O'REILLY®

Fundamentals of Software Architecture

An Engineering Approach

Mark Richards & Neal Ford

Praise for *Fundamentals of Software Architecture*

Neal and Mark aren't just outstanding software architects; they are also exceptional teachers. With Fundamentals of Software Architecture, they have managed to condense the sprawling topic of architecture into a concise work that reflects their decades of experience. Whether you're new to the role or you've been a practicing architect for many years, this book will help you be better at your job. I only wish they'd written this earlier in my career.

—Nathaniel Schutta, Architect as a Service, ntschutta.io

Mark and Neal set out to achieve a formidable goal—to elucidate the many, layered fundamentals required to excel in software architecture—and they completed their quest. The software architecture field continuously evolves, and the role requires a daunting breadth and depth of knowledge and skills. This book will serve as a guide for many as they navigate their journey to software architecture mastery.

—Rebecca J. Parsons, CTO, ThoughtWorks

Mark and Neal truly capture real world advice for technologists to drive architecture excellence. They achieve this by identifying common architecture characteristics and the trade-offs that are necessary to drive success.

—Cassie Shum, Technical Director, ThoughtWorks

Fundamentals of Software Architecture

An Engineering Approach

Mark Richards and Neal Ford

Beijing • Boston • Farnham • Sebastopol • Tokyo

Fundamentals of Software Architecture

by Mark Richards and Neal Ford

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Chris Guzikowski

Development Editors: Alicia Young and Virginia Wilson

Production Editor: Christopher Faucher

Copyeditor: Sonia Saruba

Proofreader: Amanda Kersey

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2020: First Edition

Revision History for the First Edition

- 2020-01-27: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492043454 for release details.

978-1-492-04345-4

[LSI]

Preface: Invalidating Axioms

Axiom

> *A statement or proposition which is regarded as being established, accepted, or self-evidently true.*

Mathematicians create theories based on axioms, assumptions for things indisputably true. Software architects also build theories atop axioms, but the software world is, well, *softer* than mathematics: fundamental things continue to change at a rapid pace, including the axioms we base our theories upon.

The software development ecosystem exists in a constant state of dynamic equilibrium: while it exists in a balanced state at any given point in time, it exhibits *dynamic* behavior over the long term. A great modern example of the nature of this ecosystem follows the ascension of containerization and the attendant changes: tools like Kubernetes didn't exist a decade ago, yet now entire software conferences exist to service its users. The software ecosystem changes chaotically: one small change causes another small change; when repeated hundreds of times, it generates a new ecosystem.

Architects have an important responsibility to question assumptions and axioms left over from previous eras. Many of the books about software architecture were written in an era that only barely resembles the current world. In fact, the authors believe that we must question fundamental axioms on a regular basis, in light of improved engineering practices, operational ecosystems, software development processes—everything that makes up the messy, dynamic equilibrium where architects and developers work each day.

Careful observers of software architecture over time witnessed an evolution of capabilities. Starting with the engineering practices of Extreme Programming, continuing with Continuous Delivery, the DevOps

revolution, microservices, containerization, and now cloud-based resources, all of these innovations led to new capabilities and trade-offs. As capabilities changed, so did architects' perspectives on the industry. For many years, the tongue-in-cheek definition of software architecture was "the stuff that's hard to change later." Later, the microservices architecture style appeared, where *change* is a first-class design consideration.

Each new era requires new practices, tools, measurements, patterns, and a host of other changes. This book looks at software architecture in modern light, taking into account all the innovations from the last decade, along with some new metrics and measures suited to today's new structures and perspectives.

The subtitle of our book is "An Engineering Approach." Developers have long wished to change software development from a *craft*, where skilled artisans can create one-off works, to an *engineering* discipline, which implies repeatability, rigor, and effective analysis. While software engineering still lags behind other types of engineering disciplines by many orders of magnitude (to be fair, software is a very young discipline compared to most other types of engineering), architects have made huge improvements, which we'll discuss. In particular, modern Agile engineering practices have allowed great strides in the types of systems that architects design.

We also address the critically important issue of *trade-off analysis*. As a software developer, it's easy to become enamored with a particular technology or approach. But architects must always soberly assess the good, bad, and ugly of every choice, and virtually nothing in the real world offers convenient binary choices—everything is a trade-off. Given this pragmatic perspective, we strive to eliminate value judgments about technology and instead focus on analyzing trade-offs to equip our readers with an analytic eye toward technology choices.

This book won't make someone a software architecture overnight—it's a nuanced field with many facets. We want to provide existing and burgeoning architects a good modern overview of software architecture and

its many aspects, from structure to soft skills. While this book covers well-known patterns, we take a new approach, leaning on lessons learned, tools, engineering practices, and other input. We take many existing axioms in software architecture and rethink them in light of the current ecosystem, and design architectures, taking the modern landscape into account.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`*Constant width*`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`***Constant width bold***`

Shows commands or other text that should be typed literally by the user.

`*Constant width italic*`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://fundamentalsofsoftwarearchitecture.com*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Fundamentals of Software Architecture* by Mark Richards and Neal Ford (O'Reilly). Copyright 2020 Mark Richards, Neal Ford, 978-1-492-04345-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/fundamentals-of-software-architecture*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Mark and Neal would like to thank all the people who attended our classes, workshops, conference sessions, user group meetings, as well as all the other people who listened to versions of this material and provided invaluable feedback. We would also like to thank the publishing team at O'Reilly, who made this as painless an experience as writing a book can be. We would also like to thank No Stuff Just Fluff director Jay Zimmerman for creating a conference series that allows good technical content to grow and spread, and all the other speakers whose feedback and tear-soaked shoulders we appreciate. We would also like to thank a few random oases of sanity-preserving and idea-sparking groups that have names like Pasty Geeks and the Hacker B&B.

Acknowledgments from Mark Richards

In addition to the preceding acknowledgments, I would like to thank my lovely wife, Rebecca. Taking everything else on at home and sacrificing the opportunity to work on your own book allowed me to do additional consulting gigs and speak at more conferences and training classes, giving me the opportunity to practice and hone the material for this book. You are the best.

Acknowledgments from Neal Ford

Neal would like to thank his extended family, ThoughtWorks as a collective, and Rebecca Parsons and Martin Fowler as individual parts of it. ThoughtWorks is an extraordinary group who manage to produce value for customers while keeping a keen eye toward why things work so that that we can improve them. ThoughtWorks supported this book in many myriad ways and continues to grow ThoughtWorkers who challenge and inspire every day. Neal would also like to thank our neighborhood cocktail club for

a regular escape from routine. Lastly, Neal would like to thank his wife, Candy, whose tolerance for things like book writing and conference speaking apparently knows no bounds. For decades she’s kept me grounded and sane enough to function, and I hope she will for decades more as the love of my life.

Chapter 1. Introduction

The job “software architect” appears near the top of numerous lists of best jobs across the world. Yet when readers look at the *other* jobs on those lists (like nurse practitioner or finance manager), there’s a clear career path for them. Why is there no path for software architects?

First, the industry doesn’t have a good definition of software architecture itself. When we teach foundational classes, students often ask for a concise definition of what a software architect does, and we have adamantly refused to give one. And we’re not the only ones. In his famous whitepaper “Who Needs an Architect?” Martin Fowler famously refused to try to define it, instead falling back on the famous quote:

> *Architecture is about the important stuff…whatever that is.*
>
> —Ralph Johnson

When pressed, we created the mindmap shown in Figure 1-1, which is woefully incomplete but indicative of the scope of software architecture. We will, in fact, offer our definition of software architecture shortly.

Second, as illustrated in the mindmap, the role of software architect embodies a massive amount and scope of responsibility that continues to expand. A decade ago, software architects dealt only with the purely technical aspects of architecture, like modularity, components, and patterns. Since then, because of new architectural styles that leverage a wider swath of capabilities (like microservices), the role of software architect has expanded. We cover the many intersections of architecture and the remainder of the organization in “Intersection of Architecture and…”.

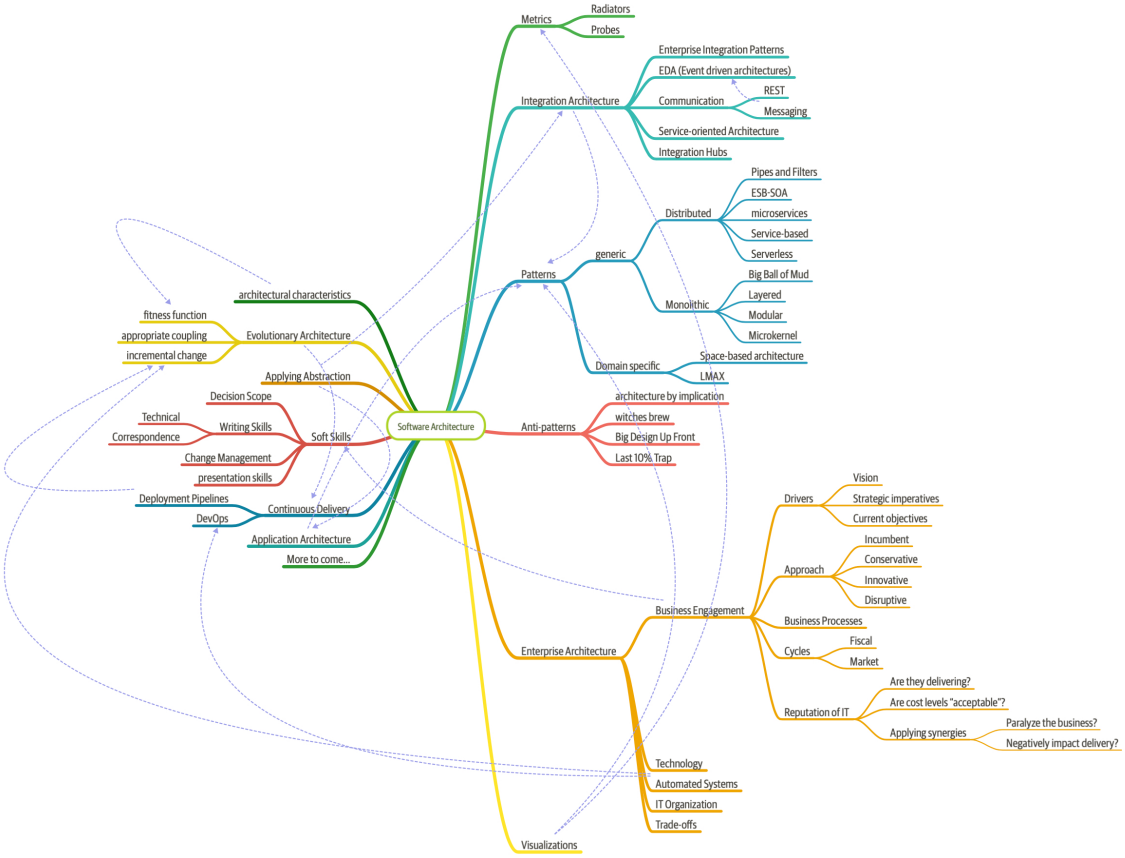

Figure 1-1. The responsibilities of a software architect encompass technical abilities, soft skills, operational awareness, and a host of others

Third, software architecture is a constantly moving target because of the rapidly evolving software development ecosystem. Any definition cast today will be hopelessly outdated in a few years. The Wikipedia definition of software architecture provides a reasonable overview, but many statements are outdated, such as “Software architecture is about making fundamental structural choices which are costly to change once implemented.” Yet architects designed modern architectural styles like microservices with the idea of incremental built in—it is no longer expensive to make structural changes in microservices. Of course, that capability means trade-offs with other concerns, such as coupling. Many books on software architecture treat it as a static problem; once solved, we can safely ignore it. However, we recognize the inherent dynamic nature of software architecture, including the definition itself, throughout the book.

Fourth, much of the material about software architecture has only historical relevance. Readers of the Wikipedia page won't fail to notice the bewildering array of acronyms and cross-references to an entire universe of knowledge. Yet, many of these acronyms represent outdated or failed attempts. Even solutions that were perfectly valid a few years ago cannot work now because the context has changed. The history of software architecture is littered with things architects have tried, only to realize the damaging side effects. We cover many of those lessons in this book.

Why a book on software architecture fundamentals now? The scope of software architecture isn't the only part of the development world that constantly changes. New technologies, techniques, capabilities…in fact, it's easier to find things that haven't changed over the last decade than to list all the changes. Software architects must make decisions within this constantly changing ecosystem. Because everything changes, including foundations upon which we make decisions, architects should reexamine some core axioms that informed earlier writing about software architecture. For example, earlier books about software architecture don't consider the impact of DevOps because it didn't exist when these books were written.

When studying architecture, readers must keep in mind that, like much art, it can only be understood in context. Many of the decisions architects made were based on realities of the environment they found themselves in. For example, one of the major goals of late 20th-century architecture included making the most efficient use of shared resources, because all the infrastructure at the time was expensive and commercial: operating systems, application servers, database servers, and so on. Imagine strolling into a 2002 data center and telling the head of operations "Hey, I have a great idea for a revolutionary style of architecture, where each service runs on its own isolated machinery, with its own dedicated database (describing what we now know as microservices). So, that means I'll need 50 licenses for Windows, another 30 application server licenses, and at least 50 database server licenses." In 2002, trying to build an architecture like microservices would be inconceivably expensive. Yet, with the advent of open source during the intervening years, coupled with updated engineering

practices via the DevOps revolution, we can reasonably build an architecture as described. Readers should keep in mind that all architectures are a product of their context.

Defining Software Architecture

The industry as a whole has struggled to precisely define "software architecture." Some architects refer to software architecture as the *blueprint* of the system, while others define it as the *roadmap* for developing a system. The issue with these common definitions is understanding what the blueprint or roadmap actually contains. For example, what is analyzed when an architect *analyzes* an architecture?

Figure 1-2 illustrates a way to think about software architecture. In this definition, software architecture consists of the *structure* of the system (denoted as the heavy black lines supporting the architecture), combined with *architecture characteristics* ("-ilities") the system must support, *architecture decisions*, and finally *design principles*.

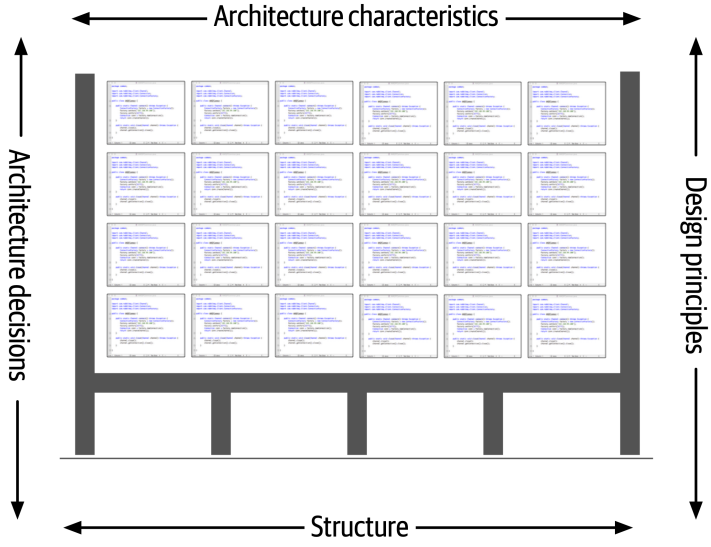

Figure 1-2. Architecture consists of the structure combined with architecture characteristics ("-ilities"), architecture decisions, and design principles

The *structure* of the system, as illustrated in Figure 1-3, refers to the type of architecture style (or styles) the system is implemented in (such as microservices, layered, or microkernel). Describing an architecture solely by the structure does not wholly elucidate an architecture. For example, suppose an architect is asked to describe an architecture, and that architect responds "it's a microservices architecture." Here, the architect is only talking about the *structure* of the system, but not the *architecture* of the system. Knowledge of the architecture characteristics, architecture decisions, and design principles is also needed to fully understand the architecture of the system.

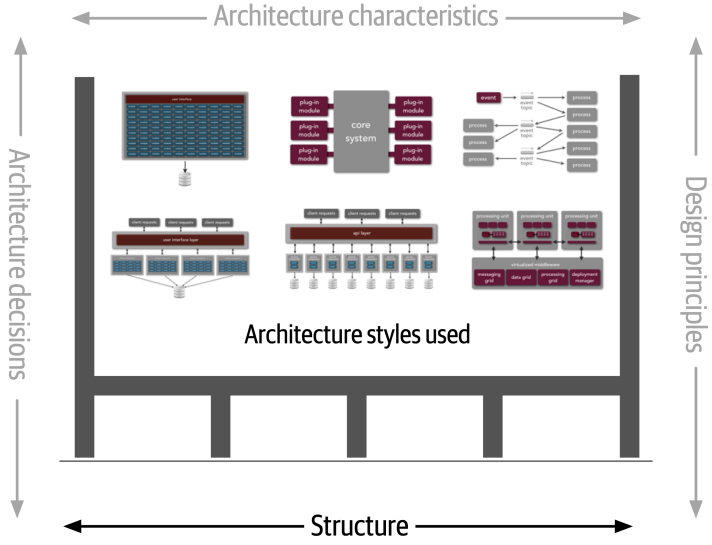

Figure 1-3. Structure refers to the type of architecture styles used in the system

Architecture characteristics are another dimension of defining software architecture (see Figure 1-4). The architecture characteristics define the success criteria of a system, which is generally orthogonal to the functionality of the system. Notice that all of the characteristics listed do not require knowledge of the functionality of the system, yet they are required in order for the system to function properly. Architecture characteristics are so important that we've devoted several chapters in this book to understanding and defining them.

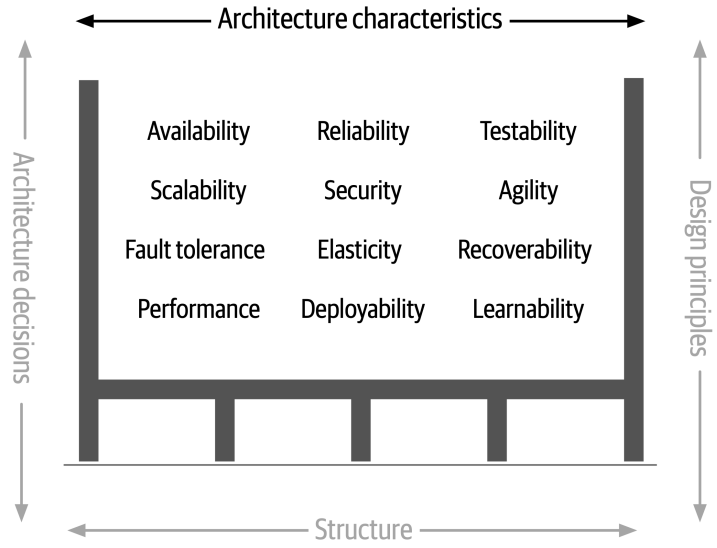

Figure 1-4. Architecture characteristics refers to the "-ilities" that the system must support

The next factor that defines software architecture is *architecture decisions*. Architecture decisions define the rules for how a system should be constructed. For example, an architect might make an architecture decision that only the business and services layers within a layered architecture can access the database (see Figure 1-5), restricting the presentation layer from making direct database calls. Architecture decisions form the constraints of the system and direct the development teams on what is and what isn't allowed.

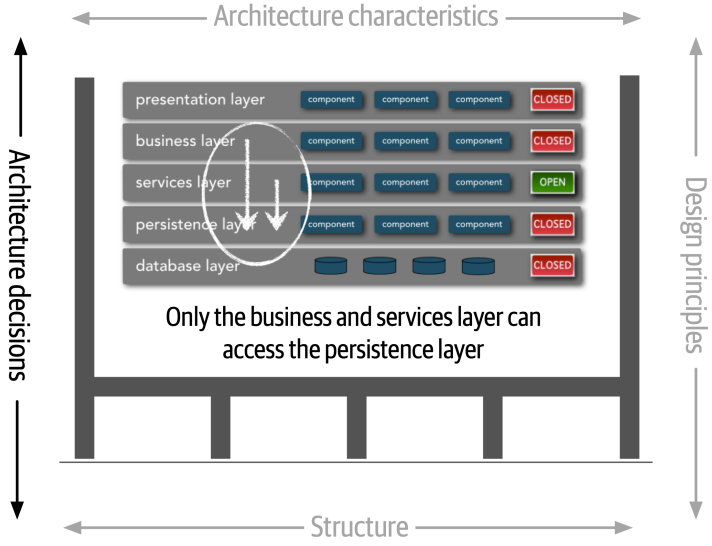

Figure 1-5. Architecture decisions are rules for constructing systems

If a particular architecture decision cannot be implemented in one part of the system due to some condition or other constraint, that decision (or rule) can be broken through something called a *variance*. Most organizations have variance models that are used by an architecture review board (ARB) or chief architect. Those models formalize the process for seeking a variance to a particular standard or architecture decision. An exception to a particular architecture decision is analyzed by the ARB (or chief architect if no ARB exists) and is either approved or denied based on justifications and trade-offs.

The last factor in the definition of architecture is *design principles*. A design principle differs from an architecture decision in that a design principle is a *guideline* rather than a hard-and-fast *rule*. For example, the design principle illustrated in Figure 1-6 states that the development teams should leverage asynchronous messaging between services within a microservices architecture to increase performance. An architecture decision (rule) could never cover every condition and option for communication between services, so a design principle can be used to provide guidance for the preferred method (in this case, asynchronous messaging) to allow the developer to choose a more appropriate communication protocol (such as REST or gRPC) given a specific circumstance.

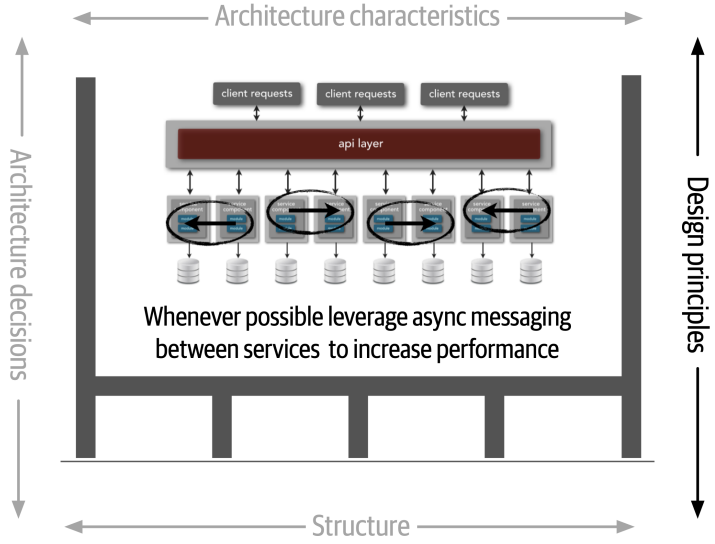

Figure 1-6. Design principles are guidelines for constructing systems

Expectations of an Architect

Defining the role of a software architect presents as much difficulty as defining software architecture. It can range from expert programmer up to defining the strategic technical direction for the company. Rather than waste time on the fool's errand of defining the role, we recommend focusing on the *expectations* of an architect.

There are eight core expectations placed on a software architect, irrespective of any given role, title, or job description:

- Make architecture decisions
- Continually analyze the architecture
- Keep current with latest trends
- Ensure compliance with decisions
- Diverse exposure and experience
- Have business domain knowledge
- Possess interpersonal skills

- Understand and navigate politics

The first key to effectiveness and success in the software architect role depends on understanding and practicing each of these expectations.

Make Architecture Decisions

An architect is expected to define the architecture decisions and design principles used to guide technology decisions within the team, the department, or across the enterprise.

Guide is the key operative word in this first expectation. An architect should *guide* rather than *specify* technology choices. For example, an architect might make a decision to use React.js for frontend development. In this case, the architect is making a technical decision rather than an architectural decision or design principle that will help the development team make choices. An architect should instead instruct development teams to use a *reactive-based framework for frontend web development*, hence guiding the development team in making the choice between Angular, Elm, React.js, Vue, or any of the other reactive-based web frameworks.

Guiding technology choices through architecture decisions and design principles is difficult. The key to making effective architectural decisions is asking whether the architecture decision is helping to *guide* teams in making the right technical choice or whether the architecture decision *makes* the technical choice for them. That said, an architect on occasion might need to make specific technology decisions in order to preserve a particular architectural characteristic such as scalability, performance, or availability. In this case it would be still considered an architectural decision, even though it specifies a particular technology. Architects often struggle with finding the correct line, so Chapter 19 is entirely about architecture decisions.

Continually Analyze the Architecture

An architect is expected to continually analyze the architecture and current technology environment and then recommend solutions for improvement.

This expectation of an architect refers to *architecture vitality*, which assesses how viable the architecture that was defined three or more years ago is *today*, given changes in both business and technology. In our experience, not enough architects focus their energies on continually analyzing existing architectures. As a result, most architectures experience elements of structural decay, which occurs when developers make coding or design changes that impact the required architectural characteristics, such as performance, availability, and scalability.

Other forgotten aspects of this expectation that architects frequently forget are testing and release environments. Agility for code modification has obvious benefits, but if it takes teams weeks to test changes and months for releases, then architects cannot achieve agility in the overall architecture.

An architect must holistically analyze changes in technology and problem domains to determine the soundness of the architecture. While this kind of consideration rarely appears in a job posting, architects must meet this expectation to keep applications relevant.

Keep Current with Latest Trends

An architect is expected to keep current with the latest technology and industry trends.

Developers must keep up to date on the latest technologies they use on a daily basis to remain relevant (and to retain a job!). An architect has an even more critical requirement to keep current on the latest technical and industry trends. The decisions an architect makes tend to be long-lasting and difficult to change. Understanding and following key trends helps the architect prepare for the future and make the correct decision.

Tracking trends and keeping current with those trends is hard, particularly for a software architect. In Chapter 24 we discuss various techniques and

resources on how to do this.

Ensure Compliance with Decisions

An architect is expected to ensure compliance with architecture decisions and design principles.

Ensuring compliance means that the architect is continually verifying that development teams are following the architecture decisions and design principles defined, documented, and communicated by the architect. Consider the scenario where an architect makes a decision to restrict access to the database in a layered architecture to only the business and services layers (and not the presentation layer). This means that the presentation layer must go through all layers of the architecture to make even the simplest of database calls. A user interface developer might disagree with this decision and access the database (or the persistence layer) directly for performance reasons. However, the architect made that architecture decision for a specific reason: to control change. By closing the layers, database changes can be made without impacting the presentation layer. By not ensuring compliance with architecture decisions, violations like this can occur, the architecture will not meet the required architectural characteristics ("-ilities"), and the application or system will not work as expected.

In Chapter 6 we talk more about measuring compliance using automated fitness functions and automated tools.

Diverse Exposure and Experience

An architect is expected to have exposure to multiple and diverse technologies, frameworks, platforms, and environments.

This expectation does not mean an architect must be an expert in every framework, platform, and language, but rather that an architect must at least be familiar with a variety of technologies. Most environments these days are heterogeneous, and at a minimum an architect should know how to

interface with multiple systems and services, irrespective of the language, platform, and technology those systems or services are written in.

One of the best ways of mastering this expectation is for the architect to stretch their comfort zone. Focusing only on a single technology or platform is a safe haven. An effective software architect should be aggressive in seeking out opportunities to gain experience in multiple languages, platforms, and technologies. A good way of mastering this expectation is to focus on technical breadth rather than technical depth. Technical breadth includes the stuff you know about, but not at a detailed level, combined with the stuff you know a lot about. For example, it is far more valuable for an architect to be familiar with 10 different caching products and the associated pros and cons of each rather than to be an expert in only one of them.

Have Business Domain Knowledge

An architect is expected to have a certain level of business domain expertise.

Effective software architects understand not only technology but also the business domain of a problem space. Without business domain knowledge, it is difficult to understand the business problem, goals, and requirements, making it difficult to design an effective architecture to meet the requirements of the business. Imagine being an architect at a large financial institution and not understanding common financial terms such as an average directional index, aleatory contracts, rates rally, or even nonpriority debt. Without this knowledge, an architect cannot communicate with stakeholders and business users and will quickly lose credibility.

The most successful architects we know are those who have broad, hands-on technical knowledge coupled with a strong knowledge of a particular domain. These software architects are able to effectively communicate with C-level executives and business users using the domain knowledge and language that these stakeholders know and understand. This in turn creates

a strong level of confidence that the software architect knows what they are doing and is competent to create an effective and correct architecture.

Possess Interpersonal Skills

An architect is expected to possess exceptional interpersonal skills, including teamwork, facilitation, and leadership.

Having exceptional leadership and interpersonal skills is a difficult expectation for most developers and architects. As technologists, developers and architects like to solve technical problems, not people problems. However, as Gerald Weinberg was famous for saying, “no matter what they tell you, it’s always a people problem.” An architect is not only expected to provide technical guidance to the team, but is also expected to lead the development teams through the implementation of the architecture. Leadership skills are at least half of what it takes to become an effective software architect, regardless of the role or title the architect has.

The industry is flooded with software architects, all competing for a limited number of architecture positions. Having strong leadership and interpersonal skills is a good way for an architect to differentiate themselves from other architects and stand out from the crowd. We’ve known many software architects who are excellent technologists but are ineffective architects due to the inability to lead teams, coach and mentor developers, and effectively communicate ideas and architecture decisions and principles. Needless to say, those architects have difficulties holding a position or job.

Understand and Navigate Politics

An architect is expected to understand the political climate of the enterprise and be able to navigate the politics.

It might seem rather strange talk about negotiation and navigating office politics in a book about software architecture. To illustrate how important and necessary negotiation skills are, consider the scenario where a developer makes the decision to leverage the strategy pattern to reduce the

overall cyclomatic complexity of a particular piece of complex code. Who really cares? One might applaud the developer for using such a pattern, but in almost all cases the developer does not need to seek approval for such a decision.

Now consider the scenario where an architect, responsible for a large customer relationship management system, is having issues controlling database access from other systems, securing certain customer data, and making any database schema change because too many other systems are using the CRM database. The architect therefore makes the decision to create what are called *application silos*, where each application database is only accessible from the application owning that database. Making this decision will give the architect better control over the customer data, security, and change control. However, unlike the previous developer scenario, this decision will also be challenged by almost everyone in the company (with the possible exception of the CRM application team, of course). Other applications need the customer management data. If those applications are no longer able to access the database directly, they must now ask the CRM system for the data, requiring remote access calls through REST, SOAP, or some other remote access protocol.

The main point is that *almost every decision an architect makes will be challenged*. Architectural decisions will be challenged by product owners, project managers, and business stakeholders due to increased costs or increased effort (time) involved. Architectural decisions will also be challenged by developers who feel their approach is better. In either case, the architect must navigate the politics of the company and apply basic negotiation skills to get most decisions approved. This fact can be very frustrating to a software architect, because most decisions made as a developer did not require approval or even a review. Programming aspects such as code structure, class design, design pattern selection, and sometimes even language choice are all part of the art of programming. However, an architect, now able to finally be able to make broad and important decisions, must justify and fight for almost every one of those decisions. Negotiation skills, like leadership skills, are so critical and necessary that

we’ve dedicated an entire chapter in the book to understanding them (see Chapter 23).

Intersection of Architecture and…

The scope of software architecture has grown over the last decade to encompass more and more responsibility and perspective. A decade ago, the typical relationship between architecture and operations was contractual and formal, with lots of bureaucracy. Most companies, trying to avoid the complexity of hosting their own operations, frequently outsourced operations to a third-party company, with contractual obligations for service-level agreements, such as uptime, scale, responsiveness, and a host of other important architectural characteristics. Now, architectures such as microservices freely leverage former solely operational concerns. For example, elastic scale was once painfully built into architectures (see Chapter 15), while microservices handled it less painfully via a liaison between architects and DevOps.

HISTORY: PETS.COM AND WHY WE HAVE ELASTIC SCALE

The history of software development contains rich lessons, both good and bad. We assume that current capabilities (like elastic scale) just appeared one day because of some clever developer, but those ideas were often born of hard lessons. Pets.com represents an early example of hard lessons learned. Pets.com appeared in the early days of the internet, hoping to become the Amazon.com of pet supplies. Fortunately, they had a brilliant marketing department, which invented a compelling mascot: a sock puppet with a microphone that said irreverent things. The mascot became a superstar, appearing in public at parades and national sporting events.

Unfortunately, management at Pets.com apparently spent all the money on the mascot, not on infrastructure. Once orders started pouring in, they weren't prepared. The website was slow, transactions were lost, deliveries delayed, and so on…pretty much the worst-case scenario. So bad, in fact, that the business closed shortly after its disastrous Christmas rush, selling the only remaining valuable asset (the mascot) to a competitor.

What the company needed was elastic scale: the ability to spin up more instances of resources, as needed. Cloud providers offer this feature as a commodity, but in the early days of the internet, companies had to manage their own infrastructure, and many fell victim to a previously unheard of phenomenon: too much success can kill the business. Pets.com and other similar horror stories led engineers to develop the frameworks that architects enjoy now.

The following sections delve into some of the newer intersections between the role of architect and other parts of an organization, highlighting new capabilities and responsibilities for architects.

Engineering Practices

Traditionally, software architecture was separate from the development process used to create software. Dozens of popular methodologies exist to build software, including Waterfall and many flavors of Agile (such as Scrum, Extreme Programming, Lean, and Crystal), which mostly don’t impact software architecture.

However, over the last few years, engineering advances have thrust process concerns upon software architecture. It is useful to separate software development *process* from *engineering practices*. By *process*, we mean how teams are formed and managed, how meetings are conducted, and workflow organization; it refers to the mechanics of how people organize and interact. Software *engineering* practices, on the other hand, refer to process-agnostic practices that have illustrated, repeatable benefit. For example, continuous integration is a proven engineering practice that doesn’t rely on a particular process.

THE PATH FROM EXTREME PROGRAMMING TO CONTINUOUS DELIVERY

The origins of Extreme Programming (XP) nicely illustrate the difference between *process* and *engineering*. In the early 1990s, a group of experienced software developers, led by Kent Beck, started questioning the dozens of different development processes popular at the time. In their experience, it seemed that none of them created repeatably good outcomes. One of the XP founders said that choosing one of the extant processes was "no more guarantee of project success than flipping a coin." They decided to rethink how to build software, and they started the XP project in March of 1996. To inform their process, they rejected the conventional wisdom and focused on the *practices* that led to project success in the past, pushed to the extreme. Their reasoning was that they'd seen a correlation on previous projects between more tests and higher quality. Thus, the XP approach to testing took the practice to the extreme: do test-first development, ensuring that all code is tested before it enters the code base.

XP was lumped into other popular Agile processes that shared similar perspectives, but it was one of the few methodologies that included engineering practices such as automation, testing, continuous integration, and other concrete, experienced-based techniques. The efforts to continue advancing the engineering side of software development continued with the book *Continuous Delivery* (Addison-Wesley Professional)—an updated version of many XP practices—and came to fruition in the DevOps movement. In many ways, the DevOps revolution occurred when operations adopted engineering practices originally espoused by XP: automation, testing, declarative single source of truth, and others.

We strongly support these advances, which form the incremental steps that will eventually graduate software development into a proper engineering discipline.

Focusing on engineering practices is important. First, software development lacks many of the features of more mature engineering disciplines. For example, civil engineers can predict structural change with much more accuracy than similarly important aspects of software structure. Second, one of the Achilles heels of software development is estimation—how much time, how many resources, how much money? Part of this difficulty lies with antiquated accounting practices that cannot accommodate the exploratory nature of software development, but another part is because we're traditionally bad at estimation, at least in part because of *unknown unknowns*.

> *…because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know.*
>
> —Former United States Secretary of Defense Donald Rumsfeld

Unknown unknowns are the nemesis of software systems. Many projects start with a list of *known unknowns*: things developers must learn about the domain and technology they know are upcoming. However, projects also fall victim to *unknown unknowns*: things no one knew were going to crop up yet have appeared unexpectedly. This is why all "Big Design Up Front" software efforts suffer: architects cannot design for unknown unknowns. To quote Mark (one of your authors):

> *All architectures become iterative because of unknown unknowns, Agile just recognizes this and does it sooner.*

Thus, while process is mostly separate from architecture, an iterative process fits the nature of software architecture better. Teams trying to build a modern system such as microservices using an antiquated process like Waterfall will find a great deal of friction from an antiquated process that ignores the reality of how software comes together.

Often, the architect is also the technical leader on projects and therefore determines the engineering practices the team uses. Just as architects must

carefully consider the problem domain before choosing an architecture, they must also ensure that the architectural style and engineering practices form a symbiotic mesh. For example, a microservices architecture assumes automated machine provisioning, automated testing and deployment, and a raft of other assumptions. Trying to build one of these architectures with an antiquated operations group, manual processes, and little testing creates tremendous friction and challenges to success. Just as different problem domains lend themselves toward certain architectural styles, engineering practices have the same kind of symbiotic relationship.

The evolution of thought leading from Extreme Programming to Continuous Delivery continues. Recent advances in engineering practices allow new capabilities within architecture. Neal's most recent book, *Building Evolutionary Architectures* (O'Reilly), highlights new ways to think about the intersection of engineering practices and architecture, allowing better automation of architectural governance. While we won't summarize that book here, it gives an important new nomenclature and way of thinking about architectural characteristics that will infuse much of the remainder of this book.

Neal's book covers techniques for building architectures that change gracefully over time. In Chapter 4, we describe architecture as the combination of requirements and additional concerns, as illustrated in Figure 1-7.

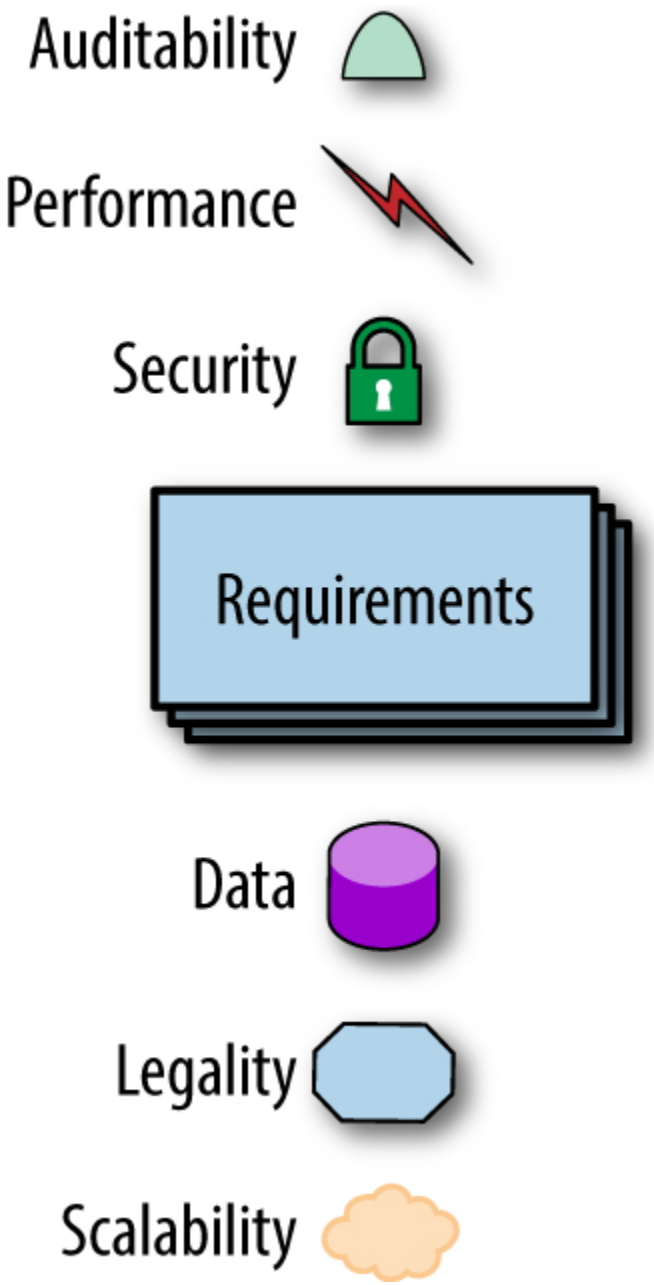

Figure 1-7. The architecture for a software system consists of both requirements and all the other architectural characteristics

As any experience in the software development world illustrates, nothing remains static. Thus, architects may design a system to meet certain criteria, but that design must survive both implementation (how can architects make sure that their design is implemented correctly) and the inevitable change driven by the software development ecosystem. What we need is an *evolutionary architecture*.

Building Evolutionary Architectures introduces the concept of using *fitness functions* to protect (and govern) architectural characteristics as change occurs over time. The concept comes from evolutionary computing. When designing a genetic algorithm, developers have a variety of techniques to mutate the solution, evolving new solutions iteratively. When designing such an algorithm for a specific goal, developers must measure the outcome to see if it is closer or further away from an optimal solution; that measure is a fitness function. For example, if developers designed a genetic algorithm to solve the traveling salesperson problem (whose goal is the shortest route between various cities), the fitness function would look at the path length.

Building Evolutionary Architectures co-opts this idea to create *architectural fitness functions*: an objective integrity assessment of some architectural characteristic(s). This assessment may include a variety of mechanisms, such as metrics, unit tests, monitors, and chaos engineering. For example, an architect may identify page load time as an importance characteristic of the architecture. To allow the system to change without degrading performance, the architecture builds a fitness function as a test that measures page load time for each page and then runs the test as part of the continuous integration for the project. Thus, architects always know the status of critical parts of the architecture because they have a verification mechanism in the form of fitness functions for each part.

We won’t go into the full details of fitness functions here. However, we will point out opportunities and examples of the approach where applicable. Note the correlation between how often fitness functions execute and the feedback they provide. You’ll see that adopting Agile engineering practices such as continuous integration, automated machine provisioning, and similar practices makes building resilient architectures easier. It also illustrates how intertwined architecture has become with engineering practices.

Operations/DevOps

The most obvious recent intersection between architecture and related fields occurred with the advent of DevOps, driven by some rethinking of architectural axioms. For many years, many companies considered operations as a separate function from software development; they often outsource operations to another company as a cost-saving measure. Many architectures designed during the 1990s and 2000s assumed that architects couldn't control operations and were built defensively around that restriction (for a good example of this, see Space-Based Architecture in Chapter 15).

However, a few years ago, several companies started experimenting with new forms of architecture that combine many operational concerns with the architecture. For example, in older-style architectures, such as ESB-driven SOA, the architecture was designed to handle things like elastic scale, greatly complicating the architecture in the process. Basically, architects were forced to defensively design around the limitations introduced because of the cost-saving measure of outsourcing operations. Thus, they built architectures that could handle scale, performance, elasticity, and a host of other capabilities internally. The side effect of that design was vastly more complex architecture.

The builders of the microservices style of architecture realized that these operational concerns are better handled by operations. By creating a liaison between architecture and operations, the architects can simplify the design and rely on operations for the things they handle best. Thus, realizing a misappropriation of resources led to accidental complexity, and architects and operations teamed up to create microservices, the details of which we cover in Chapter 17.

Process

Another axiom is that software architecture is mostly orthogonal to the software development process; the way that you build software (*process*) has little impact on the software architecture (*structure*). Thus, while the

software development process a team uses has some impact on software architecture (especially around engineering practices), historically they have been thought of as mostly separate. Most books on software architecture ignore the software development process, making specious assumptions about things like predictability. However, the process by which teams develop software has an impact on many facets of software architecture. For example, many companies over the last few decades have adopted Agile development methodologies because of the nature of software. Architects in Agile projects can assume iterative development and therefore a faster feedback loop for decisions. That in turn allows architects to be more aggressive about experimentation and other knowledge that relies on feedback.

As the previous quote from Mark observes, all architecture becomes iterative; it's only a matter of time. Toward that end, we're going assume a baseline of Agile methodologies throughout and call out exceptions where appropriate. For example, it is still common for many monolithic architectures to use older processes because of their age, politics, or other mitigating factors unrelated to software.

One critical aspect of architecture where Agile methodologies shine is restructuring. Teams often find that they need to migrate their architecture from one pattern to another. For example, a team started with a monolithic architecture because it was easy and fast to bootstrap, but now they need to move it to a more modern architecture. Agile methodologies support these kinds of changes better than planning-heavy processes because of the tight feedback loop and encouragement of techniques like the Strangler Pattern and feature toggles.

Data

A large percentage of serious application development includes external data storage, often in the form of a relational (or, increasingly, NoSQL) database. However, many books about software architecture include only a light treatment of this important aspect of architecture. Code and data have a symbiotic relationship: one isn't useful without the other.

Database administrators often work alongside architects to build data architecture for complex systems, analyzing how relationships and reuse will affect a portfolio of applications. We won't delve into that level of specialized detail in this book. At the same time, we won't ignore the existence and dependence on external storage. In particular, when we talk about the operational aspects of architecture and *architectural quantum* (see Chapter 3), we include important external concerns such as databases.

Laws of Software Architecture

While the scope of software architecture is almost impossibly broad, unifying elements do exist. The authors have first and foremost learned the *First Law of Software Architecture* by constantly stumbling across it:

> *Everything in software architecture is a trade-off.*
>
> —First Law of Software Architecture

Nothing exists on a nice, clean spectrum for software architects. Every decision must take into account many opposing factors.

> *If an architect thinks they have discovered something that isn't a trade-off, more likely they just haven't identified the trade-off yet.*
>
> —Corollary 1

We define software architecture in terms beyond structural scaffolding, incorporating principles, characteristics, and so on. Architecture is broader than just the combination of structural elements, reflected in our *Second Law of Software Architecture*:

> *Why is more important than how.*
>
> —Second Law of Software Architecture

The authors discovered the importance of this perspective when we tried keeping the results of exercises done by students during workshop as they crafted architecture solutions. Because the exercises were timed, the only artifacts we kept were the diagrams representing the topology. In other

words, we captured *how* they solved the problem but not *why* the team made particular choices. An architect can look at an existing system they have no knowledge of and ascertain how the structure of the architecture works, but will struggle explaining why certain choices were made versus others.

Throughout the book, we highlight *why* architects make certain decisions along with trade-offs. We also highlight good techniques for capturing important decisions in "Architecture Decision Records".

Part I. Foundations

To understand important trade-offs in architecture, developers must understand some basic concepts and terminology concerning components, modularity, coupling, and connascence.

Chapter 2. Architectural Thinking

An architect sees things differently from a developer's point of view, much in the same way a meteorologist might see clouds differently from an artist's point of view. This is called *architectural thinking*. Unfortunately, too many architects believe that architectural thinking is simply just "thinking about the architecture," as depicted in Figure 2-1.

Figure 2-1. Architectural thinking (iStockPhoto)

Architectural thinking is much more than that. It is seeing things with an architectural eye, or an architectural point of view. There are four main aspects of thinking like an architect. First, it's understanding the difference between architecture and design and knowing how to collaborate with development teams to make architecture work. Second, it's about having a

wide breadth of technical knowledge while still maintaining a certain level of technical depth, allowing the architect to see solutions and possibilities that others do not see. Third, it's about understanding, analyzing, and reconciling trade-offs between various solutions and technologies. Finally, it's about understanding the importance of business drivers and how they translate to architectural concerns.

In this chapter we explore these four aspects of thinking like an architect and seeing things with an architectural eye.

Architecture Versus Design

The difference between architecture and design is often a confusing one. Where does architecture end and design begin? What responsibilities does an architect have versus those of a developer? Thinking like an architect is knowing the difference between architecture and design and seeing how the two integrate closely to form solutions to business and technical problems.

Consider Figure 2-2, which illustrates the traditional responsibilities an architect has, as compared to those of a developer. As shown in the diagram, an architect is responsible for things like analyzing business requirements to extract and define the architectural characteristics ("-ilities"), selecting which architecture patterns and styles would fit the problem domain, and creating components (the building blocks of the system). The artifacts created from these activities are then handed off to the development team, which is responsible for creating class diagrams for each component, creating user interface screens, and developing and testing source code.

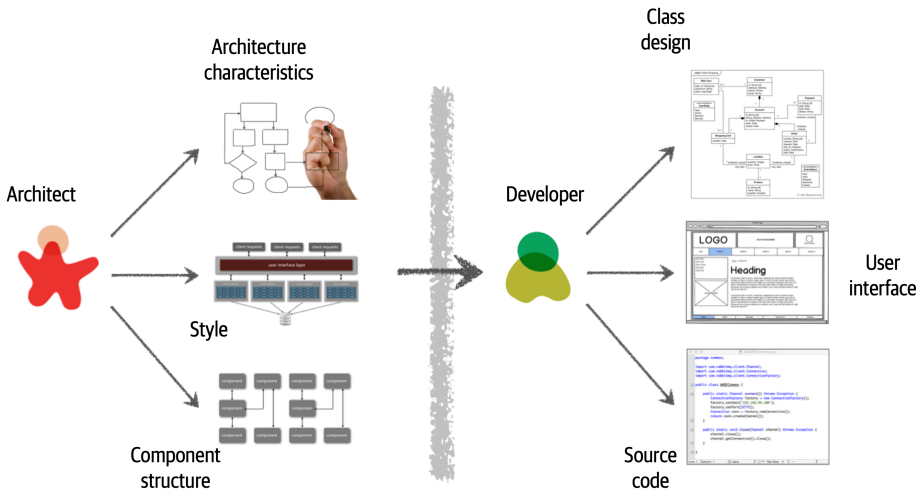

Figure 2-2. Traditional view of architecture versus design

There are several issues with the traditional responsibility model illustrated in Figure 2-2. As a matter of fact, this illustration shows exactly why architecture rarely works. Specifically, it is the unidirectional arrow passing though the virtual and physical barriers separating the architect from the developer that causes all of the problems associated with architecture. Decisions an architect makes sometimes never make it to the development teams, and decisions development teams make that change the architecture rarely get back to the architect. In this model the architect is disconnected from the development teams, and as such the architecture rarely provides what it was originally set out to do.

To make architecture work, both the physical and virtual barriers that exist between architects and developers must be broken down, thus forming a strong bidirectional relationship between architects and development teams. The architect and developer must be on the same virtual team to make this work, as depicted in Figure 2-3. Not only does this model facilitate strong bidirectional communication between architecture and development, but it also allows the architect to provide mentoring and coaching to developers on the team.

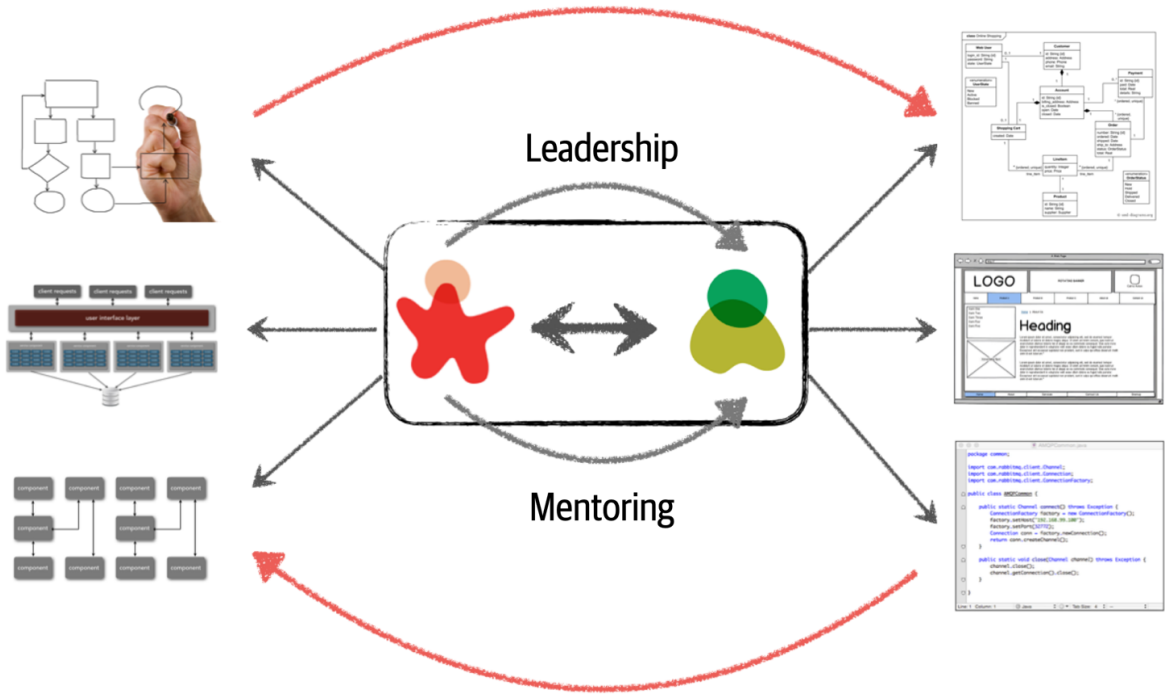

Figure 2-3. Making architecture work through collaboration

Unlike the old-school waterfall approaches to static and rigid software architecture, the architecture of today's systems changes and evolves every iteration or phase of a project. A tight collaboration between the architect and the development team is essential for the success of any software project. So where does architecture end and design begin? It doesn't. They are both part of the circle of life within a software project and must always be kept in synchronization with each other in order to succeed.

Technical Breadth

The scope of technological detail differs between developers and architects. Unlike a developer, who must have a significant amount of *technical depth* to perform their job, a software architect must have a significant amount of *technical breadth* to think like an architect and see things with an architecture point of view. This is illustrated by the knowledge pyramid shown in Figure 2-4, which encapsulates all the technical knowledge in the world. It turns out that the kind of information a technologist should value differs with career stages.

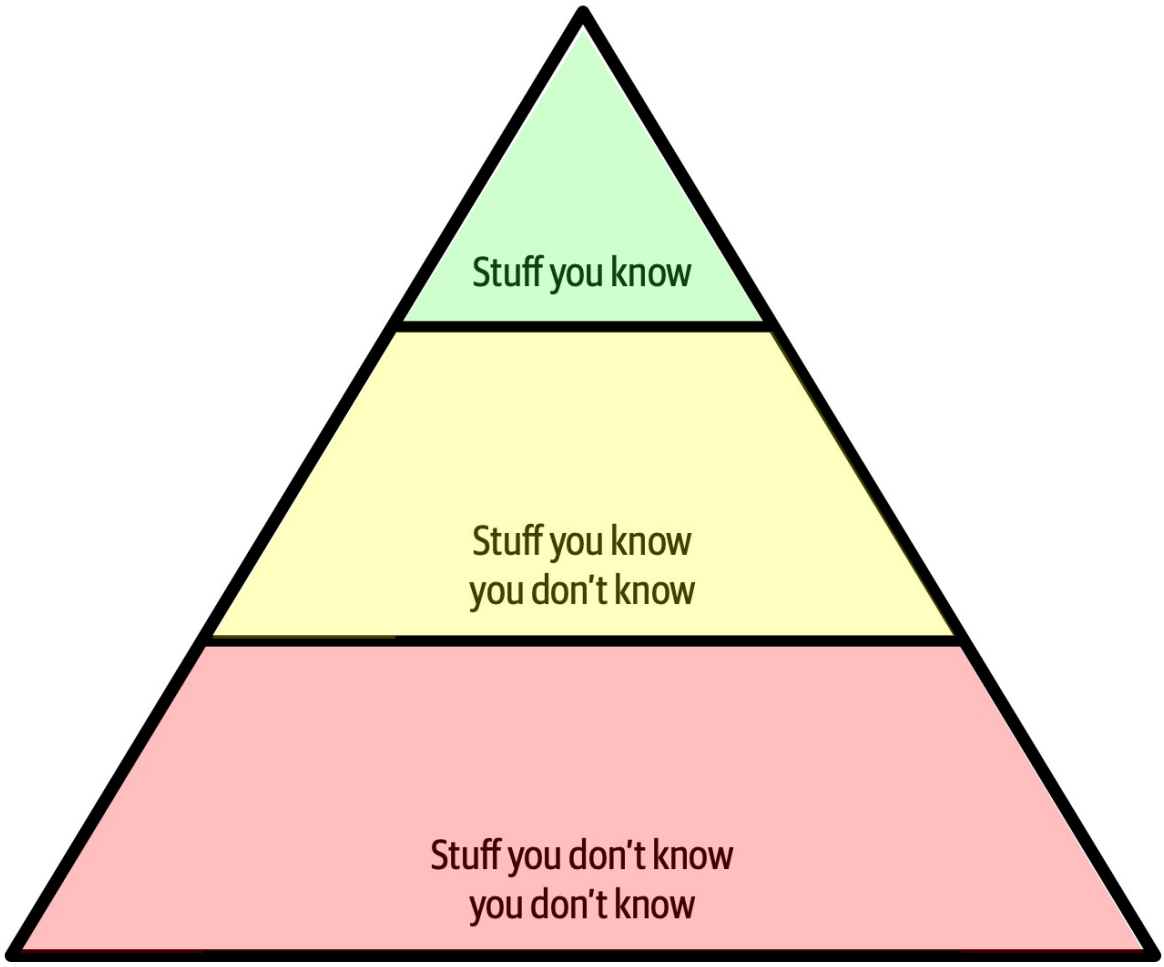

Figure 2-4. The pyramid representing all knowledge

As shown in Figure 2-4, any individual can partition all their knowledge into three sections: *stuff you know, stuff you know you don't know,* and *stuff you don't know you don't know*.

Stuff you know includes the technologies, frameworks, languages, and tools a technologist uses on a daily basis to perform their job, such as knowing Java as a Java programmer. *Stuff you know you don't know* includes those things a technologist knows a little about or has heard of but has little or no expertise in. A good example of this level of knowledge is the Clojure programming language. Most technologists have *heard* of Clojure and know it's a programming language based on Lisp, but they can't code in the language. *Stuff you don't know you don't know* is the largest part of the knowledge triangle and includes the entire host of technologies, tools, frameworks, and languages that would be the perfect solution to a problem

a technologist is trying to solve, but the technologist doesn't even know those things exist.

A developer's early career focuses on expanding the top of the pyramid, to build experience and expertise. This is the ideal focus early on, because developers need more perspective, working knowledge, and hands-on experience. Expanding the top incidentally expands the middle section; as developers encounter more technologies and related artifacts, it adds to their stock of *stuff you know you don't know*.

In Figure 2-5, expanding the top of the pyramid is beneficial because expertise is valued. However, the *stuff you know* is also the *stuff you must maintain*—nothing is static in the software world. If a developer becomes an expert in Ruby on Rails, that expertise won't last if they ignore Ruby on Rails for a year or two. The things at the top of the pyramid require time investment to maintain expertise. Ultimately, the size of the top of an individual's pyramid is their *technical depth*.

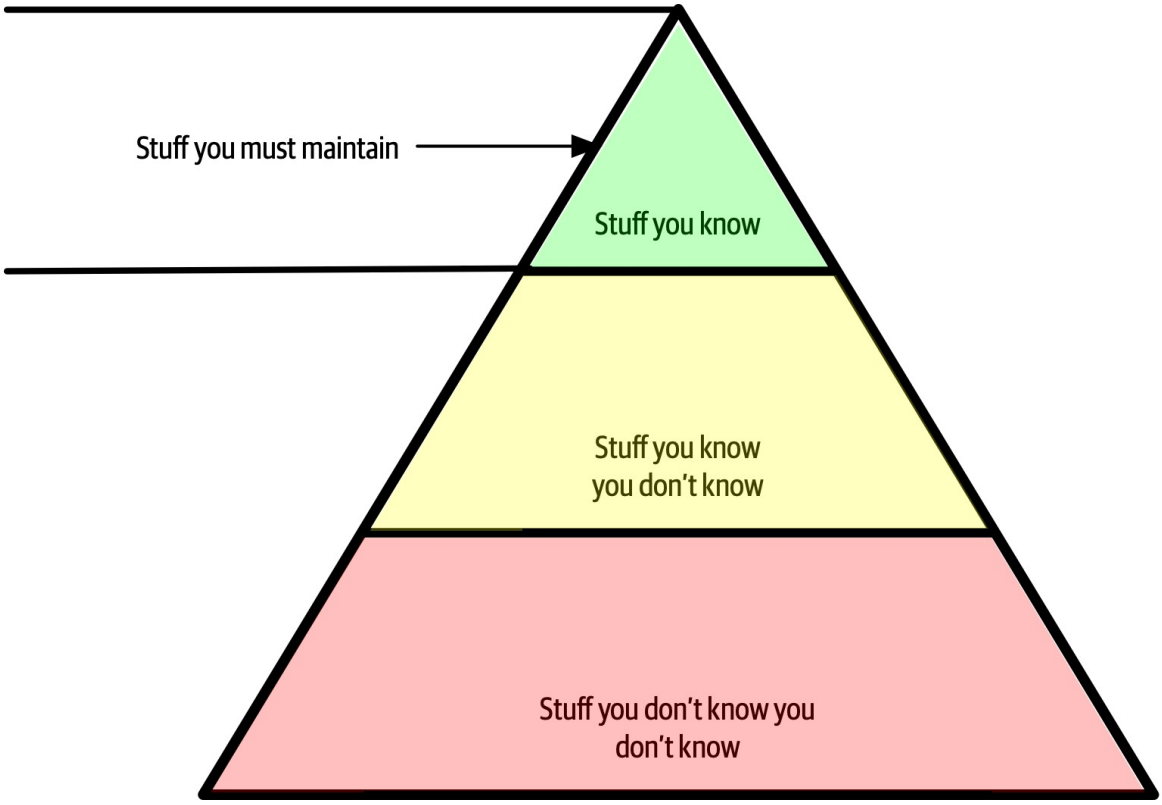

Figure 2-5. Developers must maintain expertise to retain it

However, the nature of knowledge changes as developers transition into the architect role. A large part of the value of an architect is a *broad* understanding of technology and how to use it to solve particular problems. For example, as an architect, it is more beneficial to know that five solutions exist for a particular problem than to have singular expertise in only one. The most important parts of the pyramid for architects are the top *and* middle sections; how far the middle section penetrates into the bottom section represents an architect's technical *breadth*, as shown in Figure 2-6.

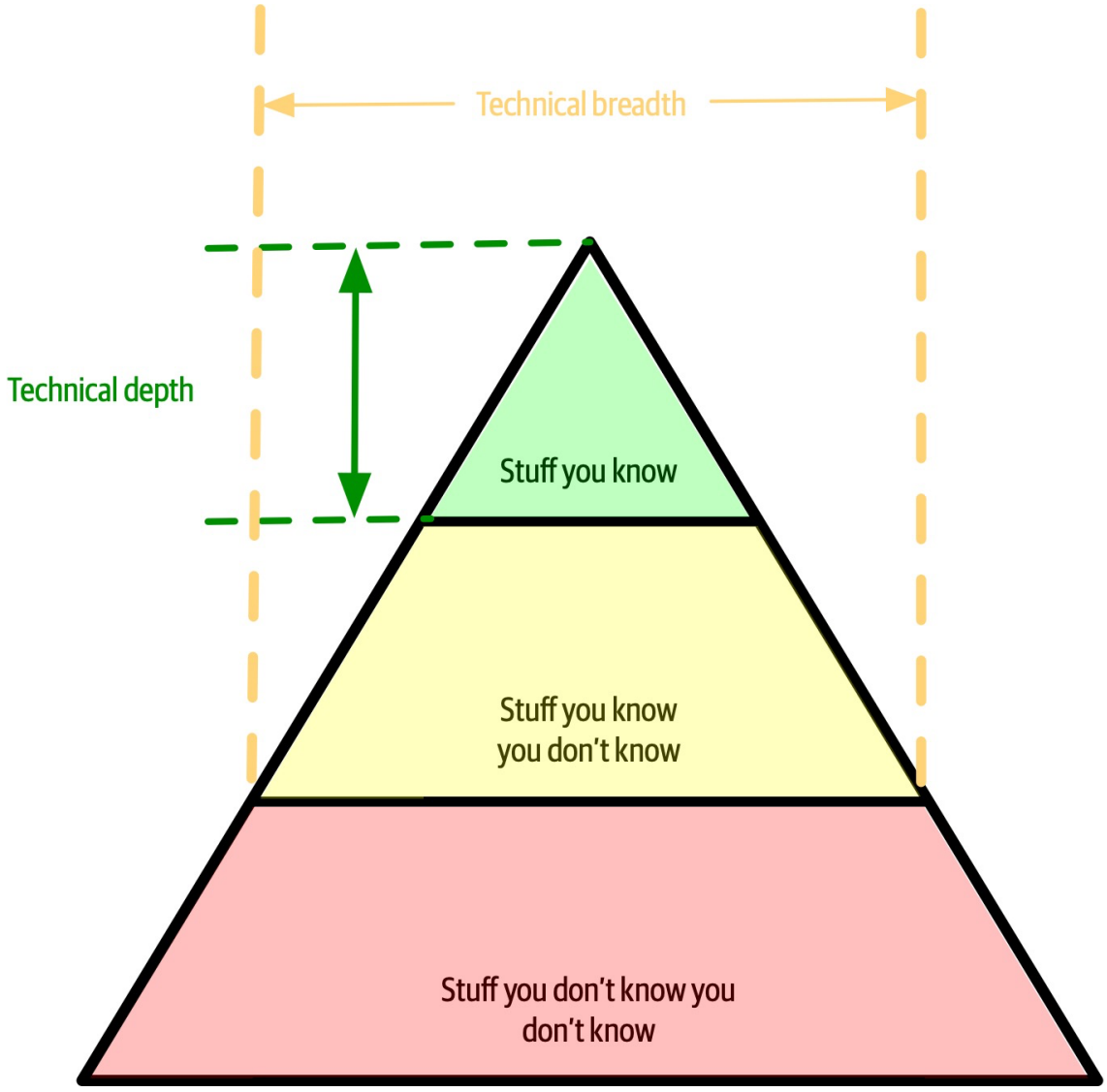

Figure 2-6. What someone knows is technical depth, and how much someone knows is technical breadth

As an architect, *breadth* is more important than *depth*. Because architects must make decisions that match capabilities to technical constraints, a broad understanding of a wide variety of solutions is valuable. Thus, for an architect, the wise course of action is to sacrifice some hard-won expertise and use that time to broaden their portfolio, as shown in Figure 2-7. As illustrated in the diagram, some areas of expertise will remain, probably in particularly enjoyable technology areas, while others usefully atrophy.

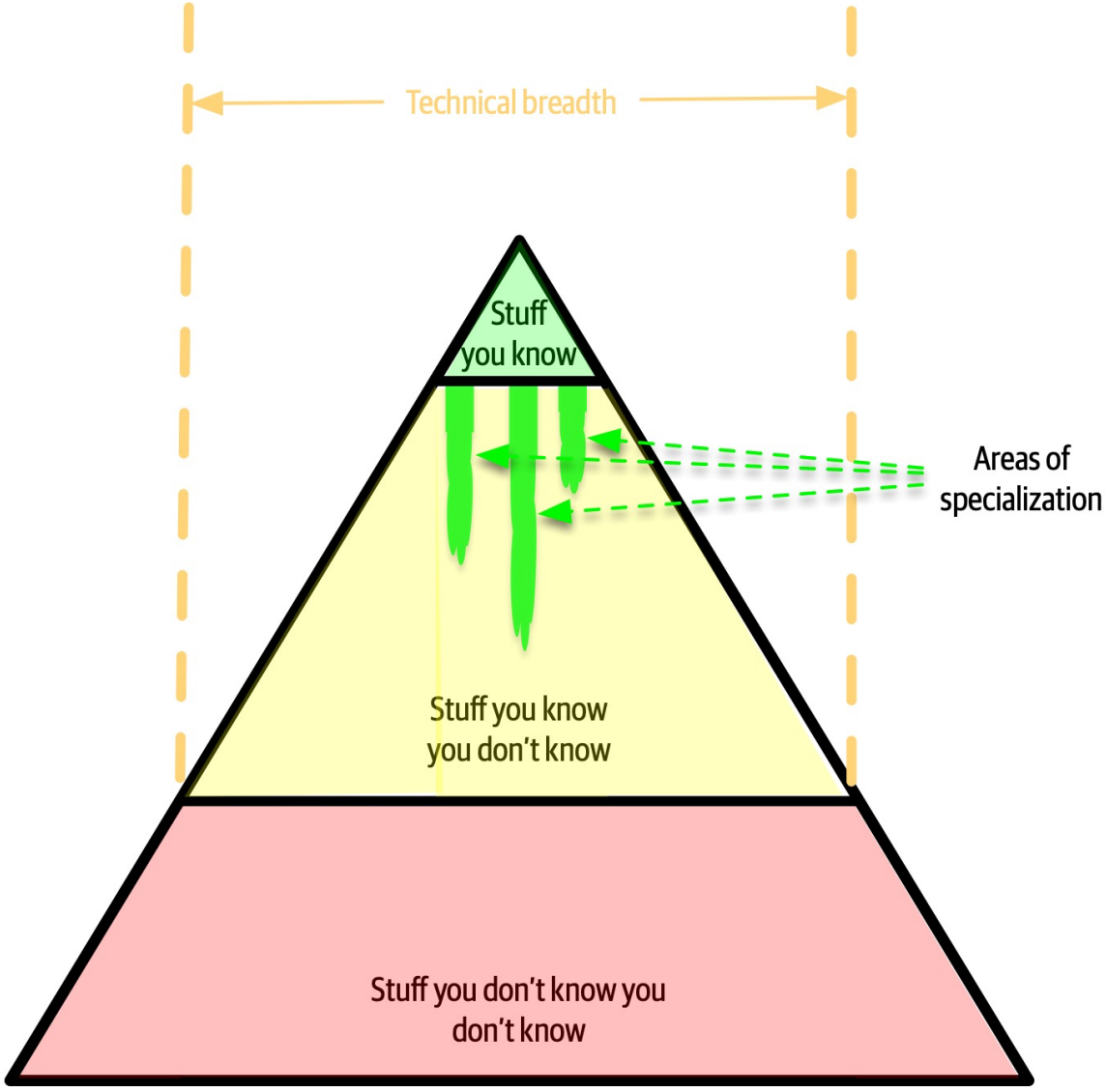

Figure 2-7. Enhanced breadth and shrinking depth for the architect role

Our knowledge pyramid illustrates how fundamentally different the role of *architect* compares to *developer*. Developers spend their whole careers

honing expertise, and transitioning to the architect role means a shift in that perspective, which many individuals find difficult. This in turn leads to two common dysfunctions: first, an architect tries to maintain expertise in a wide variety of areas, succeeding in none of them and working themselves ragged in the process. Second, it manifests as *stale expertise*—the mistaken sensation that your outdated information is still cutting edge. We see this often in large companies where the developers who founded the company have moved into leadership roles yet still make technology decisions using ancient criteria (see “Frozen Caveman Anti-Pattern”).

Architects should focus on technical breadth so that they have a larger quiver from which to draw arrows. Developers transitioning to the architect role may have to change the way they view knowledge acquisition. Balancing their portfolio of knowledge regarding depth versus breadth is something every developer should consider throughout their career.

FROZEN CAVEMAN ANTI-PATTERN

A behavioral anti-pattern commonly observed in the wild, the *Frozen Caveman Anti-Pattern*, describes an architect who always reverts back to their pet irrational concern for every architecture. For example, one of Neal's colleagues worked on a system that featured a centralized architecture. Yet, each time they delivered the design to the client architects, the persistent question was "But what if we lose Italy?" Several years before, a freak communication problem had prevented headquarters from communicating with its stores in Italy, causing great inconvenience. While the chances of a reoccurrence were extremely small, the architects had become obsessed about this particular architectural characteristic.

Generally, this anti-pattern manifests in architects who have been burned in the past by a poor decision or unexpected occurrence, making them particularly cautious in the future. While risk assessment is important, it should be realistic as well. Understanding the difference between genuine versus perceived technical risk is part of the ongoing learning process for architects. Thinking like an architect requires overcoming these "frozen caveman" ideas and experiences, seeing other solutions, and asking more relevant questions.

Analyzing Trade-Offs

Thinking like an architect is all about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine what is the best solution. To quote Mark (one of your authors):

> *Architecture is the stuff you can't Google.*

Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is "it depends." While many people get increasingly annoyed at this answer, it is unfortunately true. You cannot Google the answer to whether REST or messaging would be better,

or whether microservices is the right architecture style, because it *does* depend. It depends on the deployment environment, business drivers, company culture, budgets, timeframes, developer skill set, and dozens of other factors. Everyone's environment, situation, and problem is different, hence why architecture is so hard. To quote Neal (another one of your authors):

There are no right or wrong answers in architecture—only trade-offs.

For example, consider an item auction system, as illustrated in Figure 2-8, where someone places a bid for an item up for auction.

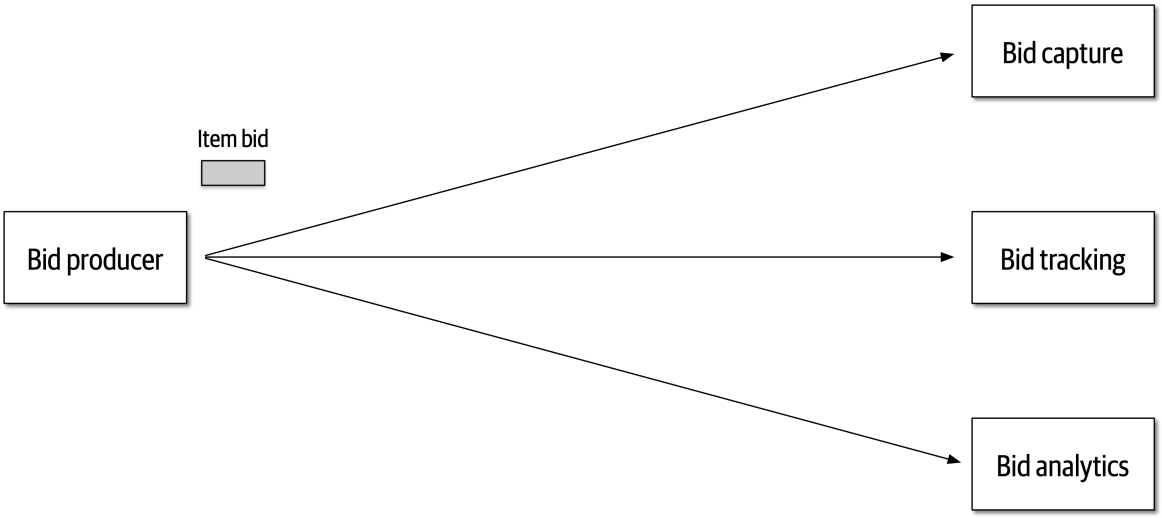

Figure 2-8. Auction system example of a trade-off—queues or topics?

The `Bid Producer` service generates a bid from the bidder and then sends that bid amount to the `Bid Capture`, `Bid Tracking`, and `Bid Analytics` services. This could be done by using queues in a point-to-point messaging fashion or by using a topic in a publish-and-subscribe messaging fashion. Which one should the architect use? You can't Google the answer. Architectural thinking requires the architect to analyze the trade-offs associated with each option and select the best one given the specific situation.

The two messaging options for the item auction system are shown in figures Figure 2-9 and Figure 2-10, with Figure 2-9 illustrating the use of a topic in

a publish-and-subscribe messaging model, and Figure 2-10 illustrating the use of queues in a point-to-point messaging model.

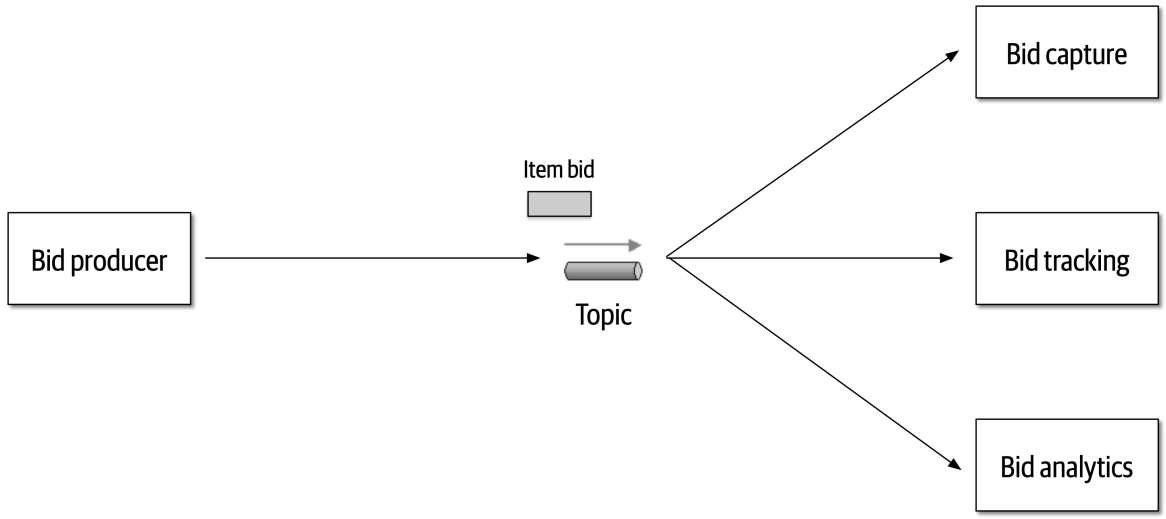

Figure 2-9. Use of a topic for communication between services

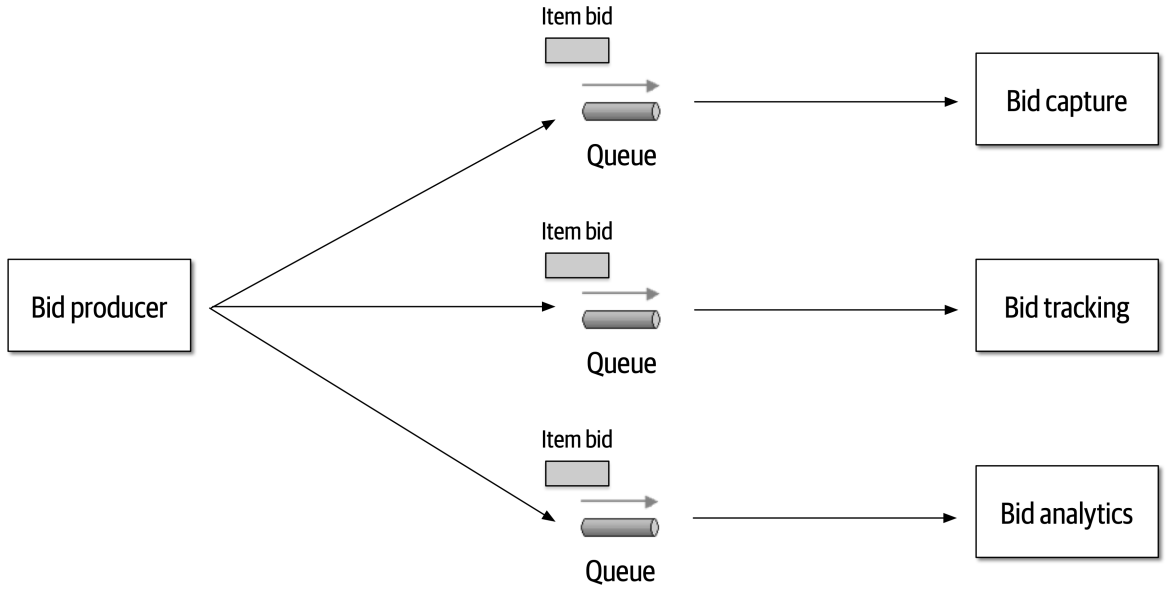

Figure 2-10. Use of queues for communication between services

The clear advantage (and seemingly obvious solution) to this problem in Figure 2-9 is that of *architectural extensibility*. The `Bid Producer` service only requires a single connection to a topic, unlike the queue solution in Figure 2-10 where the `Bid Producer` needs to connect to three different queues. If a new service called `Bid History` were to be added to this system due to the requirement to provide each bidder with a history of all

the bids they made in each auction, no changes at all would be needed to the existing system. When the new `Bid History` service is created, it could simply subscribe to the topic already containing the bid information. In the queue option shown in Figure 2-10, however, a new queue would be required for the `Bid History` service, and the `Bid Producer` would need to be modified to add an additional connection to the new queue. The point here is that using queues requires significant change to the system when adding new bidding functionality, whereas with the topic approach no changes are needed at all in the existing infrastructure. Also, notice that the `Bid Producer` is more decoupled in the topic option—the `Bid Producer` doesn't know how the bidding information will be used or by which services. In the queue option the `Bid Producer` knows exactly how the bidding information is used (and by whom), and hence is more coupled to the system.

With this analysis it seems clear that the topic approach using the publish-and-subscribe messaging model is the obvious and best choice. However, to quote Rich Hickey, the creator of the Clojure programming language:

> *Programmers know the benefits of everything and the trade-offs of nothing. Architects need to understand both.*

Thinking architecturally is looking at the benefits of a given solution, but also analyzing the negatives, or trade-offs, associated with a solution. Continuing with the auction system example, a software architect would analyze the negatives of the topic solution. In analyzing the differences, notice first in Figure 2-9 that with a topic, *anyone* can access bidding data, which introduces a possible issue with data access and data security. In the queue model illustrated in Figure 2-10, the data sent to the queue can *only* be accessed by the specific consumer receiving that message. If a rogue service did listen in on a queue, those bids would not be received by the corresponding service, and a notification would immediately be sent about the loss of data (and hence a possible security breach). In other words, it is very easy to wiretap into a topic, but not a queue.

In addition to the security issue, the topic solution in Figure 2-9 only supports homogeneous contracts. All services receiving the bidding data must accept the same contract and set of bidding data. In the queue option in Figure 2-10, each consumer can have its own contract specific to the data it needs. For example, suppose the new `Bid History` service requires the current asking price along with the bid, but no other service needs that information. In this case, the contract would need to be modified, impacting all other services using that data. In the queue model, this would be a separate channel, hence a separate contract not impacting any other service.

Another disadvantage of the topic model illustrated in Figure 2-9 is that it does not support monitoring of the number of messages in the topic and hence auto-scaling capabilities. However, with the queue option in Figure 2-10, each queue can be monitored individually, and programmatic load balancing applied to each bidding consumer so that each can be automatically scaled independency from one another. Note that this trade-off is technology specific in that the Advanced Message Queuing Protocol (AMQP) can support programmatic load balancing and monitoring because of the separation between an exchange (what the producer sends to) and a queue (what the consumer listens to).

Given this trade-off analysis, now which is the better option? And the answer? It depends! Table 2-1 summarizes these trade-offs.

Table 2-1. Trade-offs between topics and queues

Topic advantages	Topic disadvantages
Architectural extensibility	Data access and data security concerns
Service decoupling	No heterogeneous contracts
	Monitoring and programmatic scalability

The point here is that *everything* in software architecture has a trade-off: an advantage and disadvantage. Thinking like an architect is analyzing these trade-offs, then asking “which is more important: extensibility or security?”

The decision between different solutions will always depend on the business drivers, environment, and a host of other factors.

Understanding Business Drivers

Thinking like an architect is understanding the business drivers that are required for the success of the system and translating those requirements into architecture characteristics (such as scalability, performance, and availability). This is a challenging task that requires the architect to have some level of business domain knowledge and healthy, collaborative relationships with key business stakeholders. We've devoted several chapters in the book on this specific topic. In Chapter 4 we define various architecture characteristics. In Chapter 5 we describe ways to identify and qualify architecture characteristics. And in Chapter 6 we describe how to measure each of these characteristics to ensure the business needs of the system are met.

Balancing Architecture and Hands-On Coding

One of the difficult tasks an architect faces is how to balance hands-on coding with software architecture. We firmly believe that every architect should code and be able to maintain a certain level of technical depth (see "Technical Breadth"). While this may seem like an easy task, it is sometimes rather difficult to accomplish.

The first tip in striving for a balance between hands-on coding and being a software architect is avoiding the bottleneck trap. The bottleneck trap occurs when the architect has taken ownership of code within the critical path of a project (usually the underlying framework code) and becomes a bottleneck to the team. This happens because the architect is not a full-time developer and therefore must balance between playing the developer role (writing and testing source code) and the architect role (drawing diagrams, attending meetings, and well, attending more meetings).

One way to avoid the bottleneck trap as an effective software architect is to delegate the critical path and framework code to others on the development team and then focus on coding a piece of business functionality (a service or a screen) one to three iterations down the road. Three positive things happen by doing this. First, the architect is gaining hands-on experience writing production code while no longer becoming a bottleneck on the team. Second, the critical path and framework code is distributed to the development team (where it belongs), giving them ownership and a better understanding of the harder parts of the system. Third, and perhaps most important, the architect is writing the same business-related source code as the development team and is therefore better able to identify with the development team in terms of the pain they might be going through with processes, procedures, and the development environment.

Suppose, however, that the architect is not able to develop code with the development team. How can a software architect still remain hands-on and maintain some level of technical depth? There are four basic ways an architect can still remain hands-on at work without having to “practice coding from home” (although we recommend practicing coding at home as well).

The first way is to do frequent proof-of-concepts or POCs. This practice not only requires the architect to write source code, but it also helps validate an architecture decision by taking the implementation details into account. For example, if an architect is stuck trying to make a decision between two caching solutions, one effective way to help make this decision is to develop a working example in each caching product and compare the results. This allows the architect to see first-hand the implementation details and the amount of effort required to develop the full solution. It also allows the architect to better compare architectural characteristics such as scalability, performance, or overall fault tolerance of the different caching solutions.

Our advice when doing proof-of-concept work is that, whenever possible, the architect should write the best production-quality code they can. We recommend this practice for two reasons. First, quite often, throwaway

proof-of-concept code goes into the source code repository and becomes the reference architecture or guiding example for others to follow. The last thing an architect would want is for their throwaway, sloppy code to be a representation of their typical work. The second reason is that by writing production-quality proof-of-concept code, the architect gets practice writing quality, well-structured code rather than continually developing bad coding practices.

Another way an architect can remain hands-on is to tackle some of the technical debt stories or architecture stories, freeing the development team up to work on the critical functional user stories. These stories are usually low priority, so if the architect does not have the chance to complete a technical debt or architecture story within a given iteration, it's not the end of the world and generally does not impact the success of the iteration.

Similarly, working on bug fixes within an iteration is another way of maintaining hands-on coding while helping the development team as well. While certainly not glamorous, this technique allows the architect to identify where issues and weakness may be within the code base and possibly the architecture.

Leveraging automation by creating simple command-line tools and analyzers to help the development team with their day-to-day tasks is another great way to maintain hands-on coding skills while making the development team more effective. Look for repetitive tasks the development team performs and automate the process. The development team will be grateful for the automation. Some examples are automated source validators to help check for specific coding standards not found in other lint tests, automated checklists, and repetitive manual code refactoring tasks.

Automation can also be in the form of architectural analysis and fitness functions to ensure the vitality and compliance of the architecture. For example, an architect can write Java code in ArchUnit in the Java platform to automate architectural compliance, or write custom fitness functions to

ensure architectural compliance while gaining hands-on experience. We talk about these techniques in Chapter 6.

A final technique to remain hands-on as an architect is to do frequent code reviews. While the architect is not actually writing code, at least they are *involved* in the source code. Further, doing code reviews has the added benefits of being able to ensure compliance with the architecture and to seek out mentoring and coaching opportunities on the team.

Chapter 3. Modularity

First, we want to untangle some common terms used and overused in discussions about architecture surrounding modularity and provide definitions for use throughout the book.

> *95% of the words [about software architecture] are spent extolling the benefits of "modularity" and that little, if anything, is said about how to achieve it.*
>
> —Glenford J. Myers (1978)

Different platforms offer different reuse mechanisms for code, but all support some way of grouping related code together into *modules*. While this concept is universal in software architecture, it has proven slippery to define. A casual internet search yields dozens of definitions, with no consistency (and some contradictions). As you can see from the quote from Myers, this isn't a new problem. However, because no recognized definition exists, we must jump into the fray and provide our own definitions for the sake of consistency throughout the book.

Understanding modularity and its many incarnations in the development platform of choice is critical for architects. Many of the tools we have to analyze architecture (such as metrics, fitness functions, and visualizations) rely on these modularity concepts. Modularity is an organizing principle. If an architect designs a system without paying attention to how the pieces wire together, they end up creating a system that presents myriad difficulties. To use a physics analogy, software systems model complex systems, which tend toward entropy (or disorder). Energy must be added to a physical system to preserve order. The same is true for software systems: architects must constantly expend energy to ensure good structural soundness, which won't happen by accident.

Preserving good modularity exemplifies our definition of an *implicit* architecture characteristic: virtually no project features a requirement that

asks the architect to ensure good modular distinction and communication, yet sustainable code bases require order and consistency.

Definition

The dictionary defines *module* as “each of a set of standardized parts or independent units that can be used to construct a more complex structure.” We use *modularity* to describe a logical grouping of related code, which could be a group of classes in an object-oriented language or functions in a structured or functional language. Most languages provide mechanisms for modularity (`package` in Java, `namespace` in .NET, and so on). Developers typically use modules as a way to group related code together. For example, the `com.mycompany.customer` package in Java should contain things related to customers.

Languages now feature a wide variety of packaging mechanisms, making a developer’s chore of choosing between them difficult. For example, in many modern languages, developers can define behavior in functions/methods, classes, or packages/namespaces, each with different visibility and scoping rules. Other languages complicate this further by adding programming constructs such as the metaobject protocol to provide developers even more extension mechanisms.

Architects must be aware of how developers package things because it has important implications in architecture. For example, if several packages are tightly coupled together, reusing one of them for related work becomes more difficult.

MODULAR REUSE BEFORE CLASSES

Developers who predate object-oriented languages may puzzle over why so many different separation schemes commonly exist. Much of the reason has to do with backward compatibility, not of code but rather for how developers think about things. In March of 1968, Edsger Dijkstra published a letter in the *Communications of the ACM* entitled "Go To Statement Considered Harmful." He denigrated the common use of the `GOTO` statement common in programming languages at the time that allowed non-linear leaping around within code, making reasoning and debugging difficult.

This paper helped usher in the era of *structured* programming languages, exemplified by Pascal and C, which encouraged deeper thinking about how things fit together. Developers quickly realized that most of the languages had no good way to group like things together logically. Thus, the short era of *modular* languages was born, such as Modula (Pascal creator Niklaus Wirth's next language) and Ada. These languages had the programming construct of a *module*, much as we think about packages or namespaces today (but without the classes).

The modular programming era was short-lived. Object-oriented languages became popular because they offered new ways to encapsulate and reuse code. Still, language designers realized the utility of modules, retaining them in the form of packages, namespaces, etc. Many odd compatibility features exist in languages to support these different paradigms. For example, Java supports modular (via packages and package-level initialization using static initializers), object-oriented, and functional paradigms, each programming style with its own scoping rules and quirks.

For discussions about architecture, we use modularity as a general term to denote a related grouping of code: classes, functions, or any other grouping. This doesn't imply a physical separation, merely a logical one; the difference is sometimes important. For example, lumping a large number of

classes together in a monolithic application may make sense from a convenience standpoint. However, when it comes time to restructure the architecture, the coupling encouraged by loose partitioning becomes an impediment to breaking the monolith apart. Thus, it is useful to talk about modularity as a concept separate from the physical separation forced or implied by a particular platform.

It is worth noting the general concept of *namespace*, separate from the technical implementation in the .NET platform. Developers often need precise, fully qualified names for software assets to separate different software assets (components, classes, and so on) from each other. The most obvious example that people use every day is the internet: unique, global identifiers tied to IP addresses. Most languages have some modularity mechanism that doubles as a namespace to organize things: variables, functions, and/or methods. Sometimes the module structure is reflected physically. For example, Java requires that its package structure must reflect the directory structure of the physical class files.

A LANGUAGE WITH NO NAME CONFLICTS: JAVA 1.0

The original designers of Java had extensive experience dealing with name conflicts and clashes in the various programming platforms at the time. The original design of Java used a clever hack to avoid the possibility of ambiguity between two classes that had the same name. For example, what if your problem domain included a catalog *order* and an installation *order*: both named *order* but with very different connotations (and classes). The solution in Java was to create the `package` namespace mechanism, along with the requirement that the physical directory structure just match the package name. Because filesystems won't allow the same named file to reside in the same directory, they leveraged the inherent features of the operating system to avoid the possibility of ambiguity. Thus, the original `classpath` in Java contained only directories, disallowing the possibility of name conflicts.

However, as the language designers discovered, forcing every project to have a fully formed directory structure was cumbersome, especially as projects became larger. Plus, building reusable assets was difficult: frameworks and libraries must be "exploded" into the directory structure. In the second major release of Java (1.2, called Java 2), designers added the `jar` mechanism, allowing an archive file to act as a directory structure on a classpath. For the next decade, Java developers struggled with getting the `classpath` exactly right, as a combination of directories and JAR files. And, of course, the original intent was broken: now two JAR files could create conflicting names on a classpath, leading to numerous war stories of debugging class loaders.

Measuring Modularity

Given the importance of modularity to architects, they need tools to understand it. Fortunately, researchers created a variety of language-

agnostic metrics to help architects understand modularity. We focus on three key concepts: *cohesion*, *coupling*, and *connascence*.

Cohesion

Cohesion refers to what extent the parts of a module should be contained within the same module. In other words, it is a measure of how related the parts are to one another. Ideally, a cohesive module is one where all the parts should be packaged together, because breaking them into smaller pieces would require coupling the parts together via calls between modules to achieve useful results.

> *Attempting to divide a cohesive module would only result in increased coupling and decreased readability.*
>
> —Larry Constantine

Computer scientists have defined a range of cohesion measures, listed here from best to worst:

Functional cohesion

: Every part of the module is related to the other, and the module contains everything essential to function.

Sequential cohesion

: Two modules interact, where one outputs data that becomes the input for the other.

Communicational cohesion

: Two modules form a communication chain, where each operates on information and/or contributes to some output. For example, add a record to the database and generate an email based on that information.

Procedural cohesion

: Two modules must execute code in a particular order.

Temporal cohesion

Modules are related based on timing dependencies. For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are temporally cohesive.

Logical cohesion

The data within modules is related logically but not functionally. For example, consider a module that converts information from text, serialized objects, or streams. Operations are related, but the functions are quite different. A common example of this type of cohesion exists in virtually every Java project in the form of the `StringUtils` package: a group of static methods that operate on `String` but are otherwise unrelated.

Coincidental cohesion

Elements in a module are not related other than being in the same source file; this represents the most negative form of cohesion.

Despite having seven variants listed, *cohesion* is a less precise metric than *coupling*. Often, the degree of cohesiveness of a particular module is at the discretion of a particular architect. For example, consider this module definition:

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`
- `get customer orders`
- `cancel customer orders`

Should the last two entries reside in this module or should the developer create two separate modules, such as:

Customer Maintenance

- `add customer`
- `update customer`
- `get customer`
- `notify customer`

Order Maintenance

- `get customer orders`
- `cancel customer orders`

Which is the correct structure? As always, it depends:

- Are those the only two operations for `Order Maintenance`? If so, it may make sense to collapse those operations back into `Customer Maintenance`.
- Is `Customer Maintenance` expected to grow much larger, encouraging developers to look for opportunities to extract behavior?
- Does `Order Maintenance` require so much knowledge of `Customer` information that separating the two modules would require a high degree of coupling to make it functional? This relates back to the Larry Constantine quote.

These questions represent the kind of trade-off analysis at the heart of the job of a software architect.

Surprisingly, given the subjectiveness of cohesion, computer scientists have developed a good structural metric to determine cohesion (or, more specifically, the lack of cohesion). A well-known set of metrics named the Chidamber and Kemerer Object-oriented metrics suite was developed by

the eponymous authors to measure particular aspects of object-oriented software systems. The suite includes many common code metrics, such as cyclomatic complexity (see “Cyclomatic Complexity”) and several important coupling metrics discussed in “Coupling”.

The Chidamber and Kemerer Lack of Cohesion in Methods (LCOM) metric measures the structural cohesion of a module, typically a component. The initial version appears in Equation 3-1.

Equation 3-1. LCOM, version 1

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases}$$

P increases by one for any method that doesn’t access a particular shared field and *Q* decreases by one for methods that do share a particular shared field. The authors sympathize with those who don’t understand this formulation. Worse, it has gradually gotten more elaborate over time. The second variation introduced in 1996 (thus the name *LCOM96B*) appears in Equation 3-2.

Equation 3-2. LCOM 96b

$$LCOM96b = \frac{1}{a} \sum_{j=1}^{a} \frac{m - \mu(Aj)}{m}$$

We wont bother untangling the variables and operators in Equation 3-2 because the following written explanation is clearer. Basically, the LCOM metric exposes incidental coupling within classes. Here’s a better definition of LCOM:

LCOM

: The sum of sets of methods not shared via sharing fields

Consider a class with private fields `a` and `b`. Many of the methods only access `a`, and many other methods only access `b`. The *sum* of the sets of methods not shared via sharing fields (`a` and `b`) is high; therefore, this class

reports a high LCOM score, indicating that it scores high in *lack of cohesion in methods*. Consider the three classes shown in Figure 3-1.

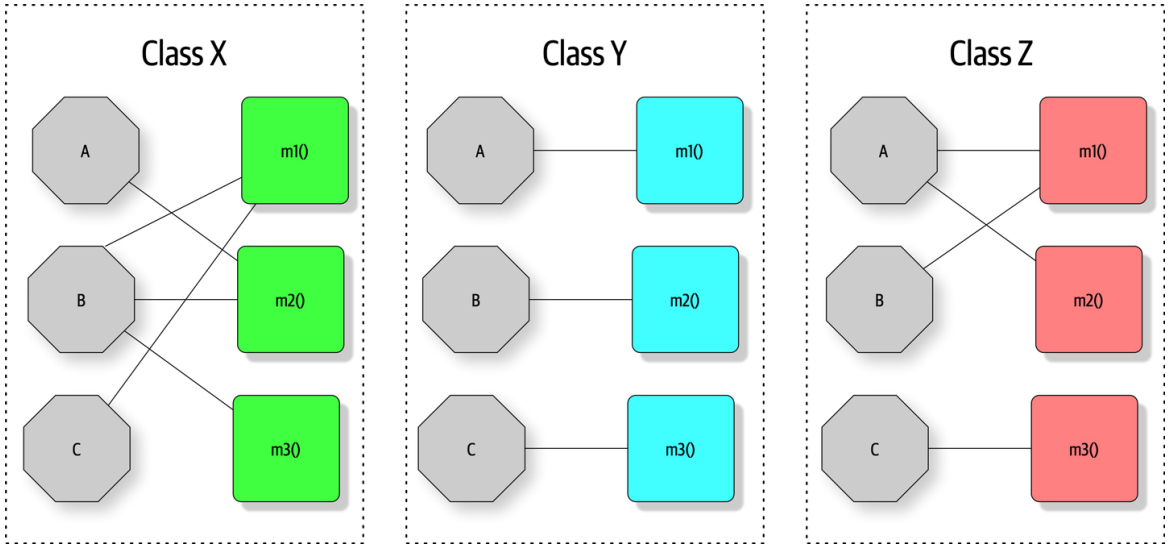

Figure 3-1. Illustration of the LCOM metric, where fields are octagons and methods are squares

In Figure 3-1, fields appear as single letters and methods appear as blocks. In `Class X`, the LCOM score is low, indicating good structural cohesion. `Class Y`, however, lacks cohesion; each of the field/method pairs in `Class Y` could appear in its own class without affecting behavior. `Class Z` shows mixed cohesion, where developers could refactor the last field/method combination into its own class.

The LCOM metric is useful to architects who are analyzing code bases in order to move from one architectural style to another. One of the common headaches when moving architectures are shared utility classes. Using the LCOM metric can help architects find classes that are incidentally coupled and should never have been a single class to begin with.

Many software metrics have serious deficiencies, and LCOM is not immune. All this metric can find is *structural* lack of cohesion; it has no way to determine logically if particular pieces fit together. This reflects back on our Second Law of Software Architecture: prefer *why* over *how*.

Coupling

Fortunately, we have better tools to analyze coupling in code bases, based in part on graph theory: because the method calls and returns form a call graph, analysis based on mathematics becomes possible. In 1979, Edward Yourdon and and Larry Constantine published *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Prentice-Hall), defining many core concepts, including the metrics *afferent* and *efferent* coupling. *Afferent* coupling measures the number of *incoming* connections to a code artifact (component, class, function, and so on). *Efferent* coupling measures the *outgoing* connections to other code artifacts. For virtually every platform tools exist that allow architects to analyze the coupling characteristics of code in order to assist in restructuring, migrating, or understanding a code base.

WHY SUCH SIMILAR NAMES FOR COUPLING METRICS?

Why are two critical metrics in the architecture world that represent opposite concepts named virtually the same thing, differing in only the vowels that sound the most alike? These terms originate from Yourdon and Constantine's *Structured Design*. Borrowing concepts from mathematics, they coined the now-common afferent and efferent coupling terms, which should have been called incoming and outgoing coupling. However, because the original authors leaned toward mathematical symmetry rather than clarity, developers came up with several mnemonics to help out: *a* appears before *e* in the English alphabet, corresponding to *incoming* being before *outgoing*, or the observation that the letter *e* in efferent matches the initial letter in *exit*, corresponding to outgoing connections.

Abstractness, Instability, and Distance from the Main

Sequence

While the raw value of component coupling has value to architects, several other derived metrics allow a deeper evaluation. These metrics were created by Robert Martin for a C++ book, but are widely applicable to other object-oriented languages.

Abstractness is the ratio of abstract artifacts (abstract classes, interfaces, and so on) to concrete artifacts (implementation). It represents a measure of abstractness versus implementation. For example, consider a code base with no abstractions, just a huge, single function of code (as in a single `main()` method). The flip side is a code base with too many abstractions, making it difficult for developers to understand how things wire together (for example, it takes developers a while to figure out what to do with an `AbstractSingletonProxyFactoryBean`).

The formula for abstractness appears in Equation 3-3.

Equation 3-3. Abstractness

$$A = \frac{\sum m^a}{\sum m^c}$$

In the equation, m^a represents *abstract* elements (interfaces or abstract classes) with the module, and m^c represents *concrete* elements (nonabstract classes). This metric looks for the same criteria. The easiest way to visualize this metric: consider an application with 5,000 lines of code, all in one `main()` method. The abstractness numerator is 1, while the denominator is 5,000, yielding an abstractness of almost 0. Thus, this metric measures the ratio of abstractions in your code.

Architects calculate *abstractness* by calculating the ratio of the sum of abstract artifacts to the sum of the concrete ones.

Another derived metric, *instability*, is defined as the ratio of efferent coupling to the sum of both efferent and afferent coupling, shown in Equation 3-4.

Equation 3-4. Instability

$$I = \frac{C^e}{C^e + C^a}$$

In the equation, c^e represents *efferent* (or outgoing) coupling, and c^a represents *afferent* (or incoming) coupling.

The *instability* metric determines the volatility of a code base. A code base that exhibits high degrees of instability breaks more easily when changed because of high coupling. For example, if a class calls to many other classes to delegate work, the calling class shows high susceptibility to breakage if one or more of the called methods change.

Distance from the Main Sequence

One of the few holistic metrics architects have for architectural structure is *distance from the main sequence*, a derived metric based on *instability* and *abstractness*, shown in Equation 3-5.

Equation 3-5. Distance from the main sequence

$$D = |A + I - 1|$$

In the equation, `A` = *abstractness* and I = *instability*.

Note that both *abstractness* and *instability* are ratios, meaning their result will always fall between 0 and 1. Thus, when graphing the relationship, we see the graph in Figure 3-2.

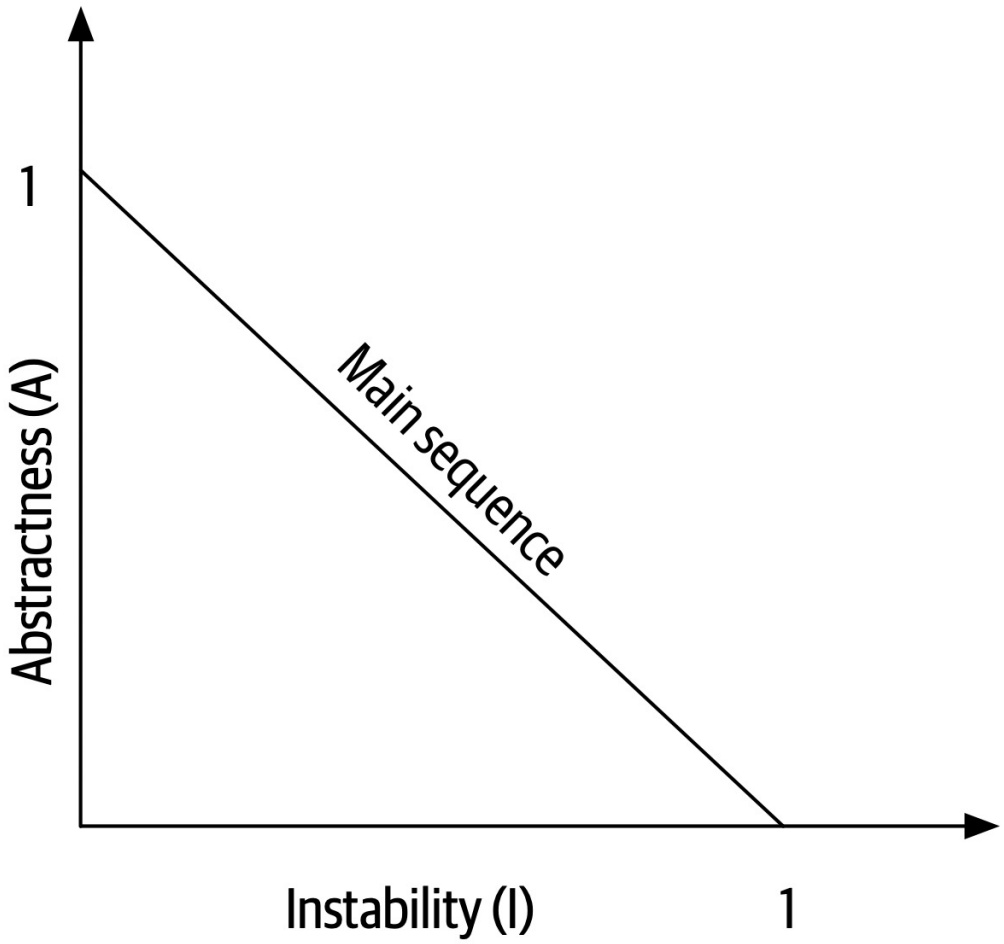

Figure 3-2. The main sequence defines the ideal relationship between abstractness and instability

The *distance* metric imagines an ideal relationship between abstractness and instability; classes that fall near this idealized line exhibit a healthy mixture of these two competing concerns. For example, graphing a particular class allows developers to calculate the *distance from the main sequence* metric, illustrated in Figure 3-3.

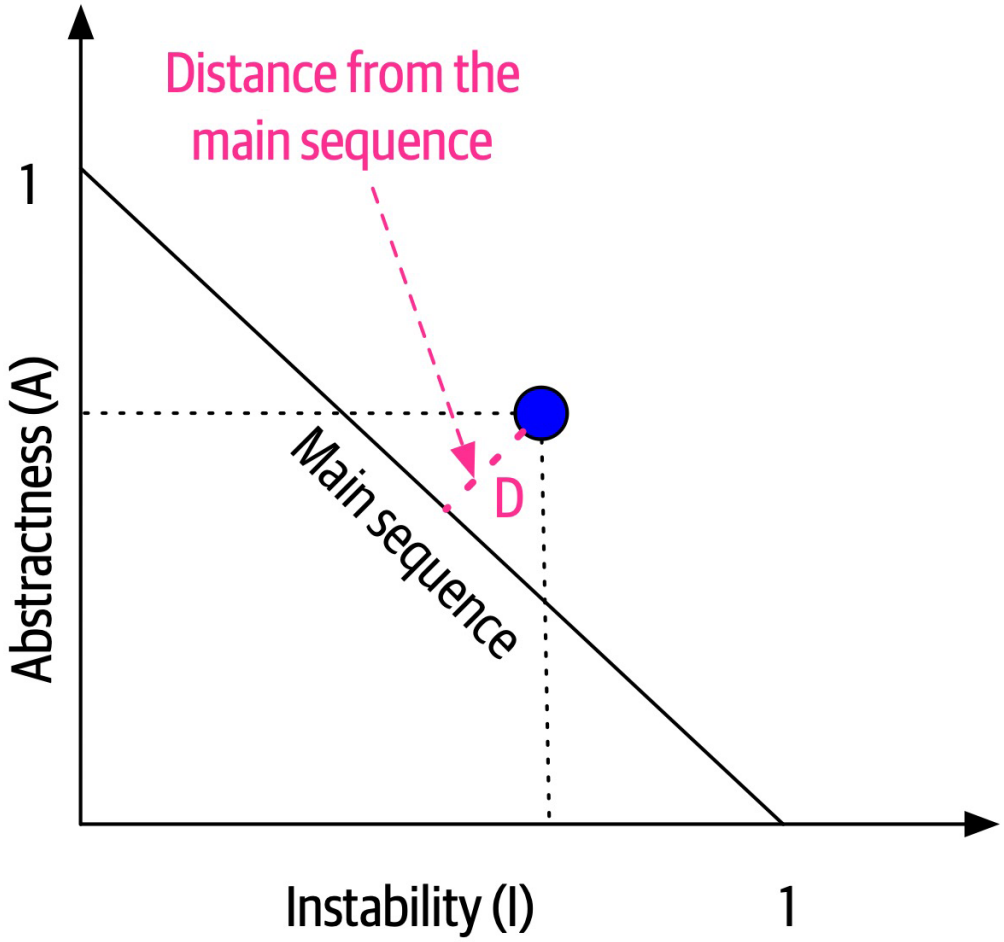

Figure 3-3. Normalized distance from the main sequence for a particular class

In Figure 3-3, developers graph the candidate class, then measure the distance from the idealized line. The closer to the line, the better balanced the class. Classes that fall too far into the upper-righthand corner enter into what architects call the *zone of uselessness*: code that is too abstract becomes difficult to use. Conversely, code that falls into the lower-lefthand corner enter the *zone of pain*: code with too much implementation and not enough abstraction becomes brittle and hard to maintain, illustrated in Figure 3-4.

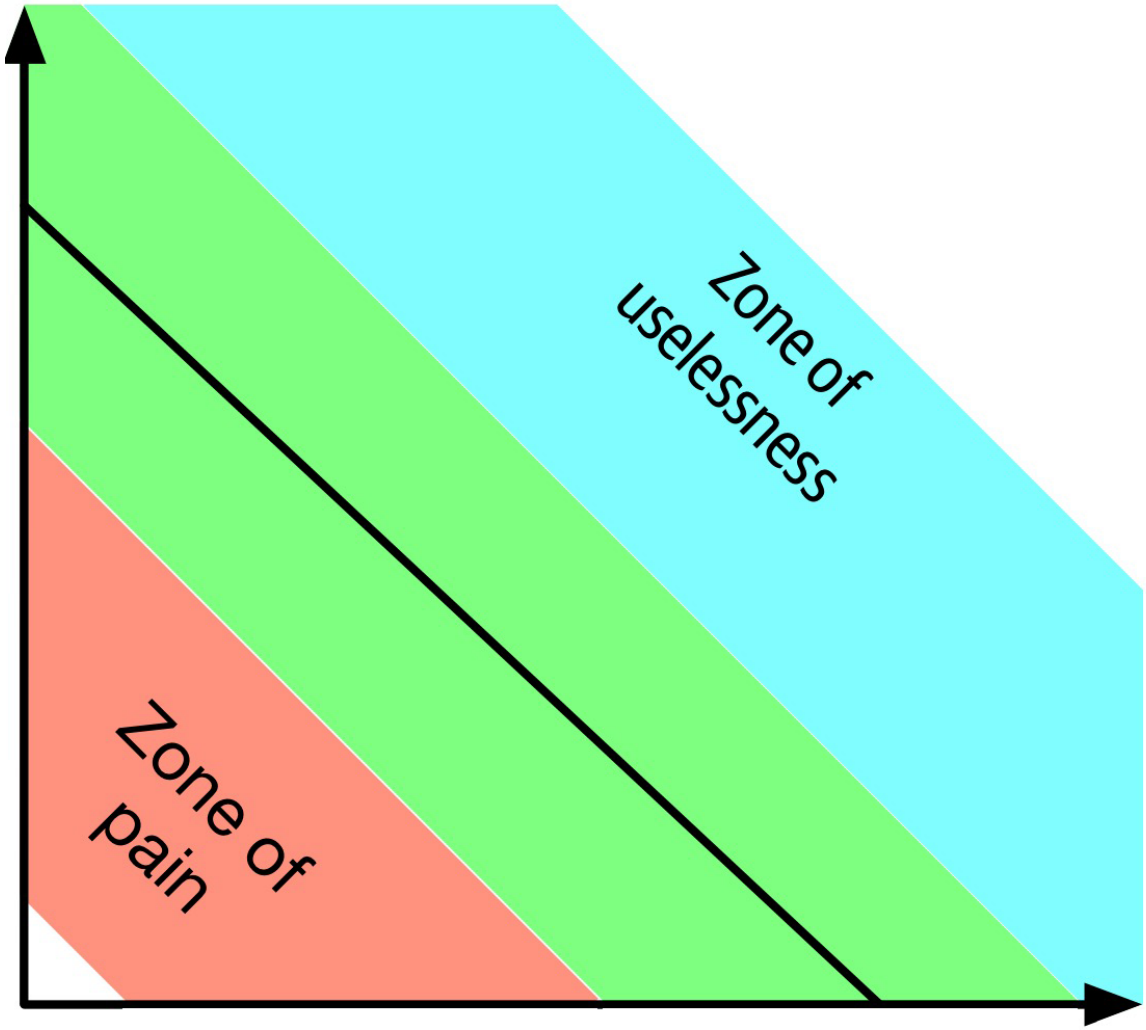

Figure 3-4. Zones of Uselessness and Pain

Tools exist in many platforms to provide these measures, which assist architects when analyzing code bases because of unfamiliarity, migration, or technical debt assessment.

LIMITATIONS OF METRICS

While the industry has a few code-level metrics that provide valuable insight into code bases, our tools are extremely blunt compared to analysis tools from other engineering disciplines. Even metrics derived directly from the structure of code require interpretation. For example, cyclomatic complexity (see “Cyclomatic Complexity”) measures complexity in code bases but cannot distinguish from *essential* complexity (because the underlying problem is complex) or *accidental complexity* (the code is more complex than it should be). Virtually all code-level metrics require interpretation, but it is still useful to establish baselines for critical metrics such as cyclomatic complexity so that architects can assess which type they exhibit. We discuss setting up just such tests in “Governance and Fitness Functions”.

Notice that the previously mentioned book by Edward Yourdon and and Larry Constantine (*Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*) predates the popularity of object-oriented languages, focusing instead on structured programming constructs, such as functions (not methods). It also defined other types of coupling that we do not cover here because they have been supplanted by *connascence*.

Connascence

In 1996, Meilir Page-Jones published *What Every Programmer Should Know About Object-Oriented Design* (Dorset House), refining the afferent and efferent coupling metrics and recasting them to object-oriented languages with a concept he named *connascence*. Here’s how he defined the term:

> *Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system.*
>
> —Meilir Page-Jones

He developed two types of connascence: *static* and *dynamic*.

Static connascence

Static connascence refers to source-code-level coupling (as opposed to execution-time coupling, covered in "Dynamic connascence"); it is a refinement of the afferent and efferent couplings defined by *Structured Design*. In other words, architects view the following types of static connascence as the *degree* to which something is coupled, either afferently or efferently:

Connascence of Name (CoN)

> Multiple components must agree on the name of an entity.
>
> Names of methods represents the most common way that code bases are coupled and the most desirable, especially in light of modern refactoring tools that make system-wide name changes trivial.

Connascence of Type (CoT)

> Multiple components must agree on the type of an entity.
>
> This type of connascence refers to the common facility in many statically typed languages to limit variables and parameters to specific types. However, this capability isn't purely a language feature—some dynamically typed languages offer selective typing, notably Clojure and Clojure Spec.

Connascence of Meaning (CoM) or Connascence of Convention (CoC)

> Multiple components must agree on the meaning of particular values.
>
> The most common obvious case for this type of connascence in code bases is hard-coded numbers rather than constants. For example, it is common in some languages to consider defining somewhere `int TRUE = 1; int FALSE = 0`. Imagine the problems if someone flips those values.

Connascence of Position (CoP)

Multiple entities must agree on the order of values.

This is an issue with parameter values for method and function calls even in languages that feature static typing. For example, if a developer creates a method `void updateSeat(String name, String seatLocation)` and calls it with the values `updateSeat("14D", "Ford, N")`, the semantics aren't correct even if the types are.

Connascence of Algorithm (CoA)

Multiple components must agree on a particular algorithm.

A common case for this type of connascence occurs when a developer defines a security hashing algorithm that must run on both the server and client and produce identical results to authenticate the user. Obviously, this represents a high form of coupling—if either algorithm changes any details, the handshake will no longer work.

Dynamic connascence

The other type of connascence Page-Jones defined was *dynamic connascence*, which analyses calls at runtime. The following is a description of the different types of dynamic connascence:

Connascence of Execution (CoE)

The order of execution of multiple components is important.

Consider this code:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

It won't work correctly because certain properties must be set in order.

Connascence of Timing (CoT)

The timing of the execution of multiple components is important.

The common case for this type of connascence is a race condition caused by two threads executing at the same time, affecting the outcome of the joint operation.

Connascence of Values (CoV)

Occurs when several values relate on one another and must change together.

Consider the case where a developer has defined a rectangle as four points, representing the corners. To maintain the integrity of the data structure, the developer cannot randomly change one of points without considering the impact on the other points.

The more common and problematic case involves transactions, especially in distributed systems. When an architect designs a system with separate databases, yet needs to update a single value across all of the databases, all the values must change together or not at all.

Connascence of Identity (CoI)

Occurs when several values relate on one another and must change together.

The common example of this type of connascence involves two independent components that must share and update a common data structure, such as a distributed queue.

Architects have a harder time determining dynamic connascence because we lack tools to analyze runtime calls as effectively as we can analyze the call graph.

Connascence properties

Connascence is an analysis tool for architect and developers, and some properties of connascence help developers use it wisely. The following is a description of each of these connascence properties:

Strength

Architects determine the *strength* of connascence by the ease with which a developer can refactor that type of coupling; different types of connascence are demonstrably more desirable, as shown in Figure 3-5. Architects and developers can improve the coupling characteristics of their code base by refactoring toward better types of connascence.

Architects should prefer static connascence to dynamic because developers can determine it by simple source code analysis, and modern tools make it trivial to improve static connascence. For example, consider the case of *connascence of meaning*, which developers can improve by refactoring to *connascence of name* by creating a named constant rather than a magic value.

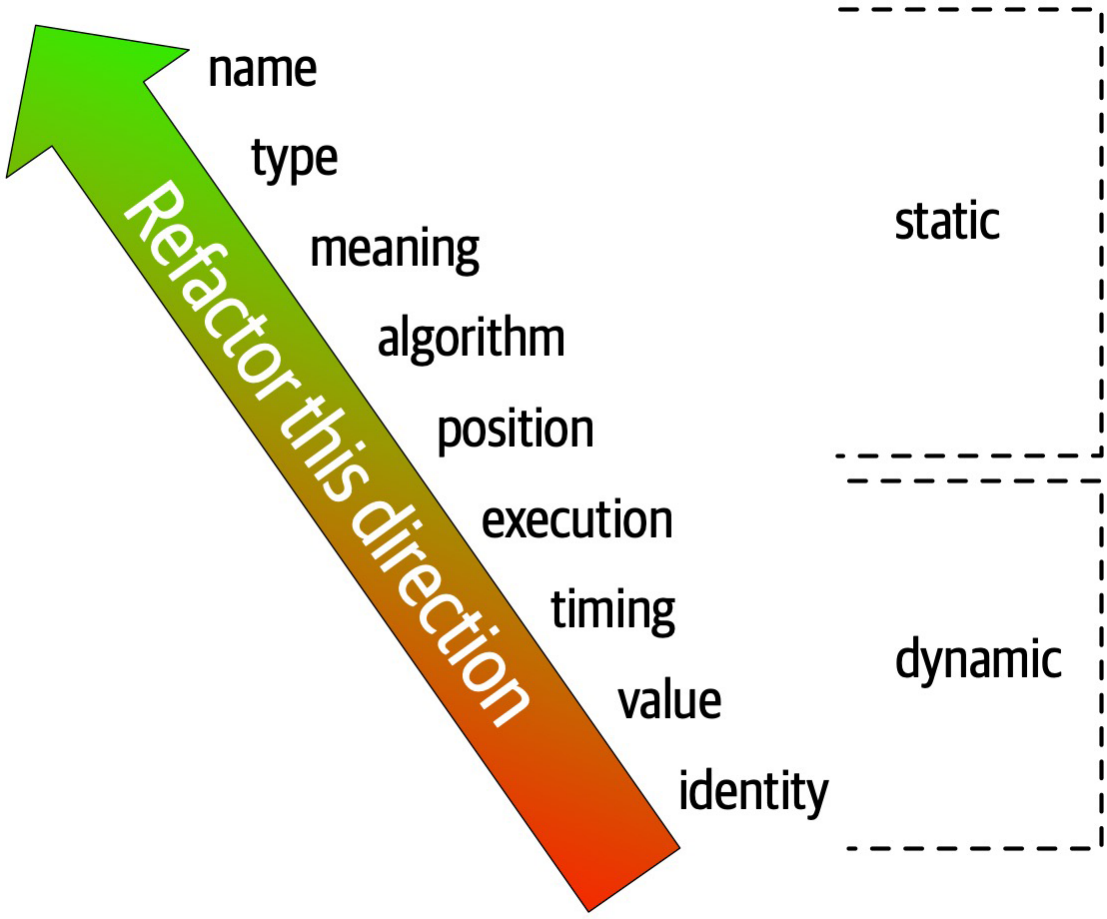

Figure 3-5. The strength on connascence provides a good refactoring guide

Locality

> The *locality* of connascence measures how proximal the modules are to each other in the code base. Proximal code (in the same module) typically has more and higher forms of connascence than more separated code (in separate modules or code bases). In other words, forms of connascence that indicate poor coupling when far apart are fine when closer together. For example, if two classes in the same component have connascence of meaning, it is less damaging to the code base than if two components have the same form of connascence.
>
> Developers must consider strength and locality together. Stronger forms of connascence found within the same module represent less code smell than the same connascence spread apart.

Degree

> The *degree* of connascence relates to the size of its impact—does it impact a few classes or many? Lesser degrees of connascence damage code bases less. In other words, having high dynamic connascence isn't terrible if you only have a few modules. However, code bases tend to grow, making a small problem correspondingly bigger.

Page-Jones offers three guidelines for using connascence to improve systems modularity:

1. Minimize overall connascence by breaking the system into encapsulated elements
2. Minimize any remaining connascence that crosses encapsulation boundaries
3. Maximize the connascence within encapsulation boundaries

The legendary software architecture innovator Jim Weirich repopularized the concept of connascence and offers two great pieces of advice:

Rule of Degree: convert strong forms of connascence into weaker forms of connascence

Rule of Locality: as the distance between software elements increases, use weaker forms of connascence

Unifying Coupling and Connascence Metrics

So far, we've discussed both coupling and connascence, measures from different eras and with different targets. However, from an architect's point of view, these two views overlap. What Page-Jones identifies as static connascence represents degrees of either incoming or outgoing coupling. Structured programming only cares about in or out, whereas connascence cares about how things are coupled together. To help visualize the overlap in concepts, consider Figure 3-6.

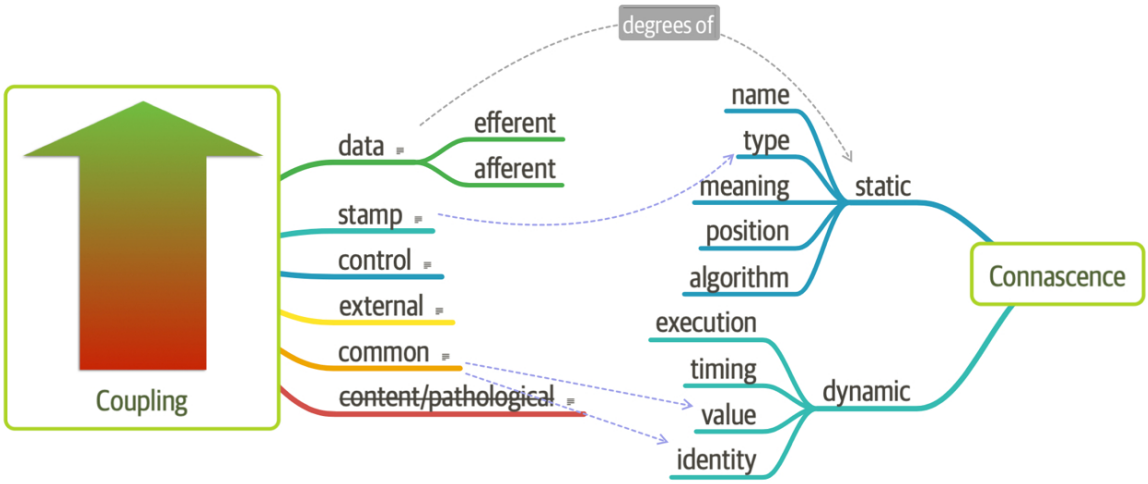

Figure 3-6. Unifying coupling and connascence

In Figure 3-6, the structured programming coupling concepts appear on the left, while the connascence characteristics appear on the right. What structured programming called *data coupling* (method calls), connascence provides advice for how that coupling should manifest. Structured programming didn't really address the areas covered by dynamic connascence; we encapsulate that concept shortly in "Architectural Quanta and Granularity".

The problems with 1990s connascence

Several problems exist for architects when applying these useful metrics for analyzing and designing systems. First, these measures look at details at a low level of code, focusing on code quality and hygiene than necessarily architectural structure. Architects tend to care more about *how* modules are coupled rather than the *degree* of coupling. For example, an architect cares about synchronous versus asynchronous communication, and doesn't care so much about how that's implemented.

The second problem with connascence lies with the fact that it doesn't really address a fundamental decision that many modern architects must make—synchronous or asynchronous communication in distributed architectures like microservices? Referring back to the First Law of Software Architecture, everything is a trade-off. After we discuss the scope of architecture characteristics in Chapter 7, we'll introduce new ways to think about modern connascence.

From Modules to Components

We use the term *module* throughout as a generic name for a bundling of related code. However, most platforms support some form of *component*, one of the key building blocks for software architects. The concept and corresponding analysis of the logical or physical separation has existed since the earliest days of computer science. Yet, with all the writing and thinking about components and separation, developers and architects still struggle with achieving good outcomes.

We'll discuss deriving components from problem domains in Chapter 8, but we must first discuss another fundamental aspect of software architecture: architecture characteristics and their scope.

Chapter 4. Architecture Characteristics Defined

A company decides to solve a particular problem using software, so it gathers a list of requirements for that system. A wide variety of techniques exist for the exercise of requirements gathering, generally defined by the software development process used by the team. But the architect must consider many other factors in designing a software solution, as illustrated in Figure 4-1.

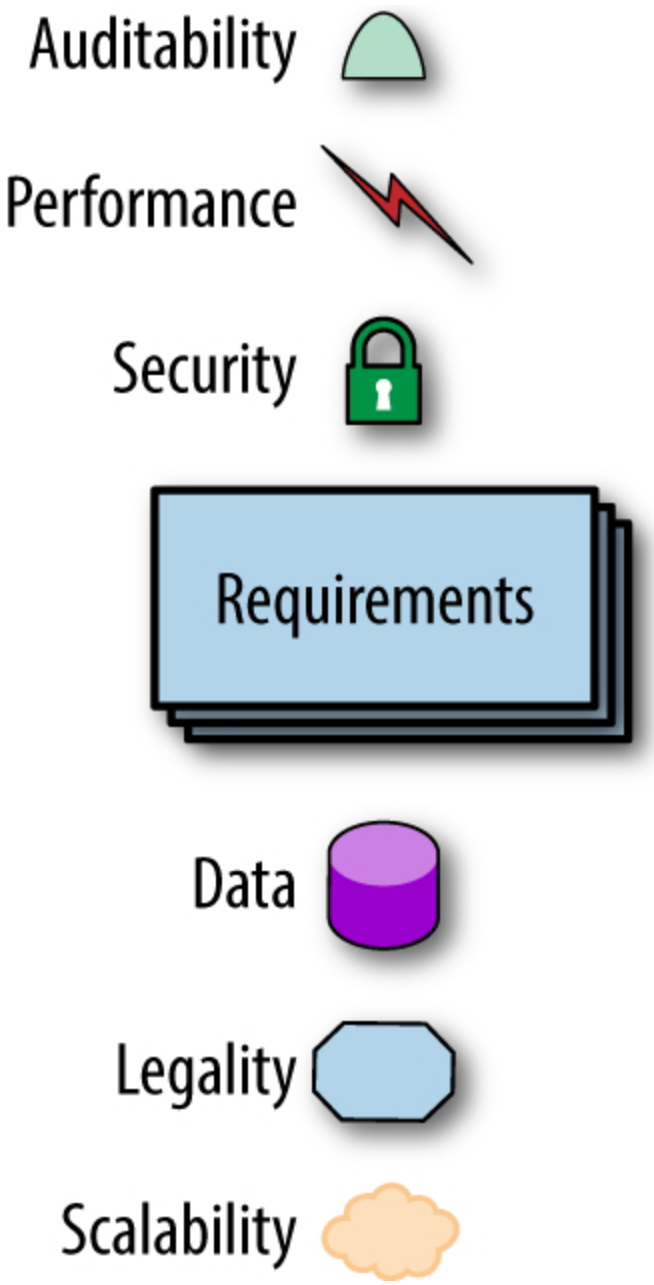

Figure 4-1. A software solution consists of both domain requirements and architectural characteristics

Architects may collaborate on defining the domain or business requirements, but one key responsibility entails defining, discovering, and otherwise analyzing all the things the software must do that isn't directly related to the domain functionality: *architectural characteristics*.

What distinguishes software architecture from coding and design? Many things, including the role that architects have in defining architectural

characteristics, the important aspects of the system independent of the problem domain. Many organizations describe these features of software with a variety of terms, including *nonfunctional requirements*, but we dislike that term because it is self-denigrating. Architects created that term to distinguish architecture characteristics from *functional requirements*, but naming something *nonfunctional* has a negative impact from a language standpoint: how can teams be convinced to pay enough attention to something "nonfunctional"? Another popular term is *quality attributes*, which we dislike because it implies after-the-fact quality assessment rather than design. We prefer *architecture characteristics* because it describes concerns critical to the success of the architecture, and therefore the system as a whole, without discounting its importance.

An architecture characteristic meets three criteria:

- Specifies a nondomain design consideration
- Influences some structural aspect of the design
- Is critical or important to application success

These interlocking parts of our definition are illustrated in Figure 4-2.

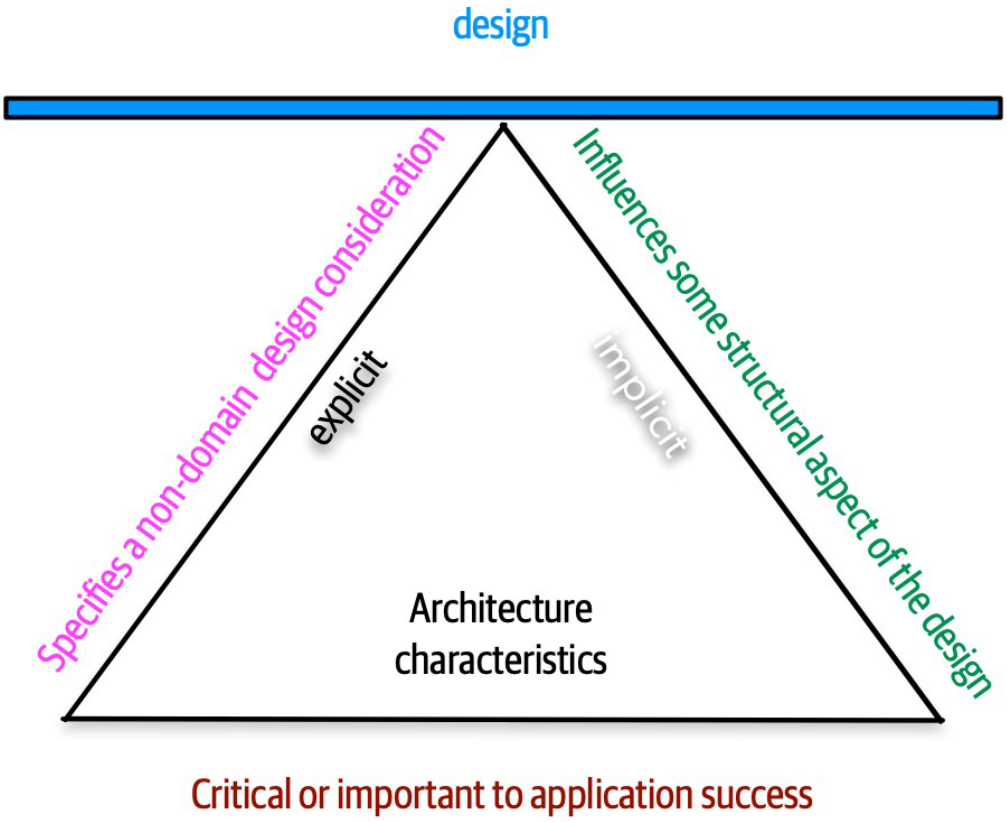

Figure 4-2. The differentiating features of architecture characteristics

The definition illustrated in Figure 4-2 consists of the three components listed, in addition to a few modifiers:

Specifies a nondomain design consideration

> When designing an application, the requirements specify what the application should do; architecture characteristics specify operational and design criteria for success, concerning how to implement the requirements and why certain choices were made. For example, a common important architecture characteristic specifies a certain level of performance for the application, which often doesn't appear in a requirements document. Even more pertinent: no requirements document states "prevent technical debt," but it is a common design consideration for architects and developers. We cover this distinction between explicit and implicit characteristics in depth in "Extracting Architecture Characteristics from Domain Concerns".

Influences some structural aspect of the design

The primary reason architects try to describe architecture characteristics on projects concerns design considerations: does this architecture characteristic require special structural consideration to succeed? For example, *security* is a concern in virtually every project, and all systems must take a baseline of precautions during design and coding. However, it rises to the level of architecture characteristic when the architect needs to design something special. Consider two cases surrounding payment in a example system:

Third-party payment processor

If an integration point handles payment details, then the architecture shouldn't require special structural considerations. The design should incorporate standard security hygiene, such as encryption and hashing, but doesn't require special structure.

In-application payment processing

If the application under design must handle payment processing, the architect may design a specific module, component, or service for that purpose to isolate the critical security concerns structurally. Now, the architecture characteristic has an impact on both architecture and design.

Of course, even these two criteria aren't sufficient in many cases to make this determination: past security incidents, the nature of the integration with the third party, and a host of other criteria may be present during this decision. Still, it shows some of the considerations architects must make when determining how to design for certain capabilities.

Critical or important to application success

Applications *could* support a huge number of architecture characteristics…but shouldn't. Support for each architecture characteristic adds complexity to the design. Thus, a critical job for

architects lies in choosing the fewest architecture characteristics rather than the most possible.

We further subdivide architecture characteristics into implicit versus explicit architecture characteristics. Implicit ones rarely appear in requirements, yet they're necessary for project success. For example, availability, reliability, and security underpin virtually all applications, yet they're rarely specified in design documents. Architects must use their knowledge of the problem domain to uncover these architecture characteristics during the analysis phase. For example, a high-frequency trading firm may not have to specify low latency in every system, yet the architects in that problem domain know how critical it is. Explicit architecture characteristics appear in requirements documents or other specific instructions.

In Figure 4-2, the choice of a triangle is intentional: each of the definition elements supports the others, which in turn support the overall design of the system. The fulcrum created by the triangle illustrates the fact that these architecture characteristics often interact with one another, leading to the pervasive use among architects of the term *trade-off*.

Architectural Characteristics (Partially) Listed

Architecture characteristics exist along a broad spectrum of the software system, ranging from low-level code characteristics, such as modularity, to sophisticated operational concerns, such as scalability and elasticity. No true universal standard exists despite attempts to codify ones in the past. Instead, each organization creates its own interpretation of these terms. Additionally, because the software ecosystem changes so fast, new concepts, terms, measures, and verifications constantly appear, providing new opportunities for architecture characteristics definitions.

Despite the volume and scale, architects commonly separate architecture characteristics into broad categories. The following sections describe a few,

along with some examples.

Operational Architecture Characteristics

Operational architecture characteristics cover capabilities such as performance, scalability, elasticity, availability, and reliability. Table 4-1 lists some operational architecture characteristics.

Table 4-1. Common operational architecture characteristics

Term	Definition
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there's a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

Operational architecture characteristics heavily overlap with operations and DevOps concerns, forming the intersection of those concerns in many software projects.

Structural Architecture Characteristics

Architects must concern themselves with code structure as well. In many cases, the architect has sole or shared responsibility for code quality concerns, such as good modularity, controlled coupling between components, readable code, and a host of other internal quality assessments.

Table 4-2 lists a few structural architecture characteristics.

Table 4-2. Structural architecture characteristics

Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

Cross-Cutting Architecture Characteristics

While many architecture characteristics fall into easily recognizable categories, many fall outside or defy categorization yet form important

design constraints and considerations. Table 4-3 describes a few of these.

Table 4-3. Cross-cutting architecture characteristics

Term	Definition
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

Any list of architecture characteristics will necessarily be an incomplete list; any software may invent important architectural characteristics based

on unique factors (see "Italy-ility" for an example).

ITALY-ILITY

One of Neal's colleagues recounts a story about the unique nature of architectural characteristics. She worked for a client whose mandate required a centralized architecture. Yet, for each proposed design, the first question from the client was "But what happens if we loose Italy?" Years ago, because of a freak communication outage, the head office had lost communication with the Italian branches, and it was organizationally traumatic. Thus, a firm requirement of all future architectures insisted upon what the team eventually called *Italy-ility*, which they all knew meant a unique combination of availability, recoverability, and resilience.

Additionally, many of the preceding terms are imprecise and ambiguous, sometimes because of subtle nuance or the lack of objective definitions. For example, *interoperability* and *compatibility* may appear equivalent, which will be true for some systems. However, they differ because *interoperability* implies ease of integration with other systems, which in turn implies published, documented APIs. *Compatibility*, on the other hand, is more concerned with industry and domain standards. Another example is *learnability*. One definition is how easy it is for users to learn to use the software, and another definition is the level at which the system can automatically learn about its environment in order to become self-configuring or self-optimizing using machine learning algorithms.

Many of the definitions overlap. For example, consider availability and reliability, which seem to overlap in almost all cases. Yet consider the internet protocol UDP, which underlies TCP. UDP is available over IP but not reliable: the packets may arrive out of order, and the receiver may have to ask for missing packets again.

No complete list of standards exists. The International Organization for Standards (ISO) publishes a list organized by capabilities, overlapping

many of the ones we've listed, but mainly establishing an incomplete category list. The following are some of the ISO definitions:

Performance efficiency

> Measure of the performance relative to the amount of resources used under known conditions. This includes *time behavior* (measure of response, processing times, and/or throughput rates), *resource utilization* (amounts and types of resources used), and *capacity* (degree to which the maximum established limits are exceeded).

Compatibility

> Degree to which a product, system, or component can exchange information with other products, systems, or components and/or perform its required functions while sharing the same hardware or software environment. It includes *coexistence* (can perform its required functions efficiently while sharing a common environment and resources with other products) and *interoperability* (degree to which two or more systems can exchange and utilize information).

Usability

> Users can use the system effectively, efficiently, and satisfactorily for its intended purpose. It includes *appropriateness recognizability* (users can recognize whether the software is appropriate for their needs), *learnability* (how easy users can learn how to use the software), *user error protection* (protection against users making errors), and *accessibility* (make the software available to people with the widest range of characteristics and capabilities).

Reliability

> Degree to which a system functions under specified conditions for a specified period of time. This characteristic includes subcategories such as *maturity* (does the software meet the reliability needs under normal operation), *availability* (software is operational and accessible), *fault tolerance* (does the software operate as intended despite hardware or

software faults), and *recoverability* (can the software recover from failure by recovering any affected data and reestablish the desired state of the system.

Security

Degree the software protects information and data so that people or other products or systems have the degree of data access appropriate to their types and levels of authorization. This family of characteristics includes *confidentiality* (data is accessible only to those authorized to have access), *integrity* (the software prevents unauthorized access to or modification of software or data), *nonrepudiation*, (can actions or events be proven to have taken place), *accountability* (can user actions of a user be traced), and *authenticity* (proving the identity of a user).

Maintainability

Represents the degree of effectiveness and efficiency to which developers can modify the software to improve it, correct it, or adapt it to changes in environment and/or requirements. This characteristic includes *modularity* (degree to which the software is composed of discrete components), *reusability* (degree to which developers can use an asset in more than one system or in building other assets), *analyzability* (how easily developers can gather concrete metrics about the software), *modifiability* (degree to which developers can modify the software without introducing defects or degrading existing product quality), and *testability* (how easily developers and others can test the software).

Portability

Degree to which developers can transfer a system, product, or component from one hardware, software, or other operational or usage environment to another. This characteristic includes the subcharacteristics of *adaptability* (can developers effectively and efficiently adapt the software for different or evolving hardware, software, or other operational or usage environments), *installability* (can

the software be installed and/or uninstalled in a specified environment), and *replaceability* (how easily developers can replace the functionality with other software).

The last item in the ISO list addresses the functional aspects of software, which we do not believe belongs in this list:

Functional suitability
: This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following subcharacteristics:

  Functional completeness
  : Degree to which the set of functions covers all the specified tasks and user objectives.

  Functional correctness
  : Degree to which a product or system provides the correct results with the needed degree of precision.

  Functional appropriateness
  : Degree to which the functions facilitate the accomplishment of specified tasks and objectives. These are not architecture characteristics but rather the motivational requirements to build the software. This illustrates how thinking about the relationship between architecture characteristics and the problem domain has evolved. We cover this evolution in Chapter 7.

THE MANY AMBIGUITIES IN SOFTWARE ARCHITECTURE

A consistent frustration amongst architects is the lack of clear definitions of so many critical things, including the activity of software architecture itself! This leads companies to define their own terms for common things, which leads to industry-wide confusion because architects either use opaque terms or, worse yet, use the same terms for wildly different meanings. As much as we'd like, we can't impose a standard nomenclature on the software development world. However, we do follow and recommend the advice from domain-driven design to establish and use a ubiquitous language amongst fellow employees to help ensure fewer term-based misunderstandings.

Trade-Offs and Least Worst Architecture

Applications can only support a few of the architecture characteristics we've listed for a variety of reasons. First, each of the supported characteristics requires design effort and perhaps structural support. Second, the bigger problem lies with the fact that each architecture characteristic often has an impact on others. For example, if an architect wants to improve *security*, it will almost certainly negatively impact *performance*: the application must do more on-the-fly encryption, indirection for secrets hiding, and other activities that potentially degrade performance.

A metaphor will help illustrate this interconnectivity. Apparently, pilots often struggle learning to fly helicopters because it requires a control for each hand and each foot, and changing one impacts the others. Thus, flying a helicopter is a balancing exercise, which nicely describes the trade-off process when choosing architecture characteristics. Each architecture characteristic that an architect designs support for potentially complicates the overall design.

Thus, architects rarely encounter the situation where they are able to design a system and maximize every single architecture characteristic. More often, the decisions come down to trade-offs between several competing concerns.

> **TIP**
>
> Never shoot for the *best* architecture, but rather the *least worst* architecture.

Too many architecture characteristics leads to generic solutions that are trying to solve every business problem, and those architectures rarely work because the design becomes unwieldy.

This suggests that architects should strive to design architecture to be as iterative as possible. If you can make changes to the architecture more easily, you can stress less about discovering the exact correct thing in the first attempt. One of the most important lessons of Agile software development is the value of iteration; this holds true at all levels of software development, including architecture.

Chapter 5. Identifying Architectural Characteristics

Identifying the driving architectural characteristics is one of the first steps in creating an architecture or determining the validity of an existing architecture. Identifying the correct architectural characteristics ("-ilities") for a given problem or application requires an architect to not only understand the domain problem, but also collaborate with the problem domain stakeholders to determine what is truly important from a domain perspective.

An architect uncovers architecture characteristics in at least three ways by extracting from domain concerns, requirements, and implicit domain knowledge. We previously discussed implicit characteristics and we cover the other two here.

Extracting Architecture Characteristics from Domain Concerns

An architect must be able to translate domain concerns to identify the right architectural characteristics. For example, is scalability the most important concern, or is it fault tolerance, security, or performance? Perhaps the system requires all four characteristics combined. Understanding the key domain goals and domain situation allows an architect to translate those domain concerns to "-ilities," which then forms the basis for correct and justifiable architecture decisions.

One tip when collaborating with domain stakeholders to define the driving architecture characteristics is to work hard to keep the final list as short as possible. A common anti-pattern in architecture entails trying to design a *generic architecture*, one that supports *all* the architecture characteristics.

Each architecture characteristic the architecture supports complicates the overall system design; supporting too many architecture characteristics leads to greater and greater complexity before the architect and developers have even started addressing the problem domain, the original motivation for writing the software. Don’t obsess over the number of charateristics, but rather the motivation to keep design simple.

CASE STUDY: THE VASA

The original story of over-specifying architecture characteristics and ultimately killing a project must be the Vasa. It was a Swedish warship built between 1626 and 1628 by a king who wanted the most magnificent ship ever created. Up until that time, ships were either troop transports or gunships—the Vasa would be both! Most ships had one deck—the Vasa had two! All the cannons were twice the size of those on similar ships. Despite some trepidation by the expert ship builders (who ultimately couldn’t say no to King Adolphus), the shipbuilders finished the construction. In celebration, the ship sailed out into the harbor and shot a cannon salute off one side. Unfortunately, because the ship was top-heavy, it capsized and sank to the bottom of the bay in Sweden. In the early 20th century, salvagers rescued the ship, which now resides in a museum in Stockholm.

Many architects and domain stakeholders want to prioritize the final list of architecture characteristics that the application or system must support. While this is certainly desirable, in most cases it is a fool’s errand and will not only waste time, but also produce a lot of unnecessary frustration and disagreement with the key stakeholders. Rarely will all stakeholders agree on the priority of each and every characteristic. A better approach is to have the domain stakeholders select the top three most important characteristics from the final list (in any order). Not only is this much easier to gain consensus on, but it also fosters discussions about what is most important and helps the architect analyze trade-offs when making vital architecture decisions.

Most architecture characteristics come from listening to key domain stakeholders and collaborating with them to determine what is important from a domain perspective. While this may seem like a straightforward activity, the problem is that architects and domain stakeholders speak different languages. Architects talk about scalability, interoperability, fault tolerance, learnability, and availability. Domain stakeholders talk about mergers and acquisitions, user satisfaction, time to market, and competitive advantage. What happens is a "lost in translation" problem where the architect and domain stakeholder don't understand each other. Architects have no idea how to create an architecture to support user satisfaction, and domain stakeholders don't understand why there is so much focus and talk about availability, interoperability, learnability, and fault tolerance in the application. Fortunately, there is usually a translation from domain concerns to architecture characteristics. Table 5-1 shows some of the more common domain concerns and the corresponding "-ilities" that support them.

Table 5-1. Translation of domain concerns to architecture characteristics

Domain concern	Architecture characteristics
Mergers and acquisitions	Interoperability, scalability, adaptability, extensibility
Time to market	Agility, testability, deployability
User satisfaction	Performance, availability, fault tolerance, testability, deployability, agility, security
Competitive advantage	Agility, testability, deployability, scalability, availability, fault tolerance
Time and budget	Simplicity, feasibility

One important thing to note is that agility does not equal time to market. Rather, it is agility + testability + deployability. This is a trap many architects fall into when translating domain concerns. Focusing on only one

of the ingredients is like forgetting to put the flour in the cake batter. For example, a domain stakeholder might say something like “Due to regulatory requirements, it is absolutely imperative that we complete end-of-day fund pricing on time.” An ineffective architect might just focus on performance because that seems to be the primary focus of that domain concern. However, that architect will fail for many reasons. First, it doesn’t matter how fast the system is if it isn’t available when needed. Second, as the domain grows and more funds are created, the system must be able to also scale to finish end-of-day processing in time. Third, the system must not only be available, but must also be reliable so that it doesn’t crash as end-of-day fund prices are being calculated. Forth, what happens if the end-of-day fund pricing is about 85% complete and the system crashes? It must be able to recover and restart where the pricing left off. Finally, the system may be fast, but are the fund prices being calculated correctly? So, in addition to performance, the architect must also equally place a focus on availability, scalability, reliability, recoverability, and auditability.

Extracting Architecture Characteristics from Requirements

Some architecture characteristics come from explicit statements in requirements documents. For example, explicit expected numbers of users and scale commonly appear in domain or domain concerns. Others come from inherent domain knowledge by architects, one of the many reasons that domain knowledge is always beneficial for architects. For example, suppose an architect designs an application that handles class registration for university students. To make the math easy, assume that the school has 1,000 students and 10 hours for registration. Should an architect design a system assuming consistent scale, making the implicit assumption that the students during the registration process will distribute themselves evenly over time? Or, based on knowledge of university students habits and proclivities, should the architect design a system that can handle all 1,000 students attempting to register in the last 10 minutes? Anyone who

understands how much students stereotypically procrastinate knows the answer to this question! Rarely will details like this appear in requirements documents, yet they do inform the design decisions.

THE ORIGIN OF ARCHITECTURE KATAS

A few years ago, Ted Neward, a well-known architect, devised architecture katas, a clever method to allow nascent architects a way to practice deriving architecture characteristics from domain-targeted descriptions. From Japan and martial arts, a *kata* is an individual training exercise, where the emphasis lies on proper form and technique.

> *How do we get great designers? Great designers design, of course.*
>
> —Fred Brooks

So how are we supposed to get great architects if they only get the chance to architect fewer than a half dozen times in their career?

To provide a curriculum for aspiring architects, Ted created the first architecture katas site, which your authors Neal and Mark adapted and updated. The basic premise of the kata exercise provides architects with a problem stated in domain terms and additional context (things that might not appear in requirements yet impact design). Small teams work for 45 minutes on a design, then show results to the other groups, who vote on who came up with the best architecture. True to its original purpose, architecture katas provide a useful laboratory for aspiring architects.

Each kata has predefined sections:

Description

: The overall domain problem the system is trying to solve

Users

: The expected number and/or types of users of the system

Requirements

: Domain/domain-level requirements, as an architect might expect from domain users/domain experts

Neal updated the format a few years later on his blog to add the *additional context* section to each kata with important additional considerations, making the exercises more realistic.

Additional context

> Many of the considerations an architect must make aren't explicitly expressed in requirements but rather by implicit knowledge of the problem domain

We encourage burgeoning architects to use the site to do their own kata exercise. Anyone can host a brown-bag lunch where a team of aspiring architects can solve a problem and get an experienced architect to evaluate the design and trade-off analysis, either on the spot or from a short analysis after the fact. The design won't be elaborate because the exercise is timeboxed. Team members ideally get feedback from the experienced architecture about missed trade-offs and alternative designs.

Case Study: Silicon Sandwiches

To illustrate several concepts, we use an *architecture kata* (see "The Origin of Architecture Katas" for the origin of the concept). To show how architects derive architecture characteristics from requirements, we introduce the Silicon Sandwiches kata.

Description

> A national sandwich shop wants to enable online ordering (in addition to its current call-in service).

Users

> Thousands, perhaps one day millions

Requirements

- Users will place their order, then be given a time to pick up their sandwich and directions to the shop (which must integrate with several external mapping services that include traffic information)
- If the shop offers a delivery service, dispatch the driver with the sandwich to the user
- Mobile-device accessibility
- Offer national daily promotions/specials
- Offer local daily promotions/specials
- Accept payment online, in person, or upon delivery

Additional context

- Sandwich shops are franchised, each with a different owner
- Parent company has near-future plans to expand overseas
- Corporate goal is to hire inexpensive labor to maximize profit

Given this scenario, how would an architect derive architecture characteristics? Each part of the requirement might contribute to one or more aspects of architecture (and many will not). The architect doesn't design the entire system here—considerable effort must still go into crafting code to solve the domain statement. Instead, the architect looks for things that influence or impact the design, particularly structural.

First, separate the candidate architecture characteristics into explicit and implicit characteristics.

Explicit Characteristics

Explicit architecture characteristics appear in a requirements specification as part of the necessary design. For example, a shopping website may aspire to support a particular number of concurrent users, which domain analysts specify in the requirements. An architect should consider each part of the

requirements to see if it contributes to an architecture characteristic. But first, an architect should consider domain-level predictions about expected metrics, as represented in the Users section of the kata.

One of the first details that should catch an architect's eye is the number of users: currently thousands, perhaps one day millions (this is a very ambitious sandwich shop!). Thus, *scalability*—the ability to handle a large number of concurrent users without serious performance degradation—is one of the top architecture characteristics. Notice that the problem statement didn't explicitly ask for scalability, but rather expressed that requirement as an expected number of users. Architects must often decode domain language into engineering equivalents.

However, we also probably need *elasticity*—the ability to handle bursts of requests. These two characteristics often appear lumped together, but they have different constraints. Scalability looks like the graph shown in Figure 5-1.

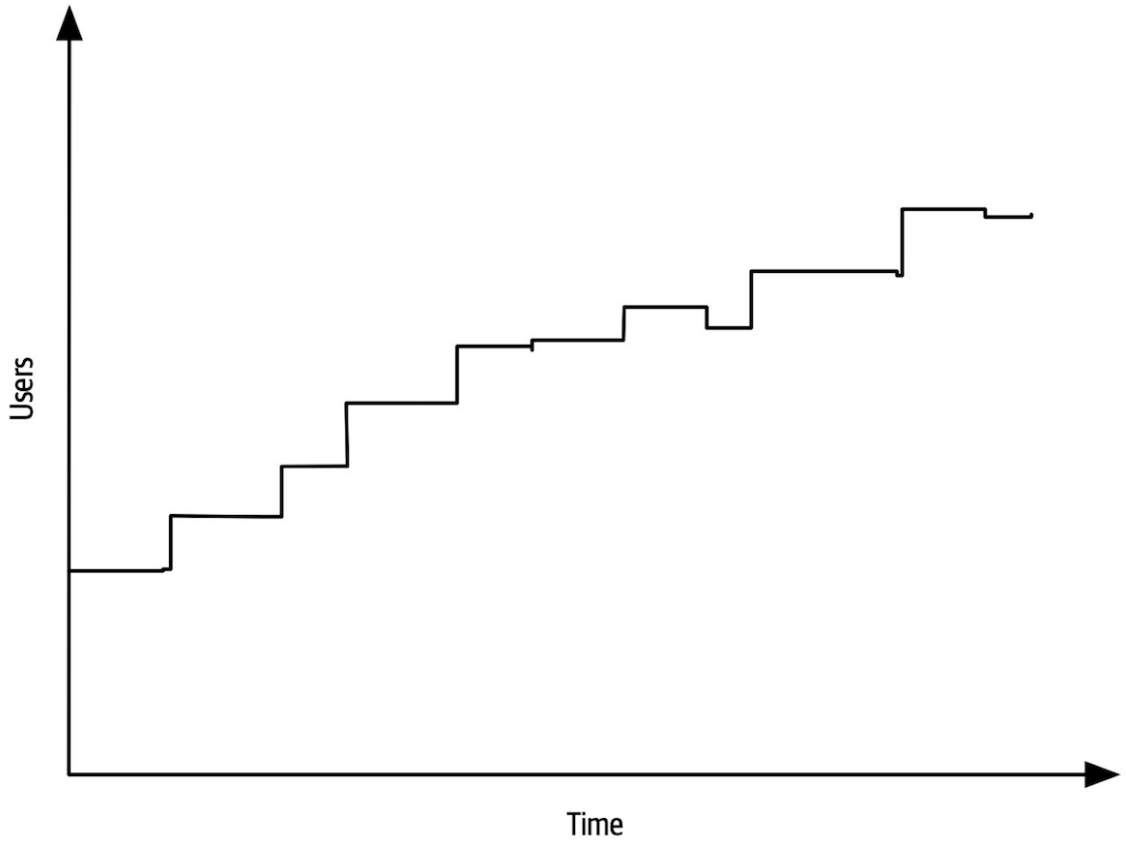

Figure 5-1. Scalability measures the performance of concurrent users

Elasticity, on the other hand, measures bursts of traffic, as shown in Figure 5-2.

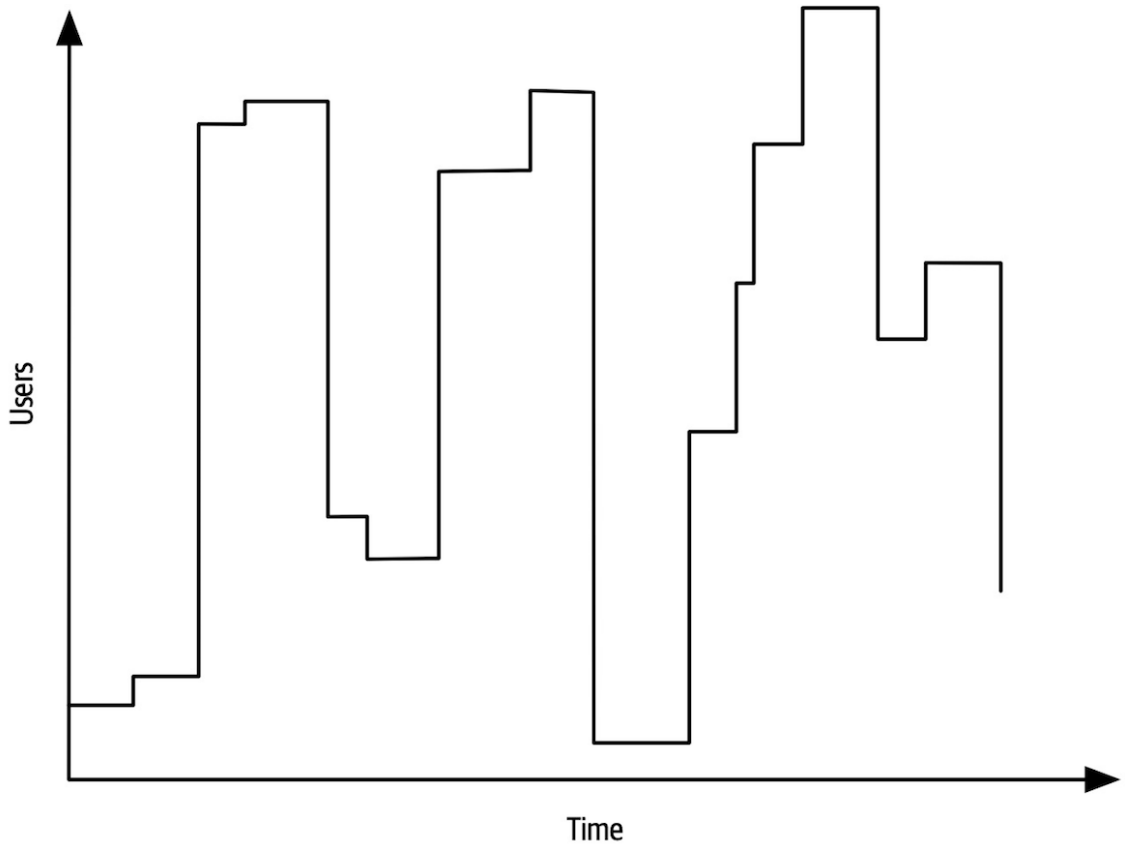

Figure 5-2. Elastic systems must withstand bursts of users

Some systems are scalable but not elastic. For example, consider a hotel reservation system. Absent special sales or events, the number of users is probably consistent. In contrast, consider a concert ticket booking system. As new tickets go on sale, fervent fans will flood the site, requiring high degrees of elasticity. Often, elastic systems also need scalability: the ability to handle bursts and high numbers of concurrent users.

The requirement for elasticity did not appear in the Silicon Sandwiches requirements, yet the architect should identify this as an important consideration. Requirements sometimes state architecture characteristics outright, but some lurk inside the problem domain. Consider a sandwich shop. Is its traffic consistent throughout the day? Or does it endure bursts of

traffic around mealtimes? Almost certainly the latter. Thus, a good architect should identify this potential architecture characteristic.

An architect should consider each of these business requirements in turn to see if architecture characteristics exist:

1. Users will place their order, then be given a time to pick up their sandwich and directions to the shop (which must provide the option to integrate with external mapping services that include traffic information).

   External mapping services imply integration points, which may impact aspects such as reliability. For example, if a developer builds a system that relies on a third-party system, yet calling it fails, it impacts the reliability of the calling system. However, architects must also be wary of over-specifying architecture characteristics. What if the external traffic service is down? Should the Silicon Sandwiches site fail, or should it just offer slightly less efficiency without traffic information? Architects should always guard against building unnecessary brittleness or fragility into designs.

2. If the shop offers a delivery service, dispatch the driver with the sandwich to the user.

   No special architecture characteristics seem necessary to support this requirement.

3. Mobile-device accessibility.

   This requirement will primarily affect the *design* of the application, pointing toward building either a portable web application or several native web applications. Given the budget constraints and simplicity of the application, an architect would likely deem it overkill to build multiple applications, so the design points toward a mobile-optimized web application. Thus, the architect may want to define some specific performance architecture characteristics for page load time and other mobile-sensitive characteristics. Notice

that the architect shouldn’t act alone in situations like this, but should instead collaborate with user experience designers, domain stakeholders, and other interested parties to vet decisions like this.

4. Offer national daily promotions/specials.
5. Offer local daily promotions/specials.

   Both of these requirements specify customizability across both promotions and specials. Notice that requirement 1 also implies customized traffic information based on address. Based on all three of these requirements, the architect may consider customizability as an architecture characteristic. For example, an architecture style such as microkernel architecture supports customized behavior extremely well by defining a plug-in architecture. In this case, the default behavior appears in the core, and developers write the optional customized parts, based on location, via plug-ins. However, a traditional design can also accommodate this requirement via design patterns (such as Template Method). This conundrum is common in architecture and requires architects to constantly weight trade-offs between competing options. We discuss particular trade-off in more detail in “Design Versus Architecture and Trade-Offs”.

6. Accept payment online, in person, or upon delivery.

   Online payments imply security, but nothing in this requirement suggests a particularly heightened level of security beyond what’s implicit.

7. Sandwich shops are franchised, each with a different owner.

   This requirement may impose cost restrictions on the architecture—the architect should check the feasibility (applying constraints like cost, time, and staff skill set) to see if a simple or sacrificial architecture is warranted.

8. Parent company has near-future plans to expand overseas.

This requirement implies *internationalization*, or *i18n*. Many design techniques exist to handle this requirement, which shouldn't require special structure to accommodate. This will, however, certainly drive design decisions.

9. Corporate goal is to hire inexpensive labor to maximize profit.

   This requirement suggests that usability will be important, but again is more concerned with design than architecture characteristics.

The third architecture characteristic we derive from the preceding requirements is *performance*: no one wants to buy from a sandwich shop that has poor performance, especially at peak times. However, *performance* is a nuanced concept—what *kind* of performance should the architect design for? We cover the various nuances of performance in Chapter 6.

We also want to define performance numbers in conjunction with scalability numbers. In other words, we must establish a baseline of performance without particular scale, as well as determine what an acceptable level of performance is given a certain number of users. Quite often, architecture characteristics interact with one another, forcing architects to define them in relation to one another.

Implicit Characteristics

Many architecture characteristics aren't specified in requirements documents, yet they make up an important aspect of the design. One implicit architecture characteristic the system might want to support is *availability*: making sure users can access the sandwich site. Closely related to availability is *reliability*: making sure the site stays up during interactions—no one wants to purchase from a site that continues dropping connections, forcing them to log in again.

Security appears as an implicit characteristic in every system: no one wants to create insecure software. However, it may be prioritized depending on criticality, which illustrates the interlocking nature of our definition. An

architect considers security an architecture characteristic if it influences some structural aspect of the design and is critical or important to the application.

For Silicon Sandwiches, an architect might assume that payments should be handled by a third party. Thus, as long as developers follow general security hygiene (not passing credit card numbers as plain text, not storing too much information, and so on), the architect shouldn't need any special structural design to accommodate security; good design in the application will suffice. Each architecture characteristic interacts with the others, leading to the common pitfall of architects of over-specifying architecture characteristics, which is just as damaging as under-specifying them because it overcomplicates the system design.

The last major architecture characteristic that Silicon Sandwiches needs to support encompasses several details from the requirements: *customizability*. Notice that several parts of the problem domain offer custom behavior: recipes, local sales, and directions that may be locally overridden. Thus, the architecture should support the ability to facilitate custom behavior. Normally, this would fall into the design of the application. However, as our definition specifies, a part of the problem domain that relies on custom structure to support it moves into the realm of an architecture characteristic. This design element isn't critical to the success of the application though. It is important to note that there are no correct answers in choosing architecture characteristics, only incorrect ones (or, as Mark notes in one of his well-known quotes):

> *There are no wrong answers in architecture, only expensive ones.*

DESIGN VERSUS ARCHITECTURE AND TRADE-OFFS

In the Silicon Sandwiches kata, an architect would likely identify customizability as a part of the system, but the question then becomes: architecture or design? The architecture implies some structural component, whereas design resides within the architecture. In the customizability case of Silicon Sandwiches, the architect could choose an architecture style like microkernel and build structural support for customization. However, if the architect chose another style because of competing concerns, developers could implement the customization using the Template Method design pattern, which allows parent classes to define workflow that can be overridden in child classes. Which design is better?

Like in all architecture, it depends on a number of factors. First, are there good reasons, such as performance and coupling, not to implement a microkernel architecture? Second, are other desirable architecture characteristics more difficult in one design versus the other? Third, how much would it cost to support all the architecture characteristics in each design versus pattern? This type of architectural trade-off analysis makes up an important part of an architect's role.

Above all, it is critical for the architect to collaborate with the developers, project manager, operations team, and other co-constructors of the software system. No architecture decision should be made isolated from the implementation team (which leads to the dreaded *Ivory Tower Architect* anti-pattern). In the case of Silicon Sandwiches, the architect, tech lead, developers, and domain analysts should collaborate to decide how best to implement customizability.

An architect could design an architecture that doesn't accommodate customizability structurally, requiring the design of the application itself to support that behavior (see "Design Versus Architecture and Trade-Offs"). Architects shouldn't stress too much about discovering the exactly correct set of architecture characteristics—developers can implement functionality

in a variety of ways. However, correctly identifying important structural elements may facilitate a simpler or more elegant design. Architects must remember: there is no best design in architecture, only a least worst collection of trade-offs.

Architects must also prioritize these architecture characteristics toward trying to find the simplest required sets. A useful exercise once the team has made a first pass at identifying the architecture characteristics is to try to determine the least important one—if you must eliminate one, which would it be? Generally, architects are more likely to cull the explicit architecture characteristics, as many of the implicit ones support general success. The way we define what's critical or important to success assists architects in determining if the application truly requires each architecture characteristic. By attempting to determine the least applicable one, architects can help determine critical necessity. In the case of Silicon Sandwiches, which architecture characteristic that we have identified is least important? Again, no absolute correct answer exists. However, in this case, the solution could lose either customizability or performance. We could eliminate customizability as an architecture characteristic and plan to implement that behavior as part of application design. Of the operational architecture characteristics, performance is likely the least critical for success. Of course, the developers don't mean to build an application that has terrible performance, but rather one that doesn't prioritize performance over other characteristics, such as scalability or availability.

Chapter 6. Measuring and Governing Architecture Characteristics

Architects must deal with the extraordinarily wide variety of architecture characteristics across all different aspects of software projects. Operational aspects like performance, elasticity, and scalability comingle with structural concerns such as modularity and deployability. This chapter focuses on concretely defining some of the more common architecture characteristics and building governance mechanisms for them.

Measuring Architecture Characteristics

Several common problems exist around the definition of architecture characteristics in organizations:

They aren't physics

> Many architecture characteristics in common usage have vague meanings. For example, how does an architect design for *agility* or *deployability*? The industry has wildly differing perspectives on common terms, sometimes driven by legitimate differing contexts, and sometimes accidental.

Wildly varying definitions

> Even within the same organization, different departments may disagree on the definition of critical features such as *performance*. Until developers, architecture, and operations can unify on a common definition, a proper conversation is difficult.

Too composite

> Many desirable architecture characteristics comprise many others at a smaller scale. For example, developers can decompose agility into characteristics such as modularity, deployability, and testability.

Objective definitions for architecture characteristics solve all three problems: by agreeing organization-wide on concrete definitions for architecture characteristics, teams create a ubiquitous language around architecture. Also, by encouraging objective definitions, teams can unpack composite characteristics to uncover measurable features they can objectively define.

Operational Measures

Many architecture characteristics have obvious direct measurements, such as performance or scalability. However, even these offer many nuanced interpretations, depending on the team's goals. For example, perhaps a team measures the average response time for certain requests, a good example of an operational architecture characteristics measure. But if teams only measure the average, what happens if some boundary condition causes 1% of requests to take 10 times longer than others? If the site has enough traffic, the outliers may not even show up. Therefore, a team may also want to measure the maximum response times to catch outliers.

THE MANY FLAVORS OF PERFORMANCE

Many of the architecture characteristics we describe have multiple, nuanced definitions. Performance is a great example. Many projects look at general performance: for example, how long request and response cycles take for a web application. However, architects and DevOps engineers have performed a tremendous amount of work on establishing performance budgets: specific budgets for specific parts of the application. For example, many organizations have researched user behavior and determined that the optimum time for first-page render (the first visible sign of progress for a webpage, in a browser or mobile device) is 500 ms—half a second; Most applications fall in the double-digit range for this metric. But, for modern sites that attempt to capture as many users as possible, this is an important metric to track, and the organizations behind them have built extremely nuanced measures.

Some of these metrics have additional implications for the design of applications. Many forward-thinking organizations place *K-weight budgets* for page downloads: a maximum number of bytes' worth of libraries and frameworks allowed on a particular page. Their rationale behind this structure derives from physics constraints: only so many bytes can travel over a network at a time, especially for mobile devices in high-latency areas.

High-level teams don't just establish hard performance numbers; they base their definitions on statistical analysis. For example, say a video streaming service wants to monitor scalability. Rather than set an arbitrary number as the goal, engineers measure the scale over time and build statistical models, then raise alarms if the real-time metrics fall outside the prediction models. A failure can mean two things: the model is incorrect (which teams like to know) or something is amiss (which teams also like to know).

The kinds of characteristics that teams can now measure are evolving rapidly, in conjunction with tools and nuanced understanding. For example, many teams recently focused on performance budgets for metrics such as

first contentful paint and *first CPU idle*, both of which speak volumes about performance issues for users of webpages on mobile devices. As devices, targets, capabilities, and myriad other things change, teams will find new things and ways to measure.

Structural Measures

Some objective measures are not so obvious as performance. What about internal structural characteristics, such as well-defined modularity? Unfortunately, comprehensive metrics for internal code quality don’t yet exist. However, some metrics and common tools do allow architects to address some critical aspects of code structure, albeit along narrow dimensions.

An obvious measurable aspect of code is complexity, defined by the *cyclomatic complexity* metric.

CYCLOMATIC COMPLEXITY

Cyclomatic Complexity (CC) is a code-level metric designed to provide an object measure for the complexity of code, at the function/method, class, or application level, developed by Thomas McCabe, Sr., in 1976.

It is computed by applying graph theory to code, specifically decision points, which cause different execution paths. For example, if a function has no decision statements (such as `if` statements), then `CC = 1`. If the function had a single conditional, then `CC = 2` because two possible execution paths exist.

The formula for calculating the CC for a single function or method is $CC = E - N + 2$, where `N` represents *nodes* (lines of code), and `E` represents *edges* (possible decisions). Consider the C-like code shown in Example 6-1.

Example 6-1. Sample code for cyclomatic complexity evaluation

```
public void decision(int c1, int c2) {
    if (c1 < 100)
        return 0;
    else if (c1 + C2 > 500)
       return 1;
    else
      return -1;
}
```

The cyclomatic complexity for Example 6-1 is 3 (=5 – 4 + 2); the graph appears in Figure 6-1.

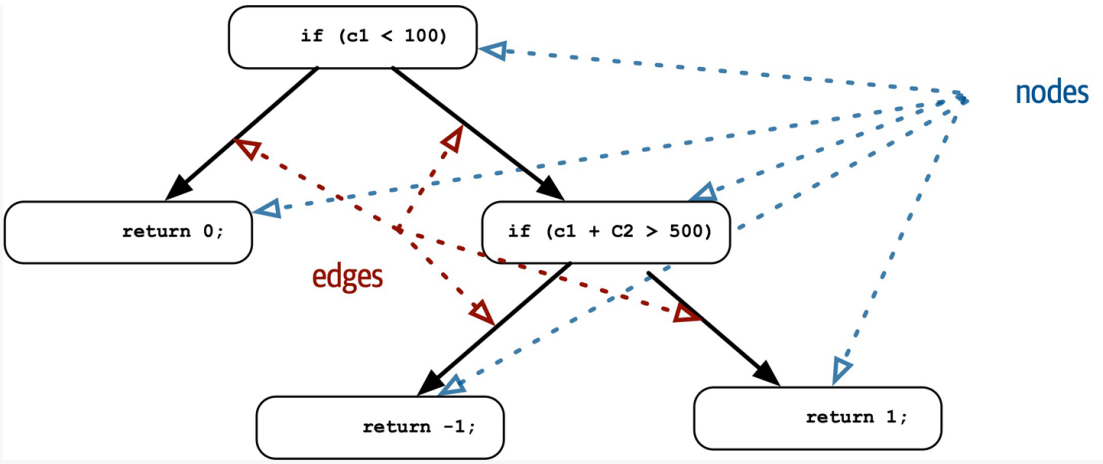

Figure 6-1. Cyclomatic Complexity for the decision function

The number 2 appearing in the cyclomatic complexity formula represents a simplification for a single function/method. For fan-out calls to other methods (known as *connected components* in graph theory), the more general formula is $CC = E - N + 2P$, where P represents the number of connected components.

Architects and developers universally agree that overly complex code represents a code smell; it harms virtually every one of the desirable characteristics of code bases: modularity, testability, deployability, and so on. Yet if teams don’t keep an eye on gradually growing complexity, that complexity will dominate the code base.

WHAT'S A GOOD VALUE FOR CYCLOMATIC COMPLEXITY?

A common question the authors receive when talking about this subject is: what's a good threshold value for CC? Of course, like all answers in software architecture: it depends! It depends on the complexity of the problem domain. For example, if you have an algorithmically complex problem, the solution will yield complex functions. Some of the key aspects of CC for architects to monitor: are functions complex because of the problem domain or because of poor coding? Alternatively, is the code partitioned poorly? In other words, could a large method be broken down into smaller, logical chunks, distributing the work (and complexity) into more well-factored methods?

In general, the industry thresholds for CC suggest that a value under 10 is acceptable, barring other considerations such as complex domains. We consider that threshold very high and would prefer code to fall under five, indicating cohesive, well-factored code. A metrics tool in the Java world, Crap4J, attempts to determine how poor (crappy) your code is by evaluating a combination of CC and code coverage; if CC grows to over 50, no amount of code coverage rescues that code from crappiness. The most terrifying professional artifact Neal ever encountered was a single C function that served as the heart of a commercial software package whose CC was over 800! It was a single function with over 4,000 lines of code, including the liberal use of `GOTO` statements (to escape impossibly deeply nested loops).

Engineering practices like test-driven development have the accidental (but positive) side effect of generating smaller, less complex methods on average for a given problem domain. When practicing TDD, developers try to write a simple test, then write the smallest amount of code to pass the test. This focus on discrete behavior and good test boundaries encourages well-factored, highly cohesive methods that exhibit low CC.

Process Measures

Some architecture characteristics intersect with software development processes. For example, agility often appears as a desirable feature. However, it is a composite architecture characteristic that architects may decompose into features such as testability, and deployability.

Testability is measurable through code coverage tools for virtually all platforms that assess the completeness of testing. Like all software checks, it cannot replace thinking and intent. For example, a code base can have 100% code coverage yet poor assertions that don't actually provide confidence in code correctness. However, testability is clearly an objectively measurable characteristic. Similarly, teams can measure deployability via a variety of metrics: percentage of successful to failed deployments, how long deployments take, issues/bugs raised by deployments, and a host of others. Each team bears the responsibility to arrive at a good set of measurements that capture useful data for their organization, both in quality and quantity. Many of these measures come down to team priorities and goals.

Agility and its related parts clearly relate to the software development process. However, that process may impact the structure of the architecture. For example, if ease of deployment and testability are high priorities, then an architect would place more emphasis on good modularity and isolation at the architecture level, an example of an architecture characteristic driving a structural decision. Virtually anything within the scope of a software project may rise to the level of an architecture characteristic if it manages to meet our three criteria, forcing an architect to make design decisions to account for it.

Governance and Fitness Functions

Once architects have established architecture characteristics and prioritized them, how can they make sure that developers will respect those priorities? Modularity is a great example of an aspect of architecture that is important

but not urgent; on many software projects, urgency dominates, yet architects still need a mechanism for governance.

Governing Architecture Characteristics

Governance, derived from the Greek word *kubernan* (to steer) is an important responsibility of the architect role. As the name implies, the scope of architecture governance covers any aspect of the software development process that architects (including roles like enterprise architects) want to exert an influence upon. For example, ensuring software quality within an organization falls under the heading of architectural governance because it falls within the scope of architecture, and negligence can lead to disastrous quality problems.

Fortunately, increasingly sophisticated solutions exist to relieve this problem from architects, a good example of the incremental growth in capabilities within the software development ecosystem. The drive toward automation on software projects spawned by Extreme Programming created continuous integration, which led to further automation into operations, which we now call DevOps, continuing through to architectural governance. The book *Building Evolutionary Architectures* (O'Reilly) describes a family of techniques, called fitness functions, used to automate many aspects of architecture governance.

Fitness Functions

The word "evolutionary" in *Building Evolutionary Architectures* comes more from evolutionary computing than biology. One of the authors, Dr. Rebecca Parsons, spent some time in the evolutionary computing space, including tools like genetic algorithms. A genetic algorithm executes and produces an answer and then undergoes mutation by well-known techniques defined within the evolutionary computing world. If a developer tries to design a genetic algorithm to produce some beneficial outcome, they often want to guide the algorithm, providing an objective measure indicating the quality of the outcome. That guidance mechanism is called a *fitness*

function: an object function used to assess how close the output comes to achieving the aim. For example, suppose a developer needed to solve the traveling salesperson problem, a famous problem used as a basis for machine learning. Given a salesperson and a list of cities they must visit, with distances between them, what is the optimum route? If a developer designs a genetic algorithm to solve this problem, one fitness function might evaluate the length of the route, as the shortest possible one represents highest success. Another fitness function might be to evaluate the overall cost associated with the route and attempt to keep cost at a minimum. Yet another might be to evaluate the time the traveling salesperson is away and optimize to shorten the total travel time.

Practices in evolutionary architecture borrow this concept to create an *architecture fitness function*:

Architecture fitness function

> Any mechanism that provides an objective integrity assessment of some architecture characteristic or combination of architecture characteristics

Fitness functions are not some new framework for architects to download, but rather a new perspective on many existing tools. Notice in the definition the phrase *any mechanism*—the verification techniques for architecture characteristics are as varied as the characteristics are. Fitness functions overlap many existing verification mechanisms, depending on the way they are used: as metrics, monitors, unit testing libraries, chaos engineering, and so on, illustrated in Figure 6-2.

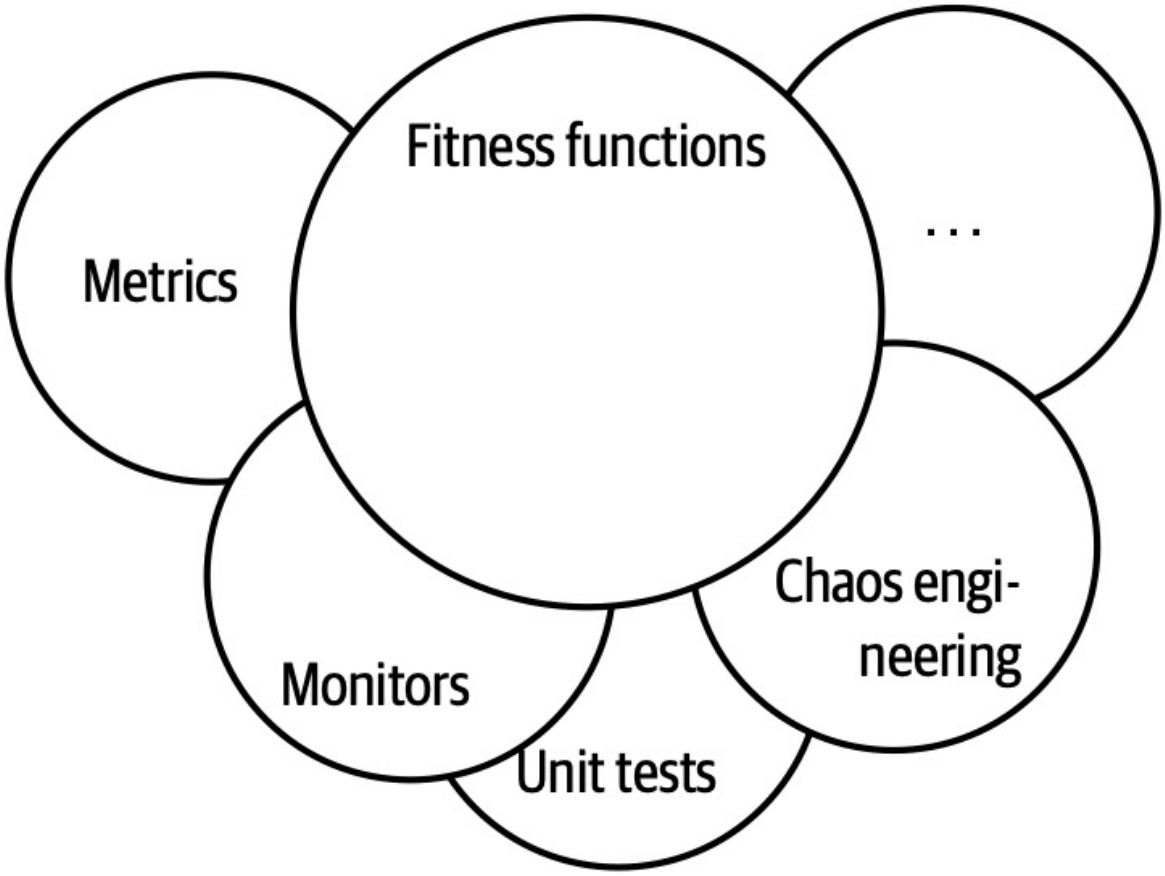

Figure 6-2. The mechanisms of fitness functions

Many different tools may be used to implement fitness functions, depending on the architecture characteristics. For example, in "Coupling" we introduced metrics to allow architects to assess modularity. Here are a couple of examples of fitness functions that test various aspects of modularity.

Cyclic dependencies

Modularity is an implicit architecture characteristic that most architects care about, because poorly maintained modularity harms the structure of a code base; thus, architects should place a high priority on maintaining good modularity. However, forces work against the architect's good intentions on many platforms. For example, when coding in any popular Java or .NET development environment, as soon as a developer references a class not already imported, the IDE helpfully presents a dialog asking the developers if they would like to auto-import the reference. This occurs so often that

most programmers develop the habit of swatting the auto-import dialog away like a reflex action. However, arbitrarily importing classes or components between one another spells disaster for modularity. For example, Figure 6-3 illustrates a particularly damaging anti-pattern that architects aspire to avoid.

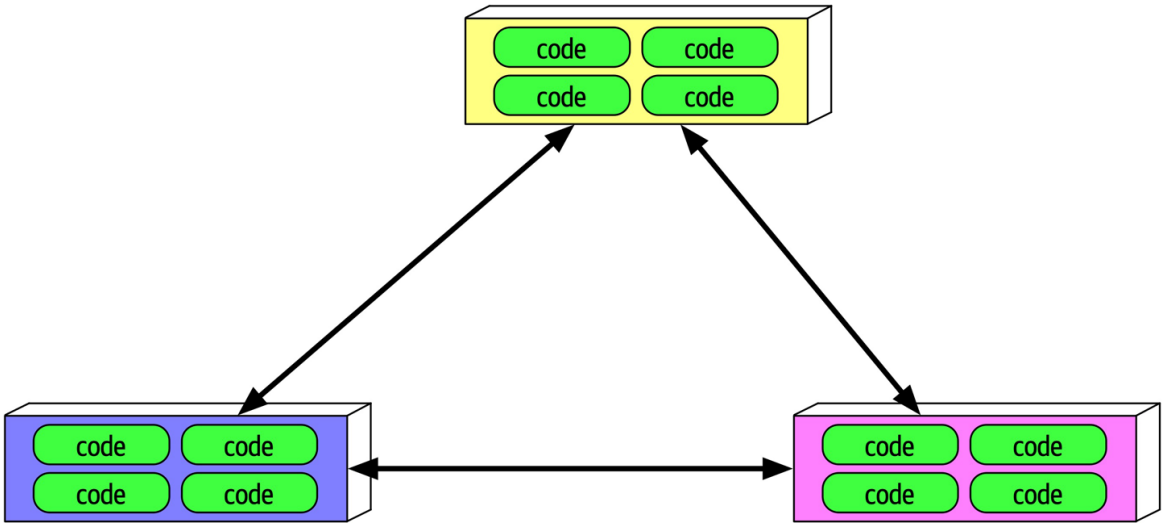

Figure 6-3. Cyclic dependencies between components

In Figure 6-3, each component references something in the others. Having a network of components such as this damages modularity because a developer cannot reuse a single component without also bringing the others along. And, of course, if the other components are coupled to other components, the architecture tends more and more toward the Big Ball of Mud anti-pattern. How can architects govern this behavior without constantly looking over the shoulders of trigger-happy developers? Code reviews help but happen too late in the development cycle to be effective. If an architect allows a development team to rampantly import across the code base for a week until the code review, serious damage has already occurred in the code base.

The solution to this problem is to write a fitness function to look after cycles, as shown in Example 6-2.

Example 6-2. Fitness function to detect component cycles

```
public class CycleTest {
    private JDepend jdepend;
```

```
    @BeforeEach
    void init() {
          jdepend = new JDepend();
          jdepend.addDirectory("/path/to/project/persistence/classes");
          jdepend.addDirectory("/path/to/project/web/classes");
          jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
          Collection packages = jdepend.analyze();
          assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}
```

In the code, an architect uses the metrics tool JDepend to check the dependencies between packages. The tool understands the structure of Java packages and fails the test if any cycles exist. An architect can wire this test into the continuous build on a project and stop worrying about the accidental introduction of cycles by trigger-happy developers. This is a great example of a fitness function guarding the important rather than urgent practices of software development: it's an important concern for architects yet has little impact on day-to-day coding.

Distance from the main sequence fitness function

In "Coupling", we introduced the more esoteric metric of distance from the main sequence, which architects can also verify using fitness functions, as shown in Example 6-3.

Example 6-3. Distance from the main sequence fitness function

```
@Test
void AllPackages() {
    double ideal = 0.0;
    double tolerance = 0.5; // project-dependent
    Collection packages = jdepend.analyze();
    Iterator iter = packages.iterator();
    while (iter.hasNext()) {
      JavaPackage p = (JavaPackage)iter.next();
      assertEquals("Distance exceeded: " + p.getName(),
            ideal, p.distance(), tolerance);
```

```
    }
}
```

In the code, the architect uses JDepend to establish a threshold for acceptable values, failing the test if a class falls outside the range.

This is both an example of an objective measure for an architecture characteristic and the importance of collaboration between developers and architects when designing and implementing fitness functions. The intent is not for a group of architects to ascend to an ivory tower and develop esoteric fitness functions that developers cannot understand.

> **TIP**
>
> Architects must ensure that developers understand the purpose of the fitness function before imposing it on them.

The sophistication of fitness function tools has increased over the last few years, including some special purpose tools. One such tool is ArchUnit, a Java testing framework inspired by and using several parts of the JUnit ecosystem. ArchUnit provides a variety of predefined governance rules codified as unit tests and allows architects to write specific tests that address modularity. Consider the layered architecture illustrated in Figure 6-4.

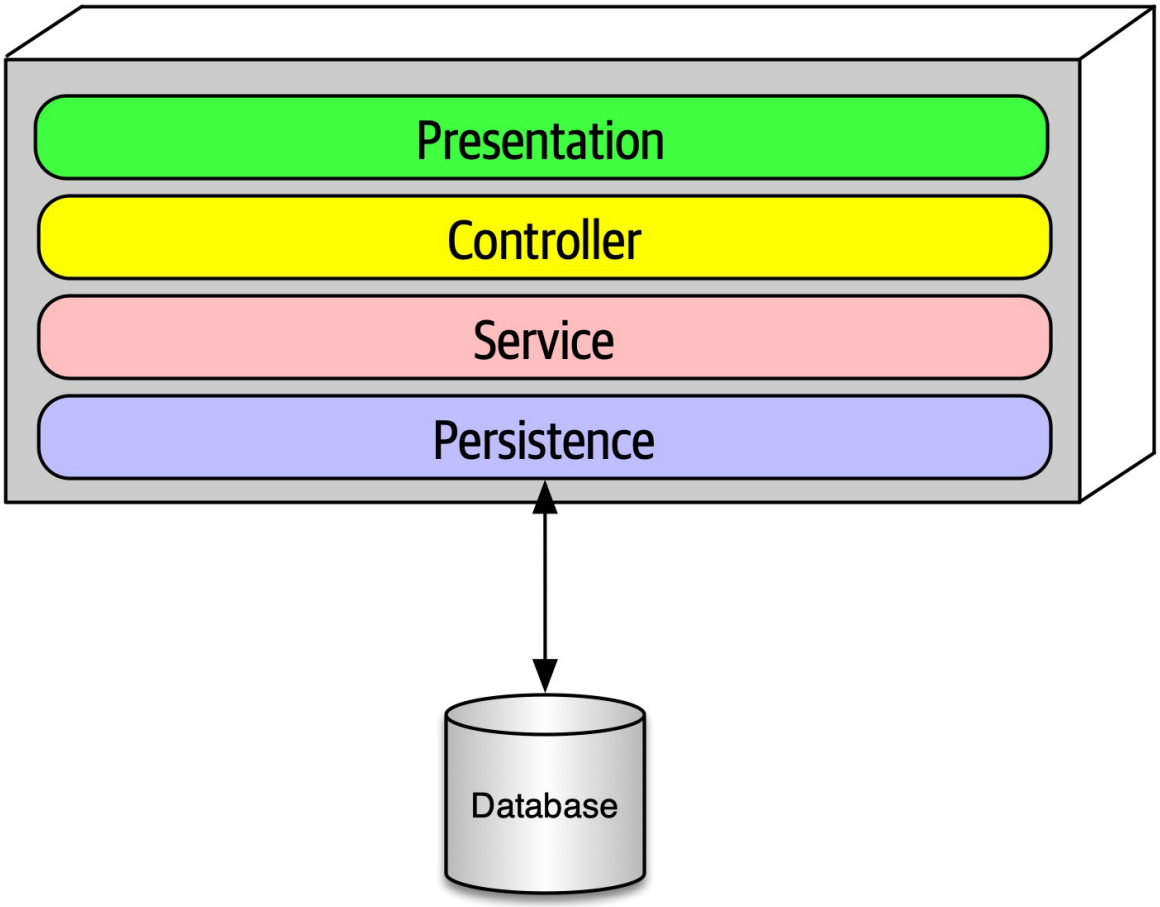

Figure 6-4. Layered architecture

When designing a layered monolith such as the one in Figure 6-4, the architect defines the layers for good reason (motivations, trade-offs, and other aspects of the layered architecture are described in Chapter 10). However, how can the architect ensure that developers will respect those layers? Some developers may not understand the importance of the patterns, while others may adopt a "better to ask forgiveness than permission" attitude because of some overriding local concern such as performance. But allowing implementers to erode the reasons for the architecture hurts the long-term health of the architecture.

ArchUnit allows architects to address this problem via a fitness function, shown in Example 6-4.

Example 6-4. ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
```

```
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In Example 6-4, the architect defines the desirable relationship between layers and writes a verification fitness function to govern it.

A similar tool in the .NET space, NetArchTest, allows similar tests for that platform; a layer verification in C# appears in Example 6-5.

Example 6-5. NetArchTest for layer dependencies

```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

Another example of fitness functions is Netflix's Chaos Monkey and the attendant Simian Army. In particular, the *Conformity*, *Security*, and *Janitor* Monkeys exemplify this approach. The Conformity Monkey allows Netflix architects to define governance rules enforced by the monkey in production. For example, if the architects decided that each service should respond usefully to all RESTful verbs, they build that check into the Conformity Monkey. Similarly, the Security Monkey checks each service for well-known security defects, like ports that shouldn't be active and configuration errors. Finally, the Janitor Monkey looks for instances that no other services route to anymore. Netflix has an evolutionary architecture, so developers routinely migrate to newer services, leaving old services running with no collaborators. Because services running on the cloud consume money, the Janitor Monkey looks for orphan services and disintegrates them out of production.

THE ORIGIN OF THE SIMIAN ARMY

When Netflix decided to move its operations to Amazon's cloud, the architects worried over the fact that they no longer had control over operations—what happens if a defect appears operationally? To solve this problem, they spawned the discipline of Chaos Engineering with the original Chaos Monkey, and eventually the Simian Army. The Chaos Monkey simulated general chaos within the production environment to see how well their system would endure it. Latency was a problem with some AWS instances, thus the Chaos Monkey would simulate high latency (which was such a problem, they eventually created a specialized monkey for it, the Latency Monkey). Tools such as the Chaos Kong, which simulates an entire Amazon data center failure, helped Netflix avoid such outages when they occured for real.

Chaos engineering offers an interesting new perspective on architecture: it's not a question of if something will eventually break, but when. Anticipating those breakages and tests to prevent them makes systems much more robust.

A few years ago, the influential book *The Checklist Manifesto* by Atul Gawande (Picador) described how professions such as airline pilots and surgeons use checklists (sometimes legally mandated). It's not because those professionals don't know their jobs or are forgetful. Rather, when professionals do a highly detailed job over and over, it becomes easy for details to slip by; a succinct checklist forms an effective reminder. This is the correct perspective on fitness functions—rather than a heavyweight governance mechanism, fitness functions provide a mechanism for architects to express important architectural principles and automatically verify them. Developers know that they shouldn't release insecure code, but that priority competes with dozens or hundreds of other priorities for busy developers. Tools like the Security Monkey specifically, and fitness functions generally, allow architects to codify important governance checks into the substrate of the architecture.

Chapter 7. Scope of Architecture Characteristics

A prevailing axiomatic assumption in the software architecture world had traditionally placed the scope of architecture characteristics at the system level. For example, when architects talk about scalability, they generally couch that discussion around the scalability of the entire system. That was a safe assumption a decade ago, when virtually all systems were monolithic. With the advent of modern engineering techniques and the architecture styles they enabled, such as microservices, the scope of architecture characteristics has narrowed considerably. This is a prime example of an axiom slowly becoming outdated as the software development ecosystem continues its relentless evolution.

During the writing of the *Building Evolutionary Architectures* book, the authors needed a technique to measure the structural evolvability of particular architecture styles. None of the existing measures offered the correct level of detail. In “Structural Measures”, we discuss a variety of code-level metrics that allow architects to analyze structural aspects of an architecture. However, all these metrics only reveal low-level details about the code, and cannot evaluate dependent components (such as databases) outside the code base that still impact many architecture characteristics, especially operational ones. For example, no matter how much an architect puts effort into designing a performant or elastic code base, if the system uses a database that doesn’t match those characteristics, the application won’t be successful.

When evaluating many operational architecture characteristics, an architect must consider dependent components outside the code base that will impact those characteristics. Thus, architects need another method to measure these kinds of dependencies. That lead the *Building Evolutionary Architectures* authors to define the term *architecture quantum*. To understand the

architecture quantum definition, we must preview one key metric here, connascence.

Coupling and Connascence

Many of the code-level coupling metrics, such as *afferent* and *efferent* coupling (described in "Structural Measures"), reveal details at a too fine-grained level for architectural analysis. In 1996, Meilir Page-Jones published a book titled *What Every Programmer Should Know About Object Oriented Design* (Dorset House) that included several new measures of coupling he named *connascence*, which is defined as follows:

Connascence

> Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system

He defined two types of connascence: *static*, discoverable via static code analysis, and *dynamic*, concerning runtime behavior. To define the architecture quantum, we needed a measure of how components are "wired" together, which corresponds to the connascence concept. For example, if two services in a microservices architecture share the same class definition of some class, like *address*, we say they are *statically* connascent with each other—changing the shared class requires changes to both services.

For dynamic connascence, we define two types: *synchronous* and *asynchronous*. Synchronous calls between two distributed services have the caller wait for the response from the callee. On the other hand, *asynchronous* calls allow fire-and-forget semantics in event-driven architectures, allowing two different services to differ in operational architecture

Architectural Quanta and Granularity

Component-level coupling isn't the only thing that binds software together. Many business concepts semantically bind parts of the system together, creating *functional cohesion*. To successfully design, analyze, and evolve software, developers must consider all the coupling points that could break.

Many science-literate developers know of the concept of quantum from physics, the minimum amount of any physical entity involved in an interaction. The word quantum derives from Latin, meaning "how great" or "how much." We have adopted this notion to define an *architecture quantum*:

Architecture quantum

> An independently deployable artifact with high functional cohesion and synchronous connascence

This definition contains several parts, dissected here:

Independently deployable

> An architecture quantum includes all the necessary components to function independently from other parts of the architecture. For example, if an application uses a database, it is part of the quantum because the system won't function without it. This requirement means that virtually all legacy systems deployed using a single database by definition form a quantum of one. However, in the microservices architecture style, each service includes its own database (part of the *bounded context* driving philosophy in microservices, described in detail in Chapter 17), creating multiple quanta within that architecture.

High functional cohesion

> *Cohesion* in component design refers to how well the contained code is unified in purpose. For example, a `Customer` component with properties and methods all pertaining to a *Customer* entity exhibits high cohesion; whereas a `Utility` component with a random collection of miscellaneous methods would not.

High functional cohesion implies that an architecture quantum does something purposeful. This distinction matters little in traditional monolithic applications with a single database. However, in microservices architectures, developers typically design each service to match a single workflow (a *bounded context*, as described in "Domain-Driven Design's Bounded Context"), thus exhibiting high functional cohesion.

Synchronous connascence

Synchronous connascence implies synchronous calls within an application context or between distributed services that form this architecture quantum. For example, if one service in a microservices architecture calls another one synchronously, each service cannot exhibit extreme differences in operational architecture characteristics. If the caller is much more scalable than the callee, timeouts and other reliability concerns will occur. Thus, synchronous calls create dynamic connascence for the length of the call—if one is waiting for the other, their operational architecture characteristics must be the same for the duration of the call.

Back in Chapter 6, we defined the relationship between traditional coupling metrics and connascence, which didn't include our new *communication connascence* measure. We update this diagram in Figure 7-1.

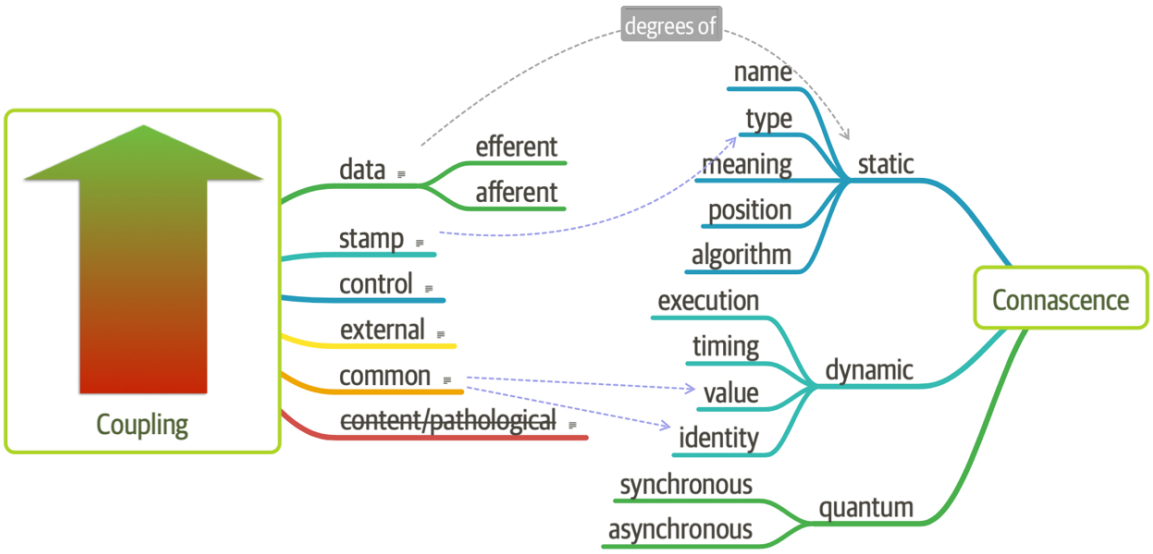

Figure 7-1. Adding quantum connascence to the unified diagram

For another example, consider a microservices architecture with a `Payment` service and an `Auction` service. When an auction ends, the `Auction` service sends payment information to the `Payment` service. However, let's say that the payment service can only handle a payment every 500 ms—what happens when a large number of auctions end at once? A poorly designed architecture would allow the first call to go through and allow the others to time out. Alternatively, an architect might design an asynchronous communication link between `Payment` and `Auction`, allowing the message queue to temporarily buffer differences. In this case, asynchronous connascence creates a more flexible architecture. We cover this subject in great detail in Chapter 14.

DOMAIN-DRIVEN DESIGN'S BOUNDED CONTEXT

Eric Evans' book *Domain-Driven Design* (Addison-Wesley Professional) has deeply influenced modern architectural thinking. *Domain-driven design* (DDD) is a modeling technique that allows for organized decomposition of complex problem domains. DDD defines the *bounded context*, where everything related to the domain is visible internally but opaque to other bounded contexts. Before DDD, developers sought holistic reuse across common entities within the organization. Yet creating common shared artifacts causes a host of problems, such as coupling, more difficult coordination, and increased complexity. The *bounded context* concept recognizes that each entity works best within a localized context. Thus, instead of creating a unified `Customer` class across the entire organization, each problem domain can create its own and reconcile differences at integration points.

The architecture quantum concept provides the new scope for architecture characteristics. In modern systems, architects define architecture characteristics at the quantum level rather than system level. By looking at a narrower scope for important operational concerns, architects may identify architectural challenges early, leading to hybrid architectures. To illustrate scoping provided by the architecture quantum measure, consider another architecture kata, *Going, Going, Gone*.

Case Study: Going, Going, Gone

In Chapter 5, we introduced the concept of an architecture kata. Consider this one, concerning an online auction company. Here is the description of the architecture kata:

Description

> An auction company wants to take its auctions online to a nationwide scale. Customers choose the auction to participate in, wait until the

auction begins, then bid as if they are there in the room with the auctioneer.

Users

Scale up to hundreds of participants per auction, potentially up to thousands of participants, and as many simultaneous auctions as possible.

Requirements

- Auctions must be as real-time as possible.
- Bidders register with a credit card; the system automatically charges the card if the bidder wins.
- Participants must be tracked via a reputation index.
- Bidders can see a live video stream of the auction and all bids as they occur.
- Both online and live bids must be received in the order in which they are placed.

Additional context

- Auction company is expanding aggressively by merging with smaller competitors.
- Budget is not constrained. This is a strategic direction.
- Company just exited a lawsuit where it settled a suit alleging fraud.

Just as in "Case Study: Silicon Sandwiches", an architect must consider each of these requirements to ascertain architecture characteristics:

1. "Nationwide scale," "scale up to hundreds of participants per auction, potentially up to thousands of participants, and as many

simultaneous auctions as possible," "auctions must be as real-time as possible."

Each of these requirements implies both scalability to support the sheer number of users and elasticity to support the bursty nature of auctions. While the requirements explicitly call out scalability, elasticity represents an implicit characteristics based on the problem domain. When considering auctions, do users all politely spread themselves out during the course of bidding, or do they become more frantic near the end? Domain knowledge is crucial for architects to pick up implicit architecture characteristics. Given the real-time nature of auctions, an architect will certainly consider performance a key architecture characteristic.

2. "Bidders register with a credit card; the system automatically charges the card if the bidder wins," "company just exited a lawsuit where it settled a suit alleging fraud."

   Both these requirements clearly point to security as an architecture characteristic. As covered in Chapter 5, security is an implicit architecture characteristic in virtually every application. Thus, architects rely on the second part of the definition of architecture characteristics, that they influence some structural aspect of the design. Should an architect design something special to accommodate security, or will general design and coding hygiene suffice? Architects have developed techniques for handling credit cards safely via design without necessarily building special structure. For example, as long as developers make sure not to store credit card numbers in plain text, to encrypt while in transit, and so on, then the architect shouldn't have to build special considerations for security.

   However, the second phrase should make an architect pause and ask for further clarification. Clearly, some aspect of security (fraud) was a problem in the past, thus the architect should ask for further input no matter what level of security they design.

3. “Participants must be tracked via a reputation index.”

   This requirement suggests some fanciful names such as “anti-trollability,” but the *track* part of the requirement might suggest some architecture characteristics such as auditability and loggability. The deciding factor again goes back to the defining characteristic—is this outside the scope of the problem domain? Architects must remember that the analysis to yield architecture characteristics represents only a small part of the overall effort to design and implement an application—a lot of design work happens past this phase! During this part of architecture definition, architects look for requirements with structural impact not already covered by the domain.

   Here’s a useful litmus test architects use to make the determination between domain versus architecture characteristics is: does it require domain knowledge to implement, or is it an abstract architecture characteristic? In the Going, Going, Gone kata, an architect upon encountering the phrase “reputation index” would seek out a business analyst or other subject matter expert to explain what they had in mind. In other words, the phrase “reputation index” isn’t a standard definition like more common architecture characteristics. As a counter example, when architects discuss *elasticity*, the ability to handle bursts of users, they can talk about the architecture characteristic purely in the abstract—it doesn’t matter what kind of application they consider: banking, catalog site, streaming video, and so on. Architects must determine whether a requirement isn’t already encompassed by the domain *and* requires particular structure, which elevates a consideration to architecture characteristic.

4. “Auction company is expanding aggressively by merging with smaller competitors.”

   While this requirement may not have an immediate impact on application design, it might become the determining factor in a

trade-off between several options. For example, architects must often choose details such as communication protocols for integration architecture: if integration with newly merged companies isn't a concern, it frees the architect to choose something highly specific to the problem. On the other hand, an architect may choose something that's less than perfect to accommodate some additional trade-off, such as interoperability. Subtle implicit architecture characteristics such as this pervade architecture, illustrating why doing the job well presents challenges.

5. "Budget is not constrained. This is a strategic direction."

   Some architecture katas impose budget restrictions on the solution to represent a common real-world trade-off. However, in the Going, Going, Gone kata, it does not. This allows the architect to choose more elaborate and/or special-purpose architectures, which will be beneficial given the next requirements.

6. "Bidders can see a live video stream of the auction and all bids as they occur," "both online and live bids must be received in the order in which they are placed."

   This requirement presents an interesting architectural challenge, definitely impacting the structure of the application and exposing the futility of treating architecture characteristics as a system-wide evaluation. Consider availability—is that need uniform throughout the architecture? In other words, is the availability of the one bidder more important than availability for one of the hundreds of bidders? Obviously, the architect desires good measures for both, but one is clearly more critical: if the auctioneer cannot access the site, online bids cannot occur for anyone. Reliability commonly appears with availability; it addresses operational aspects such as uptime, as well as data integrity and other measures of how reliable an application is. For example, in an auction site, the architect must

ensure that the message ordering is reliably correct, eliminating race conditions and other problems.

This last requirement in the Going, Going, Gone kata highlights the need for a more granular scope in architecture than the system level. Using the architecture quantum measure, architects scope architecture characteristics at the quantum level. For example, in Going, Going, Gone, an architect would notice that different parts of this architecture need different characteristics: streaming bids, online bidders, and the auctioneer are three obvious choices. Architects use the architecture quantum measure as a way to think about deployment, coupling, where data should reside, and communication styles within architectures. In this kata, an architect can analyze the differing architecture characteristics per architecture quantum, leading to hybrid architecture design earlier in the process.

Thus, for Going, Going, Gone, we identified the following quanta and corresponding architecture characteristics:

Bidder feedback

Encompasses the bid stream and video stream of bids

- Availability
- Scalability
- Performance

Auctioneer

The live auctioneer

- Availability
- Reliability
- Scalability
- Elasticity
- Performance

- Security

Bidder

Online bidders and bidding

- Reliability
- Availability
- Scalability
- Elasticity

Chapter 8. Component-Based Thinking

In Chapter 3, we discussed *modules* as a collection of related code. However, architects typically think in terms of *components*, the physical manifestation of a module.

Developers physically package modules in different ways, sometimes depending on their development platform. We call physical packaging of modules *components*. Most languages support physical packaging as well: `jar` files in Java, `dll` in .NET, `gem` in Ruby, and so on. In this chapter, we discuss architectural considerations around components, ranging from scope to discovery.

Component Scope

Developers find it useful to subdivide the concept of *component* based on a wide host of factors, a few of which appear in Figure 8-1.

Components offer a language-specific mechanism to group artifacts together, often nesting them to create stratification. As shown in Figure 8-1, the simplest component wraps code at a higher level of modularity than classes (or functions, in nonobject-oriented languages). This simple wrapper is often called a *library*, which tends to run in the same memory address as the calling code and communicate via language function call mechanisms. Libraries are usually compile-time dependencies (with notable exceptions like dynamic link libraries [DLLs] that were the bane of Windows users for many years).

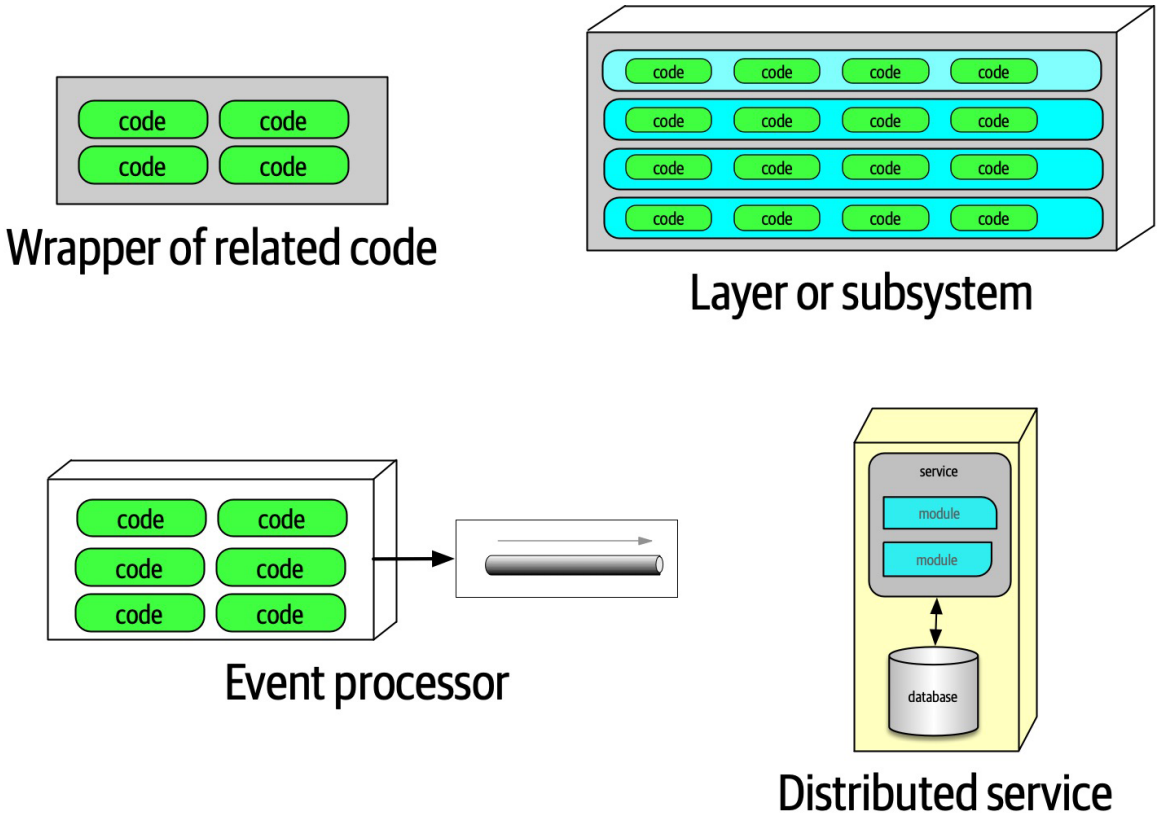

Figure 8-1. Different varieties of components

Components also appear as subsystems or layers in architecture, as the deployable unit of work for many event processors. Another type of component, a *service*, tends to run in its own address space and communicates via low-level networking protocols like TCP/IP or higher-level formats like REST or message queues, forming stand-alone, deployable units in architectures like microservices.

Nothing requires an architect to use components—it just so happens that it's often useful to have a higher level of modularity than the lowest level offered by the language. For example, in microservices architectures, simplicity is one of the architectural principles. Thus, a service may consist of enough code to warrant components or may be simple enough to just contain a small bit of code, as illustrated in Figure 8-2.

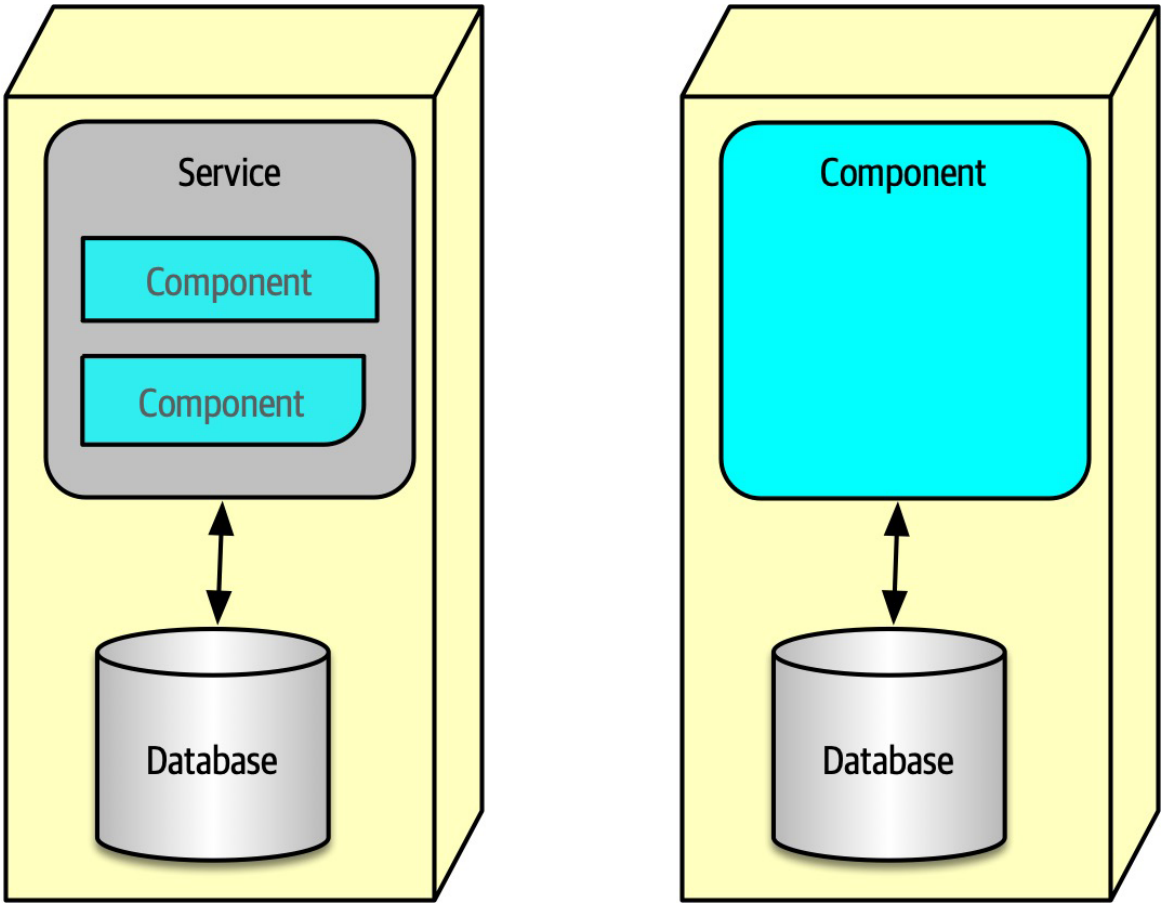

Figure 8-2. A microservice might have so little code that components aren't necessary

Components form the fundamental modular building block in architecture, making them a critical consideration for architects. In fact, one of the primary decisions an architect must make concerns the top-level partitioning of components in the architecture.

Architect Role

Typically, the architect defines, refines, manages, and governs components within an architecture. Software architects, in collaboration with business analysts, subject matter experts, developers, QA engineers, operations, and enterprise architects, create the initial design for software, incorporating the architecture characteristics discussed in Chapter 4 and the requirements for the software system.

Virtually all the details we cover in this book exist independently from whatever software development process teams use: architecture is independent from the development process. The primary exception to this rule entails the engineering practices pioneered in the various flavors of Agile software development, particularly in the areas of deployment and automating governance. However, in general, software architecture exists separate from the process. Thus, architects ultimately don't care where requirements originate: a formal Joint Application Design (JAD) process, lengthy waterfall-style analysis and design, Agile story cards…or any hybrid variation of those.

Generally the component is the lowest level of the software system an architect interacts directly with, with the exception of many of the code quality metrics discussed in Chapter 6 that affect code bases holistically. Components consist of classes or functions (depending on the implementation platform), whose design falls under the responsibility of tech leads or developers. It's not that architects shouldn't involve themselves in class design (particularly when discovering or applying design patterns), but they should avoid micromanaging each decision from top to bottom in the system. If architects never allow other roles to make decisions of consequence, the organization will struggle with empowering the next generation of architects.

An architect must identify components as one of the first tasks on a new project. But before an architect can identify components, they must know how to partition the architecture.

Architecture Partitioning

The First Law of Software Architecture states that everything in software is a trade-off, including how architects create components in an architecture. Because components represent a general containership mechanism, an architect can build any type of partitioning they want. Several common styles exist, with different sets of trade-offs. We discuss architecture styles in depth in Part II. Here we discuss an important aspect of styles, the *top-level partitioning* in an architecture.

Consider the two types of architecture styles shown in Figure 8-3.

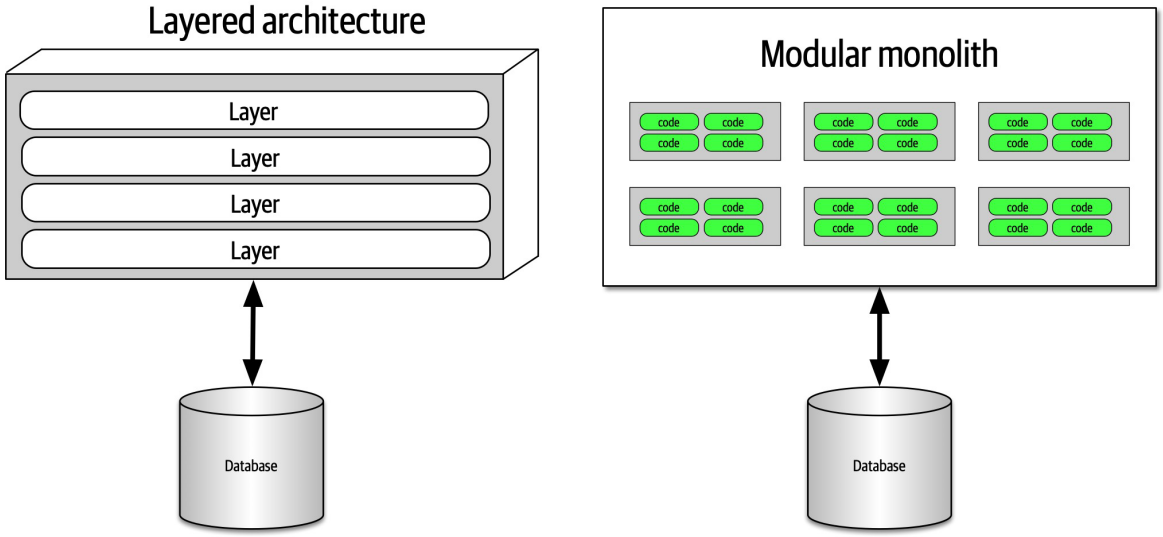

Figure 8-3. Two types of top-level architecture partitioning: layered and modular

In Figure 8-3, one type of architecture familiar to many is the *layered monolith* (discussed in detail in Chapter 10). The other is an architecture style popularized by Simon Brown called a *modular monolith*, a single deployment unit associated with a database and partitioned around domains rather than technical capabilities. These two styles represent different ways to *top-level partition* the architecture. Note that in each variation, each of the top-level components (layers or components) likely has other components embedded within. The top-level partitioning is of particular interest to architects because it defines the fundamental architecture style and way of partitioning code.

Organizing architecture based on technical capabilities like the layered architecture represents *technical top-level partitioning*. A common version of this appears in Figure 8-4.

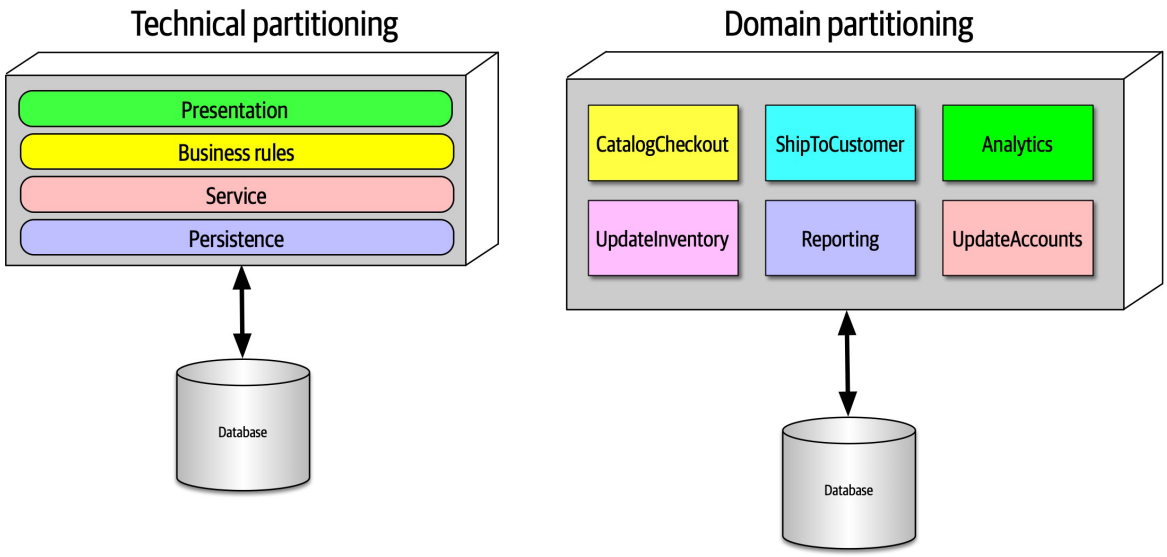

Figure 8-4. Two types of top-level partitioning in architecture

In Figure 8-4, the architect has partitioned the functionality of the system into *technical* capabilities: presentation, business rules, services, persistence, and so on. This way of organizing a code base certainly makes sense. All the persistence code resides in one layer in the architecture, making it easy for developers to find persistence-related code. Even though the basic concept of layered architecture predates it by decades, the Model-View-Controller design pattern matches with this architectural pattern, making it easy for developers to understand. Thus, it is often the default architecture in many organizations.

An interesting side effect of the predominance of the layered architecture relates to how companies seat different project roles. When using a layered architecture, it makes some sense to have all the backend developers sit together in one department, the DBAs in another, the presentation team in another, and so on. Because of *Conway's law*, this makes some sense in those organizations.

CONWAY'S LAW

Back in the late 1960s, Melvin Conway made an observation that has become known as *Conway's law*:

> *Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.*

Paraphrased, this law suggests that when a group of people designs some technical artifact, the communication structures between the people end up replicated in the design. People at all levels of organizations see this law in action, and they sometimes make decisions based on it. For example, it is common for organizations to partition workers based on technical capabilities, which makes sense from a pure organizational sense but hampers collaboration because of artificial separation of common concerns.

A related observation coined by Jonny Leroy of ThoughtWorks is the Inverse Conway Maneuver, which suggests evolving team and organizational structure together to promote the desired architecture.

The other architectural variation in Figure 8-4 represents *domain partitioning*, inspired by the Eric Evan book *Domain-Driven Design*, which is a modeling technique for decomposing complex software systems. In DDD, the architect identifies domains or workflows independent and decoupled from each other. The microservices architecture style (discussed in Chapter 17) is based on this philosophy. In a modular monolith, the architect partitions the architecture around domains or workflows rather than technical capabilities. As components often nest within one another, each of the components in Figure 8-4 in the domain partitioning (for example, *CatalogCheckout*) may use a persistence library and have a separate layer for business rules, but the top-level partitioning revolves around domains.

One of the fundamental distinctions between different architecture patterns is what type of top-level partitioning each supports, which we cover for each individual pattern. It also has a huge impact on how an architect decides how to initially identify components—does the architect want to partition things technically or by domain?

Architects using technical partitioning organize the components of the system by technical capabilities: presentation, business rules, persistence, and so on. Thus, one of the organizing principles of this architecture is *separation of technical concerns*. This in turn creates useful levels of decoupling: if the service layer is only connected to the persistence layer below and business rules layer above, then changes in persistence will only potentially affect those layers. This style of partitioning provides a decoupling technique, reducing rippling side effects on dependent components. We cover more details of this architecture style in the layered architecture pattern in Chapter 10. It is certainly logical to organize systems using technical partitioning, but, like all things in software architecture, this offers some trade-offs.

The separation enforced by technical partitioning enables developers to find certain categories of the code base quickly, as it is organized by capabilities. However, most realistic software systems require workflows that cut across technical capabilities. Consider the common business workflow of *CatalogCheckout*. The code to handle *CatalogCheckout* in the technically layered architecture appears in all the layers, as shown in Figure 8-5.

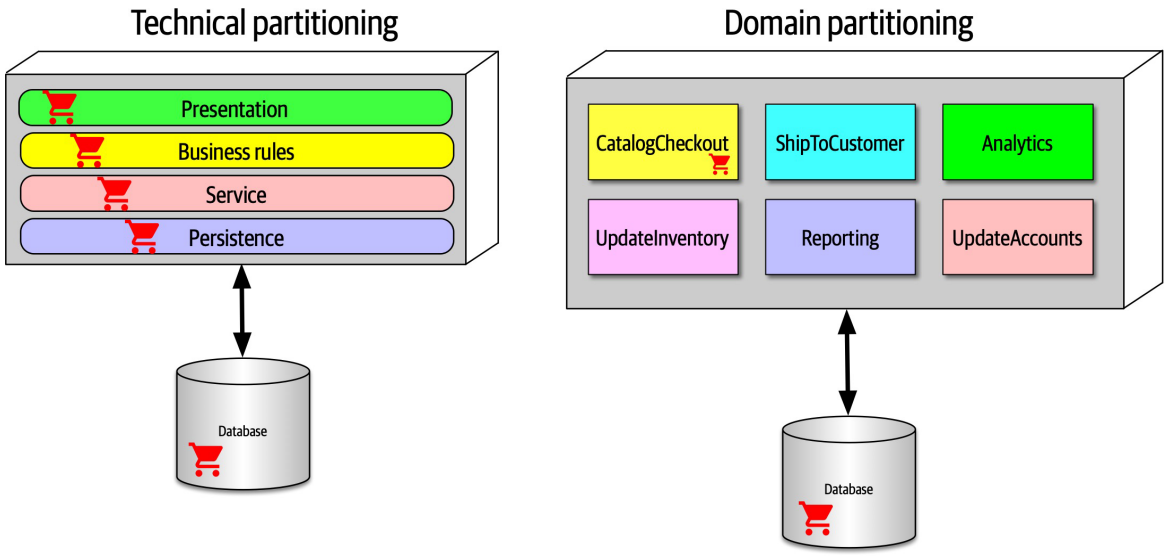

Figure 8-5. Where domains/workflows appear in technical- and domain-partitioned architectures

In Figure 8-5, in the technically partitioned architecture, *CatalogCheckout* appears in all the layers; the domain is smeared across the technical layers. Contrast this with domain partitioning, which uses a top-level partitioning that organizes components by domain rather than technical capabilities. In Figure 8-5, architects designing the domain-partitioned architecture build top-level components around workflows and/or domains. Each component in the domain partitioning may have subcomponents, including layers, but the top-level partitioning focuses on domains, which better reflects the kinds of changes that most often occur on projects.

Neither of these styles is more correct than the other—refer to the First Law of Software Architecture. That said, we have observed a decided industry trend over the last few years toward domain partitioning for the monolithic and distributed (for example, microservices) architectures. However, it is one of the first decisions an architect must make.

Case Study: Silicon Sandwiches: Partitioning

Consider the case of one of our example katas, "Case Study: Silicon Sandwiches". When deriving components, one of the fundamental decisions facing an architect is the top-level partitioning. Consider the first of two

different possibilities for Silicon Sandwiches, a domain partitioning, illustrated in Figure 8-6.

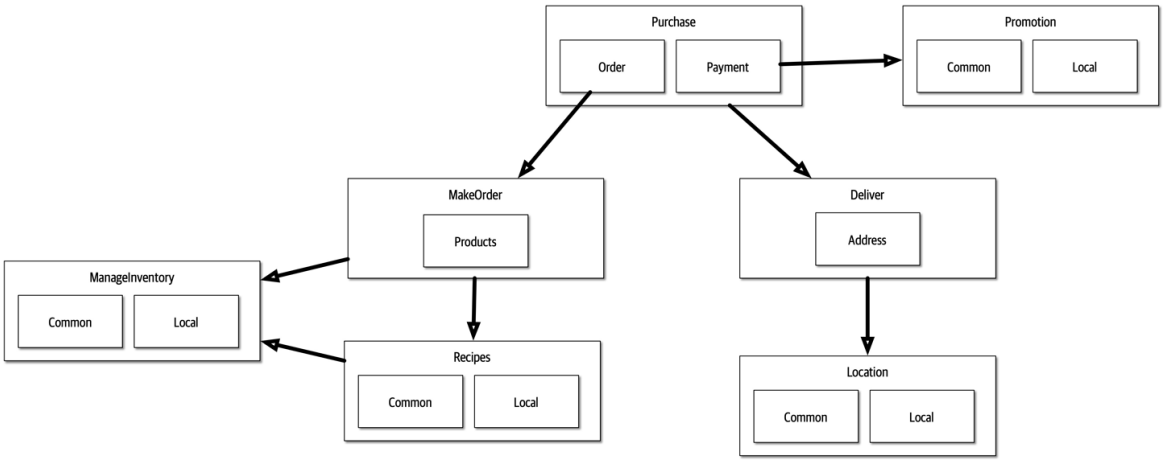

Figure 8-6. A domain-partitioned design for Silicon Sandwiches

In Figure 8-6, the architect has designed around domains (workflows), creating discrete components for `Purchase`, `Promotion`, `MakeOrder`, `ManageInventory`, `Recipes`, `Delivery`, and `Location`. Within many of these components resides a subcomponent to handle the various types of customization required, covering both common and local variations.

An alternative design isolates the `common` and `local` parts into their own partition, illustrated in Figure 8-7. `Common` and `Local` represent top-level components, with `Purchase` and `Delivery` remaining to handle the workflow.

Which is better? It depends! Each partitioning offers different advantages and drawbacks.

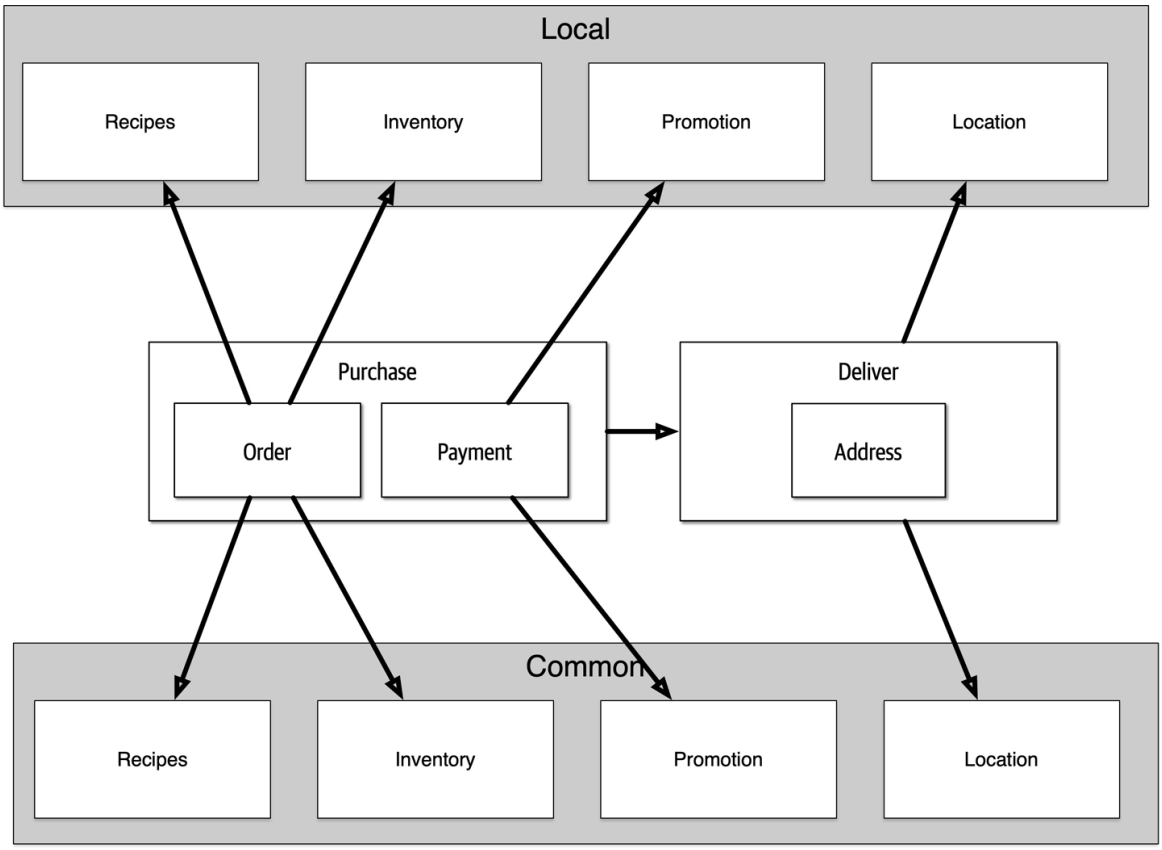

Figure 8-7. A technically partitioned design for Silicon Sandwiches

Domain partitioning

Domain-partitioned architectures separate top-level components by workflows and/or domains.

Advantages

- Modeled more closely toward how the business functions rather than an implementation detail
- Easier to utilize the Inverse Conway Maneuver to build cross-functional teams around domains
- Aligns more closely to the modular monolith and microservices architecture styles
- Message flow matches the problem domain
- Easy to migrate data and components to distributed architecture

Disadvantage

- Customization code appears in multiple places

Technical partitioning

Technically partitioned architectures separate top-level components based on technical capabilities rather than discrete workflows. This may manifest as layers inspired by Model-View-Controller separation or some other ad hoc technical partitioning. Figure 8-7 separates components based on customization.

Advantages

- Clearly separates customization code.
- Aligns more closely to the layered architecture pattern.

Disadvantages

- Higher degree of global coupling. Changes to either the `Common` or `Local` component will likely affect all the other components.
- Developers may have to duplication domain concepts in both `common` and `local` layers.
- Typically higher coupling at the data level. In a system like this, the application and data architects would likely collaborate to create a single database, including customization and domains. That in turn creates difficulties in untangling the data relationships if the architects later want to migrate this architecture to a distributed system.

Many other factors contribute to an architect's decision on what architecture style to base their design upon, covered in Part II.

Developer Role

Developers typically take components, jointly designed with the architect role, and further subdivide them into classes, functions, or subcomponents. In general, class and function design is the shared responsibility of architects, tech leads, and developers, with the lion's share going to developer roles.

Developers should never take components designed by architects as the last word; all software design benefits from iteration. Rather, that initial design should be viewed as a first draft, where implementation will reveal more details and refinements.

Component Identification Flow

Component identification works best as an iterative process, producing candidates and refinements through feedback, illustrated in Figure 8-8.

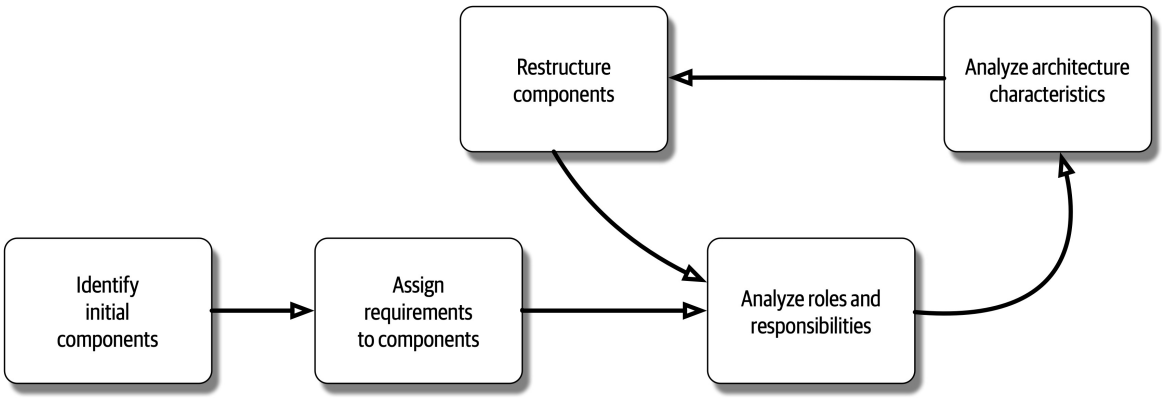

Figure 8-8. Component identification cycle

This cycle describes a generic architecture exposition cycle. Certain specialized domains may insert other steps in this process or change it altogether. For example, in some domains, some code must undergo security or auditing steps in this process. Descriptions of each step in Figure 8-8 appear in the following sections.

Identifying Initial Components

Before any code exists for a software project, the architect must somehow determine what top-level components to begin with, based on what type of top-level partitioning they choose. Outside that, an architect has the freedom to make up whatever components they want, then map domain functionality to them to see where behavior should reside. While this may sound arbitrary, it's hard to start with anything more concrete if an architect designs a system from scratch. The likelihood of achieving a good design from this initial set of components is disparagingly small, which is why architects must iterate on component design to improve it.

Assign Requirements to Components

Once an architect has identified initial components, the next step aligns requirements (or user stories) to those components to see how well they fit. This may entail creating new components, consolidating existing ones, or breaking components apart because they have too much responsibility. This mapping doesn't have to be exact—the architect is attempting to find a good coarse-grained substrate to allow further design and refinement by architects, tech leads, and/or developers.

Analyze Roles and Responsibilities

When assigning stories to components, the architect also looks at the roles and responsibilities elucidated during the requirements to make sure that the granularity matches. Thinking about both the roles and behaviors the application must support allows the architect to align the component and domain granularity. One of the greatest challenges for architects entails discovering the correct granularity for components, which encourages the iterative approach described here.

Analyze Architecture Characteristics

When assigning requirements to components, the architect should also look at the architecture characteristics discovered earlier in order to think about

how they might impact component division and granularity. For example, while two parts of a system might deal with user input, the part that deals with hundreds of concurrent users will need different architecture characteristics than another part that needs to support only a few. Thus, while a purely functional view of component design might yield a single component to handle user interaction, analyzing the architecture characteristics will lead to a subdivision.

Restructure Components

Feedback is critical in software design. Thus, architects must continually iterate on their component design with developers. Designing software provides all kinds of unexpected difficulties—no one can anticipate all the unknown issues that usually occur during software projects. Thus, an iterative approach to component design is key. First, it's virtually impossible to account for all the different discoveries and edge cases that will arise that encourage redesign. Secondly, as the architecture and developers delve more deeply into building the application, they gain a more nuanced understanding of where behavior and roles should lie.

Component Granularity

Finding the proper granularity for components is one of an architect's most difficult tasks. Too fine-grained a component design leads to too much communication between components to achieve results. Too coarse-grained components encourage high internal coupling, which leads to difficulties in deployability and testability, as well as modularity-related negative side effects.

Component Design

No accepted "correct" way exists to design components. Rather, a wide variety of techniques exist, all with various trade-offs. In all processes, an architect takes requirements and tries to determine what coarse-grained

building blocks will make up the application. Lots of different techniques exist, all with varying trade-offs and coupled to the software development process used by the team and organization. Here, we talk about a few general ways to discover components and traps to avoid.

Discovering Components

Architects, often in collaboration with other roles such as developers, business analysts, and subject matter experts, create an initial component design based on general knowledge of the system and how they choose to decompose it, based on technical or domain partitioning. The team goal is an initial design that partitions the problem space into coarse chunks that take into account differing architecture characteristics.

Entity trap

While there is no one true way to ascertain components, a common anti-pattern lurks: the *entity trap*. Say that an architect is working on designing components for our kata Going, Going, Gone and ends up with a design resembling Figure 8-9.

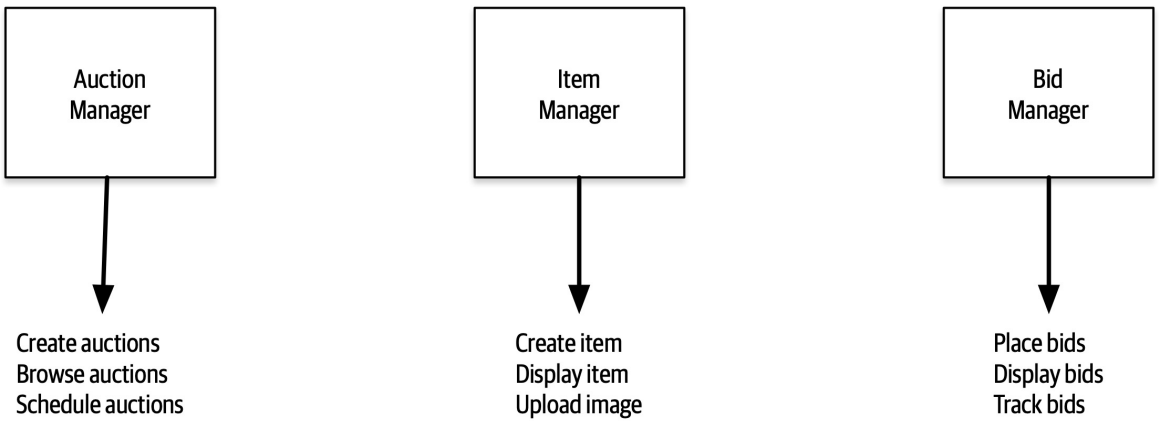

Figure 8-9. Building an architecture as an object-relational mapping

In Figure 8-9, the architect has basically taken each entity identified in the requirements and made a `Manager` component based on that entity. This isn't an architecture; it's a component-relational mapping of a framework to a database. In other words, if a system only needs simple database CRUD

operations (create, read, update, delete), then the architect can download a framework to create user interfaces directly from the database.

NAKED OBJECTS AND SIMILAR FRAMEWORKS

More than a decade ago, a family of frameworks appeared that makes building simple CRUD applications trivial, exemplified by Naked Objects (which has since split into two projects, a .NET version still called NakedObjects, and a Java version that moved to the Apache open source foundation under the name Isis). The premise behind these frameworks offers to build a user interface frontend on database entities. For example, in Naked Objects, the developer points the framework to database tables, and the framework builds a user interface based on the tables and their defined relationships.

Several other popular frameworks exist that basically provide a default user interface based on database table structure: the scaffolding feature of the Ruby on Rails framework provides the same kind of default mappings from website to database (with many options to extend and add sophistication to the resulting application).

If an architect's needs require merely a simple mapping from a database to a user interface, full-blown architecture isn't necessary; one of these frameworks will suffice.

The entity trap anti-pattern arises when an architect incorrectly identifies the database relationships as workflows in the application, a correspondence that rarely manifests in the real world. Rather, this anti-pattern generally indicates lack of thought about the actual workflows of the application. Components created with the entity trap also tend to be too coarse-grained, offering no guidance whatsoever to the development team in terms of the packaging and overall structuring of the source code.

Actor/Actions approach

The *actor/actions* approach is a popular way that architects use to map requirements to components. In this approach, originally defined by the Rational Unified Process, architects identify actors who perform activities with the application and the actions those actors may perform. It provides a technique for discovering the typical users of the system and what kinds of things they might do with the system.

The actor/actions approach became popular in conjunction with particular software development processes, especially more formal processes that favor a significant portion of upfront design. It is still popular and works well when the requirements feature distinct roles and the kinds of actions they can perform. This style of component decomposition works well for all types of systems, monolithic or distributed.

Event storming

Event storming as a component discovery technique comes from domain-driven design (DDD) and shares popularity with microservices, also heavily influenced by DDD. In event storming, the architect assumes the project will use messages and/or events to communicate between the various components. To that end, the team tries to determine which events occur in the system based on requirements and identified roles, and build components around those event and message handlers. This works well in distributed architectures like microservices that use events and messages, because it helps architects define the messages used in the eventual system.

Workflow approach

An alternative to event storming offers a more generic approach for architects not using DDD or messaging. The *workflow approach* models the components around workflows, much like event storming, but without the explicit constraints of building a message-based system. A workflow approach identifies the key roles, determines the kinds of workflows these roles engage in, and builds components around the identified activities.

None of these techniques is superior to the others; all offer a different set of trade-offs. If a team uses a waterfall approach or other older software development processes, they might prefer the Actor/Actions approach because it is general. When using DDD and corresponding architectures like microservices, *event storming* matches the software development process exactly.

Case Study: Going, Going, Gone: Discovering Components

If a team has no special constraints and is looking for a good general-purpose component decomposition, the Actor/Actions approach works well as a generic solution. It's the one we use in our case study for Going, Going, Gone.

In Chapter 7, we introduced the architecture kata for Going, Going, Gone (GGG) and discovered architecture characteristics for this system. This system has three obvious roles: the *bidder*, the *auctioneer*, and a frequent participant in this modeling technique, the *system*, for internal actions. The roles interact with the application, represented here by the system, which identifies when the application initiates an event rather than one of the roles. For example, in GGG, once the auction is complete, the system triggers the payment system to process payments.

We can also identify a starting set of actions for each of these roles:

`Bidder`

: View live video stream, view live bid stream, place a bid

`Auctioneer`

: Enter live bids into system, receive online bids, mark item as sold

`System`

: Start auction, make payment, track bidder activity

Given these actions, we can iteratively build a set of starter components for GGG; one such solution appears in Figure 8-10.

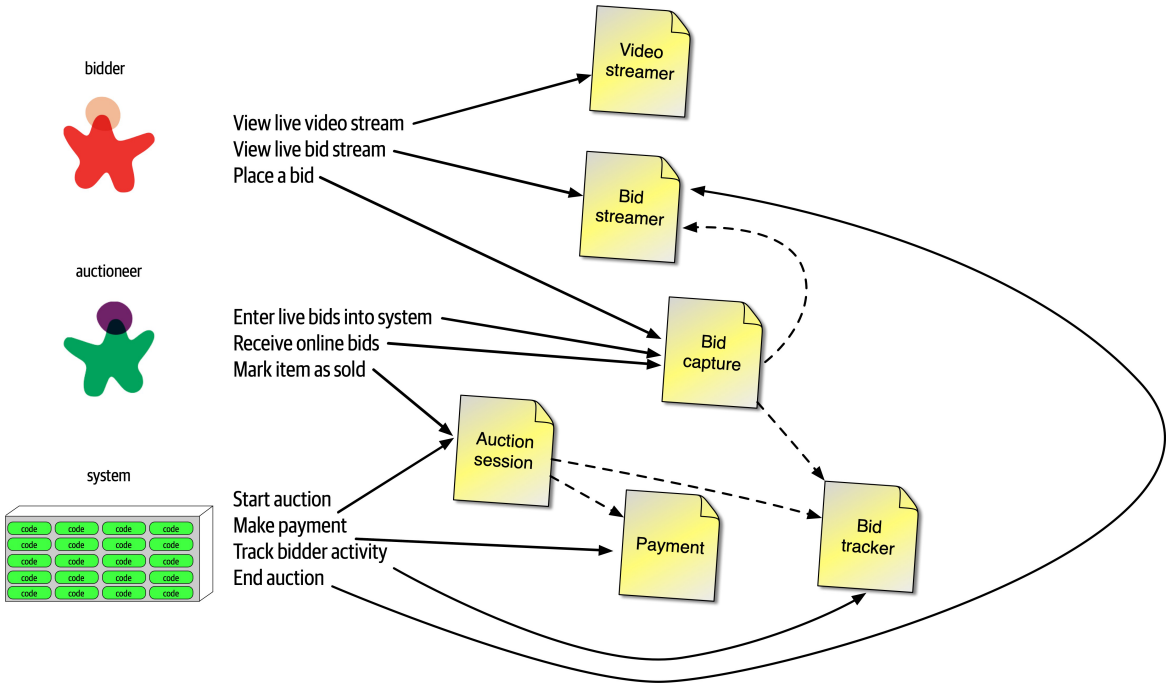

Figure 8-10. Initial set of components for Going, Going, Gone

In Figure 8-10, each of the roles and actions maps to a component, which in turn may need to collaborate on information. These are the components we identified for this solution:

`VideoStreamer`

Streams a live auction to users.

`BidStreamer`

Streams bids as they occur to the users. Both `Video Streamer` and `Bid Streamer` offer read-only views of the auction to the bidder.

`BidCapture`

This component captures bids from both the auctioneer and bidders.

`BidTracker`

Tracks bids and acts as the system of record.

AuctionSession

: Starts and stops an auction. When the bidder ends the auction, performs the payment and resolution steps, including notifying bidders of ending.

Payment

: Third-party payment processor for credit card payments.

Referring to the component identification flow diagram in Figure 8-8, after the initial identification of components, the architect next analyzes architecture characteristics to determine if that will change the design. For this system, the architect can definitely identify different sets of architecture characteristics. For example, the current design features a `BidCapture` component to capture bids from both bidders and the auctioneer, which makes sense functionally: capturing bids from anyone can be handled the same. However, what about architecture characteristics around bid capture? The auctioneer doesn't need the same level of scalability or elasticity as potentially thousands of bidders. By the same token, an architect must ensure that architecture characteristics like reliability (connections don't drop) and availability (the system is up) for the auctioneer could be higher than other parts of the system. For example, while it's bad for business if a bidder can't log in to the site or if they suffer from a dropped connection, it's disastrous to the auction if either of those things happen to the auctioneer.

Because they have differing levels of architecture characteristics, the architect decides to split the `Bid Capture` component into `Bid Capture` and `Auctioneer Capture` so that each of the two components can support differing architecture characteristics. The updated design appears in Figure 8-11.

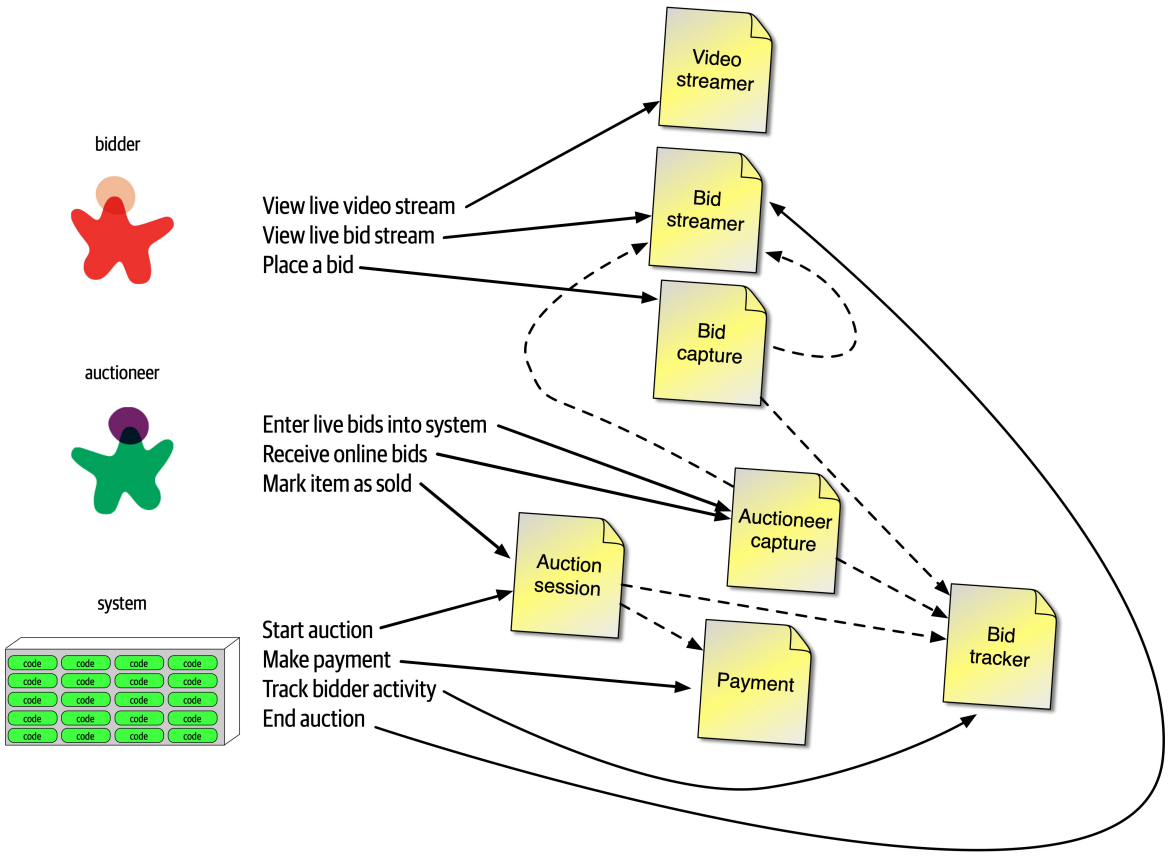

Figure 8-11. Incorporating architecture characteristics into GGG component design

In Figure 8-11, the architect creates a new component for `Auctioneer Capture` and updates information links to both `Bid Streamer` (so that online bidders see the live bids) and `Bid Tracker`, which is managing the bid streams. Note that `Bid Tracker` is now the component that will unify the two very different information streams: the single stream of information from the auctioneer and the multiple streams from bidders.

The design shown in Figure 8-11 isn't likely the final design. More requirements must be uncovered (how do people register, administration functions around payment, and so on). However, this example provides a good starting point to start iterating further on the design.

This is one possible set of components to solve the GGG problem—but it's not necessarily correct, nor is it the only one. Few software systems have only one way that developers can implement them; every design has different sets of trade-offs. As an architect, don't obsess over finding the

one true design, because many will suffice (and less likely overengineered). Rather, try to objectively assess the trade-offs between different design decisions, and choose the one that has the least worst set of trade-offs.

Architecture Quantum Redux: Choosing Between Monolithic Versus Distributed Architectures

Recalling the discussion defining architecture quantum in "Architectural Quanta and Granularity", the architecture quantum defines the scope of architecture characteristics. That in turn leads an architect toward an important decision as they finish their initial component design: should the architecture be monolithic or distributed?

A *monolithic* architecture typically features a single deployable unit, including all functionality of the system that runs in the process, typically connected to a single database. Types of monolithic architectures include the layered and modular monolith, discussed fully in Chapter 10. A *distributed* architecture is the opposite—the application consists of multiple services running in their own ecosystem, communicating via networking protocols. Distributed architectures may feature finer-grained deployment models, where each service may have its own release cadence and engineering practices, based on the development team and their priorities.

Each architecture style offers a variety of trade-offs, covered in Part II. However, the fundamental decision rests on how many quanta the architecture discovers during the design process. If the system can manage with a single quantum (in other words, one set of architecture characteristics), then a monolith architecture offers many advantages. On the other hand, differing architecture characteristics for components, as illustrated in the GGG component analysis, requires a distributed architecture to accommodate differing architecture characteristics. For example, the `VideoStreamer` and `BidStreamer` both offer read-only views of the auction to bidders. From a design standpoint, an architect would

rather not deal with read-only streaming mixed with high-scale updates. Along with the aforementioned differences between bidder and auctioneer, these differing characteristics lead an architect to choose a distributed architecture.

The ability to determine a fundamental design characteristic of architecture (monolith versus distributed) early in the design process highlights one of the advantages of using the architecture quantum as a way of analyzing architecture characteristics scope and coupling.

Part II. Architecture Styles

The difference between an architecture style and an architecture pattern can be confusing. We define an *architecture style* as the overarching structure of how the user interface and backend source code are organized (such as within layers of a monolithic deployment or separately deployed services) and how that source code interacts with a datastore. *Architecture patterns*, on the other hand, are lower-level design structures that help form specific solutions within an architecture style (such as how to achieve high scalability or high performance within a set of operations or between sets of services).

Understanding architecture styles occupies much of the time and effort for new architects because they share importance and abundance. Architects must understand the various styles and the trade-offs encapsulated within each to make effective decisions; each architecture style embodies a well-known set of trade-offs that help an architect make the right choice for a particular business problem.

Chapter 9. Foundations

Architecture styles, sometimes called architecture patterns, describe a named relationship of components covering a variety of architecture characteristics. An architecture style name, similar to design patterns, creates a single name that acts as shorthand between experienced architects. For example, when an architect talks about a layered monolith, their target in the conversation understands aspects of structure, which kinds of architecture characteristics work well (and which ones can cause problems), typical deployment models, data strategies, and a host of other information. Thus, architects should be familiar with the basic names of fundamental generic architecture styles.

Each name captures a wealth of understood detail, one of the purposes of design patterns. An architecture style describes the topology, assumed and default architecture characteristics, both beneficial and detrimental. We cover many common modern architecture patterns in the remainder of this section of the book (Part II). However, architects should be familiar with several fundamental patterns that appear embedded within the larger patterns.

Fundamental Patterns

Several fundamental patterns appear again and again throughout the history of software architecture because they provide a useful perspective on organizing code, deployments, or other aspects of architecture. For example, the concept of layers in architecture, separating different concerns based on functionality, is as old as software itself. Yet, the layered pattern continues to manifest in different guises, including modern variants discussed in Chapter 10.

Big Ball of Mud

Architects refer to the absence of any discernible architecture structure as a *Big Ball of Mud*, named after the eponymous anti-pattern defined in a paper released in 1997 by Brian Foote and Joseph Yoder:

> *A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.*
>
> *The overall structure of the system may never have been well defined.*
>
> *If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.*
>
> —Brian Foote and Joseph Yoder

In modern terms, a *big ball of mud* might describe a simple scripting application with event handlers wired directly to database calls, with no real internal structure. Many trivial applications start like this then become unwieldy as they continue to grow.

In general, architects want to avoid this type of architecture at all costs. The lack of structure makes change increasingly difficult. This type of architecture also suffers from problems in deployment, testability, scalability, and performance.

Unfortunately, this architecture anti-pattern occurs quite commonly in the real world. Few architects intend to create one, but many projects inadvertently manage to create a mess because of lack of governance around code quality and structure. For example, Neal worked with a client project whose structure appears in Figure 9-1.

The client (whose name is withheld for obvious reasons) created a Java-based web application as quickly as possible over several years. The technical visualization[1] shows their architectural coupling: each dot on the perimeter of the circle represents a class, and each line represents connections between the classes, where bolder lines indicate stronger connections. In this code base, any change to a class makes it difficult to predict rippling side effects to other classes, making change a terrifying affair.

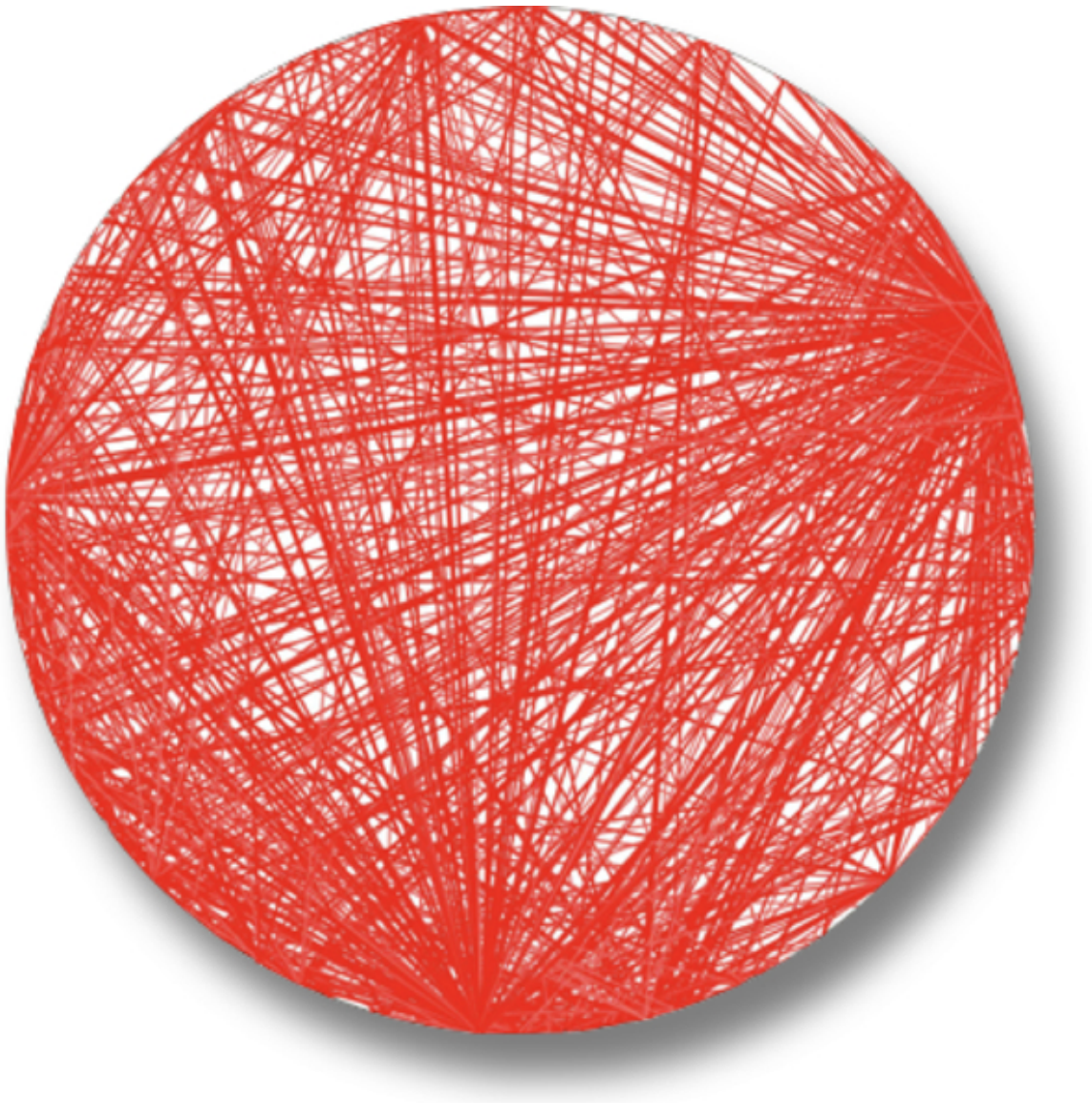

Figure 9-1. A Big Ball of Mud architecture visualized from a real code base

Unitary Architecture

When software originated, there was only the computer, and software ran on it. Through the various eras of hardware and software evolution, the two started as a single entity, then split as the need for more sophisticated capabilities grew. For example, mainframe computers started as singular systems, then gradually separated data into its own kind of system. Similarly, when personal computers first appeared, much of the commercial development focused on single machines. As networking PCs became common, distributed systems (such as client/server) appeared.

Few unitary architectures exist outside embedded systems and other highly constrained environments. Generally, software systems tend to grow in functionality over time, requiring separation of concerns to maintain operational architecture characteristics, such as performance and scale.

Client/Server

Over time, various forces required partitioning away from a single system; how to do that forms the basis for many of these styles. Many architecture styles deal with how to efficiently separate parts of the system.

A fundamental style in architecture separates technical functionality between frontend and backend, called a *two-tier*, or *client/server*, architecture. Many different flavors of this architecture exist, depending on the era and computing capabilities.

Desktop + database server

An early personal computer architecture encouraged developers to write rich desktop applications in user interfaces like Windows, separating the data into a separate database server. This architecture coincided with the appearance of standalone database servers that could connect via standard network protocols. It allowed presentation logic to reside on the desktop, while the more computationally intense action (both in volume and complexity) occurred on more robust database servers.

Browser + web server

Once modern web development arrived, the common split became web browser connected to web server (which in turn was connected to a database server). The separation of responsibilities was similar to the desktop variant but with even thinner clients as browsers, allowing a wider distribution both inside and outside firewalls. Even though the database is separate from the web server, architects often still consider this a two-tier architecture because the web and database servers run on one class of machine within the operations center and the user interface runs on the user's browser.

Three-tier

An architecture that became quite popular during the late 1990s was a *three-tier architecture*, which provided even more layers of separation. As tools like application servers became popular in Java and .NET, companies started building even more layers in their topology: a database tier using an industrial-strength database server, an application tier managed by an application server, frontend coded in generated HTML, and increasingly, JavaScript, as its capabilities expanded.

The three-tier architecture corresponded with network-level protocols such as Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) that facilitated building distributed architectures.

Just as developers today don't worry about how network protocols like TCP/IP work (they just work), most architects don't have to worry about this level of plumbing in distributed architectures. The capabilities offered by such tools in that era exist today as either tools (like message queues) or architecture patterns (such as event-driven architecture, covered in Chapter 14).

THREE-TIER, LANGUAGE DESIGN, AND LONG-TERM IMPLICATIONS

During the era in which the Java language was designed, three-tier computing was all the rage. Thus, it was assumed that, in the future, all systems would be three-tier architectures. One of the common headaches with existing languages such as C++ was how cumbersome it was to move objects over the network in a consistent way between systems. Thus, the designers of Java decided to build this capability into the core of the language using a mechanism called *serialization*. Every `Object` in Java implements an interface that requires it to support serialization. The designers figured that since three-tiered architecture would forever be the architecture style, baking it into the language would offer a great convenience. Of course, that architectural style came and went, yet the leftovers appear in Java to this day, greatly frustrating the language designer who wants to add modern features that, for backward compatibility, must support serialization, which virtually no one uses today.

Understanding the long-term implications of design decisions has always eluded us, in software, as in other engineering disciplines. The perpetual advice to favor simple designs is in many ways defense against future consequences.

Monolithic Versus Distributed Architectures

Architecture styles can be classified into two main types: *monolithic* (single deployment unit of all code) and *distributed* (multiple deployment units connected through remote access protocols). While no classification scheme is perfect, distributed architectures all share a common set of challenges and issues not found in the monolithic architecture styles, making this classification scheme a good separation between the various architecture styles.

In this book we will describe in detail the following architecture styles:

Monolithic

- Layered architecture (Chapter 10)
- Pipeline architecture (Chapter 11)
- Microkernel architecture (Chapter 12)

Distributed

- Service-based architecture (Chapter 13)
- Event-driven architecture (Chapter 14)
- Space-based architecture (Chapter 15)
- Service-oriented architecture (Chapter 16)
- Microservices architecture (Chapter 17)

Distributed architecture styles, while being much more powerful in terms of performance, scalability, and availability than monolithic architecture styles, have significant trade-offs for this power. The first group of issues facing all distributed architectures are described in *the fallacies of distributed computing*, first coined by L. Peter Deutsch and other colleagues from Sun Microsystems in 1994. A *fallacy* is something that is believed or assumed to be true but is not. All eight of the fallacies of distributed computing apply to distributed architectures today. The following sections describe each fallacy.

Fallacy #1: The Network Is Reliable

Figure 9-2. The network is not reliable

Developers and architects alike assume that the network is reliable, but it is not. While networks have become more reliable over time, the fact of the matter is that networks still remain generally unreliable. This is significant for all distributed architectures because all distributed architecture styles rely on the network for communication to and from services, as well as between services. As illustrated in Figure 9-2, `Service B` may be totally healthy, but `Service A` cannot reach it due to a network problem; or even worse, `Service A` made a request to `Service B` to process some data and does not receive a response because of a network issue. This is why things like timeouts and circuit breakers exist between services. The more a system relies on the network (such as microservices architecture), the potentially less reliable it becomes.

Fallacy #2: Latency Is Zero

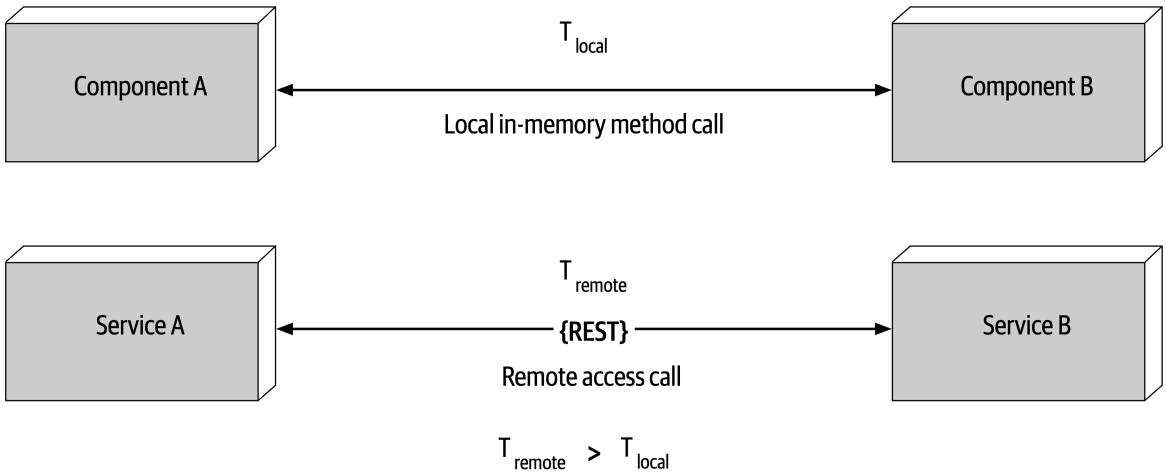

Figure 9-3. Latency is not zero

As Figure 9-3 shows, when a local call is made to another component via a method or function call, that time (`t_local`) is measured in nanoseconds or microseconds. However, when that same call is made through a remote access protocol (such as REST, messaging, or RPC), the time measured to access that service (`t_remote`) is measured in milliseconds. Therefore, `t_remote` will always be greater that `t_local`. Latency in any distributed architecture is not zero, yet most architects ignore this fallacy, insisting that they have fast networks. Ask yourself this question: do you know what the average round-trip latency is for a RESTful call in your production environment? Is it 60 milliseconds? Is it 500 milliseconds?

When using any distributed architecture, architects must know this latency average. It is the only way of determining whether a distributed architecture is feasible, particularly when considering microservices (see Chapter 17) due to the fine-grained nature of the services and the amount of communication between those services. Assuming an average of 100 milliseconds of latency per request, chaining together 10 service calls to perform a particular business function adds 1,000 milliseconds to the request! Knowing the average latency is important, but even more important is also knowing the 95th to 99th percentile. While an average latency might yield only 60 milliseconds (which is good), the 95th percentile might be 400 milliseconds! It's usually this "long tail" latency

that will kill performance in a distributed architecture. In most cases, architects can get latency values from a network administrator (see “Fallacy #6: There Is Only One Administrator”).

Fallacy #3: Bandwidth Is Infinite

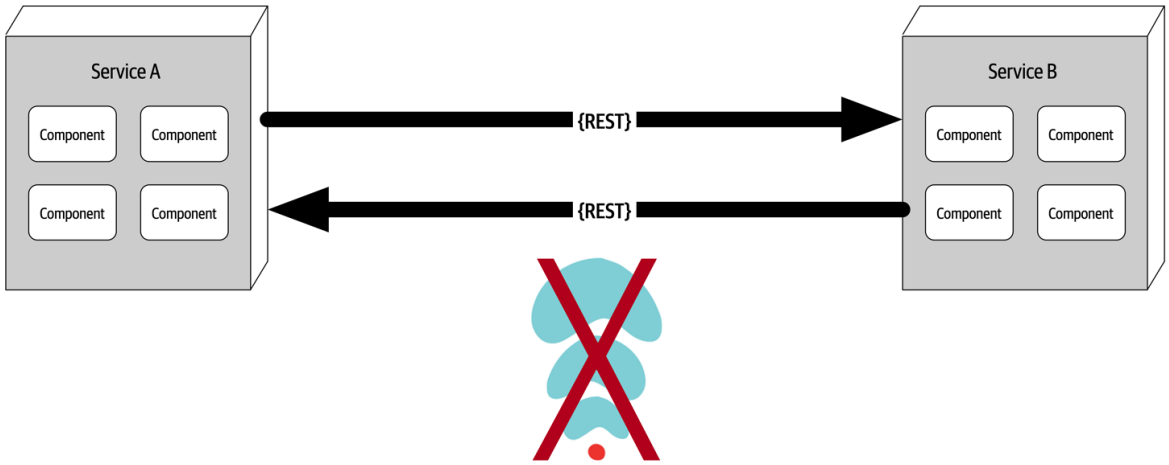

Figure 9-4. Bandwidth is not infinite

Bandwidth is usually not a concern in monolithic architectures, because once processing goes into a monolith, little or no bandwidth is required to process that business request. However, as shown in Figure 9-4, once systems are broken apart into smaller deployment units (services) in a distributed architecture such as microservices, communication to and between these services significantly utilizes bandwidth, causing networks to slow down, thus impacting latency (fallacy #2) and reliability (fallacy #1).

To illustrate the importance of this fallacy, consider the two services shown in Figure 9-4. Let’s say the lefthand service manages the wish list items for the website, and the righthand service manages the customer profile. Whenever a request for a wish list comes into the lefthand service, it must make an interservice call to the righthand customer profile service to get the customer name because that data is needed in the response contract for the wish list, but the wish list service on the lefthand side doesn’t have the name. The customer profile service returns 45 attributes totaling 500 kb to the wish list service, which only needs the name (200 bytes). This is a form

of coupling referred to as *stamp coupling*. This may not sound significant, but requests for the wish list items happen about 2,000 times a second. This means that this interservice call from the wish list service to the customer profile service happens 2,000 times a second. At 500 kb for each request, the amount of bandwidth used for that *one* interservice call (out of hundreds being made that second) is 1 Gb!

Stamp coupling in distributed architectures consumes significant amounts of bandwidth. If the customer profile service were to only pass back the data needed by the wish list service (in this case 200 bytes), the total bandwidth used to transmit the data is only 400 kb. Stamp coupling can be resolved in the following ways:

- Create private RESTful API endpoints
- Use field selectors in the contract
- Use GraphQL to decouple contracts
- Use value-driven contracts with consumer-driven contracts (CDCs)
- Use internal messaging endpoints

Regardless of the technique used, ensuring that the minimal amount of data is passed between services or systems in a distributed architecture is the best way to address this fallacy.

Fallacy #4: The Network Is Secure

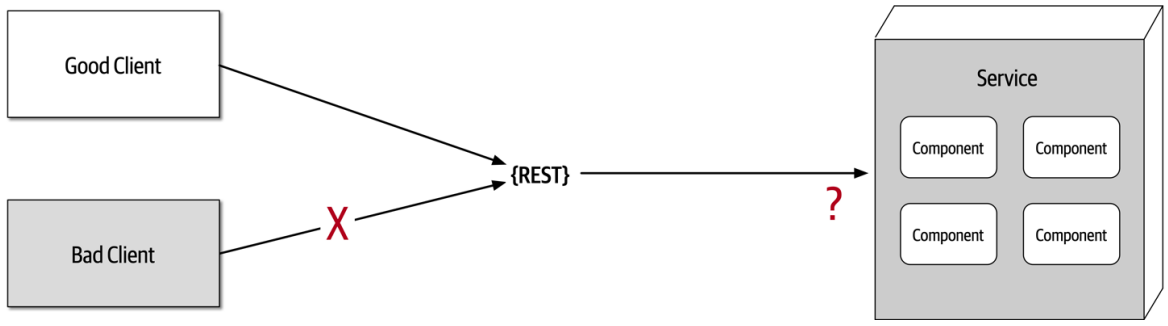

Figure 9-5. The network is not secure

Most architects and developers get so comfortable using virtual private networks (VPNs), trusted networks, and firewalls that they tend to forget about this fallacy of distributed computing: *the network is not secure*. Security becomes much more challenging in a distributed architecture. As shown in Figure 9-5, each and every endpoint to each distributed deployment unit must be secured so that unknown or bad requests do not make it to that service. The surface area for threats and attacks increases by magnitudes when moving from a monolithic to a distributed architecture. Having to secure every endpoint, even when doing interservice communication, is another reason performance tends to be slower in synchronous, highly-distributed architectures such as microservices or service-based architecture.

Fallacy #5: The Topology Never Changes

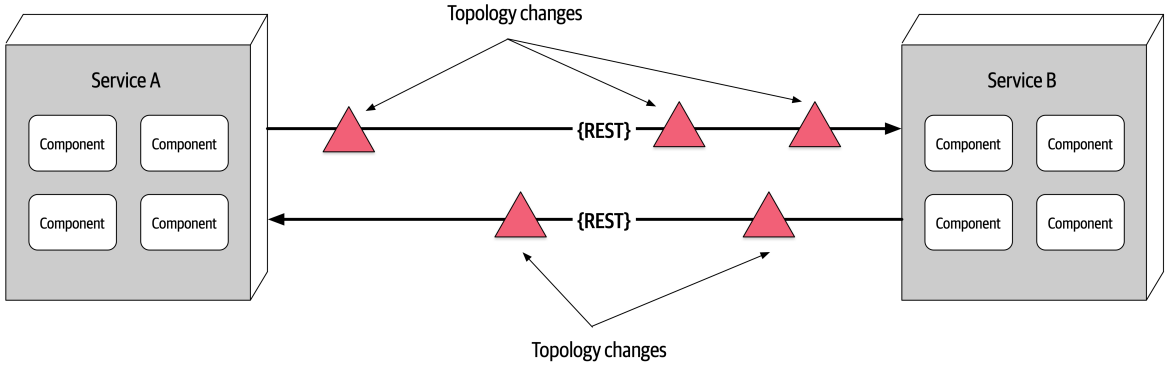

Figure 9-6. The network topology always changes

This fallacy refers to the overall network topology, including all of the routers, hubs, switches, firewalls, networks, and appliances used within the overall network. Architects assume that the topology is fixed and never changes. *Of course it changes*. It changes all the time. What is the significance of this fallacy?

Suppose an architect comes into work on a Monday morning, and everyone is running around like crazy because services keep timing out in production. The architect works with the teams, frantically trying to figure out why this is happening. No new services were deployed over the weekend. What could it be? After several hours the architect discovers that a minor network

upgrade happened at 2 a.m. that morning. This supposedly “minor” network upgrade invalidated all of the latency assumptions, triggering timeouts and circuit breakers.

Architects must be in constant communication with operations and network administrators to know what is changing and when so that they can make adjustments accordingly to reduce the type of surprise previously described. This may seem obvious and easy, but it is not. As a matter of fact, this fallacy leads directly to the next fallacy.

Fallacy #6: There Is Only One Administrator

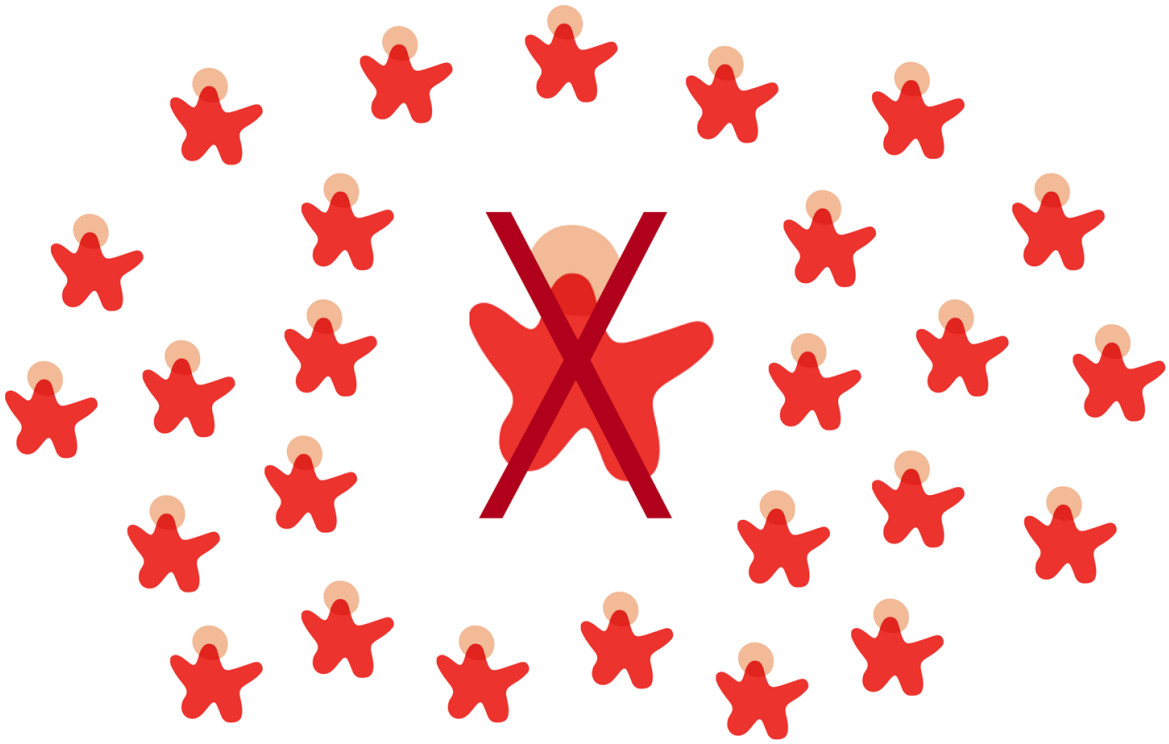

Figure 9-7. There are many network administrators, not just one

Architects all the time fall into this fallacy, assuming they only need to collaborate and communicate with one administrator. As shown in Figure 9-7, there are dozens of network administrators in a typical large company. Who should the architect talk to with regard to latency (“Fallacy #2: Latency Is Zero”) or topology changes (“Fallacy #5: The Topology Never Changes”)? This fallacy points to the complexity of distributed architecture

and the amount of coordination that must happen to get everything working correctly. Monolithic applications do not require this level of communication and collaboration due to the single deployment unit characteristics of those architecture styles.

Fallacy #7: Transport Cost Is Zero

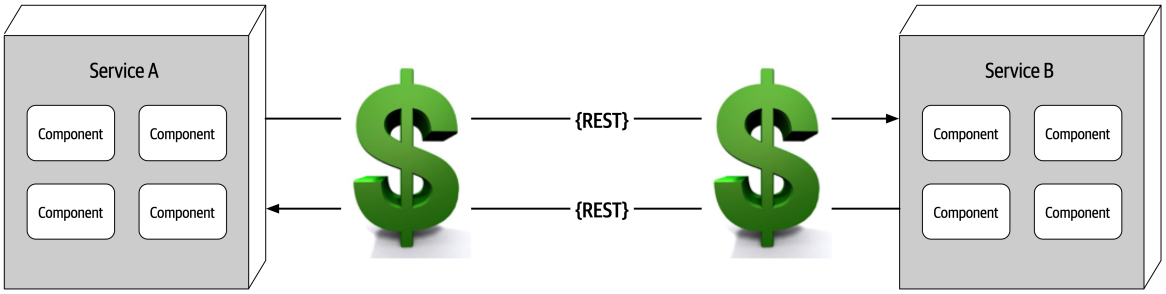

Figure 9-8. Remote access costs money

Many software architects confuse this fallacy for latency (“Fallacy #2: Latency Is Zero”). Transport cost here does not refer to latency, but rather to actual *cost* in terms of money associated with making a “simple RESTful call.” Architects assume (incorrectly) that the necessary infrastructure is in place and sufficient for making a simple RESTful call or breaking apart a monolithic application. *It is usually not*. Distributed architectures cost significantly more than monolithic architectures, primarily due to increased needs for additional hardware, servers, gateways, firewalls, new subnets, proxies, and so on.

Whenever embarking on a distributed architecture, we encourage architects to analyze the current server and network topology with regard to capacity, bandwidth, latency, and security zones to not get caught up in the trap of surprise with this fallacy.

Fallacy #8: The Network Is Homogeneous

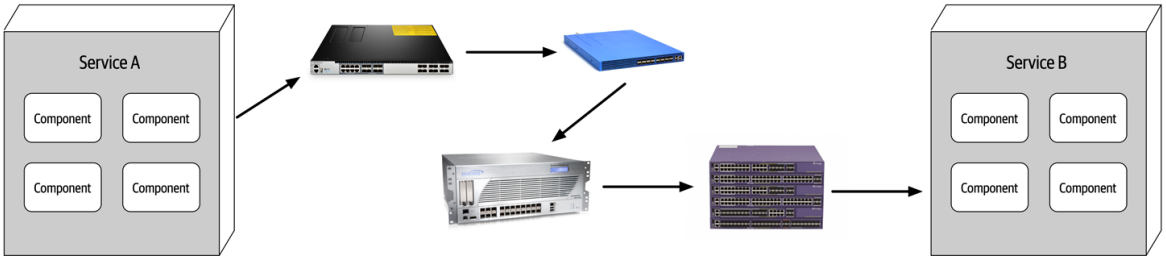

Figure 9-9. The network is not homogeneous

Most architects and developers assume a network is homogeneous—made up by only one network hardware vendor. Nothing could be farther from the truth. Most companies have multiple network hardware vendors in their infrastructure, if not more.

So what? The significance of this fallacy is that not all of those heterogeneous hardware vendors play together well. Most of it works, but does Juniper hardware seamlessly integrate with Cisco hardware? Networking standards have evolved over the years, making this less of an issue, but the fact remains that not all situations, load, and circumstances have been fully tested, and as such, network packets occasionally get lost. This in turn impacts network reliability (“Fallacy #1: The Network Is Reliable”), latency assumptions and assertions (“Fallacy #2: Latency Is Zero”), and assumptions made about the bandwidth (“Fallacy #3: Bandwidth Is Infinite”). In other words, this fallacy ties back into all of the other fallacies, forming an endless loop of confusion and frustration when dealing with networks (which is necessary when using distributed architectures).

Other Distributed Considerations

In addition to the eight fallacies of distributed computing previously described, there are other issues and challenges facing distributed architecture that aren’t present in monolithic architectures. Although the details of these other issues are out of scope for this book, we list and summarize them in the following sections.

Distributed logging

Performing root-cause analysis to determine why a particular order was dropped is very difficult and time-consuming in a distributed architecture due to the distribution of application and system logs. In a monolithic application there is typically only one log, making it easier to trace a request and determine the issue. However, distributed architectures contain dozens to hundreds of different logs, all located in a different place and all with a different format, making it difficult to track down a problem.

Logging consolidation tools such as Splunk help to consolidate information from various sources and systems together into one consolidated log and console, but these tools only scratch the surface of the complexities involved with distributed logging. Detailed solutions and patterns for distributed logging are outside the scope of this book.

Distributed transactions

Architects and developers take transactions for granted in a monolithic architecture world because they are so straightforward and easy to manage. Standard `commits` and `rollbacks` executed from persistence frameworks leverage ACID (atomicity, consistency, isolation, durability) transactions to guarantee that the data is updated in a correct way to ensure high data consistency and integrity. Such is not the case with distributed architectures.

Distributed architectures rely on what is called *eventual consistency* to ensure the data processed by separate deployment units is at some unspecified point in time all synchronized into a consistent state. This is one of the trade-offs of distributed architecture: high scalability, performance, and availability at the sacrifice of data consistency and data integrity.

Transactional sagas are one way to manage distributed transactions. Sagas utilize either event sourcing for compensation or finite state machines to manage the state of transaction. In addition to sagas, *BASE* transactions are used. BASE stands for (B)asic availability, (S)oft state, and (E)ventual consistency. BASE transactions are not a piece of software, but rather a technique. *Soft state* in BASE refers to the transit of data from a source to a

target, as well as the inconsistency between data sources. Based on the *basic availability* of the systems or services involved, the systems will *eventually* become consistent through the use of architecture patterns and messaging.

Contract maintenance and versioning

Another particularly difficult challenge within distributed architecture is contract creation, maintenance, and versioning. A contract is behavior and data that is agreed upon by both the client and the service. Contract maintenance is particularly difficult in distributed architectures, primarily due to decoupled services and systems owned by different teams and departments. Even more complex are the communication models needed for version deprecation.

---

1 Made with a now-retired tool called XRay, an Eclipse plug-in.

Chapter 10. Layered Architecture Style

The *layered* architecture, also known as the *n-tiered* architecture style, is one of the most common architecture styles. This style of architecture is the de facto standard for most applications, primarily because of its simplicity, familiarity, and low cost. It is also a very natural way to develop applications due to Conway's law, which states that organizations that design systems are constrained to produce designs which are copies of the communication structures of these organizations. In most organizations there are user interface (UI) developers, backend developers, rules developers, and database experts (DBAs). These organizational layers fit nicely into the tiers of a traditional layered architecture, making it a natural choice for many business applications. The layered architecture style also falls into several architectural anti-patterns, including the *architecture by implication* anti-pattern and the *accidental architecture* anti-pattern. If a developer or architect is unsure which architecture style they are using, or if an Agile development team "just starts coding," chances are good that it is the layered architecture style they are implementing.

Topology

Components within the layered architecture style are organized into logical horizontal layers, with each layer performing a specific role within the application (such as presentation logic or business logic). Although there are no specific restrictions in terms of the number and types of layers that must exist, most layered architectures consist of four standard layers: presentation, business, persistence, and database, as illustrated in Figure 10-1. In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (such as SQL

or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

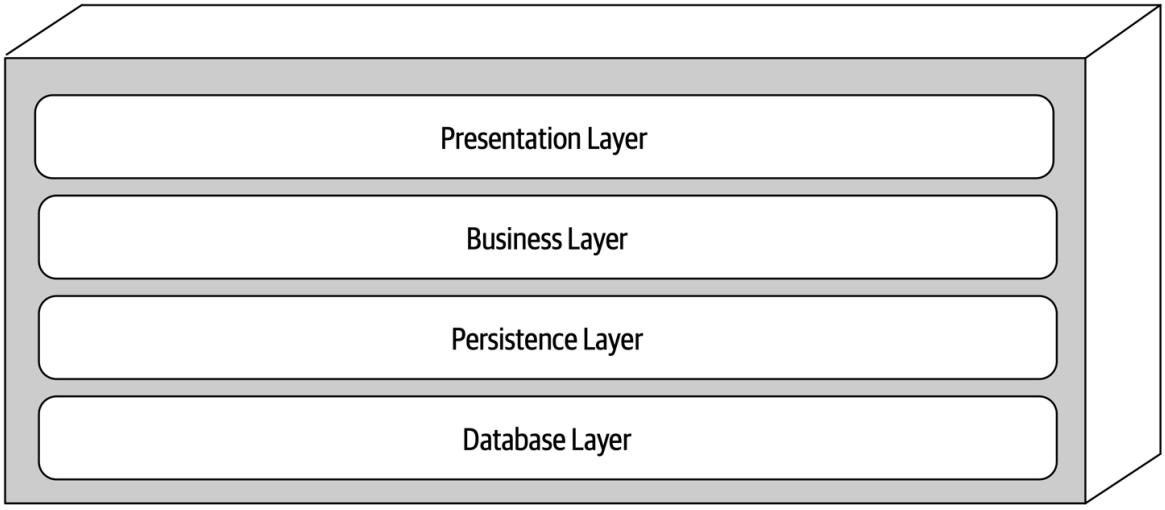

Figure 10-1. Standard logical layers within the layered architecture style

Figure 10-2 illustrates the various topology variants from a physical layering (deployment) perspective. The first variant combines the presentation, business, and persistence layers into a single deployment unit, with the database layer typically represented as a separate external physical database (or filesystem). The second variant physically separates the presentation layer into its own deployment unit, with the business and persistence layers combined into a second deployment unit. Again, with this variant, the database layer is usually physically separated through an external database or filesystem. A third variant combines all four standard layers into a single deployment, including the database layer. This variant might be useful for smaller applications with either an internally embedded database or an in-memory database. Many on-premises ("on-prem") products are built and delivered to customers using this third variant.

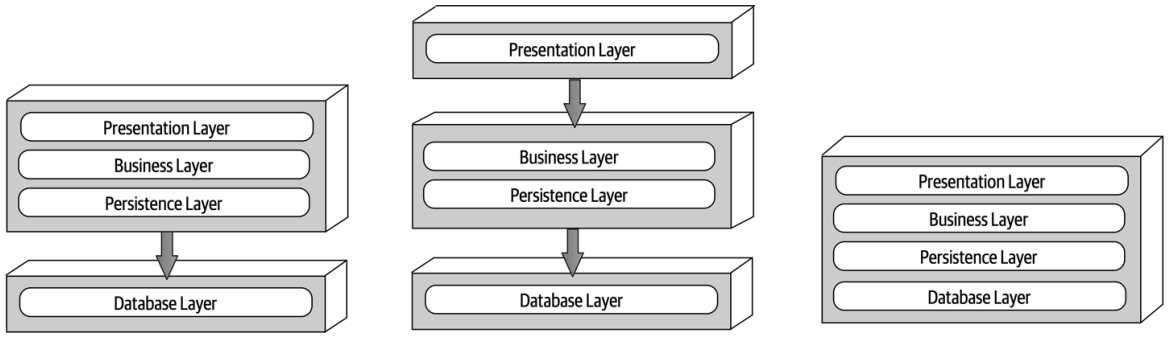

Figure 10-2. Physical topology (deployment) variants

Each layer of the layered architecture style has a specific role and responsibility within the architecture. For example, the presentation layer would be responsible for handling all user interface and browser communication logic, whereas the business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in a particular format. Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (such as calculating values or aggregating data), and pass that information up to the presentation layer.

This *separation of concerns* concept within the layered architecture style makes it easy to build effective roles and responsibility models within the architecture. Components within a specific layer are limited in scope, dealing only with the logic that pertains to that layer. For example, components in the presentation layer only handle presentation logic, whereas components residing in the business layer only handle business logic. This allows developers to leverage their particular technical expertise to focus on the technical aspects of the domain (such as presentation logic or persistence logic). The trade-off of this benefit, however, is a lack of overall agility (the ability to respond quickly to change).

The layered architecture is a *technically partitioned* architecture (as opposed to a *domain-partitioned* architecture). Groups of components, rather than being grouped by domain (such as customer), are grouped by their technical role in the architecture (such as presentation or business). As a result, any particular business domain is spread throughout all of the layers of the architecture. For example, the domain of "customer" is contained in the presentation layer, business layer, rules layer, services layer, and database layer, making it difficult to apply changes to that domain. As a result, a domain-driven design approach does not work as well with the layered architecture style.

Layers of Isolation

Each layer in the layered architecture style can be either *closed* or *open*. A closed layer means that as a request moves top-down from layer to layer, the request cannot skip any layers, but rather must go through the layer immediately below it to get to the next layer (see Figure 10-3). For example, in a closed-layered architecture, a request originating from the presentation layer must first go through the business layer and then to the persistence layer before finally making it to the database layer.

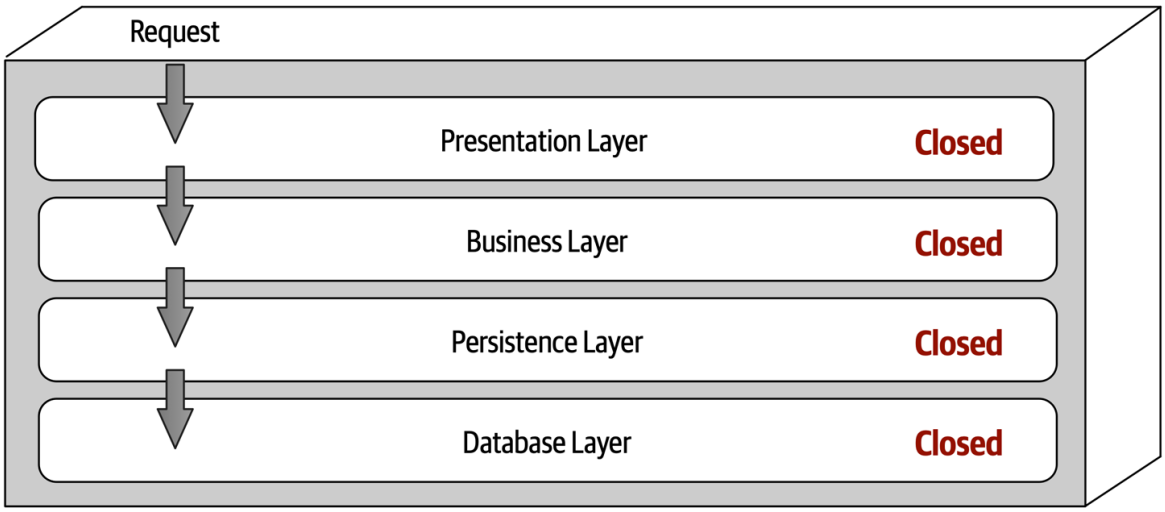

Figure 10-3. Closed layers within the layered architecture

Notice that in Figure 10-3 it would be much faster and easier for the presentation layer to access the database directly for simple retrieval requests, bypassing any unnecessary layers (what used to be known in the early 2000s as the *fast-lane reader pattern*). For this to happen, the business and persistence layers would have to be *open*, allowing requests to bypass other layers. Which is better—open layers or closed layers? The answer to this question lies in a key concept known as *layers of isolation*.

The *layers of isolation* concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers, providing the contracts between those layers remain unchanged. Each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture. However, to support layers of isolation, layers involved with the major flow of the request necessarily have to be closed. If the presentation layer can directly access the persistence layer, then changes made to the persistence layer would impact both the business layer *and* the presentation layer, producing a very tightly coupled application with layer interdependencies between components. This type of architecture then becomes very brittle, as well as difficult and expensive to change.

The layers of isolation concept also allows any layer in the architecture to be replaced without impacting any other layer (again, assuming well-defined contracts and the use of the business delegate pattern). For example, you can leverage the layers of isolation concept within the layered architecture style to replace your older JavaServer Faces (JSF) presentation layer with React.js without impacting any other layer in the application.

Adding Layers

While closed layers facilitate layers of isolation and therefore help isolate change within the architecture, there are times when it makes sense for certain layers to be open. For example, suppose there are shared objects within the business layer that contain common functionality for business components (such as date and string utility classes, auditing classes, logging

classes, and so on). Suppose there is an architecture decision stating that the presentation layer is restricted from using these shared business objects. This constraint is illustrated in Figure 10-4, with the dotted line going from a presentation component to a shared business object in the business layer. This scenario is difficult to govern and control because *architecturally* the presentation layer has access to the business layer, and hence has access to the shared objects within that layer.

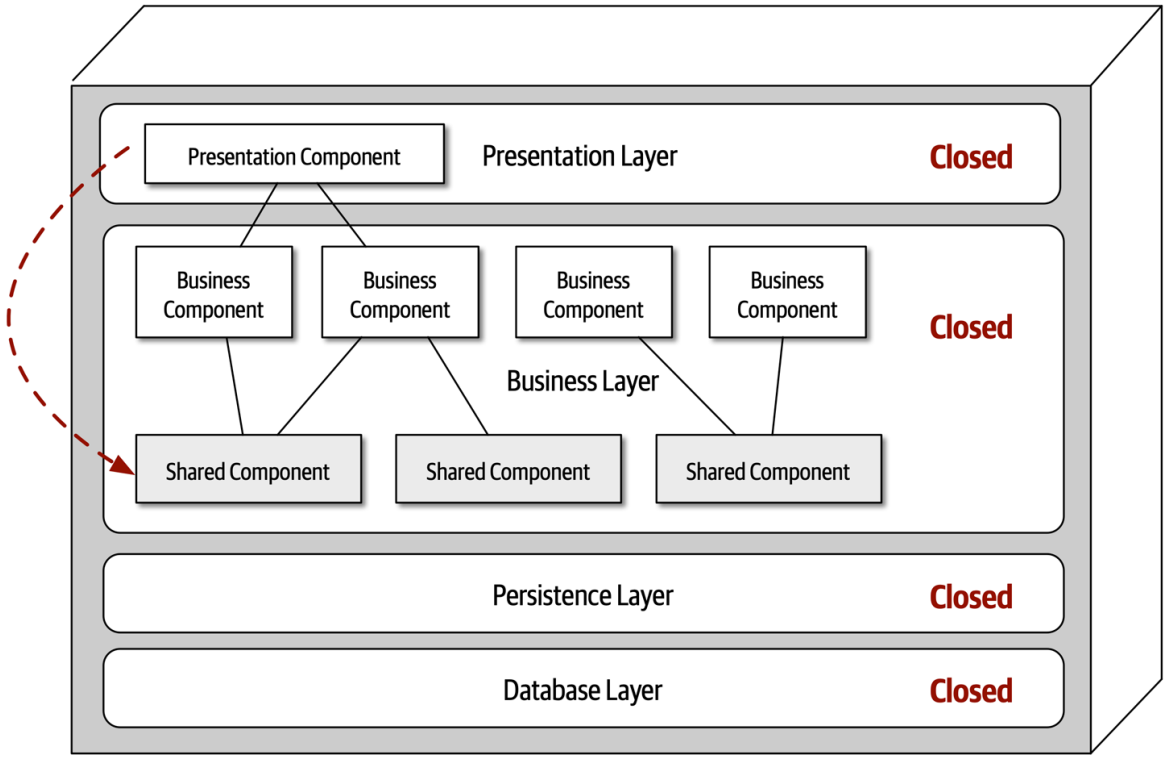

Figure 10-4. Shared objects within the business layer

One way to architecturally mandate this restriction is to add to the architecture a new services layer containing all of the shared business objects. Adding this new layer now architecturally restricts the presentation layer from accessing the shared business objects because the business layer is closed (see Figure 10-5). However, the new services layer must be marked as *open*; otherwise the business layer would be forced to go through the services layer to access the persistence layer. Marking the services layer as open allows the business layer to either access that layer (as indicated by the solid arrow), or bypass the layer and go to the next one down (as indicated by the dotted arrow in Figure 10-5).

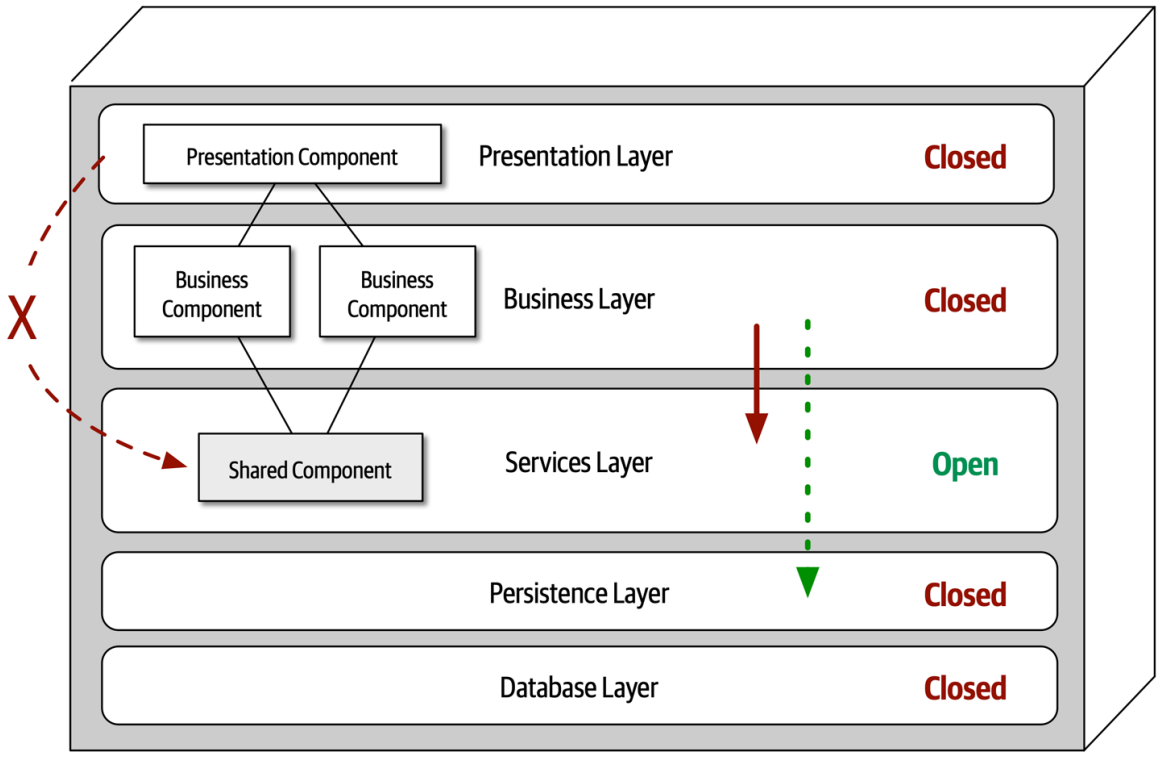

Figure 10-5. Adding a new services layer to the architecture

Leveraging the concept of open and closed layers helps define the relationship between architecture layers and request flows. It also provides developers with the necessary information and guidance to understand various layer access restrictions within the architecture. Failure to document or properly communicate which layers in the architecture are open and closed (and why) usually results in tightly coupled and brittle architectures that are very difficult to test, maintain, and deploy.

Other Considerations

The layered architecture makes for a good starting point for most applications when it is not known yet exactly which architecture style will ultimately be used. This is a common practice for many microservices efforts when architects are still determining whether microservices is the right architecture choice, but development must begin. However, when using this technique, be sure to keep reuse at a minimum and keep object hierarchies (depth of inheritance tree) fairly shallow so as to maintain a

good level of modularity. This will help facilitate the move to another architecture style later on.

One thing to watch out for with the layered architecture is the *architecture sinkhole* anti-pattern. This anti-pattern occurs when requests move from layer to layer as simple pass-through processing with no business logic performed within each layer. For example, suppose the presentation layer responds to a simple request from the user to retrieve basic customer data (such as name and address). The presentation layer passes the request to the business layer, which does nothing but pass the request on to the rules layer, which in turn does nothing but pass the request on to the persistence layer, which then makes a simple SQL call to the database layer to retrieve the customer data. The data is then passed all the way back up the stack with no additional processing or logic to aggregate, calculate, apply rules, or transform the data. This results in unnecessary object instantiation and processing, impacting both memory consumption and performance.

Every layered architecture will have at least some scenarios that fall into the architecture sinkhole anti-pattern. The key to determining whether the architecture sinkhole anti-pattern is at play is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow. For example, it is acceptable if only 20 percent of the requests are sinkholes. However, if 80 percent of the requests are sinkholes, it a good indicator that the layered architecture is not the correct architecture style for the problem domain. Another approach to solving the architecture sinkhole anti-pattern is to make all the layers in the architecture open, realizing, of course, that the trade-off is increased difficulty in managing change within the architecture.

Why Use This Architecture Style

The layered architecture style is a good choice for small, simple applications or websites. It is also a good architecture choice, particularly as a starting point, for situations with very tight budget and time constraints. Because of the simplicity and familiarity among developers and architects,

the layered architecture is perhaps one of the lowest-cost architecture styles, promoting ease of development for smaller applications. The layered architecture style is also a good choice when an architect is still analyzing business needs and requirements and is unsure which architecture style would be best.

As applications using the layered architecture style grow, characteristics like maintainability, agility, testability, and deployability are adversely affected. For this reason, large applications and systems using the layered architecture might be better suited for other, more modular architecture styles.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table (shown in Figure 10-6) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

Figure 10-6. Layered architecture characteristics ratings

Overall cost and simplicity are the primary strengths of the layered architecture style. Being monolithic in nature, layered architectures don't have the complexities associated with distributed architecture styles, are simple and easy to understand, and are relatively low cost to build and maintain. However, as a cautionary note, these ratings start to quickly diminish as monolithic layered architectures get bigger and consequently more complex.

Both deployability and testability rate very low for this architecture style. Deployability rates low due to the ceremony of deployment (effort to deploy), high risk, and lack of frequent deployments. A simple three-line change to a class file in the layered architecture style requires the entire deployment unit to be redeployed, taking in potential database changes, configuration changes, or other coding changes sneaking in alongside the original change. Furthermore, this simple three-line change is usually bundled with dozens of other changes, thereby increasing deployment risk even further (as well as increasing the frequency of deployment). The low testability rating also reflects this scenario; with a simple three-line change, most developers are not going to spend hours executing the entire regression test suite (even if such a thing were to exist in the first place), particularly along with dozens of other changes being made to the monolithic application at the same time. We gave testability a two-star rating (rather than one star) due to the ability to mock or stub components (or even an entire layer), which eases the overall testing effort.

Overall reliability rates medium (three stars) in this architecture style, mostly due to the lack of network traffic, bandwidth, and latency found in most distributed architectures. We only gave the layered architecture three stars for reliability because of the nature of the monolithic deployment, combined with the low ratings for testability (completeness of testing) and deployment risk.

Elasticity and scalability rate very low (one star) for the layered architecture, primarily due to monolithic deployments and the lack of architectural modularity. Although it is possible to make certain functions within a monolith scale more than others, this effort usually requires very complex design techniques such as multithreading, internal messaging, and other parallel processing practices, techniques this architecture isn’t well suited for. However, because the layered architecture is always a single system quantum due to the monolithic user interface, backend processing, and monolithic database, applications can only scale to a certain point based on the single quantum.

Performance is always an interesting characteristic to rate for the layered architecture. We gave it only two stars because the architecture style simply does not lend itself to high-performance systems due to the lack of parallel processing, closed layering, and the sinkhole architecture anti-pattern. Like scalability, performance can be addressed through caching, multithreading, and the like, but it is not a natural characteristic of this architecture style; architects and developers have to work hard to make all this happen.

Layered architectures don’t support fault tolerance due to monolithic deployments and the lack of architectural modularity. If one small part of a layered architecture causes an out-of-memory condition to occur, the entire application unit is impacted and crashes. Furthermore, overall availability is impacted due to the high mean-time-to-recovery (MTTR) usually experienced by most monolithic applications, with startup times ranging anywhere from 2 minutes for smaller applications, up to 15 minutes or more for most large applications.