALEPCS07*, <http://ics-web4.sns.ornl.gov/icalepcs07/WPPB03/WPPB03.PDF>.
- [Wu 04] W. Wu and T. Kelly, “Safety Tactics for Software Architecture Design,” *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC)*, 2004.
- [Wu 06] W. Wu and T. Kelly. “Deriving Safety Requirements as Part of System Architecture Definition,” in *Proceedings of 24th International System Safety Conference*. Albuquerque, NM: System Safety Society, August 2006.
- [Xiao 14] L. Xiao, Y. Cai, and R. Kazman. “Titan: A Toolset That Connects Software Architecture with Quality Analysis,” *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.
- [Xiao 16] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. “Identifying and Quantifying Architectural Debts,” *Proceedings of the International Conference on Software Engineering (ICSE) 2016*, 2016.
- [Yacoub 02] S. Yacoub and H. Ammar. “A Methodology for Architecture-Level Reliability Risk Analysis,” *IEEE Transactions on Software Engineering* 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.

This page intentionally left blank

About the Authors

Len Bass is an award-winning author who has lectured widely around the world. His books on software architecture are standards. In addition to his books on software architecture, Len has also written books on User Interface Software and DevOps. Len has over 50 years' experience in software development, 25 of those at the Software Engineering Institute of Carnegie Mellon. He also worked for three years at NICTA in Australia and is currently an adjunct faculty member at Carnegie Mellon University, where he teaches DevOps.

Dr. 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, he was a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute, where for 17 years he worked leading or co-leading projects in software product line engineering and software architecture design, documentation, and analysis. Prior to the SEI, he was a computer scientist with the U.S. Naval Research Laboratory in Washington, DC, where his work involved applying advanced software engineering principles to real-time embedded systems.

In addition to this book, Clements is the co-author of two other practitioner-oriented books about software architecture: *Documenting Software Architectures: Views and Beyond* and *Evaluating Software Architectures: Methods and Case Studies*. He also co-wrote *Software Product Lines: Practices and Patterns* and was co-author and editor of *Constructing Superior Software*. In addition, Clements has authored about a hundred papers in software engineering, reflecting his long-standing interest in the design and specification of challenging software systems.

Rick Kazman is a Professor at the University of Hawaii and a Visiting Researcher at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman has been involved in the creation of several highly influential methods and tools for architecture analysis, including the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method), and the Dali and Titan tools. In addition to this book, he is the author of over 200 publications and is co-author of three patents and eight books, including *Technical Debt: How to Find It and Fix It*, *Designing Software Architectures: A Practical Approach*, *Evaluating Software Architectures: Methods and Case*

Studies, and *Ultra-Large-Scale Systems: The Software Challenge of the Future*. His research has been cited over 25,000 times, according to Google Scholar. He is currently the chair of the IEEE TAC (Technical Activities Committee), Associate Editor for *IEEE Transactions on Software Engineering*, and a member of the ICSE Steering Committee.

Index

- A/B testing, 86
- Abort tactic, 159
- Abstract common services, 108
- Abstract data sources for testability, 189
- Abstraction, architecture as, 3
- ACID (atomic, consistent, isolated, and durable) properties, 61
- Acronym lists in documentation, 346
- Active redundancy, 66
- Activity diagrams for traces, 342–343
- Actors
 - attack, 174
 - elements, 217
- Actuators
 - mobile systems, 263, 267–268
 - safety concerns, 151–152
- Adapt tactic for integrability, 108–109, 111
- ADD method. *See* Attribute-Driven Design (ADD) method
- ADLs (architecture description languages), 331
- Aggregation for usability, 201
- Agile development, 370–373
- Agile Manifesto, 371–372
- Air France flight 447, 152
- Allocated-to relation
 - allocation views, 337
 - deployment structure, 15
- Allocation structures, 10, 15–16
- Allocation views
 - documentation, 348–350
 - overview, 337–338
- Allowed-to-use relationship, 128–129
- Alternative requests in long tail latency, 252
- Amazon service-level agreements, 53
- Analysis
 - ADD method, 295, 304–305
 - ATAM, 318–319, 321
 - automated, 363–364
- Analysts
 - documentation, 350
 - software interface documentation for, 229
- Analytic redundancy tactic
 - availability, 58
 - safety, 159
- Apache Camel project, 356–359
- Apache Cassandra database, 360–361
- Applications for quantum computing, 396–397
- Approaches
 - ATAM, 317–319, 321
 - CIA, 169
 - Lightweight Architecture Evaluation, 325
- Architects
 - communication with, 29
 - competence, 379–385
 - duties, 379–383
 - evaluation by, 311
 - knowledge, 384–385
 - mentoring, 387–388
 - mobile system concerns, 264–273
 - role. *See* Role of architects
 - skills, 383–384
- Architectural debt
 - automation, 363–364
 - determining, 356–358
 - example, 362–363
 - hotspots, 358–362
 - introduction, 355–356
 - quantifying, 363
 - summary, 364
- Architectural structures, 7–10
 - allocation, 15–16
 - C&C, 14–16
 - limiting, 18
 - module, 10–14
 - relating to each other, 15–18
 - selecting, 18
 - table of, 17
 - views, 5–6
- Architecturally significant requirements (ASRs)
 - ADD method, 289–290
 - from business goals, 282–284
 - change, 286

- introduction, 277–278
- from requirements documents, 278–279
- stakeholder interviews, 279–282
- summary, 286–287
- utility trees for, 284–286
- Architecture
 - changes, 27
 - cloud. *See* Cloud and distributed computing
 - competence. *See* Competence
 - debt. *See* Architectural debt
 - design. *See* Design and design strategy
 - documentation. *See* Documentation
 - evaluating. *See* Evaluating architecture
 - integrability, 102–103
 - modifiability. *See* Modifiability
 - patterns. *See* Patterns
 - performance. *See* Performance
 - QAW drivers, 281
 - QAW plan presentation, 280
 - quality attributes. *See* Quality attributes
 - requirements. *See* Architecturally significant requirements (ASRs); Requirements
 - security. *See* Security
 - structures. *See* Architectural structures
 - tactics. *See* Tactics
 - testability. *See* Testability
 - usability. *See* Usability
- Architecture description languages (ADLs), 331
- Architecture Tradeoff Analysis Method (ATAM), 313
 - approaches, 317–319, 321
 - example exercise, 321–324
 - outputs, 314–315
 - participants, 313–314
 - phases, 315–316
 - presentation, 316–317
 - results, 321
 - scenarios, 318
 - steps, 316–321
- Ariane 5 explosion, 151
- Artifacts
 - ADD method, 291
 - availability, 53
 - continuous deployment, 74
 - deployability, 76
 - energy efficiency, 91
 - in evaluation, 312
 - integrability, 104
 - modifiability, 120–121
 - performance, 136
 - quality attributes expressions, 43–44
 - safety, 154
 - security, 171
 - testability, 186
 - usability, 198
- Aspects for testability, 190
- ASRs. *See* Architecturally significant requirements (ASRs)
- Assertions for system state, 190
- Assurance levels in design, 164
- Asynchronous electronic communication, 375
- ATAM. *See* Architecture Tradeoff Analysis Method (ATAM)
- Atomic, consistent, isolated, and durable (ACID) properties, 61
- Attachment relation for C&C structures, 14–16
- Attachments in C&C views, 335
- Attribute-Driven Design (ADD) method
 - analysis, 295, 304–305
 - design concepts, 295–298
 - design decisions, 294
 - documentation, 301–303
 - drivers, 292–294
 - element choice, 293–294
 - element instantiation, 299–300
 - inputs, 292
 - overview, 289–291
 - prototypes, 297–298
 - responsibilities, 299–300
 - steps, 292–295
 - structures, 298–301
 - summary, 306
 - views, 294, 301–302
- Attributes. *See* Quality attributes
- Audiences for documentation, 330–331
- Audits, 176
- Authenticate actors tactic, 174
- Authorize actors tactic, 174
- Automation, 363–364
- Autoscaling in distributed computing, 258–261
- Availability
 - CIA approach, 169
 - cloud, 253–261
 - detect faults tactic, 56–59

- general scenario, 53–55
 - introduction, 51–52
 - patterns, 66–69
 - prevent faults tactic, 61–62
 - questionnaires, 62–65
 - recover from faults tactics, 59–61
 - tactics overview, 55–56
- Availability of resources tactic, 139
- Availability quality attribute, 285
- Availability zones, 248
- Backlogs in ADD method, 304
- Bandwidth in mobile systems, 267
- Bare-metal hypervisors, 235
- Barrier tactic, 159–160, 162
- Battery management systems (BMSs), 264
- BDUF (Big Design Up Front), 370–371
- Behavior
 - documenting, 340–345
 - in software architecture, 4
- Behavioral semantic distance in architecture
 - integrability, 103
- Bell, Alexander Graham, 263–264
- Best practices in design concepts, 296
- Big Design Up Front (BDUF), 370–371
- Binding
 - dynamic discovery services, 114
 - integrability, 109
 - modifiability, 122, 124–125
- Blocked time performance effects, 138–139
- Blue/green deployment pattern, 83
- BMSs (battery management systems), 264
- Bound execution times tactic, 141
- Bound queue sizes tactic, 142
- Box-and-line drawings in C&C views, 336
- Brainstorming
 - ATAM, 320
 - Lightweight Architecture Evaluation, 325
 - QAW, 281
 - scenarios, 281, 320
- Bridges pattern, 112
- Bugs, 355, 356, 362
- Buildability architecture category, 208
- Business goals
 - ASRs from, 282–284
 - ATAM, 314, 316–317
 - categorization, 283–284
 - evaluation process, 312
 - views for, 332
- Business/mission presentation in QAW, 280
- Business support, architect duties for, 382
- C&C structures. *See* Component-and-connector (C&C) patterns and structures
- Caching
 - performance, 142
 - REST, 224
- Camel project, 356–359
- Canary testing pattern, 85
- Cancel command, 200
- Capturing ASRs in utility trees, 284–286
- Car stereo systems, 344
- Cassandra database, 360–361
- Categorization of business goals, 283–284
- Central processor unit (CPU) in virtualization, 234
- Change
 - ASRs, 286
 - modifiability. *See* Modifiability
 - reasoning and managing, 27
- Change credential settings tactic, 175
- Chaos Monkey, 184–185
- Chaucer, Geoffrey, 379
- Chimero, Frank, 197
- CIA (confidentiality, integrity, and availability)
 - approach, 169
- Circuit breaker tactic, 67–68
- Classes
 - energy efficiency, 93–94
 - patches, 60
 - structure, 13
 - testability, 191
- Client/server constraints in REST, 224
- Client-server pattern, 126–127
- Cliques, 361–362
- Cloud and distributed computing
 - autoscaling, 258–261
 - basics, 248–250
 - data coordination, 258
 - failures, 251–253
 - introduction, 247
 - load balancers, 253–256
 - long tail latency, 252–253
 - mobile systems, 270

- performance, 253–261
- state management, 256–257
- summary, 261
- time coordination, 257
- timeouts, 251–252
- CNOT operations for qubits, 393
- Co-locate communicating resources tactic, 140–141
- Code, mapping to, 334
- Code on demand in REST, 225
- Cohesion
 - in modifiability, 122–123
 - in testability, 191
- Cold spare tactic, 66
- Combining views, 339–340
- Commission issues in safety, 153
- Common services in integrability, 108
- Communication
 - architect role in, 368
 - architect skills, 383
 - distributed development, 375
 - documentation for, 330
 - stakeholder, 28–30
- Communication diagrams for traces, 342
- Communication path restrictions, 107
- Communications views, 338
- Comparison tactic for safety, 158
- Compatibility
 - C&C views, 335
 - quality attributes, 211
- Competence
 - architects, 379–385
 - introduction, 379
 - mentoring, 387–388
 - program state sets, 62
 - software architecture organizations, 386–387
 - summary, 388
- Complex numbers in quantum computing, 392
- Complexity
 - quality attributes, 45–46
 - in testability, 190–191
- Component-and-connector (C&C) patterns and structures, 7–8
 - incremental architecture, 369
 - types, 14–16
 - views, combining, 339–340
 - views, documentation, 348–350
 - views, notations, 336–339
 - views, overview, 335–337
- Components, 4
 - independently developed, 34–35
 - replacing for testability, 190
- Comprehensive models for behavior
 - documentation, 341
- Comprehensive notations for state machine diagrams, 343–344
- Computer science knowledge of architects, 384–385
- Conceptual integrity of architecture, 208
- Concrete quality attribute scenarios, 43–44
- Concurrency
 - C&C views, 14, 336
 - handling, 135
 - resource management, 141
- Condition monitoring tactic
 - availability, 57
 - safety, 158
- Confidentiality, integrity, and availability (CIA)
 - approach, 169
- Configurability quality attribute, 285
- Configuring behavior for integrability, 109
- Conformity Monkey, 184
- Connectivity in mobile systems, 263, 266–267
- Connectors in C&C views, 335–337
- Consistency
 - mobile system data, 272
 - software interface design, 222
- Consolidation in QAW, 281
- Constraints
 - allocation views, 337
 - C&C views, 336
 - on implementation, 31–32
 - modular views, 333
- Contacts in distributed development, 375
- Containers
 - autoscaling, 260–261
 - virtual machines, 239–242
- Containment tactics, 158–159, 161–162
- Contention for resources tactic, 138–139
- Context diagrams, 345–346
- Contextual factors in evaluation, 312–313
- Continuous deployment, 72–75
- Control information in documentation, 346
- Control resource demand tactic, 139–141, 145

- Control tactics for testability, 188–190, 192
- Controllable deployments, 76
- Converting data for mobile system sensors, 268
- Conway, Damian, 329
- Conway's law, 37
- Coordinate tactic in integrability, 109–110, 112
- Copying qubits, 394
- Costs
 - architect role in, 368
 - of change, 118
 - distributed development, 374
 - estimates, 33–34
 - independently developed elements for, 35
 - mobile systems, 270
- Coupling
 - exchanged data representation, 226
 - in modifiability, 122–126
 - in testability, 190–191
- Cousins, Norman, 101
- CPU (central processor unit) in virtualization, 234
- Criticality in mobile systems, 270
- Crossing anti-patterns, 360
- CRUD operations in REST, 225
- Customers, communication with, 28
- Customization of user interface, 201
- Cybersecurity, quantum computing for, 396–397
- Cycle time in continuous deployment, 73–74
- Cyclic dependency, 360, 362

- DAL (Design Assurance Level), 164
- Darwin, Charles, 117
- Data coordination in distributed computing, 258
- Data model category, 13–14
- Data replication, 142
- Data semantic distance in architecture
 - integrability, 103
- de Saint-Exupéry, Antoine, 289
- Deadline monotonic prioritization strategy, 143
- Debt. *See* Architectural debt
- Decision makers on ATAM teams, 313
- Decisions
 - documenting, 347
 - mapping to quality requirements, 315
 - quality design, 48–49
- Decomposition
 - module, 10, 16
 - views, 16, 18, 340
- Defer binding tactic, 124–126
- Degradation tactic
 - availability, 60
 - safety, 159
- Demand reduction for energy efficiency, 95, 97
- Demilitarized zones (DMZs), 174
- Denial-of-service attacks, 51
- Dependencies
 - anti-patterns, 360
 - architectural debt, 356
 - architecture integrability, 102
 - on computations, 139
 - deployment, 79
 - limiting, 106–107, 111
 - modifiability, 119, 124
- Dependency injection pattern, 193
- Depends-on relation for modules, 333
- Deployability, 71
 - continuous deployment, 72–75
 - general scenarios, 76–77
 - overview, 75–76
 - patterns, 81–86
 - questionnaires, 80–81
 - tactics, 78–80
- Deployment pipelines, 72, 79–80
- Deployment structure, 15
- Deployment views
 - combining, 340
 - purpose, 332
- Deprecation of software interfaces, 220
- Design and design strategy, 289
 - ADD. *See* Attribute-Driven Design (ADD)
 - method
 - assurance levels, 164
 - early decisions, 31
 - quality attributes, 214
 - software interfaces, 222–228
- Design Assurance Level (DAL), 164
- Design structure matrices (DSMs), 356–358
- Designers, documentation for, 349
- Detect attacks tactics, 172–174, 177
- Detect faults tactic, 56–59, 63
- Detect intrusion tactic, 172
- Detect message deliveries anomalies tactic, 174
- Detect service denial tactic, 172
- Developers, documentation for, 229, 348
- Development, incremental, 33

- Development distributability attribute, 208–209
- Development environments, 72
- Deviation, failure from, 51
- Devices in mobile systems, 272
- DevOps, 74–75
- Discovery
 - energy efficiency, 94
 - integrability, 108–109
- Disk storage in virtualization, 234
- Displaying information in mobile systems, 270–271
- Distances
 - architecture integrability, 102–103
 - mobile system connectivity, 266
- Distributed computing. *See* Cloud and distributed computing
- Distributed development, 373–375
- DMZs (demilitarized zones), 174
- DO-178C document, 164
- Doctor Monkey, 185
- Documentation
 - ADD decisions, 294
 - ADD method, 301–303
 - architect duties, 381
 - behavior, 340–345
 - contents, 345–346
 - distributed development, 375
 - introduction, 329
 - notations, 331–332
 - practical considerations, 350–353
 - rationale, 346–347
 - software interfaces, 228–229
 - stakeholders, 347–350
 - summary, 353
 - traceability, 352–353
 - uses and audiences for, 330–331
 - views. *See* Views
- Domain knowledge of architects, 385
- Don't repeat yourself principle, 222
- Drivers
 - ADD method, 292–294
 - QAW, 281
- Duties, 379–383
- Dynamic allocation views, 338
- Dynamic classification in energy efficiency, 94
- Dynamic discovery pattern, 114
- Dynamic environments, documenting, 352
- Dynamic priority scheduling strategies, 143–144
- E-scribes, 314
- Earliest-deadline-first scheduling strategy, 143
- Early design decisions, 31
- EC2 cloud service, 53, 184
- ECUs (electronic control units) in mobile systems, 269–270
- Edge cases in mobile systems, 271
- Education, documentation as, 330
- Efficiency, energy. *See* Energy efficiency
- Efficient deployments, 76
- Einstein, Albert, 385
- Electric power for cloud centers, 248
- Electronic control units (ECUs) in mobile systems, 269–270
- Elements
 - ADD method, 293–294, 299–300
 - allocation views, 337
 - C&C views, 336
 - defined, 4
 - modular views, 333
 - software interfaces, 217–218
- Emergent approach, 370–371
- Emulators for virtual machines, 236
- Enabling quality attributes, 26
- Encapsulation in integrability, 106
- Encrypt data tactic, 175
- Encryption in quantum computing, 394–395
- End users, documentation for, 349–350
- Energy efficiency, 89–90
 - general scenario, 90–91
 - patterns, 97–98
 - questionnaire, 95–97
 - tactics, 92–95
- Energy for mobile systems, 263–265
- Entanglement in quantum computing, 393–394
- Enterprise architecture vs. system architecture, 4–5
- Environment
 - allocation views, 337–338
 - availability, 54
 - continuous deployment, 72
 - deployability, 76
 - energy efficiency, 91
 - integrability, 104
 - modifiability, 120–121
 - performance, 136
 - quality attributes expressions, 43–44
 - safety, 154

- security, 171
- software interfaces, 217
- testability, 186
- usability, 198
- virtualization effects, 73
- Environmental concerns with mobile systems, 269
- Errors
 - description, 51
 - error-handling views, 339
 - software interface handling of, 227–228
 - in usability, 197
- Escalating restart tactic, 60–61
- Estimates, cost and schedule, 33–34
- Evaluating architecture
 - architect duties, 311, 381
 - ATAM. *See* Architecture Tradeoff Analysis Method (ATAM)
 - contextual factors, 312–313
 - key activities, 310–311
 - Lightweight Architecture Evaluation, 324–325
 - outsider analysis, 312
 - peer review, 311–312
 - questionnaires, 326
 - risk reduction, 309–310
 - summary, 326–327
- Events
 - performance, 133
 - software interfaces, 219–220
- Evolution of software interfaces, 220–221
- Evolutionary dependencies in architectural debt, 356
- Exception detection tactic, 58–59
- Exception handling tactic, 59
- Exception prevention tactic, 62
- Exception views, 339
- Exchanged data in software interfaces, 225–227
- Executable assertions for system state, 190
- Experience in design, 296
- Expressiveness concern for exchanged data representation, 225
- Extendability in mobile systems, 273
- EXtensible Markup Language (XML), 226
- Extensions for software interfaces, 220
- External interfaces, 300–301
- Externalizing change, 125
- Failures
 - availability. *See* Availability
 - cloud, 251–253
 - description, 51
- Fault tree analysis (FTA), 153
- Faults
 - description, 51–52
 - detection, 55
 - prevention, 61–62
 - recovery from, 59–61
- Feature toggle in deployment, 80
- FIFO (first-in/first-out) queues, 143
- Firewall tactic, 159
- First-in/first-out (FIFO) queues, 143
- First principles from tactics, 47
- Fixed-priority scheduling, 143
- Flexibility
 - defer binding tactic, 124
 - independently developed elements for, 35
- Follow-up phase in ATAM, 316
- Forensics, documentation for, 330
- Formal documentation notations, 331
- Forward error recovery pattern, 68
- Foster, William A., 39
- FTA (fault tree analysis), 153
- Fuller, R. Buckminster, 1
- Function patches, 59
- Function testing in mobile systems, 272
- Functional redundancy
 - availability, 58
 - containment, 159
- Functional requirements, 40–41
- Functional suitability of quality attributes, 211
- Functionality
 - C&C views, 336
 - description, 40
- Fusion of mobile system sensors, 268
- Future computing. *See* Quantum computing
- Gateway elements in software interfaces, 223
- Gehry, Frank, 367
- General Data Protection Regulation (GDPR)
 - cloud, 248
 - privacy concerns, 170
- Generalization structure, 13
- Get method for system state, 188
- Gibran, Kahlil, 169

- Glossaries in documentation, 346
- Goals. *See* Business goals
- Good architecture, 19–20
- Graceful degradation, 60
- Granular deployments, 75
- Granularity of gateway resources, 223
- Grover’s algorithm, 395

- HAD operations for qubits, 393
- Hardware in mobile systems, 271
- Harrow, Aram W., 396
- Hashes in quantum computing, 395
- Hassidim, Avinatan, 396
- Hawking, Stephen, 89
- Health checks for load balancers, 255–256
- Heartbeats for fault detection, 57, 318
- Hedged requests in long tail latency, 252
- HHL algorithm, 396
- Hiatus stage in ATAM, 320
- High availability. *See* Availability
- Highway systems, 144
- Hosted hypervisors, 235–236
- Hot spare tactic, 66
- Hotspots
 - architectural debt, 358–362
 - identifying, 362–363
- Hotz, Robert Lee, 217
- HTTP commands for REST, 225
- Hubs for mobile system sensors, 267
- Human body structure, 5–6
- Human resource management, architect role for, 368
- Hybrid clouds, 248
- Hypertext for documentation, 351
- Hypervisors for virtual machines, 235–237
- Hyrum’s law, 229

- Identify actors tactic, 174
- IEEE standards for mobile system connectivity, 266
- Ignore faulty behavior tactic, 60
- Images for virtual machines, 238, 260
- Implementation
 - constraints, 31–32
 - modules, 334
 - structure, 15
- Implicit coupling, 226

- In-service software upgrade (ISSU), 60
- Increase cohesion tactic, 125
- Increase competence set tactic, 62
- Increase efficiency tactic, 144
- Increase efficiency of resource usage tactic, 141
- Increase resources tactic, 141, 144
- Increase semantic coherence tactic, 122–123
- Incremental architecture, 369–370
- Incremental development, 33
- Inform actors tactic, 176
- Informal contacts in distributed development, 375
- Informal notations for documentation, 331
- Infrastructure support personnel, documentation for, 350
- Inheritance anti-pattern, 360
- Inherits-from relation, 13
- Inhibiting quality attributes, 26
- Inputs in ADD method, 292
- Instances in cloud, 253–261
- Integrability
 - architecture, 102–103
 - general scenario, 104–105
 - introduction, 101–102
 - patterns, 112–114
 - questionnaires, 110–112
 - tactics, 105–110
- Integration environments, 72
- Integration management, architect role in, 368
- Integrators, documentation for, 349
- Integrity in CIA approach, 169
- Intercepting filter pattern, 194
- Intercepting validator pattern, 179
- Interfaces
 - ADD method, 300–301
 - anti-patterns, 360
 - mismatch in deployability, 85
 - mobile system connectivity, 266
 - software. *See* Software interfaces
- Interlock tactic, 160
- Intermediaries in integrability, 107
- Intermediate states in failures, 51
- Intermittent mobile system connectivity, 267
- Internal interfaces, 301
- Internet Protocol (IP) addresses
 - cloud, 260
 - virtualization, 234

- Interoperability in exchanged data representation, 225
- Interpersonal skills, 383
- Interviewing stakeholders, 279–282
- Introduce concurrency tactic, 141
- Intrusion prevention system (IPS) pattern, 179–180
- Iowability, 212
- IP (Internet Protocol) addresses
 - cloud, 260
 - virtualization, 234
- Is-a relation, 333
- Is-a-submodule-of relation, 10
- Is-an-instance-of relation, 13
- Is-part-of relation, 333
- ISO 25010 standard, 40, 209–212
- ISSU (in-service software upgrade), 60
- Issue information in architectural debt, 356
- Iterations
 - ADD method, 295, 304
 - agile development, 370–371
- Janitor Monkey, 185
- Jarre, Jean-Michel, 51
- JavaScript Object Notation (JSON), 226–227
- Kanban boards, 304–305
- Kill abnormal tasks pattern, 97–98
- Knowledge
 - architects, 379–381, 384–385
 - design concepts, 296
- Labor availability and costs in distributed development, 374
- LAE (Lightweight Architecture Evaluation) method, 324–325
- LAMP stacks, 240
- Lamport, Leslie, 247, 258
- Latency in cloud, 252–253
- Latency Monkey, 184
- Lawrence Livermore National Laboratory, 45
- Layer structures, 11–12
- Layered views, 332
- Layers pattern, 128–129
- Leaders on ATAM teams, 314
- Learning issues in usability, 197
- Least-slack-first scheduling strategy, 143
- Levels, restart, 60–61
- Life cycle in mobile systems, 263, 270–273
- Lightweight Architecture Evaluation (LAE) method, 324–325
- Likelihood of change, 117
- Limit access tactic, 174
- Limit complexity tactic, 190–192
- Limit consequences tactic, 159, 162
- Limit dependencies tactic, 106–107, 111
- Limit event response tactic, 140
- Limit exposure tactic, 175
- Limit nondeterminism tactic, 191
- Limit structural complexity tactic, 190–191
- Lloyd, Seth, 396
- Load balancer pattern for performance, 147
- Load balancers
 - description, 141
 - distributed computing, 253–256
- Local changes, 27
- Localize state storage for testability, 189
- Location factors in mobile systems, 270
- Location independence in modifiability, 119
- Locks in data coordination, 258
- Logical threads in concurrency, 14
- Logs for mobile systems, 273
- Long tail latency in cloud, 252–253
- Longfellow, Henry Wadsworth, 25
- Loss of mobile system power, 265
- Macros for testability, 190
- Maintain multiple copies tactic, 144
- Maintain multiple copies of computations tactic, 141
- Maintain multiple copies of data tactic, 142
- Maintain system model tactic, 201
- Maintain task model tactic, 201
- Maintain user model tactic, 201
- Maintainability quality attribute, 211, 285
- Maintainers, documentation for, 229, 349
- Manage deployed system tactic, 79–80
- Manage event rate tactic, 144
- Manage resources tactic, 141–142, 145–146
- Manage sampling rate tactic
 - performance, 139–140
 - quality attributes, 47
- Manage service interactions tactic, 79
- Manage work requests tactic, 139–140

- Management information in modules, 334
- Managers, communication with, 29
- Map function, 148–149
- Map-reduce pattern, 148–149
- Mapping
 - to requirements, 315
 - to source code units, 334
 - between views, 345
- Market knowledge in distributed development, 374
- Marketability category for quality, 208
- “Mars Probe Lost Due to Simple Math Error,” 217
- Masking tactic, 159
- Matrix inversion in quantum computing, 396
- MCAS software, 152–153
- Mean time between failures (MTBF), 52
- Mean time to repair (MTTR), 52
- Mediators pattern, 113
- Meetings in distributed development, 375
- Memento pattern, 205
- Memory
 - quantum computing, 395–396
 - virtualization, 234
- Mentoring and architects, 387–388
- Metering in energy efficiency, 93
- Microkernel pattern, 127–128
- Microservice architecture pattern, 81–82
- Migrates-to relation, 15
- Missile launch incident, 152
- Mixed initiative in usability, 197
- Mobile systems
 - energy usage, 263–265
 - introduction, 263–264
 - life cycle, 270–273
 - network connectivity, 266–267
 - resources, 268–270
 - sensors and actuators, 267–268
 - summary, 273–274
- Model-View-Controller (MVC) pattern, 203–204
- Modeling tools, documentation for, 351
- Models
 - quality attributes, 213–214
 - transferable and reusable, 34
- Modifiability
 - general scenario, 120–121
 - introduction, 117–119
 - managing, 27
 - mobile system connectivity, 266
 - patterns, 126–130
 - questionnaires, 125–126
 - tactics, 121–126
 - in usability, 201
- Modularity violations, 360
- Modules and module patterns, 7, 9
 - coupling, 122
 - description, 2–3
 - documentation, 348–350
 - incremental architecture, 369
 - types, 10–14
 - views, 333–334
- Monitor-actuator pattern, 163
- Monitor tactic, 56–57
- Monitoring mobile system power, 264–265
- MTBF (mean time between failures), 52
- MTTR (mean time to repair), 52
- Multiple instances in cloud, 253–261
- Multiple software interfaces, 218
- Multitasking, 135
- MVC (Model-View-Controller) pattern, 203–204
- Names for modules, 334
- Nash, Ogden, 355
- National Institute of Standards and Technology (NIST)
 - PII, 170
 - quantum computing, 394
- Near Field Communication (NFC), 266
- Netflix
 - map-reduce, 148
 - Simian Army, 184–185
- Network connectivity
 - mobile systems, 263, 266–267
 - virtualization, 234
- Network Time Protocol (NTP) for time coordination, 257
- Network transitions in mobile systems, 271
- Networked services, 35
- NFC (Near Field Communication), 266
- NIST (National Institute of Standards and Technology)
 - PII, 170
 - quantum computing, 394
- Nondeterminism in testability, 191

- Nonlocal changes, 27
- Nonrepudiation tactic, 176
- Nonrisks in ATAM, 314–315
- Nonstop forwarding tactic, 61
- NOT operations for qubits, 393
- Notations
 - C&C views, 336–339
 - documentation, 331–332
- Notifications for failures, 51
- NTP (Network Time Protocol) for time
 - coordination, 257
- Object-oriented systems in testability, 190
- Objects in sequence diagrams, 341
- Observability of failures, 52
- Observe system state tactics, 188–190, 192
- Observer pattern, 204
- Off-the-shelf components, 35
- Omissions as safety factor, 153
- Open system software, 35
- Operating systems with containers, 241–242
- Operations in software interfaces, 219–220
- Orchestrate tactic, 109–110
- Organizations, architecture influence on, 32
- Out of sequence events as safety factor, 153
- Outages. *See* Availability
- Outputs in ATAM, 314–315
- Outsider evaluation, 312
- Overlay views, 339
- Package cycles anti-pattern, 360
- Package dependencies in deployment, 79
- PALM method, 283
- Parameter fence tactic, 58
- Parameter typing tactic, 58
- Parity, environment, 73
- Partial replacement of services patterns, 85–86
- Partial system deployment in mobile systems, 273
- Partnership and preparation phase in ATAM, 315
- Passive redundancy, 66
- Patches, 59–60
- Patterns
 - ADD method, 299
 - architectural, 18
 - availability, 66–69
 - C&C. *See* Component-and-connector (C&C)
 - patterns and structures
 - deployability, 81–86
 - documenting, 345
 - energy efficiency, 97–98
 - integrability, 112–114
 - modifiability, 126–130
 - partial replacement of services, 85–86
 - performance, 146–149
 - quality attributes tactics, 46–47
 - safety, 163–164
 - security, 179–180
 - testability, 192–194
 - usability, 203–205
- Pause/resume command, 201
- Peer review, 311–312
- People management, architect duties for, 382
- Performance
 - C&C views, 335
 - cloud, 253–261
 - control resource demand tactics, 139–141
 - efficiency, 211
 - exchanged data representation, 225
 - general scenario, 134–137
 - introduction, 133–134
 - manage resources tactics, 141–142
 - patterns, 146–149
 - quality attribute, 47, 211, 285
 - questionnaires, 145–146
 - tactics overview, 137–139
 - views, 339
 - virtual machines, 237
- Periodic cleaning tactic, 141
- Personally identifiable information (PII), 170
- Personnel-related competence, 386
- Petrov, Stanislav Yevgrafovich, 152
- Phases
 - ATAM, 315–316
 - quantum computing, 392–393
- PII (personally identifiable information), 170
- Ping/echo tactic, 57
- Pipelines, deployment, 72, 79–80
- Platforms, architect knowledge about, 385
- Plug-in pattern, 127–128
- PMBOK (Project Management Body of Knowledge), 368

- Pods in virtualization, 242–243
- Pointers, smart, 62
- Policies, scheduling, 143–144
- Portability
 - containers, 242
 - modifiability, 119
 - quality attributes, 42, 211
- Power for mobile systems, 264–265
- Power monitor pattern, 98
- Power station catastrophe, 151
- Predicting system qualities, 28
- Predictive model tactic
 - availability, 62
 - safety, 157
- Preemptible processes, 143
- Preparation-and-repair tactic, 59–60
- Preprocessor macros, 190
- Presentation
 - ATAM, 314–317
 - Lightweight Architecture Evaluation, 325
 - QAW, 280
- Prevent faults
 - questionnaire, 65
 - tactics, 61–62
- Principle of least surprise, 222
- Principles, design fragments from, 47
- Prioritize events tactic, 140, 144
- Prioritizing
 - ATAM scenarios, 320
 - Lightweight Architecture Evaluation
 - scenarios, 325
 - QAW, 281
 - schedules, 143–144
- Privacy issues, 170
- Private clouds, 248
- Probabilities in quantum computing, 392–393
- Process pairs pattern, 68
- Process recommendations, 19
- Process-related competence, 387
- Processing time in performance, 138
- Procurement management, architect role in, 368
- Production environments, 72
- Programming knowledge of architects, 384
- Project management, architect duties for, 382
- Project Management Body of Knowledge (PMBOK), 368
- Project managers
 - documentation for, 347–348
 - working with, 367–368
- Project roles. *See* Role of architects
- Properties
 - ADD method, 300
 - software interfaces, 219–220
- Protocol Buffer technology, 227
- Protocols for mobile system connectivity, 266
- Prototypes in ADD method, 297–298
- Public clouds, 248
- Publicly available apps, 35
- Publish-subscribe connectors, 335
- Publish-subscribe pattern, 129–130
- Publisher role, 335
- QAW (Quality Attribute Workshop), 280–281
- QPU, 392–393
- QRAM (quantum random access memory), 395–396
- Quality Attribute Workshop (QAW), 280–281
- Quality attributes, 207
 - architecture, 208
 - ASRs, 280–281
 - ATAM, 317–318
 - capture scenarios, 213
 - considerations, 41–42
 - design approaches, 214
 - development distributability, 208–209
 - inhibiting and enabling, 26
 - introduction, 39
 - Lightweight Architecture Evaluation, 325
 - models, 213–214
 - quality design decisions, 48–49
 - requirements, 42–45
 - standard lists, 209–212
 - summary, 49
 - system, 209
 - tactics, 45–46
 - X-ability, 212–214
- Quality design decisions, 48–49
- Quality management, architect role for, 368
- Quality of products as business goal, 283
- Quality requirements, mapping decisions to, 315
- Quality views, 338–339
- Quantifying architectural debt, 363

- Quantum computing
 - algorithms, 395–396
 - applications, 396–397
 - encryption, 394–395
 - future of, 397
 - introduction, 391–392
 - matrix inversion, 396
 - qubits, 392–393
 - teleportation, 394
- Quantum random access memory (QRAM), 395–396
- Qubits
 - description, 392–393
 - teleportation, 394
- Questioners on ATAM teams, 314
- Questionnaires
 - architecture evaluation, 326
 - availability, 62–65
 - deployability, 80–81
 - energy efficiency, 95–97
 - integrability, 110–112
 - modifiability, 125–126
 - performance, 145–146
 - quality attributes, 48–49
 - safety, 160–162
 - security, 176–178
 - testability, 192
 - usability, 202–203
- Bound queue sizes tactic, 142
- Race conditions, 135
- Rate monotonic prioritization strategy, 143
- Rationale
 - documentation, 346–347
 - views, 346
- Raw data with mobile system sensors, 268
- React to attacks tactics, 175–176, 178
- READ operations for qubits, 393
- Reconfiguration tactic, 60
- Record/playback method for system state, 189
- Recover from attacks tactics, 176, 178
- Recover from faults tactics, 59–61, 64–65
- Recovery tactic, 160, 162
- Redistribute responsibilities tactic, 122–123
- Reduce computational overhead tactic, 140, 144
- Reduce coupling tactic, 123–126
- Reduce function in performance, 148–149
- Reduce indirection tactic, 140
- Redundancy tactics
 - availability, 58–59, 66–67
 - safety, 158–159, 161–162
- Redundant sensors pattern, 163
- Reference architectures in ADD method, 299
- Refined scenarios in QAW, 281
- Refinement in ADD method, 293
- Regions in cloud, 248
- Reintroduction tactics, 60–61
- Rejuvenation tactic, 61
- Relations
 - ADD elements, 294, 300
 - allocation views, 337
 - architectural structures, 16–18
 - C&C views, 336
 - modular views, 333
- Release strategy, documenting, 351
- Reliability
 - C&C views, 335
 - independently developed elements for, 35
 - quality attributes, 211
 - quality views, 339
- Remote Procedure Call (RPC), 224
- Removal from service tactic, 61
- Repair tactic, 160
- Repeatability in continuous deployment, 74
- Replacement of services patterns, 82–85
- Replication tactic
 - availability, 58
 - safety, 159
- Report method for system state, 188
- Representation and structure of exchanged data, 225–227
- Representation of architecture, 3
- Representational State Transfer (REST)
 - protocol, 224–225
- Requirements
 - architect duties, 382
 - ASRs. *See* Architecturally significant requirements (ASRs)
 - functional, 40–41
 - mapping to, 315
 - quality attributes, 42–45
 - system availability, 53
- Reset method for system state, 188
- Resist attacks tactics, 174–175, 177–178

- Resource distance in architecture integrability, 103
- Resources
 - C&C views, 335
 - contention for, 138
 - integrability management of, 110
 - mobile systems, 263, 268–271
 - monitoring in energy efficiency, 93–96
 - in performance, 138
 - sandboxing, 189
 - software interfaces, 217, 219
 - virtualization, 234
- Response
 - availability, 54
 - deployability, 76
 - energy efficiency, 91
 - integrability, 104
 - modifiability, 120–121
 - performance, 136
 - quality attribute expressions, 43–44
 - safety, 154
 - security, 171
 - testability, 186
 - usability, 199
- Response measure
 - availability, 54
 - deployability, 77
 - energy efficiency, 91
 - integrability, 104
 - modifiability, 120–121
 - performance, 137
 - quality attribute expressions, 43–44
 - safety, 155
 - security, 171
 - testability, 187
 - usability, 199
- Responsibilities
 - ADD method, 300
 - modules, 334
- REST (Representational State Transfer) protocol, 224–225
- Restart tactic, 60–61
- Restrict dependencies tactic, 124
- Restrict login tactic, 175–176
- Restrictions on vocabulary, 35–36
- Results
 - ATAM, 321
 - evaluation, 312
 - Lightweight Architecture Evaluation, 325
- Retry tactic, 60
- Reusable models, 34
- Reviews, peer, 311–312
- Revision history
 - architectural debt, 356
 - modules, 334
- Revoke access tactic, 175
- Risk
 - architect role in managing, 368
 - ATAM, 314–315
 - evaluation process, 309–310
- Role of architects, 367
 - agile development, 370–373
 - distributed development, 373–375
 - incremental architecture, 369–370
 - project manager interaction, 367–368
 - summary, 376
- Rollback tactic
 - deployment, 79
 - fault recovery, 59
 - safety, 160
- Rolling upgrade deployment pattern, 83–84
- Round-robin scheduling strategy, 143
- Rounds in ADD method, 291
- RPC (Remote Procedure Call), 224
- Runtime engines in containers, 239
- Runtime extensibility in C&C views, 336
- Rutan, Burt, 183
- SAFe (Scaled Agile Framework), 373
- Safety
 - general scenario, 154–155
 - introduction, 151–153
 - mobile systems, 269, 272–273
 - patterns, 163–164
 - questionnaires, 160–162
 - tactics, 156–160
- Sampling rate tactic, 139–140
- Sandbox tactic, 189
- Sanity checking tactic
 - availability, 57
 - safety, 158
- Satisfaction in usability, 197
- Scalability in modifiability, 119
- Scale rollouts, 79

- Scaled Agile Framework (SAFe), 373
- Scaling in distributed computing, 258–261
- Scenario scribes, 314
- Scenarios
 - ATAM, 318–320
 - availability, 53–55
 - deployability, 76–77
 - energy efficiency, 90–91
 - integrability, 104–105
 - Lightweight Architecture Evaluation, 325
 - modifiability, 120–121
 - performance, 134–137
 - QAW, 281
 - quality attributes, 42–45, 213
 - safety, 154–155
 - security, 170–172
 - testability, 186–187
 - usability, 198–199
- Schedule resources tactic
 - performance, 142
 - quality attributes, 47
- Scheduled downtimes, 52
- Schedules
 - estimates, 33–34
 - policies, 143–144
 - of resources for energy efficiency, 94
- Scope
 - architect management role in, 368
 - software interfaces, 223
- Script deployment commands, 79
- Security
 - C&C views, 336
 - general scenario, 170–172
 - introduction, 169
 - mobile system connectivity, 267
 - patterns, 179–180
 - privacy issues, 170
 - quality attributes, 211
 - questionnaires, 176–178
 - tactics, 172–176
 - views, 338
- Security Monkey, 185
- Security quality attribute, 285
- Selection
 - design concepts, 296–297
 - tools and technology, 382
- Self-test tactic, 59
- Semantic importance strategy, 143
- Semantics, resource, 219
- Semiformal documentation notations, 331
- Sensitivity points in ATAM, 315
- Sensor fusion pattern, 97
- Sensors in mobile systems, 263, 267–268
- Separate entities tactic, 175
- Separated safety pattern, 163–164
- Separation of concerns
 - testability, 191
 - virtual machines, 238
- Sequence diagrams for traces, 341–342
- Sequence omission and commission as safety
 - factor, 153
- Serverless architecture in virtualization, 243–244
- Service impact of faults, 52
- Service-level agreements (SLAs)
 - Amazon, 53
 - availability in, 52–53
- Service mesh pattern, 146–147
- Service-oriented architecture (SOA) pattern,
 - 113–114
- Service structure, 14
- Set method for system state, 188
- 737 MAX aircraft, 152–153
- Shadow tactic, 60
- Shared resources in virtualization, 234
- Shushenskaya hydroelectric power station, 151
- Simian Army, 184–185
- Size
 - modules, 122
 - queue, 142
- Skeletal systems, 33
- Sketches in ADD method, 301–302
- Skills
 - architects, 379–381, 383–384
 - distributed development, 374
- SLAs (service-level agreements)
 - Amazon, 53
 - availability in, 52–53
- Small interfaces principle, 222
- Smart pointers, 62
- Smoothing data for mobile system sensors, 268
- SOA (service-oriented architecture) pattern,
 - 113–114
- Software architecture importance, 25–26
 - change management, 27

- constraints, 31–32
- cost and schedule estimates, 33–34
- design decisions, 31
- incremental development, 33
- independently developed elements, 34–35
- organizational structure, 32
- quality attributes, 26
- stakeholder communication, 28–30
- summary, 36–37
- system qualities prediction, 28
- training basis, 36
- transferable, reusable models, 34
- vocabulary restrictions, 35–36
- Software architecture overview, 1. *See also*
 - Architecture
 - as abstraction, 3
 - behavior in, 4
 - competence, 386–387
 - definitions, 2
 - good and bad, 19–20
 - patterns, 18
 - as set of software structures, 2–3
 - structures and views, 5–18
 - summary, 21
 - system architecture vs. enterprise, 4–5
- Software Engineering Body of Knowledge (SWEBOK), 278
- Software for mobile systems, 272
- Software interfaces
 - designing, 222–228
 - documentation, 228–229
 - error handling, 227–228
 - evolution, 220–221
 - introduction, 217–218
 - multiple, 218
 - operations, events, and properties, 219–220
 - representation and structure of exchanged data, 225–227
 - resources, 219
 - scope, 223
 - styles, 224–225
 - summary, 230
- Software rejuvenation tactic, 61
- Software upgrade tactic, 59–60
- Source
 - architectural debt, 356
 - deployability, 76
 - energy efficiency, 91
 - integrability, 104
 - modifiability, 120–121
 - performance, 136
 - safety, 154
 - security, 170
 - testability, 186
 - usability, 198
- Source code, mapping to, 334
- Spare tactic, 66
- Specialized interfaces tactic, 188–189
- Spikes in agile development, 370
- Split module tactic, 122
- Staging environments, 72
- Stakeholders
 - on ATAM teams, 313–314
 - communication among, 28–30, 330
 - documentation, 347–350
 - evaluation process, 312
 - incremental architecture, 369–370
 - interviewing, 279–282
- Standards in integrability, 107–108
- State, system, 188–190, 192
- State machine diagrams, 343–345
- State management in distributed computing, 256–257
- State resynchronization tactic, 60
- Stateless interactions in REST, 224
- Static allocation views, 338
- Static classification for energy efficiency, 93–94
- Static scheduling, 144
- Stein, Gertrude, 144
- Stimulus
 - availability, 53
 - deployability, 76
 - energy efficiency, 91
 - integrability, 104
 - modifiability, 120–121
 - performance, 136
 - quality attributes expressions, 42–44
 - safety, 154
 - security, 171
 - testability, 186
 - usability, 198

- Storage
 - for testability, 189
 - virtualization, 234
- Strategy pattern for testability, 193–194
- Stroustrup, Bjarne, 277
- Structural complexity in testability, 190–191
- Structures in ADD method, 298–301
- Stuxnet virus, 151
- Styles for software interfaces, 224–225
- Submodules, 334
- Subscriber role, 335
- Substitution tactic, 156–157
- Subsystems, 6
- Super-tactics, 47
- Superposition in quantum computing, 392
- Support system initiative tactic, 201–203
- SWEBOK (Software Engineering Body of Knowledge), 278
- Syntactic distance in architecture integrability, 102–103
- Syntax for resources, 219
- System analysis and construction, documentation for, 330
- System architecture vs. enterprise architecture, 4–5
- System availability requirements, 53
- System efficiency in usability, 197
- System exceptions tactic, 58
- System initiative in usability, 197
- System qualities, predicting, 28
- System quality attributes, 209
- System values as safety factor, 153
- Systems integrators and testers, software interface documentation for, 229
- Tactics
 - ADD method, 299–300
 - architecture evaluation, 326
 - availability, 55–65
 - deployability, 78–81
 - energy efficiency, 92–97
 - integrability, 105–112
 - modifiability, 121–125
 - performance, 137–146
 - quality attributes, 45–46, 48–49
 - safety, 156–162
 - security, 172–178
 - testability, 187–192
 - usability, 200–203
- Tailor interface tactic, 109
- Team building skills, 383
- Teams in ATAM, 313–314
- Technical debt. *See* Architecture debt
- Technology knowledge of architects, 385
- Technology-related competence, 387
- Teleportation in quantum computing, 394
- Temporal distance in architecture integrability, 103
- Temporal inconsistency in deployability, 85
- 10-18 Monkey, 185
- Test harnesses, 184
- Testability
 - general scenario, 186–187
 - introduction, 183–185
 - patterns, 192–194
 - questionnaires, 192
 - tactics, 187–191
- Testable requirements, 278
- Testers, documentation for, 349
- Tests and testing
 - continuous deployment, 72–73
 - mobile systems, 271–272
 - modules, 334
- Therac 25 radiation overdose, 151
- Therapeutic reboot tactic, 61
- Thermal limits in mobile systems, 269
- Threads
 - concurrency, 135
 - virtualization, 234
- Throttling mobile system power, 265
- Throttling pattern for performance, 148
- Throughput of systems, 137
- Tiered system architectures in REST, 225
- Time and time management
 - architect role, 368
 - performance, 133
- Time coordination in distributed computing, 257
- Time to market, independently developed elements for, 35
- Timeout tactic
 - availability, 58–59
 - safety, 157–158

- Timeouts in cloud, 251–252
- Timestamp tactic
 - availability, 57
 - safety, 158
- Timing as safety factor, 153
- TMR (triple modular redundancy), 67
- Traceability
 - continuous deployment, 74
 - documentation, 352–353
- Traces for behavior documentation, 341–342
- Tradeoffs in ATAM, 315
- Traffic systems, 144
- Training, architecture for, 36
- Transactions in availability, 61
- Transducers in mobile systems, 267
- Transferable models, 34
- Transforming existing systems, 381
- Transparency in exchanged data representation, 226
- Triple modular redundancy (TMR), 67
- Two-phase commits, 61
- Type 1 hypervisors, 235
- Type 2 hypervisors, 235

- UML. *See* Unified Modeling Language (UML)
- Unambiguous requirements, 278
- Undo command, 200–201
- Unified Modeling Language (UML)
 - activity diagrams, 342–343
 - C&C views, 336–337
 - communication diagrams, 342
 - sequence diagrams, 341–342
 - state machine diagrams, 343–345
- Uniform access principle, 222
- Uniform interface in REST, 224
- Unity of purpose in modules, 122
- Unsafe state avoidance tactic, 156–157, 161
- Unsafe state detection tactic, 157–158, 161
- Unstable interfaces anti-pattern, 360
- Updates for mobile systems, 272–273
- Usability
 - general scenario, 198–199
 - introduction, 197–198
 - patterns, 203–205
 - quality attributes, 211
 - questionnaires, 202–203
 - tactics, 200–202
- Usability quality attribute, 285
- Usage
 - allocation views, 337
 - C&C views, 336
 - modular views, 333
 - reducing in energy efficiency, 94
- Use an intermediary tactic, 47
- Use cases for traces, 341
- User initiative in usability, 197
- User interface customization, 201
- User needs in usability, 197
- Users, communication with, 28
- Uses
 - for documentation, 330–331
 - views for, 332
- Uses structure in decomposition, 10–12
- Utility trees
 - ASRs, 284–286
 - ATAM, 317–318, 320
 - Lightweight Architecture Evaluation, 325

- Validate input tactic, 175
- Variability guides for views, 346
- Variability in modifiability, 119
- Vector clocks for time coordination, 257
- Verify message integrity tactic, 174
- Versioning in software interfaces, 220
- Views, 332–333
 - ADD method, 294, 301–302
 - allocation, 337–338
 - architectural structures, 5–6
 - C&C overview, 335–337
 - combining, 339–340
 - documentation, 348–350
 - mapping between, 345
 - module, 333–334
 - notations, 336–339
 - quality, 338–339
- Virtualization and virtual machines
 - autoscaling, 259–260
 - cloud, 249–250
 - containers, 239–242
 - environment effects from, 73
 - images, 238
 - introduction, 233
 - layers as, 11
 - Pods, 242–243

- in sandboxing, 189
- serverless architecture, 243–244
- shared resources, 234
- summary, 244
- virtual machine overview, 235–238
- Vocabulary
 - quality attributes, 42
 - restrictions, 35–36
- Voting tactic, 57–58
- Vulnerabilities in security views, 338
- Warm spare tactic, 66
- Watchdogs, 57
- Waterfall model, 370
- Web-based system events, 133
- West, Mae, 133
- Wikis for documentation, 351
- WiMAX standards, 266
- Work assignment structures, 15–16
- Work-breakdown structures, 32
- Work skills of architect, 384
- Wrappers pattern, 112
- Wright, Frank Lloyd, 309
- X-ability, 212–214
- XML (EXtensible Markup Language), 226
- Z operations for qubits, 393

This page intentionally left blank

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 Sivi. "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 Jeromy 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.

Much of the material in Chapter 5 is adapted from Deployment and Operations for Software Engineers by Len Bass and John Klein [Bass 19] and from R. Kazman, P. Bianco, J. Ivers, J. Klein, “Maintainability”, CMU/SEI-2020-TR-006, 2020.

Much of the material for Chapter 7 was inspired by and drawn from R. Kazman, P. Bianco, J. Ivers, J. Klein, "Integrability", CMU/SEI-2020-TR-001, 2020.

This page intentionally left blank



**Carnegie
Mellon
University**
Software
Engineering
Institute

The Leader in Software Engineering and Cybersecurity

Operated by Carnegie Mellon University, the Software Engineering Institute has been a leader in the fields of software engineering and cybersecurity since 1984. We research and solve complex, long-term problems for the Department of Defense, government agencies, and private industry, and we are always working to transition solutions to the software and systems engineering communities throughout the world.

Learn more at sei.cmu.edu



Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community



informIT[®]
the trusted technology learning source

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Que • Sams • Peachpit Press

