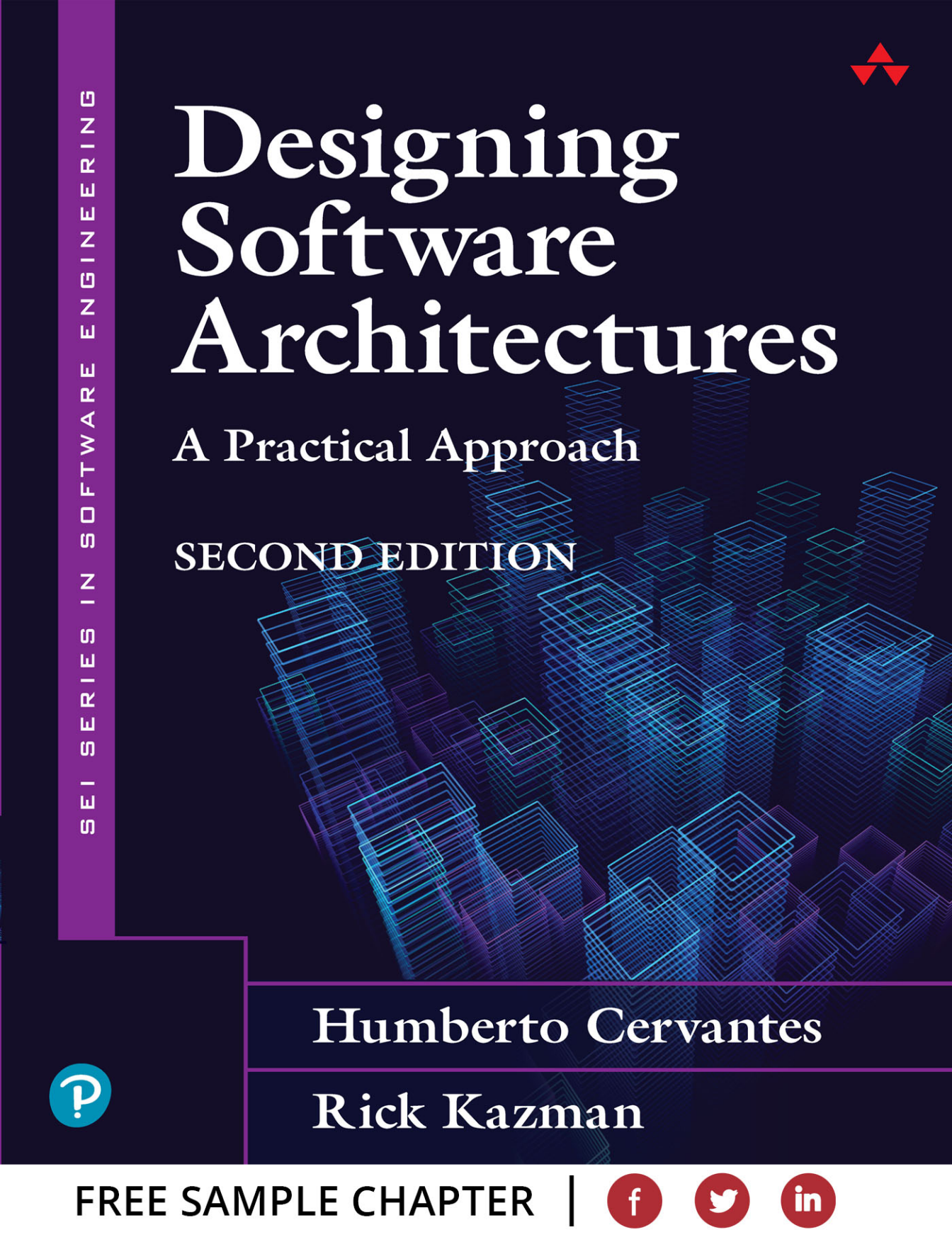
SEI SERIES IN SOFTWARE ENGINEERING
Designing Software Architectures
A Practical Approach
SECOND EDITION
Humberto Cervantes
Rick Kazman
FREE SAMPLE CHAPTER

Designing Software Architectures

2nd 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

Pearson
Addison-Wesley

informIT.com
the trusted technology learning source

Designing Software Architectures

2nd Edition

A Practical Approach

Humberto Cervantes
Rick Kazman

Addison-Wesley

Software Engineering Institute | Carnegie Mellon

SEI Series in Software Engineering

Figures 7.1, 7.2, 7.3, 9.8b: Amazon Web Services, Inc.

Figures 9.5, 9.6: Litmus Automation Inc.

Figures 9.9, 9.10: SoftServe Inc.

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: 2024937507

Copyright © 2024 Pearson Education, Inc.

Hoboken, NJ

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.pearsoned.com/permissions/. Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

ISBN-13: 978-0-13-810802-1

ISBN-10: 0-13-810802-1

$PrintCode

I would like to dedicate this book to my parents, Ilse and Humberto; to my wife Gabriela; and to my sons Julian and Alexis. Thank you for all your love, support, and inspiration.

Humberto Cervantes

I would like to dedicate this book to my wife Hong-Mei, and to my kids Jonah, Mia, and Sam, who have always supported me in my endeavors and tolerated my space cadet moments.

Rick Kazman

This page intentionally left blank

Contents

Preface *xi*

Acknowledgments *xiii*

About the Authors *xv*

CHAPTER 1 Introduction 1

1.1 Motivations 1

1.2 Software Architecture 2

1.3 The Role of the Architect 6

1.4 A Brief History of ADD 7

1.5 Summary 8

1.6 Further Reading 8

1.7 Discussion Questions 9

CHAPTER 2 Architectural Design 11

2.1 Design in General 11

2.2 Design in Software Architecture 12

2.3 Why Is Architectural Design So Important? 16

2.4 Architectural Drivers 17

2.5 Summary 26

2.6 Further Reading 26

2.7 Discussion Questions 27

CHAPTER 3 Making Design Decisions 29

3.1 Making Design Decisions 29

3.2 Design Concepts: The Building Blocks for Creating Structures 32

3.3 Design Concepts to Support Performance 39
3.4 Design Concepts to Support Availability 41
3.5 Design Concepts to Support Modifiability 46
3.6 Design Concepts to Support Security 49
3.7 Design Concepts to Support Integrability 53
3.8 Summary 57
3.9 Further Reading 57
3.10 Discussion Questions 58

CHAPTER 4 The Architecture Design Process 59
4.1 The Need for a Principled Method 59
4.2 Attribute-Driven Design 3.0 60
4.3 Applying ADD to Different System Contexts 66
4.4 Identifying and Selecting Design Concepts 70
4.5 Producing Structures 74
4.6 Defining Interfaces 77
4.7 Creating Preliminary Documentation During Design 81
4.8 Tracking Design Progress 85
4.9 Summary 87
4.10 Further Reading 88
4.11 Discussion Questions 89

CHAPTER 5 API-centric Design 91
5.1 Business Agility 91
5.2 API-centric Design 93
5.3 API-centric Design and ADD 107
5.4 Summary 110
5.5 Further Reading 110
5.6 Discussion Questions 111

CHAPTER 6 Designing for Deployability 113
6.1 Deployability Principles and Architectural Design 113
6.2 Design Decisions to Support Deployability 119
6.3 Deployability and ADD 128

6.4 Summary 129
6.5 Further Reading 130
6.6 Discussion Questions 130

CHAPTER 7 Designing Cloud-Based Solutions 133
7.1 Introduction to the Cloud 133
7.2 Drivers and the Cloud 136
7.3 Cloud-Based Design Concepts 139
7.4 ADD in the Design of Cloud-Based Solutions 147
7.5 Summary 148
7.6 Further Reading 148
7.7 Discussion Questions 149

CHAPTER 8 Case Study: Hotel Pricing System 151
8.1 Business Case 151
8.2 System Requirements 153
8.3 Development and Operations Requirements 156
8.4 The Software Design Process 158
8.5 Summary 188
8.6 Further Reading 189
8.7 Discussion Questions 189

CHAPTER 9 Case Study: Digital Twin Platform 191
9.1 Business Case 192
9.2 System Requirements 193
9.3 The Design Process 195
9.4 Summary 223
9.5 Further Reading 224
9.6 Discussion Questions 225

CHAPTER 10 Technical Debt in Architectural Design 227
10.1 Technical Debt 227
10.2 The Roots of Technical Debt in Design 228
10.3 Refactoring and Redesign 231
10.4 Technical Debt and ADD 233

10.5 Summary 235
10.6 Further Reading 236
10.7 Discussion Questions 237

CHAPTER 11 Analysis in the Design Process 239
11.1 Analysis and Design 239
11.2 Why Analyze? 242
11.3 Analysis Techniques 243
11.4 Tactics-Based Analysis 244
11.5 Reflective Questions 246
11.6 Scenario-Based Design Reviews 247
11.7 Summary 250
11.8 Further Reading 251
11.9 Discussion Questions 251

CHAPTER 12 The Architecture Design Process in the Organization 253
12.1 Architecture Design and the Development Life Cycle 253
12.2 Architecture Design and the Organization 263
12.3 Summary 268
12.4 Further Reading 269
12.5 Discussion Questions 270

CHAPTER 13 Final Thoughts 273
13.1 On the Need for Methods 273
13.2 Future Directions 275
13.3 Next Steps 276
13.4 Further Reading 277
13.5 Discussion Questions 277

Appendix Tactics-Based Questionnaires 279
Index 301

Preface

It has now been nearly 8 years since the first edition of *Designing Software Architectures* appeared. Much has changed in the world of technology since then—cloud architectures, Internet of Things (IoT) architectures, DevOps, the rise of artificial intelligence/machine learning (AI/ML), containers, microservices, and much more. Is our advice from 8 years ago still relevant? Well, yes and no.

The good news, from our perspective, is that the principles and practices of designing software architectures have not changed. The Attribute-Driven Design (ADD) method, which provides the scaffolding for the entire book, did not change *at all*. Over this 8-year span, it has been taught to thousands of practitioners and used successfully in many industrial projects. In all this time, the feedback has been positive and we have not had requests to change the method. That was reassuring. What did change, though, was the technical environment and the contexts in which we design today.

It is rare today that someone designs a standalone system. At the least, you will be building on top of existing frameworks and toolkits, and incorporating off-the-shelf components, some of which are likely open source. But you may be building a system that is interacting with other systems in real time, possibly sharing resources with them. It is likely that you are building a system using some kind of Agile methodology, and that your system will be frequently modified, with regular updates and releases. It is possible that the architectures you build will include IoT devices, will be mobile, will be cloud-based and containerized, and will be adaptive. And it is also possible that the software you are working with is old, and has accumulated technical debt over the years.

For all these reasons, we felt that it was time to write a second edition, one that acknowledges the many new contexts in which architectural decisions are made today. In this new edition, we now have chapters on supporting business agility through API-centric design, on deployability, on cloud-based solutions, and on technical debt in design.

We also created two new case studies for this edition of the book, which build upon and explore the challenges in these new contexts. The Hotel Pricing System case study in Chapter 8 implements much of its functionality in microservices, is deployed to a cloud infrastructure, is meant to be portable across environments, and was designed to be highly available. The digital twin platform case study presented in Chapter 9 explores even more challenges:

IoT, cloud computing, Big Data and analytics, AI/ML, extended reality (XR), simulation, advanced automation, and often robotics. This is a very large, complex system and requires the coordinated efforts of not just a single architect, but a large architecture team with varied expertise. In each case we show how the ADD method helps to transform design challenges into reality in a disciplined way.

We sincerely hope that this book gives you the confidence you need to tackle any design challenge, no matter how large!

Register your copy of ***Designing Software Architectures, 2nd 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 (**9780138108021**) 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.

Acknowledgments

Almost nothing worth doing can or should be done alone. It definitely took a small village of supporters to get this book done. But in particular we want to acknowledge and thank the following people who really made a substantial contribution to the book.

Marty Barrett, Mario Benitez, Jeff Gitter, James Ivers, Andrew Kotov, Stefan Malich, and Ipek Ozkaya kindly reviewed and provided feedback on this new edition of the book. Thank you very much for your valuable and insightful comments.

Thank you Serge Haziyev, Yaroslav Pidstryhach, Rodion Myronov, Taras Bachynskyy, Lyubomyr Demkiv, and Martin Vesper, who kindly provided a new case study for this book. The team from SoftServe have been long-time supporters and contributors of ideas and feedback on our work, and we are very grateful for that.

I (Humberto) wish to thank Ricardo Ivison and Luis Castro for all the opportunities they have provided me in recent years to be a practicing architect. Thank you also to the architects, developers, and managers in different companies with whom I have had the opportunity to collaborate and exchange ideas. I have learned, and continue to learn, a lot from them. I also want to give recognition to my university, Universidad Autónoma Metropolitana Iztapalapa, as it has always supported my work. Thanks to my colleagues and students who have accompanied me across the years in this architectural journey. I also wish to thank my co-author Rick Kazman for being such a nice person and colleague; it is always a pleasure to work and exchange opinions with him. Finally, I want to thank all the people outside of the software development world who lovingly support, motivate, and inspire me in many different ways. Your presence in my life is enriching and invaluable.

I (Rick) wish to thank James Ivers and his group at the Software Engineering Institute. They have given me many opportunities to engage with real-world projects and I have always benefited greatly from these. I would like to thank my wife, who has always been supportive of this work, even though it often means I am sitting at my desk, staring into space, lost in thought! And I would like to thank my co-author Humberto, who has always been energetic, positive, and a true pleasure to work with.

Finally, we want to extend a special thank you to Serge Haziyev, who has always been so supportive of our work.

This page intentionally left blank

About the Authors

Humberto Cervantes

Humberto Cervantes is a practicing software architect and a professor at Universidad Autónoma Metropolitana Iztapalapa in Mexico City. His primary research interest is software architecture and, more specifically, the development of methods and tools to aid in the design process. He is active in promoting the adoption of these methods and tools in the software industry. Since 2006, Cervantes has collaborated with several software development companies as a consultant and a software architect. He has worked in several projects for different industries, including telecommunications, hospitality, finance, and retail. He has authored numerous research papers and popularization articles, and has also co-authored one of the few books in Spanish on the topic of software architecture.

Cervantes received a master's degree and a PhD from Université Joseph Fourier (now Université Grenoble Alpes) in France. He holds the Software Architecture Professional and ATAM Evaluator certificates from the Software Engineering Institute. Besides software engineering, Cervantes enjoys spending time with his loved ones, playing with his dogs, exercising, and traveling.

Rick Kazman

Rick Kazman is the Danny and Elsa Lui Distinguished Professor of Information Technology Management 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 technical debt. Kazman has been involved in the creation of several highly influential methods and tools for architecture analysis, including the *ATAM* (Architecture Tradeoff Analysis Method) and the *Titan* and *DV8* tools. He is the author of more than 250 publications, co-author of three patents and nine books, including *Software Architecture in Practice*, *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 methods and tools have been adopted by many Fortune 1000 companies and has been cited more than 30,000 times, according to Google Scholar. He is currently a member of the IEEE Computer Society's Board of Governors, and a member of the ICSE Steering Committee.

Kazman received a B.A. (English/music) and M.Math (computer science) from the University of Waterloo, an M.A. (English) from York University, and a Ph.D. (computational linguistics) from Carnegie Mellon University. How he ever became a software engineering researcher is anybody's guess. When not doing architecture things, Kazman may be found cycling, singing acapella music, gardening, playing the piano, or practicing Tae Kwon Do.

3

Making Design Decisions

Design is the process of making decisions. Which decisions? Ideally, the ones that will lead to project success! But decision-making is hard. It is fraught with uncertainty. We are often uncertain about our project requirements and project resources. We are uncertain about the evolutionary trajectories of the technologies we build upon. We are uncertain about our planning horizon. We may be uncertain about the costs of the choices we are making. We may even be uncertain about our own knowledge (although architects and others often fail in the other direction, being overconfident rather than underconfident—this is known as overconfidence bias).

As architects we are responsible for the decisions that we make. This is as it should be. So which practices are available to us to ensure that the decisions that we make are the best ones, given our abilities? That is the subject of this chapter. We begin by reviewing a set of decision-making techniques. Then we turn to the body of knowledge that we can employ, as architects, to make better, more confident decisions.

Why Read This Chapter?

You come to this chapter, we are certain, with some preconceived ideas about what design is and what role it should play. But very few people have the depth and breadth of experience to help them make design decisions confidently. In this chapter, we provide a sampling of the kinds of knowledge that experienced designers possess, and we show how this knowledge forms the basis for making good design decisions with confidence. Reading this chapter won't make you an expert in design concepts—that is too much to expect of a single chapter—but you'll understand this essential tool that will be used throughout the book.

3.1 Making Design Decisions

We are seldom explicitly taught how to make decisions. It is somehow assumed that, given a proper education, we will make prudent choices. But this is not necessarily so. The field of

design is littered with examples of "What were they thinking?!" The architects who are really good decision makers are often ones with decades of experience. They achieved their status through years of trial and error, by making mistakes and learning from their failures. This is clearly a suboptimal form of training—expensive and inefficient and not very scalable. Can we do any better? We think that we can!

3.1.1 Decision-Making Principles

In what follows we present nine general decision-making principles. By following these principles, or strategies, you are not guaranteed to make a perfect decision. That is clearly too high a bar. But you can certainly increase the likelihood that the decision you are making is a sound, well-reasoned one—which is why we recommend these principles. And while all of them are "common sense", we see them violated over and over again in practice. The result of these (frequently avoidable) violations is projects with high technical debt, poor performance, poor scalability, stability issues, poor usability, and so on.

Principle 1: Use Facts

A fact is something that we believe to be true. Facts, and the evidence for them, are the foundations of logical reasoning. Incorrect or incomplete information may lead to invalid conclusions. An architect may hear praise for a new technology and base a decision on that, instead of testing it themselves. For example, a colleague of ours chose a NoSQL database in the Query side of a CQRS-based system (see Section 3.3.2) he was designing because he felt it was a good fit; he had not done any prototyping or analysis, but made this decision based on hearsay, experience, and gut feelings. Was this the right choice? Time will tell. To check the facts, he could have consciously considered: "What evidence supports the assumption that NoSQL will meet the system's requirements now and in the future?" When we cannot have all the facts, we make assumptions, as we will discuss next. We make a similar decision, as you will see, in Chapter 8 (the case study where this decision was made). Was this the right decision?

Principle 2: Check Assumptions

In the absence of facts, we often make assumptions so that we can continue with the process of design. For instance, we may not know if a technology can perform adequately until we prototype the software. When we consciously make assumptions, we are saying that these assumptions will be good ones. *Explicit assumptions* can be checked. *Implicit assumptions*, by comparison, are more devious. We may even be unaware we have made them. For example, if we build our application with node.js based on an existing three-tier client-server architecture (which contains a monolithic database) without assessing the compatibility with the database's performance characteristics, then this implicit assumption would not be checked, and that might create a performance risk for the system.

Principle 3: Explore Contexts

Contexts are *conditions* that influence software decisions. There are many contextual factors, such as development resources, financial pressures, legal obligations, industry norms, user

expectations, and past decisions. For example, we might want to implement a scalable and highly reliable database system, but have a limited budget. Even though the budget is not a system requirement, it affects our decision on database license procurement. Some contexts will end up being constraints on design, such as team experience or expectations. Design contexts shape our decisions implicitly, yet are not necessarily technology related. Exploring contextual factors can broaden our design considerations. To check that we have considered contexts we may ask: "What are the contexts that could influence X?", "Have I missed any contexts?", and "Does the team have the experience in implementing X?"

Principle 4: Anticipate Risks

A risk is the possibility of an undesirable outcome. A documented risk contains an estimate of the *size of the loss* and the *probability of the loss*. There are many risks that an architect needs to estimate, such as extreme spikes of demand and security attacks. Anticipating and quantifying risks is the process of exploring the unknowns, including estimating the possibility of risks occurring and, if they occur, their impacts. Architects may reflect on risks by asking questions such as "What are the potential undesirable outcomes?" or "Is there a chance that X would not work?"

Principle 5: Assign Priorities

Priorities quantify the *relative importance* of choices, choices such as which requirement to implement or which solution to use. If we can afford to implement only one of the two requirements, which one is more important? Prioritization is required when the things that we desire are competing for the same limited resource (e.g., time, money, developer skills, CPU, memory, network bandwidth). Some of these priorities emanate from contextual factors. To sort out our priorities, we can ask: "Which requirement is more important?", "What can we do without?", and "What should we use this resource for?"

Principle 6: Define Time Horizon

The time horizon defines the *time period* relevant to a decision and its impacts. Risks, benefits, costs, needs, and impacts can change over time, and we want to anticipate how they might change. For example, we might estimate that the system processing load will reach 85 percent capacity in 3 years, or we might choose to deploy the system on premises in the short term while keeping alive the option to deploy it on a public cloud if that becomes more cost-effective.

Defining the time horizon allows architects to explicitly state and evaluate the pros and cons of specific actions (and nonactions) in terms of their short- and long-term impacts. Without explicit considerations of the time horizon and the reasoning that goes along with it, long-term considerations may be undermined, or short-term needs may be ignored. To define the time horizon, the architect may ask questions such as "What would be affected in the short and long term if I decide on X?" or "What needs to be considered in different time horizons for X?"

Principle 7: Generate Multiple Solution Options

Some architects will go with the first solution they think of without considering further options. If the architect is experienced and the problem is well understood and low risk, this approach

may be ideal. But in more challenging contexts, considering only a single solution may be risky; the first solution is not necessarily the best one, especially when an architect is inexperienced or faces an unfamiliar situation. This behavior may be due to anchoring bias—a refusal to let go of the first idea. Generating multiple solution options helps an architect broaden the menu of solution choices and stimulate creativity. We can create a larger set of solution ideas by asking ourselves questions such as "Are there other solutions to this problem?" or "Can I find a better solution than X?"

Principle 8: Design Around Constraints

Constraints are *limitations* that set the boundaries of our design options. Constraints may come from requirements, contexts, technologies, and existing design choices. For instance, a CPU can compute only W instructions per second; the budget of the project is $X; the number of concurrent users supported by a software license is Y; platform Z doesn't support a certain protocol; developers have no experience with some technology; and so forth. In software development, we often find interconnected sets of constraints: If we choose component A, we must also use component B. When there are no apparent solutions, architects must design around constraints and introduce novel solutions, relax constraints, or manipulate the contexts. An architect can check on constraints by asking, "If I choose X, would it limit other design options?" or "Are there any constraints that could impede this design choice?"

Principle 9: Weigh the Pros and Cons

Pros and cons represent the arguments *for and against* each of the choices in a selection. Weighing the pros and cons is heavily bound up with examining tradeoffs. The evaluation of the pros and cons allows architects to decide what to choose and what to avoid. A quantitative evaluation can be based on measurable elements such as costs, benefits, priorities, immediacy (i.e., time horizon), and risks. However, some pros and cons cannot be easily quantified. Consider the navigation menu design of a mobile app: How can one quantify the pros and cons of a hamburger menu versus a set of tabs? In this case, qualitative arguments such as ease of access, ease of learning, and effort to implement can be marshaled. Weighing the pros and cons offers architects the chance to think about the relative benefits and drawbacks and who they might affect. To check tradeoffs, the architect can ask questions such as "Are there more benefits from solution X than from solution Y?" or "Have we done a thorough tradeoff evaluation?"

Now that we have enumerated a set of principles, let us turn our attention to the design concepts that we use to structure the system and to support our most important quality attributes.

3.2 Design Concepts: The Building Blocks for Creating Structures

Design is not random, but rather is planned, intentional, rational, and directed. The process of design might seem daunting at first. When facing the "blank page" at the beginning of any

design iteration, the space of possibilities might seem impossibly huge and complex; however, there is some help here. Over the course of decades, the software architecture community has created and evolved a body of generally accepted design principles that can guide us to create high-quality designs with predictable outcomes.

For example, some well-documented design principles are oriented toward the achievement of specific quality attributes:

- To help achieve high modifiability, aim for good modularity, which means high cohesion and low coupling.
- To help achieve high availability, avoid having any single point of failure.
- To help achieve scalability, avoid having any hard-coded limits for critical resources.
- To help achieve security, limit the points of access to critical resources.
- To help achieve testability, externalize state.
- . . . and so forth.

In each case, these principles have been evolved over decades of dealing with those quality attributes in practice. But these principles are rather abstract. To make them more usable in practice, a set of *design concepts* have been cataloged. These concepts serve as the building blocks of architectures. Different types of design concepts exist, and in the following subsections we discuss some of the most commonly used, including reference architectures, patterns, tactics, and externally developed components (such as frameworks). Whereas the first three are conceptual in nature, the last one is concrete.

3.2.1 Reference Architectures

Reference architectures are blueprints that provide an overall logical structure for particular classes of applications. A reference architecture is a reference model mapped onto one or more architectural patterns. Its utility has been proven in business and technical contexts, and it typically comes with a set of supporting artifacts that facilitates its use.

An example of a reference architecture for the development of service applications (such as microservices) is shown in Figure 3.1. This reference architecture establishes the main layers for this type of application—service, business, and data—as well as the types of elements that occur within the layers and the responsibilities of these elements, such as service interfaces, business components, data access components, service agents, and so on. Also, this reference architecture introduces cross-cutting concerns, such as security and communication, that need to be addressed. As this example shows, when you select a reference architecture for your application, you also adopt a set of concerns that you need to address during design. You may not have an explicit requirement related to communications or security, but the fact that these elements are part of the reference architecture guides you to make design decisions about them.

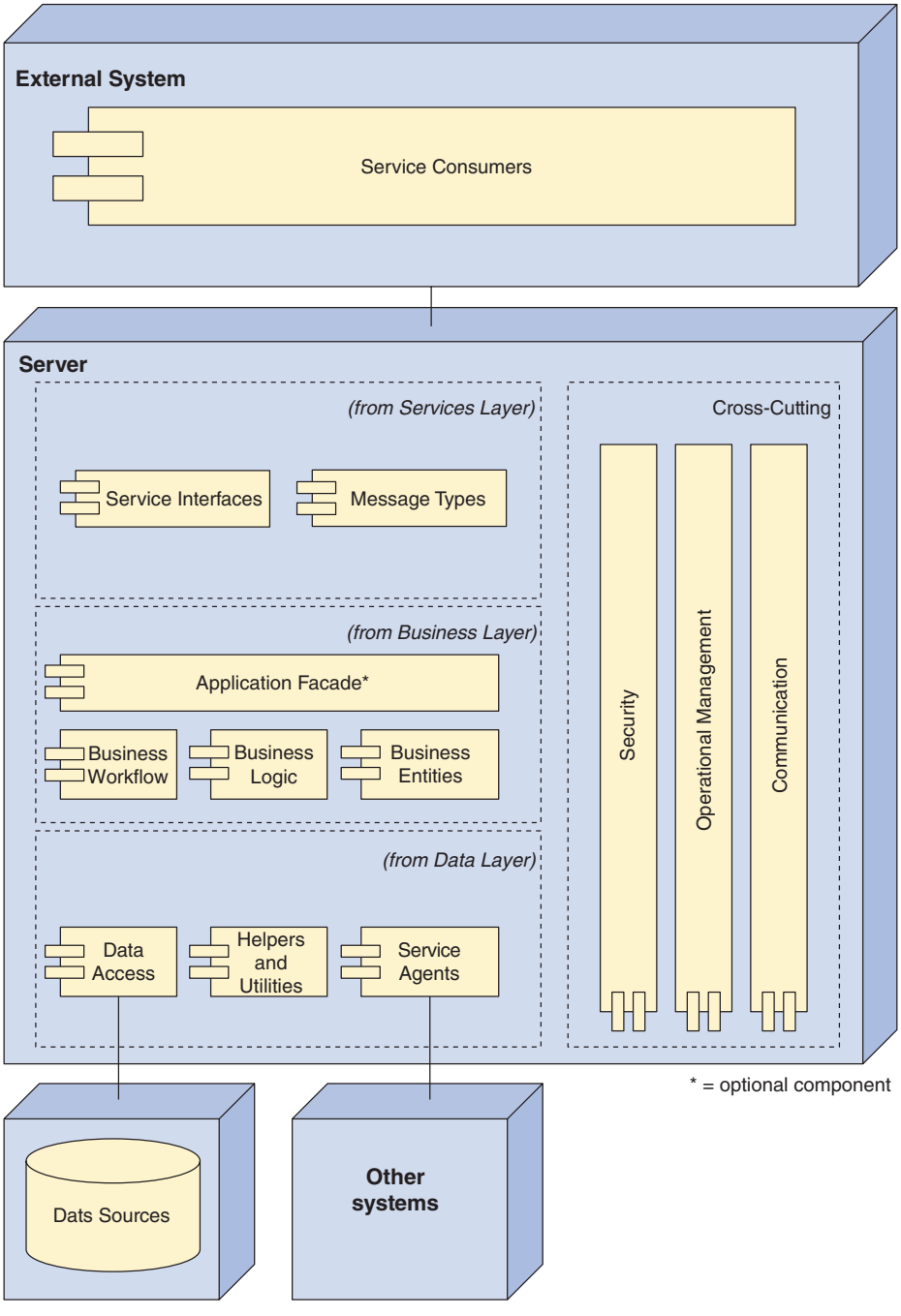

FIGURE 3.1 Service application reference architecture

Reference architectures may be confused with architectural styles, but these two concepts are different. Architectural styles (such as "pipe and filter" and "client-server") are patterns that define types of components and connectors in a specified topology that are useful for structuring an application either logically or physically. Reference architectures, in contrast, provide a more detailed structure for entire applications, and they may embody multiple patterns. Reference architectures are preferred by practitioners—which is also why we favor them in our list of design concepts.

Many reference architectures are available, but there is no single authoritative catalog of them. In the context of cloud development, providers have created a number of useful catalogs that collect reference architectures focused on cloud resources. In addition, many catalogs of patterns have been created across the years, and new ones appear periodically.

3.2.2 Patterns

Design/architectural patterns are conceptual solutions to recurring design problems that exist in a defined context. Although design patterns originally focused on decisions at the object scale, including instantiation, structuring, and behavior, today there are catalogs of patterns that address decisions at varying levels of granularity. In addition, there are specific patterns to address quality attributes such as security, availability, performance, and integrability.

Some people might differentiate between what they consider to be architectural patterns and the more fine-grained design patterns. However, we believe there is no principled difference that can be solely attributed to scale. We consider a pattern to be architectural when its use directly and substantially influences the satisfaction of some of a system's architectural drivers.

Another important type of pattern that has significant implications for the architectural design process is deployment patterns. These patterns package key decisions that shape the system's infrastructure and have profound implications for quality attributes such as availability, performance, modifiability, and usability. Such decisions are often bound early in the product life cycle. We will discuss deployment patterns in Chapter 6.

We provide examples of patterns for a number of quality attributes in Sections 3.3 through 3.8. Note that these are just examples—many more patterns exist for each quality attribute, and for other qualities.

3.2.3 Tactics

Architects can also employ fundamental design techniques to help achieve a response goal for a particular quality attribute. We call these architectural design primitives *tactics*. Tactics, like patterns, are techniques that architects have been using for years. We do not invent tactics, but simply capture what architects have done in actual practice, over the decades, to manage quality attribute response goals.

Tactics are design decisions that influence the control of a quality attribute response. For example, if you want to design a system to have good usability, you would need to make a set of design decisions that support this quality attribute, as represented in Figure 3.2.

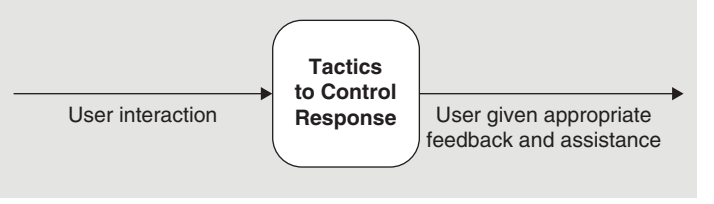

FIGURE 3.2 Usability tactics mediate user interactions and responses

Tactics are simpler than patterns. They focus on the control of a single quality attribute response (although they may, of course, trade off this response with other quality attribute goals). Patterns, in contrast, typically focus on resolving and balancing multiple forces—that is, multiple quality attribute goals. By way of analogy, we can say that a tactic is an atom, whereas a pattern is a molecule.

Tactics provide a top-down way of thinking about design. A tactics categorization begins with a set of design objectives related to the achievement of a quality attribute, and presents the architect with a set of options from which to choose.

For example, in Figure 3.3, the design objectives for usability are "Support user initiative" and "Support system initiative". An architect who wants to create a system with "good" usability needs to choose one or more of these options. That is, the architect needs to decide if they want to support a user in their tasks by allowing for features such as undo, cancel (for long-running or hung operations), aggregation (of similar UI objects so that a user-issued command applies to all of them), or pause/resume (again, typically used for long-running operations). The architect may also want to support system initiatives, such as maintaining a model of the task (so that, for example, context-appropriate help and guidance can be given), maintaining a model of the user (so that user-appropriate feedback is offered; for example, a novice user might want far more guidance, whereas an expert user might want shortcuts for common operations), or maintaining a system model (for example, being able to accurately estimate time remaining for long-running operations).

Each of these tactics is an option for the architect. They may be instantiated via coding, via patterns, or via external components such as frameworks—but they are architectural. To offer undo capabilities, the system must maintain a set of transactions, representing changes to the system state, and be able to roll back any of those transactions to undo—that is, to return the system to its prior state. To be able to offer cancel capabilities, the system must be able to revert its state to whatever it was before the operation was initiated. These sorts of capabilities require architectural thinking, and tactics provide a starting point for this decision-making process. As we will see in Chapter 4, the choice, combination, and tailoring of tactics and patterns are some of the key steps of the ADD process.

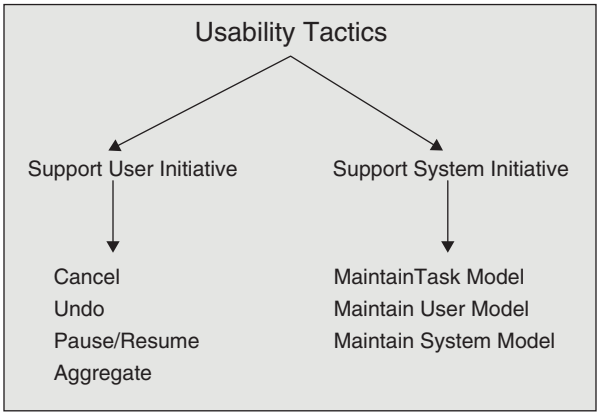

FIGURE 3.3 Usability tactics categorization

Tactics categorizations have been established for the quality attributes of availability, deployability, energy efficiency, interoperability, modifiability, performance, safety, security, testability, and usability. Collectively, these categorizations cover the vast majority of the design decisions that an architect needs to make in practice.

3.2.4 Externally Developed Components

Both patterns and tactics are abstract in nature. When you are designing a software architecture, however, you need to make these design concepts concrete and closer to the actual implementation. There are two ways to achieve this: You can code the elements obtained from tactics and patterns, or you can associate technologies with one or more of these elements in the architecture. This "buy versus build" choice is one of the most important decisions you will make as an architect.

We consider technologies to be *externally developed components*, because they are not created as part of the development project. Several types of externally developed components exist:

- *Technology families.* A technology family represents a class of technologies with common functional purposes. It can serve as a placeholder until a specific product or framework is selected. Examples include a relational database management system (RDBMS) and an object-oriented to relational mapper (ORM).
- *Products.* A product (or software package) comprises a self-contained piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding. An example is a relational database management system, such

as Oracle or PostgreSQL, which belongs to the RDBMS technology family. APIs (application programming interfaces) from external systems are one commonly encountered type of product; we discuss them in Chapter 5. We discuss another type of product, cloud capabilities made available by cloud provider platforms, in Chapter 7.

- *Application frameworks.* An application framework (or just framework) is a reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Frameworks, when carefully chosen and properly implemented, increase the productivity of programmers. They do so by enabling programmers to focus on business logic and end-user value, rather than on the underlying technologies and their implementations. As opposed to products, framework functions are generally invoked from the application code or are "injected" using some type of dependency injection approach. Frameworks usually require extensive configuration, typically through XML files or other approaches such as annotations in Java. One example of a framework is Hibernate, which is used to perform object-oriented to relational mapping in Java. Several types of frameworks are available: Full-stack frameworks, such as Spring, are usually associated with reference architectures and address general concerns across the different elements of the reference architecture, whereas non-full-stack frameworks, such as JPA (Java Persistence API), address specific functional or quality attribute concerns.
- *Platforms.* A platform provides a complete infrastructure upon which to build and execute applications. Examples of current cloud platforms include Amazon Elastic Beanstalk, Google App Engine, and Azure App Service, all of which are Platform as a Service (PaaS) offerings.

The selection of externally developed components, which is a key aspect of the design process, can be a challenging task because of their extensive number. Here are a few criteria you should consider when selecting externally developed components:

- *Problem that it addresses.* Is it something specific, such as a framework for object-oriented to relational mapping, or is it something more generic, such as a complete platform?
- *Cost.* What is the cost of the license? If it is free, what is the cost of support and education?
- *Type of license.* Does it have a license that is compatible with the project goals?
- *Vendor lock-in.* Will the inclusion of the technology result in a dependency on the organization that produces it?
- *Learning curve.* Is this a technology that is difficult to learn? Is it difficult to find developers who have some expertise with it?
- *Community support.* Is this technology well supported by a robust community or by its vendor?

In the following sections, we focus our attention on two categories of design concepts: tactics and patterns. We intentionally omit externally developed components, even though the choice of these components is clearly an important design decision. However, the rapid evolution of technologies makes externally developed components prone to becoming quickly outdated, which is why we do not discuss them extensively here.

3.3 Design Concepts to Support Performance

Performance is about time and the software system's ability to meet timing requirements. Generally speaking, faster is better!

When events occur—interrupts, messages, requests from users or other systems, or clock events—the system must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system's response to those events is the starting point for discussing performance. All systems have performance requirements, even if they are not explicitly expressed.

Performance is often linked to scalability—increasing the system's capacity, while ensuring that it still performs well. Often, performance is considered after you have constructed something and found its performance to be inadequate. You can fix this by architecting your system consciously with performance in mind.

3.3.1 Performance Tactics

The performance tactics categorization is shown in Figure 3.4. This set of tactics, like all tactics, helps an architect reason about the quality attribute. There are two major categories of performance tactics: *Control Resource Demand* and *Manage Resources.*

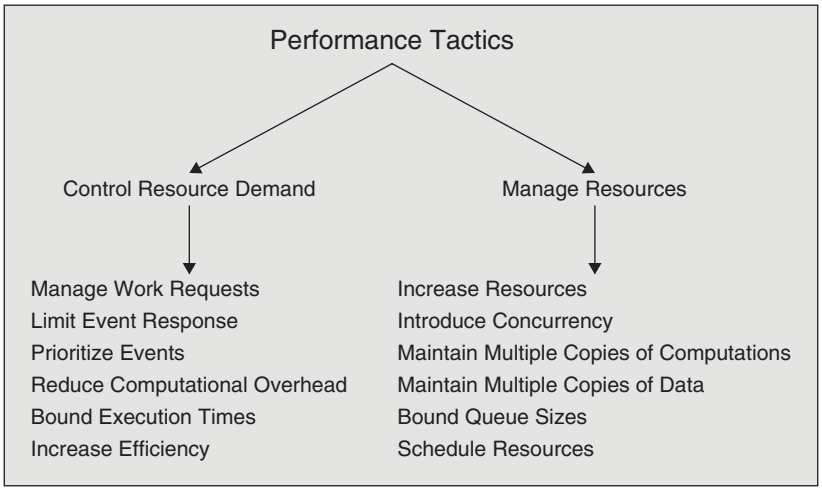

FIGURE 3.4 Performance tactics categorization

Within the *Control Resource Demand* category, the tactics are:

- *Manage work requests.* One way to reduce work is to reduce the number of requests coming into the system. Ways to do that include managing work requests (i.e., limiting the number of requests the system will accept in a given time period) and managing the sampling rate (e.g., switching to a lower frame rate for streaming video).
- *Limit event response.* When events arrive too rapidly to be processed, they must be queued until they can be processed, or until they are discarded. You may choose to process events only up to a set maximum rate, thereby ensuring predictable processing for other events.
- *Prioritize events.* If not all events are equally important, you can impose a priority scheme that ranks events according to how urgently you want to service them.
- *Reduce computational overhead.* For events that make it into the system, you can reduce the amount of work in handling each event by reducing indirection (making direct calls rather than going through an intermediary, for example), by co-locating communicating resources, and by periodic cleaning (such as garbage collection).
- *Bound execution times.* You can place a limit on how much execution time is used to respond to an event.
- *Increase efficiency of resource usage.* Improving the efficiency of algorithms and data structures (adding an index to database, for example) in critical areas can decrease latency and improve throughput and resource consumption.

Within the *Manage Resources* category, the tactics are:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential to improve performance.
- *Introduce concurrency.* If requests can be processed in parallel, blocked time can be reduced.
- *Maintain multiple copies of computations.* This tactic reduces the contention that would occur if all requests were allocated to a single instance.
- *Maintain multiple copies of data.* Two common examples of maintaining multiple copies of data are data replication and caching.
- *Bound queue sizes.* This tactic controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- *Schedule resources.* Whenever contention for a resource occurs, the resource should be scheduled. Your concern is to understand the characteristics of each resource's use and choose an appropriate scheduling strategy.

These tactics cover the spectrum of architectural concerns with respect to performance. Now we turn our attention to more complex design structures, patterns.

3.3.2 Performance Patterns

In what follows we provide a small selection of architectural patterns that address performance concerns. We make no attempt here to provide a comprehensive catalog; that is not the purpose

of this book. In-depth coverage of performance patterns can be found in many other resources. We merely provide a few patterns here to stimulate thinking and to provide examples of the kinds of resources that are available to support the architect in reasoning about design for performance.

3.3.2.1 Load Balancer Pattern

A load balancer is an intermediary that handles messages from clients and determines which instance of a service should respond. The load balancer serves as a single point of contact for incoming messages and farms out requests to a pool of (typically stateless) providers.

By sharing the load among a pool of providers, latency can be kept lower and more predictable for clients. It is a simple matter to add more resources to the resource pool, and no client needs to be aware of this event (an instance of the increase resources, maintain multiple copies of computations, and introduce concurrency tactics). Moreover, any failure of a server is invisible to clients, assuming there are still some remaining processing resources. Although this is an availability benefit, we note once again that patterns often address multiple quality attributes, whereas tactics address just a single quality attribute.

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 often replicated (and even load balanced).

3.3.2.2 Throttling Pattern

The throttling pattern packages the manage work requests tactic. It is used to limit access to an important service. In this pattern, an intermediary—a throttler—monitors the service and determines whether an incoming request can be serviced. 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.

But, once again, there are tradeoffs to consider: 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 large, or you may lose requests. Also, this pattern can be difficult to add to an existing system where clients and servers are tightly coupled.

3.4 Design Concepts to Support Availability

Availability is a system property. A highly available system is compliant with its specifications—always ready to carry out its tasks when you need it to be. A *failure* in a system occurs when the system no longer delivers a service consistent with its specification. A *fault* has the potential to cause a failure. Availability encompasses the ability of a system to mask or repair faults so they do not become failures. It builds on the quality attribute of reliability by adding the notion of recovery (repair). The goal is to minimize service outage time by mitigating faults.

3.4.1 Availability Tactics

Availability tactics enable a system to endure faults. The tactics keep faults from becoming failures (or bound the effects of the fault and make repairs). A failure's cause is a fault. A fault can be internal or external to the system. Faults can be prevented, tolerated, removed, or forecast. Through these actions, a system can become "resilient" to faults.

Figure 3.5 shows the availability tactics categorization. There are three major categories of availability tactics: *Detect Faults*, *Recover from Faults*, and *Prevent Faults*. It is difficult to envision a system that could achieve high availability if it does not address all of these categories.

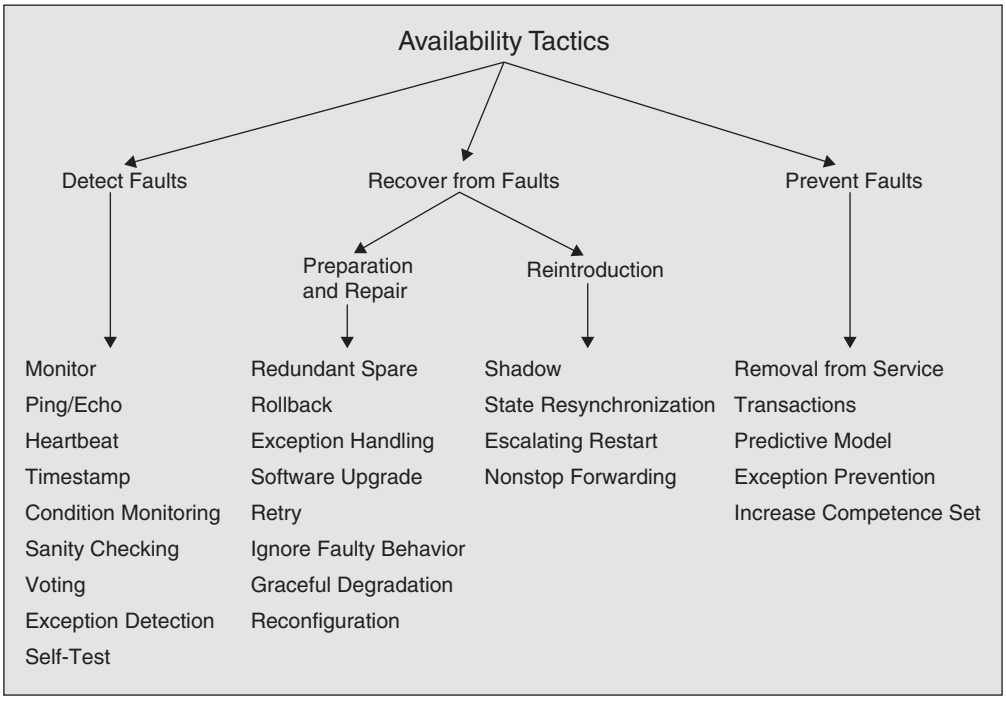

FIGURE 3.5 Availability tactics categorization

When designing for availability, we are concerned with how faults are detected, how frequently they occur, what happens when they occur, how long a system may be out of operation, how faults or failures can be prevented, and which notifications are required when a failure occurs. Each of these concerns can be addressed architecturally, through the appropriate selection of availability tactics.

Within the *Detect Faults* category, the tactics are:

- *Monitor:* a component used 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.
- *Ping/echo:* asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.
- *Heartbeat:* a periodic message exchange between a system monitor and a process being monitored.
- *Timestamp:* used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- *Condition monitoring:* checks conditions in a process or device, or validates assumptions made during the design.
- *Sanity checking:* checks the validity or reasonableness of a component's operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.
- *Voting:* checks that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, and analytic redundancy.
- *Exception detection:* detection of a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, timeout).
- *Self-test:* procedure for a component to test itself for correct operation.

Within the *Recover from Faults* category, there are two subcategories: "Preparation and Repair" and "Reintroduction". Let us first look at the tactics for "Preparation and Repair":

- *Redundant spare:* 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, as we will discuss in Section 3.4.2.)
- *Rollback:* revert to a previous known good state, referred to as the "rollback line".
- *Exception handling:* dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- *Software upgrade:* in-service upgrades to executable code images in a non-service-affecting manner.
- *Retry:* where a failure is transient, retrying the operation may lead to success.
- *Ignore faulty behavior:* ignoring messages sent from a source when it is determined that those messages are spurious.
- *Graceful degradation:* maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- *Reconfiguration:* reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

Within the "Reintroduction" subcategory, the tactics are:

- *Shadow:* running a previously failed/upgraded component in "shadow mode" prior to reverting it to an active role.

- *State resynchronization:* partner to the redundant spare tactic, where state is sent from active to standby components.
- *Escalating restart:* recover from faults by varying the granularity of the component(s) restarted.
- *Nonstop forwarding:* functionality is split into supervisory and data components. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered.

The final category of availability tactics is *Prevent Faults*. The tactics within this category are:

- *Removal from service:* temporarily placing a system component in an out-of-service state for the purpose of mitigating potential failures.
- *Transactions:* bundling state updates so that messages exchanged between components are atomic, consistent, isolated, and durable.
- *Predictive model:* monitoring the state of a process to ensure that the system is operating properly; taking corrective action when conditions are predictive of likely future faults.
- *Exception prevention:* preventing system exceptions from occurring by masking a fault, or preventing it via mechanisms such as smart pointers, abstract data types, and wrappers.
- *Increase competence set:* designing a component to handle more cases—faults—as part of its normal operation.

3.4.2 Availability Patterns

A number of redundancy patterns are commonly used to achieve high availability. In this section, we will look at them as a group, as this helps us understand their strengths and weaknesses, and the scope of the design space around this issue. These patterns deploy a group of active components and a group of redundant spare components. In general, the greater the level of redundancy, the higher the availability and the cost and complexity will be.

The benefit of using a redundant spare is that the system continues to function correctly with only a brief delay after a failure. The alternative is a system that stops functioning correctly (or altogether) until the failed component is repaired.

All of these patterns incur additional cost and complexity from 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.

3.4.2.1 Hot Spare (Active Redundancy)

In the hot spare configuration, all of the components belong to the active group and receive and process identical inputs in parallel, allowing the redundant spares to maintain a synchronous state with the active components. Because the redundant spare possesses an identical state to the active component, it can take over from a failed component in a matter of milliseconds.

3.4.2.2 Warm Spare (Passive Redundancy)

In the warm spare variant, only the members of the active group process input. One of their duties is to provide the redundant spares with periodic state updates. Because this state is loosely coupled with the active components, the redundant components are referred to as warm spares.

Passive redundancy achieves a balance between the more highly available but more compute-intensive (and expensive) hot spare pattern and the less available but less complex (and cheaper) cold spare pattern.

3.4.2.3 Cold Spare (Spare)

In the cold spare configuration, spares remain out of service until a failure occurs, at which point a power-on-reset is initiated on the spare prior to it being placed in service. Due to its poor recovery performance, and hence its high MTTR (mean time to recovery), this pattern is poorly suited to systems with high availability requirements.

3.4.2.4 Tri-Modular Redundancy

Tri-modular redundancy (TMR) is a widely used implementation in which the voting tactic is combined with the hot spare pattern; it employs three identical components (or, more generally, *N* identical components in *N*-modular redundancy). In the example shown in Figure 3.6, each component receives the same inputs and forwards its output to the voter. If the voter detects any inconsistency, it reports a fault. For this reason, the value of *N* is usually set to an odd number, to avoid ties in the voting. The voter must also decide which output to use. Typical choices are letting the majority rule or choosing a computed average of the outputs.

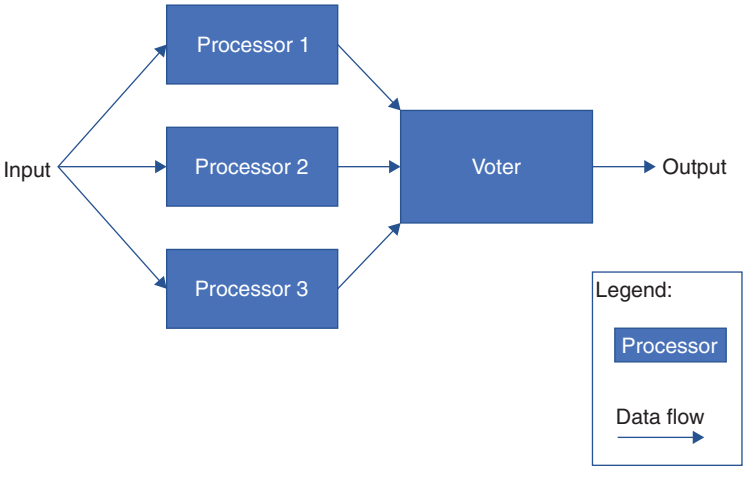

FIGURE 3.6 Tri-modular redundancy pattern

The TMR pattern is simple to understand and implement. It is independent of what might be causing disparate results and is only concerned with making a choice so that the system can continue to function.

As with the other redundancy patterns, there is a tradeoff between increasing the level of replication, which raises the cost, and the resulting availability. The statistical likelihood of two or more components failing is small, and three components is a common sweet spot between availability and cost.

3.4.2.5 Circuit Breaker

The Circuit Breaker pattern is sometimes seen as a performance pattern, but actually addresses both performance and availability concerns. (As this example shows, the label on a pattern is less important than the response measures that it helps you to achieve.) This pattern does not use redundancy, but it is important for achieving high availability in networked contexts, such as microservice architectures. A common availability tactic in networked systems is retry. For example, in the event of a timeout or fault when invoking a service, an invoker may try again (and again and again). A circuit breaker keeps the invoker from retrying infinitely many times, waiting for a response that may never come. This breaks the endless retry cycle when the circuit breaker deems that the system is dealing with a fault. Until the circuit is "reset", subsequent invocations will return immediately without requesting the service.

One important benefit of this pattern is that it removes from individual components the policy of how many retries to allow before declaring a failure. The circuit breaker, in conjunction with software that listens to it and begins recovery, prevents cascading failures.

But care must be taken in choosing the timeout (or retry) values. If the timeout is too long, then unnecessary latency is added. If the timeout is too short, then the circuit breaker will trip when it does not need to, which can lower the availability and performance of services.

3.5 Design Concepts to Support Modifiability

Modifiability is about change. As architects, we need to consider and plan for the cost and risk of making anticipated changes. To be able to plan for modifiability, an architect has to consider three big questions: (1) What can change? (2) What is the likelihood of the change? and (3) When is the change made and who makes it?

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 scalability, variability, portability, and location independence, but there are many others.

3.5.1 Modifiability Tactics

Architects need to worry about modifiability to make the system easy to understand, debug, and extend. Tactics can help address these concerns.

The modifiability tactics categorization is shown in Figure 3.7. There are three major categories of modifiability tactics: *Increase Cohesion*, *Reduce Coupling*, and *Defer Binding*. We will examine these in turn.

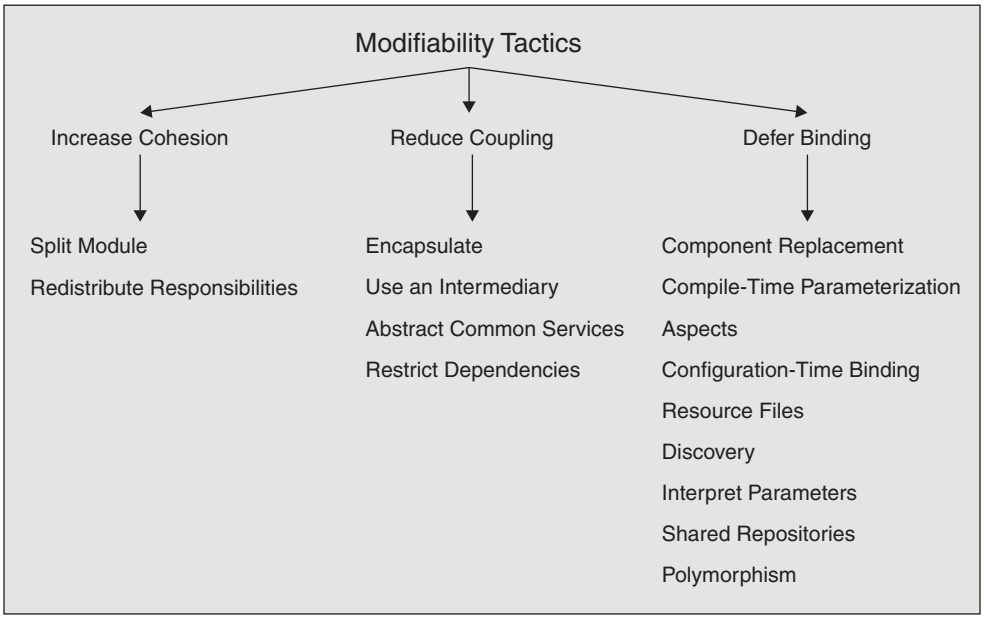

FIGURE 3.7 Modifiability tactics categorization

Within the *Increase Cohesion* category, the tactics are:

- *Split module.* If the module being modified includes responsibilities that are not cohesive, modification costs will likely be high. Refactoring the module to separate the responsibilities should reduce the average cost of future changes.
- *Redistribute responsibilities.* If (related) responsibilities A, A′, and A″ are sprinkled across several distinct modules, they should be grouped together into a single module.

Within the *Reduce Coupling* category, the tactics are:

- *Encapsulate.* Encapsulation introduces an explicit interface to an element, hiding its implementation details. This ensures that changes to the implementation do not affect the clients, as long as the interface remains stable. All access to the element must then pass through this interface.
- *Use an intermediary.* Intermediaries, such as proxies, bridges, or adapters, are used for breaking dependencies between a set of components C_i or between C_i and the system S.

- *Abstract common services.* When multiple elements provide services that are similar, it may be useful to hide them behind a common abstraction. The resulting encapsulation hides the details of the elements from other components in the system.
- *Restrict dependencies.* This tactic restricts which modules a given module interacts with or depends on.

And within the *Defer Binding* category, we distinguish the tactics according to how late in the life cycle functionality is bound to the system:

- *Tactics to bind values at compile or build time:* component replacement, compile-time parameterization, aspects.
- *Tactics to bind values at deployment, startup, or initialization time:* configuration-time binding, resource files.
- *Tactics to bind values at runtime:* discovery, interpret parameters, shared repositories, polymorphism.

3.5.2 Modifiability Patterns

In Section 3.2.3, we said that patterns differ from tactics in that tactics typically have just a single goal—the control of a quality attribute response—whereas patterns often attempt to achieve multiple goals and balance multiple competing forces. You might not think of some of the patterns that we present in this section as modifiability patterns at all. For example, we present the client-server pattern in this section. But first we begin with the uber modifiability pattern—layers.

3.5.2.1 Layered Pattern

In complex systems, there is a need to develop and evolve portions independently. For this reason, developers need a clear and well-documented separation of concerns, so that modules may be independently developed and maintained. The software therefore needs to be segmented in such a way that the modules can be evolved separately with little interaction among the parts. To achieve this separation of concerns, we divide the software into units called *layers.* Each layer is a grouping of modules that offers a cohesive set of services. Layers completely partition the software, and each partition is exposed through a public interface. There is, ideally, a unidirectional *allowed-to-use* relation among the layers.

But layering has its costs and tradeoffs. The addition of layers adds up-front effort and complexity to a system. Layers also add runtime overhead; there is a performance penalty to every interaction that crosses multiple layers. In Section 5.2.3, we will see how this pattern may be realized in terms of layered APIs.

3.5.2.2 Strategy

The Strategy pattern allows the selection of an algorithm, typically from a family of related algorithms, at runtime. This pattern is a realization of the defer binding tactic. It makes it easy to define a rich set of behaviors or policies and switch between them, typically at runtime,

as needed. This pattern eliminates the need to clutter the code with complex switch or if statements to provide for the various options. It also eases the burden for programmers who might later want to evolve the system, adding even more options. The only drawback to this pattern is that it adds some up-front complexity. Thus, it should not be used if only a small and stable number of alternative algorithms or policies need to be implemented.

3.5.2.3 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 browser clients. You might not think of this pattern as a modifiability pattern at all. But patterns are complex and they can serve multiple purposes. Indeed, the Client-Server pattern offers many benefits:

- The connection between a server and its clients is established dynamically.
- There is no coupling among the clients.
- The number of clients can easily scale and is constrained only by the capacity of the server.
- Server functionality can be scaled if needed with no impact on the clients.
- Clients and servers can evolve independently as long as their APIs remain stable.
- Common services can be shared among multiple clients.
- For interactive systems, the interaction with a user is isolated to the client.

Not all of these benefits relate to modifiability, but certainly the aspects of this pattern that limit coupling and defer binding are obvious instantiations of modifiability tactics. A server typically has no prior knowledge of its clients.

The tradeoffs that this pattern introduces primarily relate to performance, availability, and security. When this pattern is implemented, communication occurs over a network. 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, provisions must be made for security. In addition, servers may be single points of failure.

3.6 Design Concepts to Support Security

Security is a measure of the system's ability to protect data and resources from unauthorized access while still providing legitimate access to authorized actors. The most common approaches to characterizing and analyzing security focus on three important characteristics: confidentiality, integrity, and availability (CIA):

- Confidentiality is the property that data or services are protected from unauthorized access.
- Integrity is the property that data or services are not subject to unauthorized manipulation.
- Availability is the property that the system will be there for legitimate use, consistent with its specifications.

The security tactics provide strategies for achieving confidentiality and integrity. We have already discussed availability.

3.6.1 Security Tactics

Secure facilities in the physical world permit only limited access to certain resources (e.g., by building walls, having locked doors and windows, and establishing security checkpoints), have means of detecting intruders (e.g., requiring visitors to wear badges, using motion detectors), have deterrence mechanisms (e.g., armed guards, razor wire), have reaction mechanisms (e.g., automatic locking of doors), and have recovery mechanisms (e.g., off-site backups). These strategies are all relevant to computer-based systems and so lead to our four categories of security tactics: *Detect Attacks*, *Resist Attacks*, *React to Attacks*, and *Recover from Attacks*, as shown in Figure 3.8.

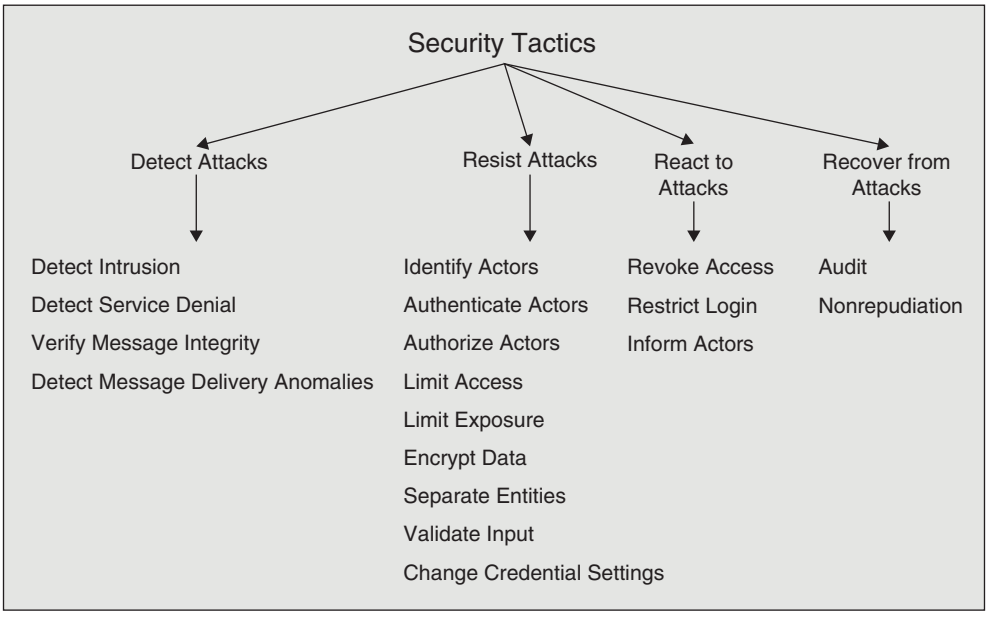

FIGURE 3.8 Security tactics categorization

Within the *Detect Attacks* category, the tactics are:

- *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.

- *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.
- *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.
- *Detect message delivery anomalies.* This tactic is used to detect man-in-the-middle attacks. If message delivery times are normally stable, then by checking the amount of 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.

Within the *Resist Attacks* category, the tactics are:

- *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 keys, 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, digital certificates, two-factor authentication, and biometric identification provide means for authentication.
- *Authorize actors.* Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. This mechanism is usually enabled by providing some access control mechanisms within a system.
- *Limit access.* This tactic involves limiting access to computer resources. Limiting access might mean restricting the number of access points to 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.
- *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 or in-transit data beyond that available from authorization.
- *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.
- *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.

- *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.

Within the *React to Attacks* category, the tactics are:

- *Revoke access.* If the system or an administrator believes that an attack is under way, then access can be limited to sensitive resources, even for normally legitimate users and uses.
- *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.
- *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.

Finally, within the *Recover from Attacks* category, the tactics are:

- *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.

In addition, all of the availability tactics aid in recovering from attacks.

3.6.2 Security Patterns

Many security patterns have been developed over the years. Here we touch on just two of them, as examples: Intercepting Validator and Intrusion Detection.

3.6.2.1 Intercepting Validator

This pattern inserts a software element—an adapter—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, or detect message delivery anomalies.

The benefit of this pattern is that, 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. The tradeoffs are that, as always, introducing an intermediary exacts a performance price.

Attack vectors change and evolve over time, so this component must be kept up-to-date to ensure that it maintains its effectiveness. This imposes a maintenance obligation on the

organization responsible for the system. Of course, this maintenance cost must be paid in any case, if the system is to maintain its level of security.

3.6.2.2 Intrusion Detection

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 often implement most of the "detect attacks" and "react to attacks" tactics.

There are some tradeoffs associated with this pattern, however. For example, the patterns of activity that an IPS looks for change and evolve over time, so the patterns database must be constantly updated. Also, systems employing an IPS incur a performance cost. In addition, IPSs are typically created from commercial off-the-shelf components, which might mean that they are not tailored for your specific application.

3.7 Design Concepts to Support Integrability

Software architects need to be concerned about more than just making separately developed components cooperate: They must also consider the costs and technical risks of integration tasks. These risks may be related to schedule, performance, or technology. Consider that a project needs to integrate a set of components $C_1, C_2, \ldots, C_n$ into a system S. The task, then, is to design for, and analyze the costs and technical risks of, integrating additional, but not yet specified, components $\{C_n + 1, \ldots, C_m\}$. We assume we have control over S, but the development of the unspecified $\{C_i\}$ components may be outside our control.

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, and distance is the difficulty of resolving differences in each of the dependencies. Distance may be any of the following: syntactic distance, data semantic distance, behavioral semantic distance, temporal distance, or even resource distance. Although we may not capture size and distance as precise metrics, this understanding creates a frame of reference for thinking about integrability.

3.7.1 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 to fulfill evolutionary requirements. These tactics help to reduce size and distance attributes.

There are three categories of integrability tactics: *Limit Dependencies*, *Adapt*, and *Coordinate*, as shown in Figure 3.9.

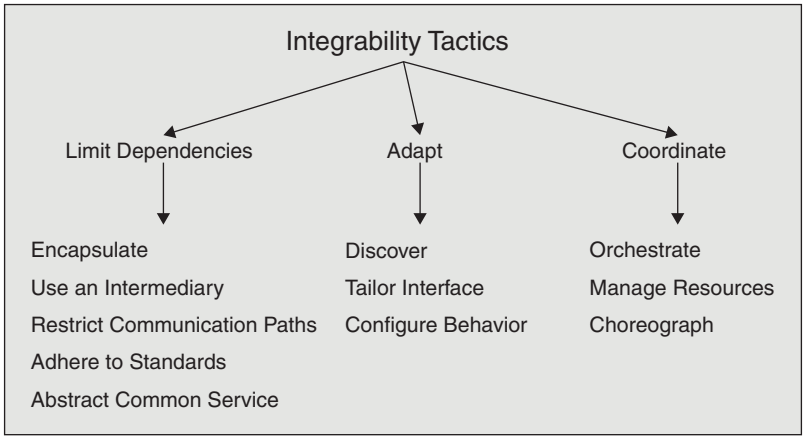

FIGURE 3.9 Integrability tactics categorization

The first category, *Limit Dependencies*, may remind you of the *Reduce Coupling* category of modifiability tactics. This is not surprising because limiting dependencies can be achieved, in large part, by reducing coupling. Within the *Limit Dependencies* category, the tactics are:

- *Encapsulate.* 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.
- *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 (e.g., syntactic, behavior, data semantic).
- *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).
- *Adhere to standards.* Standardization in system implementations is a primary enabler of integrability and interoperability, across both platforms and vendors. Some standards focus on defining syntax and data semantics. Others include richer descriptions, such as those describing protocols that include behavioral and temporal semantics.
- *Abstract common services.* Where multiple elements provide services that are similar, it may be useful to hide them behind a common abstraction. This abstraction might be realized as a common interface implemented by them all, or it might involve an intermediary that translates requests for the abstract service to more specific requests. The resulting encapsulation hides the details of the elements from other components in the system.

Within the *Adapt* category, the tactics are:

- *Discover.* A discovery service is the mechanism by which applications and services locate each other. 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.
- *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.
- *Configure behavior.* 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).

Finally, within the *Coordinate* category, the tactics are:

- *Orchestrate.* This tactic uses a control mechanism to coordinate and manage the invocation of 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.
- *Choreograph.* Choreography is an alternative control mechanism in which a process is implemented as steps in a flow, each triggered by an event message upon completion of the previous step and unaware of the other steps in the flow. This results in composable, loosely coupled services without an orchestrator. Orchestration is centralized, whereas choreography is distributed.
- *Manage resources.* A resource manager is a 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 manager.

3.7.2 Integrability Patterns

Many patterns exist to support integrability. In this section, we briefly describe just a few of them. The first three are related—adapters, bridges, and mediators.

3.7.2.1 Adapters, Bridges, and Mediators

An adapter is a form of encapsulation whereby some component is encased within an alternative abstraction. An adapter is the only element allowed to use that component; every other piece of software uses the component's services by going through the adapter. The adapter transforms the data or control information for the component it wraps.

A bridge translates some "requires" assumptions of one arbitrary component to some "provides" assumptions of another component. The key difference between a bridge and an adapter 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.

Mediators exhibit properties of both bridges and adapters. 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.

All three of these patterns allow access to an element without forcing a change to the element or its interface. But these benefits, of course, come at a cost. Creating any of these patterns requires up-front development work. Moreover, all of the patterns introduce some performance overhead while accessing the element, although typically this overhead is small.

As a side note, a fourth pattern with a similar intent is Facade. We will discuss an example of the Facade pattern in Chapter 5, where it is instantiated as an API gateway.

3.7.2.2 Services

Another pattern often used to ease integrability concerns is services. A service (whether "micro" or not) is an independent decoupled software component that can be developed, deployed, and scaled independently. The Services pattern describes a collection of distributed components that provide and consume services. Components have interfaces that describe the services that they request and that they provide. A service's quality attributes can be specified and guaranteed with a service level agreement (SLA). Components perform their computations by requesting services from one another.

The benefits of the Services pattern are that services are designed to be used by a variety of clients, leading them to be more generic. Also, services are independent: The only method for accessing a service is through its interface and through asynchronous messages over a network. This means that services typically have loose coupling with other services and with their environment. Finally, services can be implemented heterogeneously, using whatever languages and technologies are most appropriate.

However, service-based architectures, because of their heterogeneity and distinct ownership, come with many interoperability mechanisms such as Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP). This adds some up-front complexity and overhead. (We will discuss APIs and API-centric design in Chapter 5.)

3.7.2.3 Dynamic Discovery

Dynamic discovery applies the discover tactic to support finding service providers at runtime. A dynamic discovery capability sets the expectation that the system will advertise the services available for integration and the information that will be available for each service.

The main benefit of this pattern is that it allows for flexibility in binding services together into a cooperating whole. For example, services may be chosen at startup or dynamically at runtime based on their pricing or availability or other properties. The cost of this flexibility is that dynamic discovery registration and de-registration must be automated, and tools for this purpose must be acquired or generated.

3.8 Summary

This chapter, though quite long, barely scratched the surface of making design decisions. Even so, it identified some key principles for making design decisions—a way to reason rigorously about decision-making. In addition, we provided several examples of the design concepts (patterns and tactics) that serve as the building blocks of design. We hope that you are now prepared to examine other quality attributes, or other patterns for the quality attributes discussed here, in exactly the same way as demonstrated in this chapter. The ones we presented here are merely the most common ones, but they are not special in any other way.

Technologies (externally developed components) are also design concepts, though we did not discuss these extensively here. In subsequent chapters of this book (for example, Chapter 7, where we discuss cloud-based solutions), we will explore other specific design concerns such as technology choices.

This is all good news for the architect. Design, it turns out, is not mysterious. The design decisions that you can make for quality attributes encompass a substantial body of knowledge, but it is both tractable and learnable.

3.9 Further Reading

The nine decision-making principles presented here were first described in A. Tang and R. Kazman, "Decision-Making Principles for Better Software Design Decisions", *IEEE Software*, 38, November–December 2021.

There are many references for design and architecture patterns; we identified a few of these in Chapter 2. Among the best known are the "original" design patterns book: E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* by Addison-Wesley, 1994; and the book by F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, 1996. The latter book was the beginning of a series that eventually totaled five volumes. But the set of patterns is never complete; new patterns continue to be created all the time. For example, you can find a catalog of microservice patterns here: https://microservices.io/patterns/.

You can read about the tactics presented here, and other tactics, in greater detail, by L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th edition, Addison-Wesley, 2021.

The SWEBOK summarizes a set of nine software design principles: Hironori Washizaki, ed., *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, Version 4.0, IEEE Computer Society, 2024; www.swebok.org.

Several online catalogs of reference architectures have been developed by different cloud providers. They can be found at the following links:

- https://learn.microsoft.com/en-us/azure/architecture/browse/
- https://aws.amazon.com/architecture/
- https://cloud.google.com/architecture

3.10 Discussion Questions

1. Humans are inevitably biased. How do the nine decision-making principles help to combat bias? How might you use these principles in practice?
2. If you had to design an architecture for a quality attribute driver that was not listed in this chapter, how would you go about determining (and validating) the tactics and patterns, including their costs, benefits, and tradeoffs?
3. How could you use a tactic to modify and improve a design pattern? Give a concrete example.
4. What is the relationship between tactics, patterns, and externally developed components? How can you use this knowledge to select and analyze components? Give an example.
5. Tactics are foundational, but the categorization can never be said to be "done". New designs and design approaches continually emerge over time. Can you identify a tactic that was not included in the categorizations presented here?

This page intentionally left blank

Index

Numerics

2-tier deployment, 124
4-tier deployment, 125

A

abstraction, common, 48, 54
active redundancy, 44
AD&D Hotels. *See* HPS (Hotel Pricing System)
Adapt tactics, 55
adapter pattern, 55
ADD (Attribute-Driven Design), 7, 165, 274.
    See also digital twinplatform; HPS (Hotel Pricing System); performance
  allocate responsibilities, 64
  analyze the current design, 65, 163–165
  API-centric design, 109
  applying to different system contexts, 66
  and architect roles, 266
  associating responsibilities, 76
  brownfield, 69
  choose design concepts that satisfy the selected drivers, 63
    API-centric design, 108
    digital twin platform, 197–198, 216–217
    HPS (Hotel Pricing System), 159–160
  choose elements of the system to refine, 63
    API-centric design, 108
    HPS (Hotel Pricing System), 159
  cloud-based solutions, 147–148
  decisions, recording, 65
  and deployability, 128–129
  design concepts
    alternatives, 71
    identifying, 70–71
    selection, 71–72, 73–74
  design to replace a legacy application, 69–70
  elements, instantiation, 75–76
  establish iteration goal by selecting drivers, 62
    API-centric design, 108
    digital twin platform, 197
    HPS (Hotel Pricing System), 159
  establishing relationships between elements, 76–77
  greenfield
    mature domain, 66–68
    novel domain, 68–69
  instantiate architectural elements, 63–64, 160–161
    API-centric design, 108
    digital twin platform, 198–199
  interface
    defining, 64, 77, 108
    external, 77
    identification, 80–81
    internal, 78, 80
  iteration, 65
  Kanban board, 86–87
  properties, identifying, 76
  recording design decisions, 84–85
  review inputs, 60–62
    API-centric design, 108
    digital twin platform, 196
    HPS (Hotel Pricing System), 158–159
  review the design goals and purpose, 65
  roadmap, 68
  sketch views and record design decisions, 64
    API-centric design, 109
    digital twin platform, 220–222
    HPS (Hotel Pricing System), 162–163, 168–172
  structures, producing, 74–75
  and technical debt, 233–235
  tracking design progress, 85
  version 2.0, 7
  version 2.5, 7
  version 3.0, 7
Agile/agility, 5, 16, 91
  in architecture, 258
  business, 91
    drivers. *See* drivers, business agility
    focus on products, 92
  design approaches, 257–259
  Manifesto, 16

agreements, 16–17
AI, 208
  architect, 265
  digital twin platform
    constraints, 213–214
    design concepts, 216–217
    use case/s, 212
  generative, 220
algorithm, selection, 48–49
allocation structures, 75
alternatives, design concept, 71
Amazon QuickSight, 218
Amazon Timestream, 218
analysis, 239
  ATAM (architecture tradeoff analysis method), 247–249
  back-of-the-envelope, 241
  cost/benefit, 72–73
  current design, 65
  definition, 6
  and design, 239–241
  lightweight ATAM, 247–249
  reasons for, 242
  reflective questions, 246–247
  scenario-based design reviews, 247
  tactics-based, 244–246
  techniques, 243
analytical skills, 6
analytic/s
  cloud capabilities, 142
  models, 241
anchoring bias, 31–32
Apache Kafka, 161, 166
API/s
  asynchronous, 102–103
  business, 94
  -centric design, 93–94, 109
    Back-End for Front-End pattern, 103
    choose design concepts that satisfy the selected drivers, 108
    choose one or more elements of the system to refine, 108
    design of an implementation mechanism, 109
    establish iteration goal by selecting drivers, 108
    implementing APIs, 105
    instantiate architectural element, 108
    interface, defining, 108
    layering, 103–104
    review inputs, 108
    sketch views and record design decisions, 109
  and composability, 94–95
  experience, 104
  -first design, 95
    documentation, 99
    error management, 96
    granularity, 96
    methods, 99–100
    operations, 99–100
    parameters, 100
    principles, 96
    purpose of API, 95
    return values, 100
    security, 97
    specifying an API, 97–99
    supporting API evolution, 96
  gateway, 107, 161
  Gateway pattern, 104
  management, 105–106
    analytics, 107
    API testing, 107
    developer portal, 106–107
    policies, 107
  platform, 94, 109–110
  process, 104
  prototype, 109
  query-oriented, 102
  REST (Representational State Transfer), 101, 167
  RPC (Remote Procedure Call), 101–102
  security, 161
  system, 104
  versioning, 96
  WebSocket, 103
application
  cloud-native, 138
  framework, 38
  legacy, ADD (Attribute-Driven Design), 69–70
  platform, 38
  Rich Internet, 160
  service, 33
  web, 140
architect, 265–266

AI/ML, 265
blockchain, 265
data, 265
infrastructure, 264–265
role of, 6, 266
security, 265
software, 264
architecture/architectural, 2–3, 11–12. *See also* element/s; microservice/s
agility, 258–259
agreements, 16–17
backlog, 85–86
concerns, 24–25
HPS, development and operations requirements, 157
HPS, system requirements, 155
constraints, 25–26
decisions, 13–14, 15, 16
deployability, 117
design, 3–4, 13
element interaction, 14
element internals, 14–15
importance of, 16–17
iterations, 60
prototype, 17, 18
rounds, 60
documentation, 4
drivers, 12, 17, 21, 61–62
architectural concerns, 24–25
constraints, 25–26
design purpose, 17–18
primary functionality, 23–24
quality attributes, 18–20
evaluation, 4, 249
groups, 268
implementation, 4
intentional, 259
Lambda, 216
microservice, 121–122
monolithic, 122
patterns, 35
performance, 40–41
throttling, 41
plan, 21
refactoring, 69
reference, 13, 25, 33–35, 75
cloud-based, 35, 145
digital twin platform, 196–202
requirements, 3–4
structures, 63
style, 35
tension between emergence and planning, 259–260
ARID (active reviews for intermediate design), 81
artifact, 19, 141
ASR (architecturally significant requirements), 3–4
asset hierarchy service, 219
assumptions
explicit, 30
implicit, 30
ATAM (architecture tradeoff analysis method), 247–250
attack/s
man-in-the-middle, 51
reacting to, 52
recovery, 52
resisting, 51–52
authentication, 51
authorization, 51
automation, continuous deployment (CD), 113–114
availability, 41, 49, 179
cloud, 136
high, 44
patterns, 44
Circuit Breaker, 46
cold spare, 45
hot spare, 44
tri-modular redundancy, 45–46
warm spare, 45
tactics, 42
-based questions, 244–246, 280–283
cloud-based, 143–144
Detect Faults, 43
Prevent Faults, 44
Recover from Faults, 43–44

B

A/B testing, 127–128
backlog
architectural, 85–86
project, 231
back-of-the-envelope analysis, 241
BDUF (big design up front), 80, 257–258
behavior, component, 55

best-case scenario, 72
BFF (back-end for front-end) pattern, 103
bias, anchoring, 31–32
Big Data, 208
blockchain architect, 265
blue/green pattern, 126
blueprint, 4
Booch, Grady, 13
Books, Fred, 274
brainstorming, scenario, 21
bridge pattern, 55–56
brownfield, ADD (Attribute-Driven Design), 69
brownfield development, 66
bug, 231
business
  agility, 91
    drivers. *See* drivers, business agility
    focus on products, 92
  APIs, 94
  goals, 18

C

C&C (component and connector) structures, 75
canary testing, 127
CASE (computer-aided software engineering), 82
CBAM (cost/benefit analysis method), 72–73
CD (continuous deployment), 113–114, 116–117, 118
CDN (content delivery network), 140
checklists, 241
choreography, 55
CI (continuous integration), 116
CIA (confidentiality, integrity, and availability), 49
Circuit Breaker pattern, 46
Clements, Paul, "A Rational Design Process: How and Why to Fake It", 1–2
Client-Server pattern, 49
"clone and own", 228
cloud/cloud computing, 134
  ADD (Attribute-Driven Design), 147–148
  analytics capabilities, 142
  computing capabilities, 139
  constraints
    cloud native and vendor lock-in, 138–139
    cost optimization, 137
    legal, 139
  database capabilities, 141
  development and DevOps capabilities, 141
  elements, digital twin platform, 201
  FaaS (function as a service), 135
  hybrid, 134
  IaaS (infrastructure as a service), 134
  managed resources, 135–136
  networking capabilities, 140
  PaaS (platform as a service), 135
  patterns
    design, 145–147
    reference architecture, 145
  private, 134
  provider managed services, 167–168
  public, 134
  quality attributes
    availability, 136
    deployability, 137
    operability, 137
    performance and scalability, 136
    security, 137
  reference architecture, 35
  regional capabilities, 139
  SaaS (software as a service), 135
  security capabilities, 142
  service models, 134–135
  storage capabilities, 141
  tactics
    availability, 143–144
    deployability, 144
    operability, 144–145
    performance and scalability, 143
    security, 144–145
code
  "clone and own", 228
  repository, 141
cold spare, 45
communication, 6, 54
complexity, 2
component/s
  behavior, 55
  externally developed, 37–38, 76
  interface, 56
composability, 92–93, 94–95
computational overhead, 40
computing capabilities, cloud, 139

concepts
cloud capability
analytics, 142
computing, 140
database, 141
development and DevOps, 141
networking, 140
regional, 139
security, 142
storage, 141
design, 12, 33
choosing, 63, 71–72, 73–74
digital twin platform, 197–198
externally developed components, 37–38
HPS (Hotel Pricing System), 166, 185
IIoT (Industrial IoT), 204–207
instantiation, 75
patterns, 35
reference architecture, 33–35
tactics, 35–37
concerns
cross-cutting, 33
general, 24
improper separation of, 228
specific, 25
concurrency, 40
condition monitoring, 43, 143
cone of uncertainty, 254–255
confidentiality, 49, 51
conformance checking, 4
constraints, 32
architectural, 25–26, 74
cloud computing
cloud native and vendor lock-in, 138–139
cost optimization, 137
legal, 139
development and operations requirement, HPS (Hotel Pricing System), 157
digital twin platform
Big Data & AI/ML, 213–214
IIoT, 204
system requirement
digital twin platform, 195
HPS (Hotel Pricing System), 155
container, 140, 161
contexts, 30–31, 70–71
continuous deployment, 113–114
Control Resource Demand tactics, 40
Conway's law, 263
Coordinate tactics, 55
cost optimization, cloud computing, 137
CQRS (Command and Query Responsibility Segregation), 146, 160
credentials, 52
cross-cutting concerns, 33

D

data architect, 265
data lake, 142
data sources, digital twin platform, 210–211
data warehouse, 142
database
cloud-based, 141
nonrelational, 167
relational, 166
decisions/decision making, 29
"adequacy" approach, 240
architectural, 15, 16
"buy versus build" choice, 37
choosing externally developed components, 38
deployability, 119
functionality, 24
instantiation, 160–161, 180
digital twin platform, 198–199, 208
HPS (Hotel Pricing System), 185
principles
anticipate risks, 31
assign priorities, 31
check assumptions, 30
define time horizon, 31
design around constraints, 32
explore contexts, 30–31
generate multiple solution options, 31–32
use facts, 30
weigh the pros and cons, 32
recording, 65, 84–85
tactics, 35–37
decomposition, 159, 166
Defer Binding tactics, 48
dependency/ies, 48
inversion principle, 229
limiting, 54
microservice, 122

package, 120
tangled, 228
deployability, 93
and ADD, 128–129
cloud, 137
defining, 113–116
design decisions, 119
designing the system to support, 117–118
DevOps, 116
general scenario, 115–116
patterns, 121
A/B testing, 127–128
blue/green, 126
canary testing, 127
load-balanced cluster, 125–126
microservice architecture, 121–122
monoliths and modular monoliths, 122–123
rolling upgrade, 126–127
tactics, 119–120
-based questions, 283–284
Manage Deployment Pipeline, 120
deployment
continuous, 113–114
infrastructure, 118
N-tier, 123–125
patterns, 35, 75–76
pipeline, 113–114, 118
rollback, 117
design, 1–2, 11, 239. *See also* ADD (Attribute-Driven Design); concept/s, design;performance
Agile approaches, 257–259
and analysis, 239–241. *See also* analysis
API-centric, 93–94, 109
asynchronous APIs, 102–103
choose design concepts that satisfy the selected drivers, 108
greenfield API platform, 109–110
implementing APIs, 105
interface, defining, 108
layering, 103–104
queries, 102
REST (Representational State Transfer), 101
RPC APIs, 101–102
WebSockets, 103
API-first, 95
documentation, 99
error management, 96
granularity, 96
methods, 99–100
operations, 99–100
parameters, 100
purpose of API, 95
return values, 100
security, 97
specifying an API, 97–99
supporting API evolution, 96
architectural, 3–4, 13, 15, 16–17
Attribute-Driven. *See* ADD (Attribute-Driven Design)
Books on, 274
complex, 2
concepts, 12, 33
catalog, 267–268
choosing, 63, 71–72, 73–74
cloud-based, 139–143
digital twin platform, 197–198
HPS (Hotel Pricing System), 166, 185
identifying, 70–71
IIoT (Industrial IoT), 204–207
instantiation, 75
constraints, 25–26, 32, 74
contexts, 30–31
debt, 228–231
decision making, 29
“adequacy” approach, 240
anticipate risks, 31
check assumptions, 30
constraints, 32
define time horizon, 31
explore contexts, 30–31
generate multiple solution options, 31–32
use facts, 30
weigh the pros and cons, 32
detailed, 15
difficulty, 12
element interaction, 14
element internals, 14–15
emergent, 259
goals, 62
Herbert on, 274
high-level, 15
house, 11–12

integrability, 53–55
iteration, 60, 62, 64
modifiability, 46. *See also* modifiability
novel, 12, 18
patterns, 35, 36, 145–147
performing, 59–60
principles, 33
priority, 31
progress
digital twin platform, 201–202, 211–212, 222–223
HPS (Hotel Pricing System), 187–188
tracking, 85
purpose, 17–18, 60, 65
reference architecture, 33–35
role of the architect, 6
round, 60, 62
skill, 7, 12
software architecture, 12–13
tactics, 35–37, 120
team approach, 263–264
tradeoffs, 13
detailed design, 15
Detect Attacks tactics, 50–51
Detect Faults tactics, 43
developer portal, 106–107
development and operations requirements. *See also* software development
HPS (Hotel Pricing System), 156, 184–188
architectural concerns, 157
quality attribute scenarios, 156–157
DevOps, 116
CD (continuous deployment), 116–117, 118
cloud capabilities, 141
iteration 0 + spikes approach, 258–259
pipeline, 141
production environment, 116
diagram, 64
package, 186
sequence, 78–79, 173–174
difficulty, integration, 53
digital twin platform
analysis of current design, 222–223
Big Data & AI/ML
asset hierarchy service, 219
constraints, 213–214
design concepts, 216–217
edge and cloud technologies, 217
elements to refine, 215
generative AI, 220
historical data repository, 219
instantiation decisions, 217–220
use case/s, 212–213
business case, 192
constraints, 195
data sources, 210–211
design concepts, 197–198
design progress, 201–202, 211–212, 222–223
IIoT (Industrial IoT)
constraints, 204
design concepts, 204–207
drivers, 202–203
elements to refine, 204
instantiating architectural elements, 207–208
inputs, 196
instantiation decisions, 198–199
quality attribute scenarios, 194–195, 213–214
reference architecture and overall system structure, 196–202
cloud elements, 201
drivers, 197
edge elements, 200
system requirements, 193
use cases, 193–194
dimensions, scenario, 20
discovery
dynamic, 49
prototyping, 20
service, 55
DNS (domain name server), 140
Docker containers, 161
documentation
API, 99
architectural, 4
purpose, 83
scenario-based, 83–84
views, 81–83
domain model, HPS (Hotel Pricing System), 166, 168–169
drivers
architectural, 12, 17, 21, 61–62
design purpose, 17–18
digital twin platform, 197

primary functionality, 23–24
quality attributes, 18–20
business agility, 92
composability, 92–93
deployability, 93
HPS (Hotel Pricing System), 159, 165
iteration, 65
dynamic discovery, 49

E

edge elements, digital twin platform, 200
element/s
HPS (Hotel Pricing System), 175–176, 183
instantiation, 63–64, 75–76, 108, 160–161
interaction design, 14
interface
external, 77
identification, 80–81
internal, 78, 80
internals design, 14–15
refactoring, 69
refinement, 159
relationships between, 64, 76–77
responsibilities, 82
allocating, 64
associating, 76
emergence, 3, 257–258, 259–260
encapsulation, 47, 54
encryption, 51
energy efficiency, tactics-based questions, 284–286
enterprise architect, 266
environments
execution, 140
HPS (Hotel Pricing System), 156
integration, 114
production, 114, 116
staging, 114
escalating restart, 44
estimation process, 18, 80, 254
cone of uncertainty, 254–255
standard components technique, 256–257
evaluation, 4, 242, 249
event/s
priority, 40
response, 40
evolution
API, 96
software, 232
unplanned, 229
exception
detection, 43
handling, 43
prevention, 44
execution environments, 140
execution time, 40
experience
API, 104
leveraging, 71
experiments, 241, 259
explicit assumptions, 30
export-side microservice modules, HPS (Hotel Pricing System), 172
external interface, 77
externally developed components, 37–38, 76

F

FaaS (function as a service), 135
facts, 30
failure
retry, 43
system, 41
falsifiable hypotheses, 20
faults, 41
features, 16, 120
firewall, 142
framework, 38, 66–67
front-end, HPS (Hotel Pricing System), 160
full-stack framework, 38, 66–67
functional requirement, 19
functionality
decisions, 24
primary, 23–24, 153–154

G

gateway
API, 161
network, 140
general concerns, 24
generative AI, 220
goals, 13
business, 18
design, 62, 66–68

iteration, 159
reviewing, 65
short-term, 230
graceful degradation, 43
granularity
disintegration, 121
integration, 121–122
greenfield design/development, 63, 66
API platform, 109–110
mature domain, 66–68
novel domain, 68–69

H

heartbeat, 43
Hibernate, 38
high availability, 44
high-level design, 15
historical data repository, digital twin platform, 219
hot spare, 44
HPS (Hotel Pricing System)
addressing reliability and availability quality attributes, 179
analyze the current design, 183–184
changes in the infrastructure to support quality attributes, 181–183
communication failure with the channel management system, 181–183
design concepts, 180–183
drivers, 179
refinement, 179
business case, 151–153
deployment elements, 183
design concepts, 166, 185
design progress, 187–188
development and operations requirements, 156, 184–188
architectural concerns, 157
constraints, 157
quality attribute scenarios, 156–157
domain model, 166
environments, 156
establishing an overall system structure
analyze the current design, 163–165
choose design concepts that satisfy the selected drivers, 159–160
choose elements of the system to refine, 159
drivers, 159
front-end, 160
instantiate architectural element, 160–161
sketch views and record design decisions, 162–163
identify structures to support primary functionality, 165
analysis of current design, 177–179
choose elements of the system to refine, 165
drivers, 165
instantiate architectural element, 166–168
sketch views and record design decisions, 168–172
inputs, 158–159
instantiation decisions, 180, 185
methods, 175–176
modules
command-side, 170
query-side, 171
quality attribute scenarios, 154–155
software design process, 158
system requirements
architectural concerns, 155
constraints, 155
primary functionality, 153–154
quality attribute scenarios, 154–155
hybrid cloud, 134

I

IaaS (infrastructure as a service), 134
IDL (interface design language), 101–102
IIoT (Industrial IoT), digital twin platform
constraints, 204
design concepts, 204–207
drivers, 202–203
elements to refine, 204
instantiating architectural elements, 207–208
implementation
API, 109
architectural, 4
implicit assumptions, 30
improper separation of concerns, 228
Increase Cohesion tactics, 47
infrastructure
architect, 264–265
as code services, 141

deployment, 118
as a service, 134
inputs, digital twin platform, 196
instantiation
decisions
digital twin platform, 198–199, 208, 217–220
HPS (Hotel Pricing System), 180, 185
design concept, 75
element, 63–64, 75–76, 108, 160–161, 166–168
integrability/integration, 53
difficulty, 53
environment, 114
granularity, 121–122
patterns
adapter, 55
bridge, 55–56
dynamic discovery, 49
mediator, 56
services, 56
tactics, 53–54
Adapt, 55
Coordinate, 55
Limit Dependencies, 54
questionnaire, 287–288
integrity, 49, 51
intentional architecture, 259
Intercepting Validator pattern, 52–53
interface, 56
defining, 64, 77, 108
design language, 101–102
external, 77
identification, 80–81
internal, 78, 80
mismatch, 126–127
operations, 80
segregation principle, 229
tailoring, 55
intermediary, 47, 54
internal interface, 78, 80
internal requirements, 25
interview, tactics-based, 244–246
Intrusion Detection pattern, 53
issues, 25
iteration, 62, 65
0 + spikes approach, 258–259
goals, 62, 159
micro-, 64

J-K

JAD (Joint Application Design), 20
JRP (Joint Requirements Planning), 20

Kanban board, 86–87
Kinesis Data Analytics, 218

L

Lambda architecture, 216
layered pattern, 48
layers, 48, 64, 104
leadership, 6
learning from example, 1–2
legacy application, ADD (Attribute-Driven Design), 69–70
legal constraints, cloud computing, 139
Lehman's Laws of Software Evolution, 232
leveraging, experience, 71
life-cycle activities, software architecture, 3–5
lightweight ATAM (architecture tradeoff analysis method), 249–250
Liskov substitution principle, 229
Litmus Edge reference architecture, 206–207, 208
LLM (large language model), 275
load balanced cluster pattern, 125–126
load balancer pattern, 41

M

Manage Deployed System tactics, 120
Manage Deployment Pipeline tactics, 120
Manage Resources tactics, 40
managed relational database, 141
man-in-the-middle attack, 51
mature domain, greenfield system design, 66–68
mediator pattern, 56
in-memory cache, 141
message
integrity, 51
queue, 216, 217
methodology
ADD (Attribute-Driven Design). *See* ADD (Attribute-Driven Design)
Agile, 5, 16
cost/benefit analysis, 72–73
principled, 59

methods
API, 99–100
HPS (Hotel Pricing System), 175–176
need for, 273–275
micro-iteration, 64
microservice/s, 123, 160, 161
architecture, 121–122
container, 161
dependencies, 122
element internals design, 15
Mission Thread Workshop, 18
modifiability, 46
patterns, 48
Client-Server, 49
layered, 48
strategy, 48–49
tactics, 46–47
Defer Binding, 48
Increase Cohesion, 47
questionnaire, 288–289
Reduce Coupling, 47–48
modular monolith, 122–123
module/s
HPS (Hotel Pricing System)
command-side, 170
export-side, 172
query-side, 171
interface, identification, 80–81
structures, 75
view. sketching, 82
monitoring
condition, 43, 143
services, 141
monolithic architecture, 122
MVP (Minimum Viable Product), 92

N

negotiation, 6
networking capabilities, cloud, 140
non-full-stack framework, 38
nonprimary use case, 80
nonrelational database, 167
nonstop forwarding, 44
NoSQL database, 141
novel design, 12, 18
N-tier deployment, 123–125

O

OpenAPI specification, 97–99, 167
open-closed principle, 229
operability, cloud, 137, 144–145
operations, 80, 99–100
orchestration, 55
organizations/organizational
architecture groups, 268
architecture guidelines, 267
design concept catalog, 267–268
goals, 18

P

PaaS (platform as a service), 135
package dependencies, 120
parameters, API, 100
Parnas, David, “A Rational Design Process: How and Why to Fake It”, 1–2
passive redundancy, 45
pattern/s, 36, 75–76
API Gateway, 104
API Layering, 104
availability, 44
Circuit Breaker, 46
cold spare, 45
hot spare, 44
tri-modular redundancy, 45–46
warm spare, 45
Back-End for Front-End, 103
cloud/cloud computing
design, 145–147
reference architecture, 145
CQRS (Command and Query Responsibility Segregation), 160
deployability/deployment, 35, 121
A/B testing, 127–128
blue/green, 126
canary testing, 127
load-balanced cluster, 125–126
microservice architecture, 121–122
monoliths and modular monoliths, 122–123
N-tier deployment, 123–125
rolling upgrade, 126–127
design/architectural, 35

integrability
adapter, 55
bridge, 55–56
dynamic discovery, 49
mediator, 56
services, 56
modifiability, 48
Client-Server, 49
layered, 48
strategy, 48–49
performance, 40–41
load balancer, 37–41
throttling, 41
security
Intercepting Validator, 52–53
Intrusion Detection, 53
Strangler Fig, 69–70
performance, 39. *See also* availability
patterns, 40–41
load balancer, 41
throttling, 41
quality attribute, 136
tactics, 39
-based questions, 289–291
cloud-based, 143
Control Resource Demand, 40
Manage Resources, 40
performing design, 59–60
ping/echo, 43
planning, tension with emergence, 259–260
platform, 38. *See also* digital twin platform
API, 94
greenfield API, 109–110
as a service, 135
policies, 25, 107
portal, developer, 106–107
predictive model, 44
Prevent Faults tactics, 44
primary functionality, 23–24, 153–154
primary use case, 14
principles
Agile, 257–258
API-first design, 96
decision making
anticipate risks, 31
assign priorities, 31
check assumptions, 30
define time horizon, 31
design around constraints, 32
explore contexts, 30–31
generate multiple solution options, 31–32
use facts, 30
weigh the pros and cons, 32
design, 32–33
SOLID, 229
priority/ies
assigning, 31
scenario, 20, 22, 23
private cloud, 134
process API, 104
producing, structures, 74–75
production environment, 114, 116
products, 37–38, 92
project/s
backlog, 231
versus products, 92
proposal, 16
properties, identifying, 76
pros and cons, weighing, 32
prototype/prototyping, 17, 18, 241
API, 109
discovery, 20
throwaway, 73–74
public cloud, 134
purpose
API, 95
design, 17–18, 60, 65
of documentation, 83

Q

QAW (Quality Attribute Workshop), 18, 21–22
quality attribute/s, 18–20
availability, 136, 179
composability, 92–93
deployability, 93, 113–116
design principles, 33
digital twin platform, 194–195
falsifiable hypotheses, 20
HPS (Hotel Pricing System), 154–155
operability, 137
performance, 136
reliability, 179
scalability, 136, 179
scenarios, 19–20, 24, 25
digital twin platform, 213–214
HPS (Hotel Pricing System), 154–155

security, 137
query-oriented APIs, 102
query-side modules, HPS (Hotel Pricing System), 171
questions
reflective, 241, 246–247
tactics-based, 244–246
availability, 280–283
deployability, 283–284
energy efficiency, 284–286
integrability, 287–288
modifiability, 288–289
performance, 289–291
safety, 291–293
security, 293–296
testability, 296–297
usability, 297–299

R

reconfiguration, 43
recording design decisions, 84–85
Recover from Faults tactics, 43–44
recovery, 41, 52
Reduce Coupling tactics, 47–48
redundancy
active, 44
passive, 45
tri-modular, 45–46
redundant spare, 43, 44
refactoring, 69, 232–233, 260
reference architecture, 13, 25, 33–35, 75
cloud-based, 35, 145
digital twin platform, 196–202
Litmus Edge, 206–207, 208
refinement, element, 63, 68, 159
reflective questions, 241, 246–247
regional capabilities, cloud, 139
relational database, 141, 166
reliability, 179
requirement/s. *See also* development and operations requirements; system requirements
architecturally significant, 3–4
development and operations, HPS (Hotel Pricing System), 156–157, 184–188
elicitation techniques, 20
functional, 19
internal, 25
scenario, 20
system
digital twin platform, 193
HPS (Hotel Pricing System), 153–155
Resist Attacks tactics, 51–52
resource/s
managed, 135–136
manager, 55
scheduling, 40
responsibilities
associating, 76
command-side microservices module, 170
element, 76, 80, 82
improper separation of concerns, 228
redistribution, 47
REST (Representational State Transfer), 101
REST API, 167
restart, escalating, 44
retry, 43
return values, API, 100
reusability, 92–93
Rich Internet Application, 160
risk/s
anticipating, 31
managing, 242
ROI (return on investment), 72
roles, architect, 6
rollback, 43, 117, 120, 144
rolling upgrade, 126–127
round, design, 60, 62
RPC APIs, 101–102
RUP (Rational Unified Process), 260–261

S

SaaS (software as a service), 135
safety, tactics questionnaire, 291–293
sanity checking, 43
satisfice, 26
scalability, 39, 136, 179
scaled rollout, 120
scenario, 19
-based design review, 247
-based documentation, 83–84
best-case, 72
brainstorming, 21
consolidation, 21
dimensions, 20
priority, 20, 22, 23

quality attribute, 19–20, 24, 25
digital twin platform, 213–214
HPS (Hotel Pricing System), 154–155, 156–157
requirements, 20
worst-case, 72
scheduling, resources, 40
script deployment commands, 120
search and query services, 142
security, 49–50
API, 97, 161
architect, 265
cloud, 137, 142
patterns
Intercepting Validator, 52–53
Intrusion Detection, 53
tactics, 50
-based questions, 293–296
cloud-based, 144–145
Detect Attacks, 50–51
React to Attacks, 52
Recover from Attacks, 52
Resist Attacks, 51–52
self-service, 94, 106–107
self-test, 43
sequence diagram, 78–79
sequential development, 260–261
serverless execution environment, 140
service/s
application, 33
asset hierarchy, 219
cloud provider-managed, 167–168
discovery, 55
-level agreement (SLA), 56
mesh, 140
micro-, 160, 161. *See also* module/s
architecture, 121–122
versus monoliths, 117–123
monitoring, 141
pattern, 56
search and query, 142
visualization, 218
shadow mode, 43
Simon, Herbert, 274
single-responsibility principle, 229
sketching, views, 64, 81–83, 234
digital twin platform, 220–222
HPS (Hotel Pricing System), 162–163, 168–172
skill/s
architect, 6
design, 7, 12
smart camera, 208
"Smart Decisions" game, 276
software
bug, 231
evolution, 232
upgrade, 43
software architecture, 2, 264
design, 12–13
importance of, 2–3
layers, 48
life-cycle activities, 3–5
LLM (large language model), 275
refactoring, 232–233
Software Architecture in Practice, 2, 18
software development
Agile, 91, 257–259
deployability, 113–116
environments, 114
outcomes, 230
sequential phase approach, 260–261
testing strategy, 262–263
velocity, 230
water-Scrum-fall, 261
SOLID principles, 229
solution/s, 31–32
specific concerns, 25
specification, API, 97–99
split module, 47
Spring Framework, 167
staging environment, 114
standard components technique, 256–257
standardization, 54
state resynchronization, 44
stimulus/response, 19–20
storage capabilities, cloud, 141
Strangler Fig pattern, 69–70
Strategy pattern, 48–49
structures
allocation, 75
C&C, 75
module, 75
producing, 74–75

style, architectural, 35
Swagger, 167
system
  API, 104
  availability, 41
  composability, 92–93
  contexts, 70–71
  element interaction design, 14
  failure, 41
  greenfield
    mature domain, 66–68
    novel domain, 68–69
  scenario, 19
  separating entities, 51
  structures, producing, 74–75
system requirements
  digital twin platform, 193
  HPS (Hotel Pricing System)
    architectural concerns, 155
    constraints, 155
    primary functionality, 153–154
    quality attribute scenarios, 154–155

T

table, design decision, 85
tactics, 76. *See also* pattern/s
  availability, 42
    cloud-based, 143–144
    Detect Faults, 43
    Prevent Faults, 44
    questionnaire, 280–283
    Recover from Faults, 43–44
  -based analysis, 244–246
  deployability, 119–120
    cloud-based, 144
    Manage Deployment Pipeline, 120
    questionnaire, 283–284
  design, 35–37
  energy efficiency, questionnaire, 284–286
  integrability, 53–54
    Adapt, 55
    Coordinate, 55
    Limit Dependencies, 54
    questionnaire, 287–288
  modifiability, 46–47
    Defer Binding, 48
    Increase Cohesion, 47
    questionnaire, 288–289
    Reduce Coupling, 47–48
  operability, cloud-based, 144–145
  performance, 39
    cloud-based, 143
    Control Resource Demand, 40
    questionnaire, 289–291
  safety, questionnaire, 291–293
  security, 50
    cloud-based, 144–145
    Detect Attacks, 50–51
    questionnaire, 293–296
    React to Attacks, 52
    Recover from Attacks, 52
    Resist Attacks, 51–52
  testability, questionnaire, 296–297
  usability, 36–37, 297–299
tailoring an interface, 55
tangled dependencies, 228
team approach to design, 263–264
technical debt, 16, 227–228
  ADD and, 233–235
  in design, 228–231
technical skills, 6
temporal inconsistency, 126
test/ing
  API, 107
  A/B, 127–128
  canary, 127
  self-, 43
  strategy, 262–263
  tactics questionnaire, 296–297
thought experiment, 241
throttling pattern, 41
throwaway prototype, 73–74
time horizon, 31
timestamp, 43
tools
  analysis and debugging, 141
  API design and development, 106
  API management, 105–106
  CASE (computer-aided software engineering), 82
  deployment pipeline, 118
  Kanban board, 86–87
tracking, design progress, 85
tradeoffs, design, 13
transactions, 44
tri-modular redundancy, 45–46

U

unplanned evolution, 229
up-front work, 257–258
upgrade, rolling, 126–127
usability
  questionnaire, 297–299
  tactics, 36–37
use case/s
  digital twin platform, 192, 193–194
  HPS (Hotel Pricing System), 192
  nonprimary, 80
  primary, 14
user stories, 20, 167
utility, 72–73
Utility Tree, 18, 22–23

V

velocity, software, 230
vendor lock-in, 139
versioning, API, 96
views, 81
  sketching, 64, 81–83, 168–172, 234
    digital twin platform, 220–222
    HPS (Hotel Pricing System), 162–163
virtual machine, 140
visualization service, 218
voting, 43
VPN (virtual private network), 140

W-X-Y-Z

warm spare, 45
waterfall model, 257–258
water-Scrum-fall development, 261
web application execution environment, 140
WebSockets, 103
work
  requests, 40
  up-front, 257–258
workshop
  Mission Thread, 18
  Quality Attribute, 18
worst-case scenario, 72