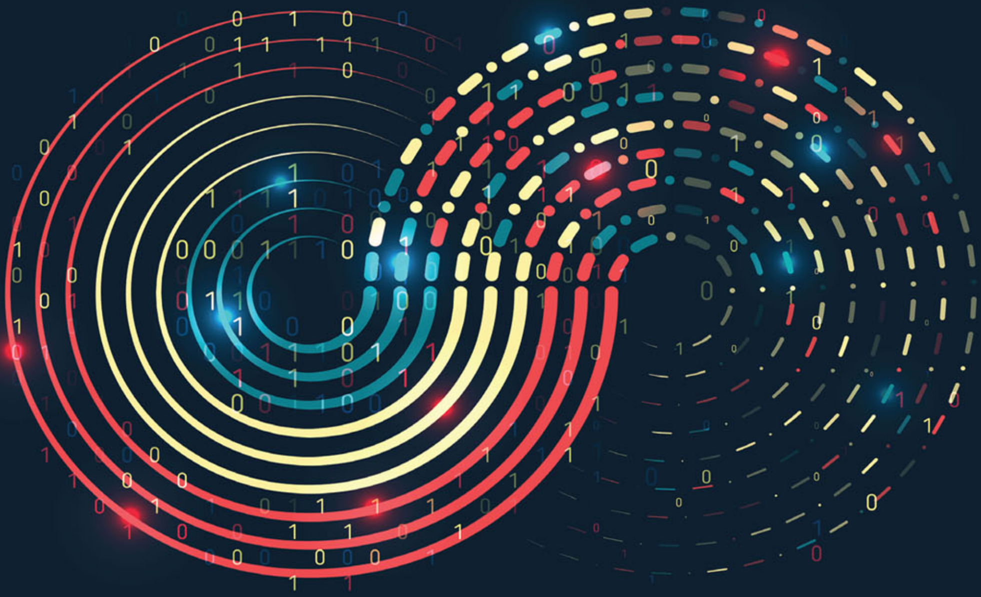

EFFECTIVE SOFTWARE ARCHITECTURE

Building Better Software Faster

OLIVER GOLDMAN

This isn't just another book about architecture. I'd have titled it "How to be a software architect." Goldman guides you on all aspects of the role, from making decisions that will impact a product for years to come to picking good names, taking aim at conventional wisdom along the way.

Daniel Jackson
Professor of Computer Science, MIT

Architects often struggle to have the intended impact within their organizations, despite having the necessary domain expertise and generating good ideas. Oliver's book offers key insights and pragmatic advice for individual architects and architecture teams—helping them to deliver successful outcomes in real world scenarios.

Dan Foygel
Senior Principal Architect, Adobe

Effective Software Architecture hits home with the most meaningful aspects of delivering software at scale. The considerations, concepts, and approaches described apply to everyone I've worked with, not just architecture teams. Oliver offers insights that are well written and immediately useful in both theory and practice.

Noah Edelstein
VP of Product Management, Smartsheet

The most frustrating projects in my career have been updating systems where the initial architecture grew without deliberation or documentation. Oliver Goldman's *Effective Software Architecture* dives into why thinking clearly about software architecture is so important and offers tools to do so. I wish everybody thought so deeply about architecture!

Andrew Certain
Distinguished Engineer, Amazon Web Services

This page intentionally left blank

Effective Software Architecture

Effective Software Architecture

Building Better Software Faster

Oliver Goldman

Addison-Wesley

Cover Credit: ArtHead/Shutterstock

Library of Congress Control Number: 2024934795

ISBN-13: 978-0-13-824932-8
ISBN-10: 0-13-824932-6

$PrintCode

To Gloria, for her love and partnership

This page intentionally left blank

Contents

Acknowledgments

This book contains the accumulation of knowledge and experience gained over decades of education and work. Far more people have influenced and informed those years than I can enumerate here. There are, however, some key players who should not go unmentioned.

I'll begin with my parents, Bernadine and Terry, and my siblings—Elizabeth, Leah, and Matthew. My parents purchased a Commodore 64 when I was nine and it's easy to trace my career in software back to its arrival in our family room. My childhood home was also full of books, thinking, and lots of word play. Somewhere in there I got the notion that I'd like to see my name on a book one day. Now the Commodore 64 has helped me realize that goal, too.

In high school, two excellent English teachers, Jeff Laing and Rick Thalman, taught me that I can write—and how to do it. I am forever grateful for their feedback and encouragement. And a big thank you to Tom Laeser, who gave me free reign in the computer lab.

At college, I was fortunate to have Mendel Rosenblum as my advisor, as well as the instructor for my operating system classes—those were my favorites. During the summers and for a time after college I also had the privilege of working for George Zweig. George put a lot of trust in a much younger, more impetuous version of me. To this day, I look back fondly at those early experiences.

I spent the bulk of my career at Adobe, with so many amazing colleagues that it's unfair to mention just a few and impossible to list them all. Nonetheless, I feel obliged to call out Winston Hendrickson and Abhay Parasnis as the two managers who gave me the biggest opportunities; I hope I have lived up to their expectations. Boris Prüßmann, Dan Foygel, Leonard Rosenthol, Roey Horns, and Stan Switzer were inspiring collaborators on all manner of projects, including the development and refinement of many of the ideas in this text.

Brett Adam, Dan Foygel, Kevin Stewart, and Roey Horns all reviewed early drafts of this book and provided valuable feedback. To Manjula Anaskar, Haze Humbert, Menka Mehta, Mary Roth, Jayaprakash P., and others behind the scenes at Pearson: Thank you for taking a chance with a new author and your guidance along the way. You have helped me achieve one of my lifelong goals and I am deeply grateful.

Most of all, I want to thank my wife, Gloria, and our four boys. This book has been written from the comfort of the home we've made together and I wouldn't have it any other way.

About the Author

Oliver Goldman leads the AEC software architecture practice at Autodesk. He has thirty years of industry experience delivering innovative products in distributed real-time interaction, scientific computing, financial systems, mobile application development, and cloud architecture at Adobe and other employers. He holds two degrees in computer science from Stanford University, is an inventor on over 50 US software patents, and has previously contributed to *Dr. Dobb's Journal*.

Introduction

When I graduated from college with a degree in computer science, it's fair to say I had a layperson's understanding of the science and theory of creating software. I had studied databases, algorithms, compilers, graphics, CPU architecture, operating systems, concurrency, and more. And I had, to some extent, a framework that related these technologies to each other.

Because I was writing software outside of classes—mostly for summer jobs—I also knew that translating academic knowledge into building a product was a distinct challenge. More accurately, it presented challenges, plural. Selecting and implementing the right algorithm was generally the easy part. Working with a large code base, creating a functional user experience, testing for quality and performance, and coordinating with a whole team operating in parallel on the same product—these were hard things to do.

Upon graduation, I worked at a series of jobs developing software products. Most of them did not result in successful products, but that didn't stop me from learning from those experiences. If the cliches about learning from failure are correct, I learned a lot during that stretch of time.

Over the course of several different projects, I noticed that I often had a better view of the product as a system—that is, of the components within it and how they relate—than most of my peers. Although I didn't appreciate it at the time, this ability to see and reason about the "big picture" is a relatively rare but quite useful skill set.

To comprehensively address all the components of a software system and how they relate is to practice architecture. Obviously, the term *architecture* is not unique to software—indeed, it's borrowed from the realm of buildings, and can be applied to products of all types. Houses have an architecture, and so do cars, TVs, and rockets. Had I grown up to be a rocket scientist, I expect I'd have found myself more interested in how all the different parts of a rocket fit together than the design of a specific valve or nozzle. But we'll never know for sure, given that I ended up in the software field.

Over my decades in the industry, the complexity of the software we produce has grown immensely. When I started, a viable software product—that is, something that could be sold for a profit—fit on a floppy disk, ran on one machine at a time, and didn't connect to the Internet. Today, a software product running "in the cloud" might easily consist of hundreds of coordinating programs running at multiple, geographically distributed sites, updated multiple times a day, and expected to operate forever without interruption (an expectation of which we admittedly sometimes fall short). The nature of a software product has evolved radically in a short time.

This evolution has made software architecture both more difficult and more critical than ever. More difficult because there are orders of magnitude more components and relationships to keep track of. More critical because, unless these relationships are managed effectively, the complexity of a system will inevitably become a limiting factor on its dependability and the velocity of further development. For most products, this is the beginning of the end. And I've seen it happen.

Software architecture isn't just about managing complexity, but if I had to pick the most valuable outcome of architecture as a discipline, that would be it. Complexity undermines everything that software should do for us. It creates unpredictable behavior, and thus undermines user trust. It leads to defects, harming dependability. It propagates failures, turning small errors into large outages. And it hampers understanding, eventually defeating any attempt to return to a simpler state or structure. Complexity is software's enemy, and a disciplined architecture practice its best defense.

Later in my career, I had the privilege and responsibility of leading architecture teams for a couple of large, complex software products. Neither product was entirely new; both had been around for more than a decade. I began by doing the sorts of things that are part of any architect's job description: understanding the system's current architecture, assessing its suitability for the current and anticipated requirements, and proposing and evaluating changes. I'll have more to say about how to do these things later in the book.

Although those activities are clearly necessary, equating them with software architecture is like saying I knew how to write software because I'd taken some computer science classes. It's a great start, but there's a whole lot more to making software architecture an integral and successful part of software development. And that's what this book is about: how to practice architecture in a software development organization.

Focus

This is not a book of software architectures. There are no explanations of client–server, domain-driven design, sense–compute–control, and other architectural styles. I do not discuss how to select a database technology, regionalize your deployments, or design for scale. Those are all important topics. There are many books, blogs, and other resources about them; there are many architects well versed in these topics.

But just as knowing how to implement a merge sort algorithm falls short of what's required to write an application, simple familiarity with a given architecture is woefully inadequate to create a system that uses it. And while adopting that merge sort might be the scope of a task that falls to a single engineer, system architecture inevitably involves a larger cast of characters.

This book aims to describe how you can take your architectural skills and knowledge and apply them to the much larger, much messier process of developing a product. Without focusing on a specific style, it defines software architecture, placing it and defining its role among the other specializations of a product development team. It identifies the many touchpoints that architecture has with the concepts, processes, standards, and so on that surround it.

We then dive deeply into the topic of change. Identifying, managing, and designing changes to a system are core to architecture practice. Design sometimes seems like a black box, with conversations going in one side and a complete design popping out the other. In reality, the whole process of change is ongoing and consists of discrete steps.

Everything we can do to make those steps apparent, to make them visible, and to steward them along this path will improve the process.

Engineering is about trade-offs, and the process of developing and evolving a system through change involves an endless litany of design decisions. Each decision opens some pathways and closes others—or perhaps it reverses an earlier decision when we discover a path has come to a dead end. How these decisions are made is itself a key skill. The more good decisions a project team can make, the less time is spent backing up. And the more quickly good decisions can be made, the more quickly the project can move ahead.

On projects of any significant size, logistics and communication also become critical considerations. Which decisions have been made, and which ones are pending? What's the vocabulary we use to describe our system? Why did we select the architecture that we're using? All these become concerns of tools, process, and communication.

Finally, we consider the architecture team in an organizational context, including the definition of software architect as a discrete role. Options for structuring an architecture team are considered, as well as how architects should engage with other disciplines within the organization. This section also considers how to identify, nurture, and develop architectural talent.

Motivation

Software grows ever more complex. We've grown accustomed to products that put the information and tools we want to use at our fingertips across multiple devices, anywhere on the globe, while supporting billions of users. The challenges involved in creating and operating these systems are far beyond the simple, stand-alone software products of just a few decades ago.

Software architecture plays a unique and critical role in this work. While only one of many disciplines that work together to conceive, realize, and run these massive systems, it uniquely demands an ability to see the "big picture," to understand how all the elements of a system come together, and to evolve that structure over time. Over the last two decades or so, architects have made great strides in developing technologies and techniques to meet these challenges. The better an organization is at doing software architecture, the better it will be at delivering quality software on time.

Even so, most product development organizations are not as good at doing software architecture as they could be. This observation crystallized for me one day when I took on a new role running a team of experienced architects on a project that was new to me. At the individual level, these architects were all adept at handling the challenges of designing software. However, they weren't packaging their skills to benefit the larger effort. They were underinvested in documentation, process, and communication.

As a result, that architecture team was underperforming. They struggled to prioritize their work, and sometimes engaged with the wrong problems. Without a strong decision-making process, they struggled to make decisions—and to keep them made. They were inconsistent in documenting their work, which sometimes led to work being ignored or redone. The project was complex and important; it warranted a significant architectural investment. But the team, despite having, collectively, more than a hundred years of architectural experience, was letting the organization down.

When I spoke with the members of my new team, I realized that they could see the symptoms—they knew they were struggling—but they could not identify the cause. They knew, individually, how to do software architecture. But collectively, they did not understand how to run an effective software architecture practice. They lacked the structure necessary to knit their individual efforts into a team effort and to integrate that work into the larger organization.

That experience led directly to this book. If these architects, with their decades of experience, didn't understand how to structure an effective architectural practice, then it was likely that many others struggled with the same issues. And while the endeavor of running an architecture team is not entirely ignored in software literature, it is also not a topic with extensive coverage. For example, *Software Architecture: Foundations, Theory, and Practice* (Taylor, Medvidovic, and Dashofy 2010) devotes 3% of its 675 pages to "People, Roles, and

Teams." I have a reasonably extensive library of books on software architecture; this is about par for the course. I decided to fill that gap.

Audience

This book is for software architects, the managers who lead them, and their counterparts in product management, user experience, program management, and other related disciplines. Building software is a joint space in which all these disciplines cooperate. I hope they will all benefit from these explanations of software architecture as a discipline, its role in software development, and how architects and architecture teams operate.

Practicing architects will find guidance that they can compare to their own methods. No matter how numerous their years of experience, they may yet find something new here. Software architecture is too young a field to have a broadly known corpus and consistent or regulated practice.

This book is also for everyone who works with a software architecture team. As projects expand, roles become differentiated: Product managers focus on requirements, testing teams create test plans, security teams develop threat models. Everyone has their own area of expertise. And yet all of those activities must be stitched back together into a cohesive whole—and that requires that everyone see how these functions fit together. In short, they must understand the system's architecture. In this book, everyone involved in software projects will find an accessible description of the role software architecture plays in achieving their goals.

Finally, this book is for the executives responsible for managing or creating an architecture team. In explaining how software architecture works, it provides the background that executives need to determine if their current architectural function is fit-for-purpose and what to look for when hiring new talent.

Success

An effective software architecture function helps product development organizations produce better software faster. As a discipline, software architecture tackles some of the hardest parts of producing software: organizing each system, managing change and complexity, designing for efficiency and dependability. Software systems with good architecture work well and continue to work well over time. Systems with poor architecture fail—often in spectacular fashion.

A successful software architecture practice also integrates these abilities with the broader challenges of product development. Architects are uniquely positioned to aggregate requirements, and thus design a cohesive whole instead of a collection of parts. And thanks to that same overarching view, they are also well positioned to communicate to everyone how those pieces come together.

To do that well requires more than a degree in computer science, and more than experience with relevant architectural styles. It requires the ability to create a predictable, repeatable change process; to make decisions expediently and effectively; and to build a team that can do these things ever better over time.

Simply put, software architecture has an ever-increasing impact on our ability to develop and deliver fit-for-purpose software. I hope this book will help guide you and your organization to a more effective software architecture practice.

Register your copy of ***Effective Software Architecture: Building Better Software Faster*** on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (**9780138249328)** and click Submit.

Chapter 1
Software Architecture

An effective software architecture practice helps product development organizations produce better software faster. But before we can discuss an effective practice, we need an understanding of software architecture. It's a term that's used frequently in the software industry, and often with a certain laxness. Tightening up our definition is important, as the practices in this book are closely aligned with a strict and complete definition of architecture.

Software architecture is often equated with software design, but the two are actually quite distinct. A *design* is a specific, point-in-time arrangement of software components that, collectively, form a software system. As we develop each release of that system and determine how that release will function, we are *designing* that release.

What happens when we create the next iteration of that same system? We'll revise its design, of course: That's how we introduce the changes that distinguish one release from the next. But we won't throw it out and start over, either; each subsequent design is related to the one that came before.

An *architecture* is a template for the iterative creation of a set of related designs. Architectures are also designed, but they are more than *a design*. Thus, an effective software architecture practice doesn't just create one good design; it lays the foundation for creating tens, hundreds, and thousands of designs. That's the potential of software architecture, and that's the promise of an effective software architecture practice.

What, then, constitutes an architecture? Standards play an important part in architecture, so it feels appropriate to start our discussion of architecture with a definition from an IEEE standard:

> *[An architecture is] the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [1]*

Let's take this definition apart, piece by piece, to understand it thoroughly.

Fundamental Organization

Imagine that you have a software product consisting of a hundred components. The precise nature of these components doesn't matter; they could be services, libraries, containers, functions, or plugins. The point is that your product is composed of these components, and the interactions between these components realize the features and functions of the product.

Now imagine that for each component, you used a random number generator to determine its type (service, library, etc.) and its method of communication. Often these are linked—for example, a code library is designed to be invoked via a local procedure call, and a service is not. That's okay; we'll pick, randomly, from the methods that reasonably apply to each component.

You've probably already realized that getting these components to work together will be a challenge. Different components require different implementation technologies, tooling, and deployment.

We'll have lots of mismatches when we try to wire components up, and we'll have to translate between local calls and remote calls and message-passing and function invocation. In starting with randomness, we've constructed a system that lacks any fundamental organization. Fortunately, this system exists only in our imagination.

No one works this way, and every real system has some fundamental organization to it. The fundamental organization of a system often is, to some extent, imposed by external factors. For example, if you're building a mobile application, your elements will primarily be libraries and they'll mostly communicate via local procedure calls. In contrast, if you're building a cloud-based product, you might organize your system around services.

When speaking of the architecture of a system, however, we are generally referring to a fundamental organization that goes beyond these external constraints. For example, cloud services necessarily communicate via the network. Is that communication organized as request-response or message-passing? Any system that chooses one method over the other is fundamentally organized around that specific approach.

Figure 1.1 illustrates the impact of each approach to establishing a system's fundamental organization. As illustrated in the leftmost diagram, a randomized system consists of different types of components (indicated by different shapes) communicating via different mechanisms (indicated by different line types), and with arbitrary relationships. External constraints tend to dictate component types and communication mechanisms, as indicated by the uniformity of shapes and line types in the middle diagram. However, external constraints rarely impose organization on relationships within the system. The final diagram, on the right side of Figure 1.1, illustrates a system with a clear fundamental organization. It uses a consistent component type, consistent communication mechanism, and has structured relationships.

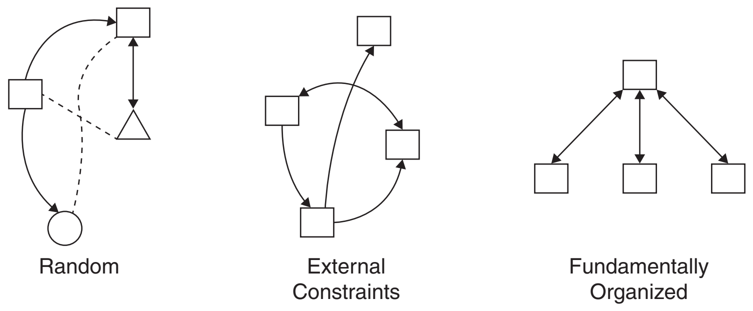

Figure 1.1
Levels of fundamental organization. A random system (left) has a mix of component and relationship types. Most systems are subject to at least some external constraints (middle) on those types. Organized systems (right) bring additional consistency through further constraints.

Of a System

We're going to use the word "system" a lot in this book. The term appears in our definition of architecture and has already been mentioned half a dozen times in this chapter. But what is a system?

For our purposes, a *system* is any set of software components that work together to provide one or more capabilities. Systems can be large: They might contain hundreds or thousands of components and execute on a similar number of computers. But they can also be small: The embedded software running on a battery-powered wireless sensor is also a system.

Systems need not work in isolation. If you are developing wireless sensor software, then, for your purposes, the boundaries of your system can be set according to which software runs on the sensor. That sensor—along with others—will send data to some *other* system that processes that data. For you, the processing system may form part of your environment, not your system; it may, for example, be developed by a different team.

Systems may also be composed of other systems. In other words, a new system can be defined as the composition of two or more smaller systems, perhaps with some additional components added to the mix. For example, a system of wireless sensors, combined with a system for data processing, can be composed into a single system providing a monitoring capability.

Thus, when we use the term *system* regarding architecture, we're allowing for the system boundaries to be set as befits the relevant scope of concern. Every aspect of software architecture covered by this book applies to any of these systems, regardless of its scale. Granted, some concerns are more readily addressed at smaller scales, so the extent and rigor with which you apply architectural practice can be adjusted according to the needs of the system at hand.

Embodied in Its Components

The fundamental organization of a system isn't easy to change. The implied decisions regarding technology, communication, structure, and so on become embodied in the components that conform to that organization. After all, it's the implementation of this organization in the components that gives the whole system form.

It is all too easy to underestimate how deep these decisions run. As desktop computing gave way to mobile- and cloud-based solutions, companies with deep investments in desktop code looked for options to bring their massive investments in these code bases into these new forms.

Initially, that challenge might have looked like a porting problem. Maybe the code needed to run on a new CPU architecture or adapt to a different operating system. Those changes are not necessarily easy to make, but they're not impossible. Furthermore, many of these code bases had already been through at least one similar port in the past, such as from Windows to macOS.

But the fundamental organization of most desktop applications includes much more than just a CPU instruction set or even an operating system. For example, most of these desktop applications are organized around the assumption that they have access to a fast, reliable, local disk. This point is so foundational that many desktop application architects wouldn't have even called it out. There was no other option to consider; it could go unstated.

As a result, in many of these applications, the assumption of fast and reliable access to a disk is embodied not just in every component but in every line of code in those components. Need to read some configuration data? User preferences? Save progress? No problem! A couple of calls to the filesystem API, problem solved.

Moving this data to the cloud breaks this assumption, and every line of code that depends on it. The data might exist, but retrieving it might be slow (because it's over the network) and unreliable (because it's over the network). Or it might be impossible right now (because the network is down), although later it will be possible again.

Figure 1.2 illustrates how these assumptions become embodied in a system's components. On the left, components in a desktop application connect directly and independently to a file system; they assume and depend on fast and immediate access to their data.

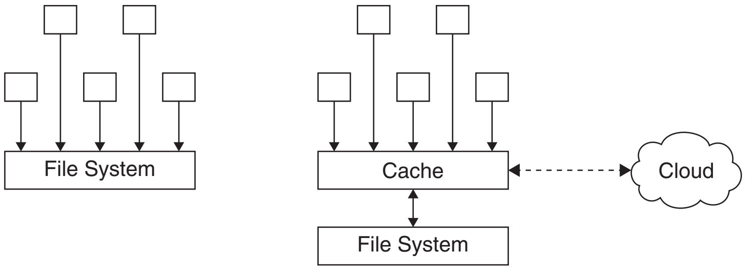

Figure 1.2
The fundamental organization of a system is embodied in its components. On the left, the organization of the system around a file system is embodied in every component. On the right, that's been shifted to a cache that mediates access to data.

Components can be written to deal with this uncertainty, but they must embody a different fundamental organization. They must be aware that data has a source that may be slow to access or even inaccessible. As a result, they tend to be organized around a local cache instead, and a good deal of time and attention go into managing how data moves between the cache and storage.

On the right side of Figure 1.2, an alternative organization binds components to a cache that, in turn, mediates between local and cloud storage. In this architecture, components also embody the assumption that their data may or may not be in the cache. When a cache miss occurs, accessing their data will be either slow (it's retrieved over the network) or impossible (if the network is down).

Organizing around a filesystem abstraction doesn't really help resolve this problem. It's not too hard to build a filesystem abstraction that can bridge the differences between different desktop operating systems, or even between mobile and desktop operating systems. But it's the wrong abstraction for data in the cloud because it continues to assume fast, local access. These foundational assumptions can be just as easily embodied in interfaces and abstractions as code.

To be clear, how an architecture's foundational organization is embodied in its components is about much more than storage and file systems. This is one example of how an architecture lays down assumptions, and those assumptions can become wired into each line of code in each component. We'll return to this point more than once, as it's a key aspect of what makes architecture valuable—and difficult.

Their Relationships to Each Other

As programmers, we tend to place an emphasis on components over connections. The components feel material—as much as anything in software ever does—because they consist of the code we write. Components can be compiled, packaged, distributed, and delivered. They feel almost tangible.

But components aren't interesting on their own. Software comes to life when those components are connected to each other in meaningful ways. How and which connections are formed should therefore be intentional, not happenstance.

Some well-known architectures place relationships front and center. The Unix shell architecture, for example, consists of two primitives: programs and streams. Streams are directional; they have an input side and an output side. Programs read zero or more inputs and write zero or more outputs. The shell's job is to link outputs and inputs, from one program to the next, forming pipelines through which data flows.

The Unix architecture doesn't have much to say about how these programs operate. They can be written in different languages, handle binary data or textual data, and so on. Most of the programs are relatively small, focused on a single job. (The emphasis on small, narrow programs is a Unix architectural principle—more on principles in a moment.)

The relationships between the programs get more attention. By default, every program has one input stream (stdin) and two output streams. The output streams are divided into the "standard" output (stdout) and a special stream for errors (stderr). It's possible but somewhat tedious to deal with more streams, whether for input or output.

The beauty of this approach is that it's simple yet powerful. Programs written at different times by different authors can be readily combined by the user to achieve new and unexpected outcomes. And it's possible not because of constraints on the programs, but rather because of their relationships with each other.

This result, in which components are combined post-development to achieve a new result, is an example of network effects. Network effects are exciting because they produce a combinatorial explosion of value for linear inputs. And while this book isn't really about platforms and network effects, there is a deep connection between platforms, network effects, and architecture.

However, the combinatorial magic of connections can also work against an architecture. In the Unix model, programs can be combined—but they do not intrinsically depend on each other. When a system contains many interconnected components that depend on each other, these relationships become a hindrance, not a help.

When relationships are not governed, the dependency count tends to grow. And while these dependencies may be introduced one at a time, the complexity of the resulting system can grow much more quickly. It can quickly grow beyond anyone's ability to comprehend, let alone manage.

If you have ever worked on a system that had components that couldn't be touched for fear of breaking some other component, then you've worked on a system where the relationships between components have become too entangled. These scenarios demonstrate how managing relationships is just as fundamental as managing the components themselves.

Their Relationships to the Environment

Systems never operate in a vacuum. In some cases, they may be the only software running on the hardware at hand, in which case that hardware is their primary environmental concern. However, most of the time, systems run on top of or as part of some other system.

For example, consider the relationship between a program (a system) and the operating system on which it runs (also a system). Operating systems *impose* a fundamental organization on the programs they host. This organization is inescapable: The programs are started by the operating system and, to a lesser or greater extent, the operating system monitors and controls the program during execution and through termination. Without some basic agreement between the two, these programs would never run.

Operating systems vary immensely in regard to how much of the fundamental organization of a program they define. In the Unix model, for example, the imposed organization is quite limited. Programs are started by invoking a function with a well-known name ("main") and set of arguments, so those elements must exist. To be sure, the typical structure of Unix programs encompasses many conventions and APIs. But very little of that is truly required. As a result, Unix successfully and easily supports writing programs of many types, languages, and structures.

The iOS platform, by comparison, is much more opinionated. iOS applications do not have a single entry point but rather a whole set of functions they are expected to respond to. Much of this has to do with the life cycle of these applications. In the Unix model, programs start, run until they're done with the task at hand, and then exit. On iOS, applications are started, brought to the foreground, moved to the background, stopped to preserve resources, restarted due to user interactions or notifications, and so on. It's much more complicated!

On Unix, you can use the fundamental organization of an iOS program if you want to. Again, Unix doesn't impose its organization; the program designer has significant discretion. But on iOS, your application largely needs to be fundamentally organized around a model dictated by iOS. This relationship of the program to the iOS environment becomes a major driver of the program's architecture.

Relationships to Multiple Environments

More opinionated environments create tension with code reuse. Building complex applications is expensive, and many software producers would like to write a system once and use it in multiple environments. From an architect's perspective, the problem of managing the relationship to the systems' environment becomes one of managing the relationship to multiple environments.

This can be difficult to accomplish when environments impose different—and in the worst case, conflicting—fundamental organizations. There are a few standard ways in which this issue is tackled:

- Ignore the environment and organize the software in some other way. As a rule, it's expensive to develop systems this way because a good deal of time and effort go into reproducing behaviors that you could get "for free" from the environment. Furthermore, those reproductions are never perfect, and differences tend to look like defects to your users.
- Create an abstraction layer that adapts two or more environments to a single model. This strategy can work well for systems that don't need deep integration with capabilities provided by the environment. For example, it often works well with games. The abstraction layer might be part of the system, or it might be externalized. These layers are sometimes products themselves.
- Split the system into two subsystems: an "environment-specific" core that is written separately for each target, and an "environment-agnostic" edge that is shared across each. While this may sound superficially

> like the abstraction layer approach, it is a discrete strategy because the environment capabilities are not abstracted. In this case, the environment-specific layer becomes as deep as necessary (but, ideally, no more than that) to connect with the environment-agnostic logic.
>
> As with engineering in general, there's no universal correct answer to the challenge of managing these environmental relationships.

Principles Governing Its Design

Until this point, the IEEE definition of architecture has focused on describing a system's current state: its fundamental organization, its components, their relationships. Now, it turns its attention to *why* it's organized this way, has these components, and includes these relationships.

Design is a decision-making activity in which each choice determines some aspect of form and function. Principles are rules or beliefs that guide decisions. Thus, architectural principles are the rules and beliefs that guide decisions about a system, helping determine its fundamental organization.

Good architectural principles assert what's important to the system—perhaps reliability, security, scalability, and so on—and guide design toward those qualities. For example, a principle might assert that a notification delivery system should favor speed over reliability. In turn, that can support a decision to use a faster but less-reliable message-passing technology.

Principles can also accelerate decision making by designating a preferred approach out of many available options. For example, a principle might state that horizontal scaling is preferred over vertical scaling. We may be tempted to think of software engineering as driven purely by facts and analysis—that is, a system either does or does not meet its requirements—but often many designs may meet the requirements. Thus, design also involves judgment calls in which we choose among a set of acceptable alternatives.

Importantly, principles govern not just what we can do, but also what we cannot do. For example, consider a system composed of a set of services. A reasonable principle for the design of this system might hold that each service should be individually deployable without downtime. On the one hand, this principle frees architects for individual services, allowing them to determine when and how they will deploy updates. On the other hand, it constrains them: Their deployment strategy must not require that other services are updated at the same time, or that those services are temporarily disabled during the update. Thus, this principle simultaneously opens up some options for consideration while closing off others.

Without constraints, teams may spend too much time exploring a potential design space, which tends to slow down decision making while producing only marginally better outcomes. It's a classic case of diminishing returns: Each new option takes significant time to explore, yet doesn't deliver meaningful benefits over options already considered. A constraint that limits this exploration will save substantial time and effort while producing just as good a result. Of course, for this approach to work, the constrained designed space must encompass suitable outcomes—so these principles do need to be carefully considered.

Too much freedom in choice also tends to result in a lack of alignment, thus undermining the system's fundamental organization. Schematically, suppose that three or four subsystems need to choose between approach A and approach B. If both approaches are roughly equivalent and no further guidance is available, the system is likely to end up with a mix of the two approaches. Because that outcome increases overall system complexity with a corresponding benefit, it's a net-negative and should be avoided. A principle that guides all of these decisions toward the same option speeds decision making and constrains the outcome, driving simplicity and alignment.

When we're not intentional about setting our principles, they tend to be set for us. And that means that we typically end up with implied principles such as "working within current organization boundaries," "minimizing scope," and "fastest time to market." None of these principles speaks to developing a better product, and they won't produce one. Establishing and adhering to principles developed through an intentional process is one of the more impactful activities an architect can undertake.

Architecture versus Design

No part of the IEEE definition better illuminates the difference between architecture and design than its assertion that *principles govern design*.

Any reasonably complex system contains hundreds or thousands of designs. These are typically arranged hierarchically: a high-level design for the system, somewhat more detailed designs for subsystems, and so on. Services, libraries, interfaces, classes, schemas—each of these requires a design.

These designs relate to each other because they describe system elements that must work together. When we consider the hierarchy of a system, we often approach design in a top-down manner for this reason. First, we'll create subsystems, defining the boundaries and their basic behaviors. Then, within each subsystem, we'll divide it further.

Each of those items (services, etc.) requires its own design, but they are not created in a vacuum. They must fit within the larger design of the subsystem. They must cooperate with their peers to realize the subsystem. Finally, they must articulate their own interior structure.

Of course, these designs don't all pop into existence, fully formed, at the same moment in time. We work in teams, so designs proceed in parallel. We expand, update, and revise so new designs are created, and existing designs are revised, over time. Unless we've mothballed a system, design is an ongoing process.

Architecture manages those designs over time. To do that, architects must think about more than just the designs. They must establish governing principles such that those myriad designs will come together into a coherent whole, both at each point in time and over the course of time.

This is not to say that architects shouldn't do design. Establishing principles for a project does no good if those principles can't be realized in the designs, and the only way this can be known is if we put them into practice. But an architect can't spend all their time on design because then they won't be practicing architecture.

And Evolution

The systems we build aren't static. Release cadences vary, but any successful system must evolve to stay relevant. This may occur through the accumulation of small changes, less-frequent major changes, or some combination of the two. But it must happen.

And so also everything we've described so far—the organization of a system, its components and their relationships, its principles and designs—must also evolve. How does that happen?

Ideally, that evolution will be governed with intention, by a set of carefully considered, fit-for-purpose principles. We can understand architectural principles as working in two ways: governing design and governing the evolution of design. One set of principles; two methods of operation.

For example, we might hold a principle that services should be loosely coupled via well-defined interfaces (a commonly adopted principle for anyone designing cloud software). This is entirely reasonable, and quite actionable when designing those services.

However, it tells us little about the evolution of those services. As we add functionality and even new services, we can maintain this property, and that acts as a sort of basic constraint on their evolution. But it doesn't address critical questions such as how to modify an existing interface, when to add functionality to existing services, or when to create a new service.

Adding new functionality to an existing system is one of the easiest forms of system evolution, and yet it can still go wrong. For example, suppose that two teams are both expanding their services with new functions. One team might choose to add a new service, favoring a larger set of smaller services. The other team might add new functions to an existing service.

When scenarios like this occur, the evolution of the system is proceeding in such a way as to undermine its fundamental organization. The problem here isn't that one approach was right and the other wrong. In the abstract, either approach could be an appropriate response to the need to expand the system's functionality. The damage arises from the differing responses, which make the system unnecessarily more complex. The principles governing the evolution of a system should head this off by articulating which approach will be used.

Adding new functionality is perhaps the easiest version of the evolutionary problem. A much harder version occurs when the principles themselves need to change. For example, perhaps an earlier principle focused on speed of delivery, so it favored designs that added code to a single, monolithic service. Later, the team might have adopted a principle of developing smaller, more loosely coupled services that can be independently updated and deployed.

Often this problem arises when teams shift from unstated design principles focused on minimizing change and quick delivery to intentional principles concerned with dependability, maintainability, and quality. When this happens, it's not enough to apply the new principles to new and updated designs. Properties such as security, scalability, and cost tend to be constrained by their weakest link—that is, the components that don't address these concerns. At the limit, to realize these kinds of principles requires every existing design to be reworked.

This is one of the hardest problems to solve in software development: how to evolve a system from one set of principles to another. At the same time, it's also one of the most common problems because the principles we care about when we're getting the first version or two out the door are often very different from what we care about once we have a successful, proven product. Our priorities quite naturally shift from shipping quickly to creating a quality, sustainable product.

One response to this challenge is a grand "re-architecture" project in which every element of the system is rebuilt to align with these new priorities. But this approach is not evolutionary—it's revolutionary. It's also rarely successful because it requires development teams to make a sustained investment in the old and the new at the same time. Add in the overhead of running both efforts, and you've easily tripled the cost of ongoing development. Few teams can sustain this kind of investment.

The good news is that an effective software architecture practice can address even this most challenging scenario. An effective software architecture team can lay out an evolutionary path for change. The key is to understand evolution not as something foisted on a system under duress, but rather as the natural state of the system. An effective architectural process makes change intrinsic, predictable, and controllable. Ultimately, the ability to govern the evolution of a system is the essence of architecture's role.

Summary

Software systems consist of components and their relationships. A system's architecture is the organization of its components and relationships, along with the principles that govern their design and evolution. An architecture describes a system in both its current and future states.

When a system's organization is not governed, decisions tend to be driven by external factors. Often these involve respecting reporting structures, minimizing the scope of change, and shipping quickly. These can be important factors, but they can work against creating a system with a clear fundamental organization.

We can better manage a system's fundamental organization by applying principles to its design and evolution. These architectural principles don't need to replace other factors—shipping quicky may still be important—but they need to be part of the conversation. Software architecture, like any engineering discipline, involves tradeoffs between competing goals.

Evolution is intrinsic to software architecture. Teams that use architectural principles to drive change over time—including adopting new principles when appropriate—can evolve their systems to deliver new capabilities and improve the security, dependability, maintainability, and other aspects of their system.

Ultimately, architecture's role in software development is to take a holistic and intentional view of the system. It begins by identifying the fundamental organization of a system, describing its components and relationships. It sets principles that govern design to realize that fundamental organization. Most of all, it's about establishing the principles via which those designs, components, and relationships will change over time.

This page intentionally left blank

Index