Chapter 2. Architectural Thinking

Architectural thinking is about seeing things with an architect's eye—in other words, from an architectural point of view. Understanding how a particular change might impact overall scalability, paying attention to how different parts of a system interact, and knowing which third-party libraries and frameworks would be most appropriate for a given situation are all examples of thinking architecturally.

Being able to think like an architect first involves understanding what software architecture is and the differences between architecture and design. It then involves having a wide breadth of knowledge to see solutions and possibilities that others do not see; understanding the importance of business drivers and how they translate to architectural concerns; and understanding, analyzing, and reconciling trade-offs between various solutions and technologies.

In this chapter, we explore these aspects of thinking like an architect.

Architecture Versus Design

Take a moment and picture your dream house in your mind. How many floors does it have? Is the roof flat or peaked? Is it a big sprawling, single-story ranch house or a multistory contemporary house? How many bedrooms does it have? All of these things define the overall *structure* of the house—in other words, its architecture. Now take a moment to picture the inside of the house. Does it have carpeting or hardwood floors? What color are the walls? Are there floor lamps or lights hanging from the ceiling? All of these things relate to the *design* of the house.

Similarly, software architecture is less about a system's appearance and more about its structure, whereas design is more about a system's appearance and less about its structure. For example, the choice to use microservices defines the structure and shape of the system (its architecture), whereas the look and feel of the user interface (UI) screen define the design of the system.

But what about decisions such as whether to break apart a service into smaller parts or deciding on a UI framework? Unfortunately, most decisions such as these fall somewhere on a *spectrum* between architecture and design, making it difficult to determine what should be considered architecture.

Leveraging the following criteria can help determine whether something is more about architecture or more about design:

- Is it more strategic in nature or more tactical?
- How much effort will it take to change or construct?
- How significant are the trade-offs?

These factors are shown in Figure 2-1, illustrating the spectrum between architecture and design to help determine where a decision lies and who should

have the responsibility for it.

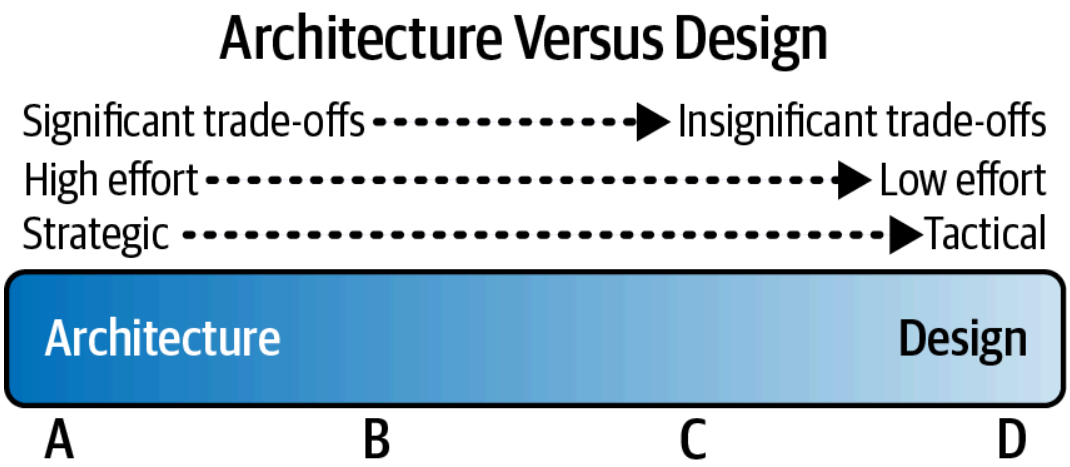

Figure 2-1. The spectrum between architecture and design

Strategic Versus Tactical Decisions

The more strategic a decision, the more architectural it becomes. Conversely, the more tactical a decision, the more it's likely to be about design. *Strategic* decisions are generally long-term, whereas *tactical* decisions are generally short-term and usually independent of other actions or decisions.

One good way to determine whether a decision is more strategic or tactical is to consider the following questions:

How much thought and planning is involved in the decision?

> A decision that takes a couple of minutes is more likely to be tactical and hence more about design, whereas a decision requiring weeks of planning is likely to be more strategic and hence more about architecture.

How many people are involved in the decision?

> A decision made alone or with a colleague is likely more tactical and on the design side of the spectrum, whereas a decision requiring lots of meetings with many different stakeholders is likely more strategic and on the architectural side of the spectrum.

Is the decision a long-term vision or a short-term action?

> A decision that is likely to change soon is usually tactical in nature and would be more about design, whereas one that will last a very long time is typically more strategic and more about architecture.

While these questions are a bit subjective, they nevertheless help in determining whether something is strategic or tactical, and thus more about architecture or design.

Level of Effort

In his famous paper "Who Needs an Architect?", software architect Martin Fowler writes that architecture is "the stuff that's hard to change." The harder something is to change, generally, the more effort is required, placing that decision or activity toward the architectural side of the spectrum. Conversely, something that requires

minimal effort to implement or change places it more on the design side of the spectrum.

For example, moving from a monolithic layered architecture to microservices would require significant effort, so it would be more about architecture. Rearranging fields on a screen would require minimal effort, and therefore be more about design.

The Significance of Trade-Offs

Analyzing the trade-offs of a particular decision can help a lot in determining whether it is more about architecture or design. The more significant the trade-offs are, the more architectural the decision tends to be. For example, choosing to use the microservices architecture style provides better scalability, agility, elasticity, and fault tolerance. However, this architecture is highly complex, is very expensive, has poor data consistency, and doesn't perform well due to service coupling. These are some pretty significant trade-offs. We can conclude that this decision is more on the architectural side of the spectrum than design.

Even design decisions have trade-offs. For example, breaking apart a class file provides better maintainability and readability—at the cost of managing more classes. These trade-offs are not overly significant (especially compared to microservices' trade-offs), so this decision is more on the design side of the spectrum.

Technical Breadth

Unlike developers, who must have a significant amount of *technical depth* to perform their jobs, software architects must have a significant amount of *technical breadth* to see things from an architectural point of view. Technical depth is all about having deep knowledge of a particular programming language, platform, framework, product, and so on, whereas technical breadth is all about knowing a little bit about a lot of things.

To better understand the difference, consider the knowledge pyramid shown in Figure 2-2. It encapsulates all the technical knowledge in the world, which can be broken down into three levels: *stuff you know*, *stuff you know you don't know*, and *stuff you don't know you don't know*.

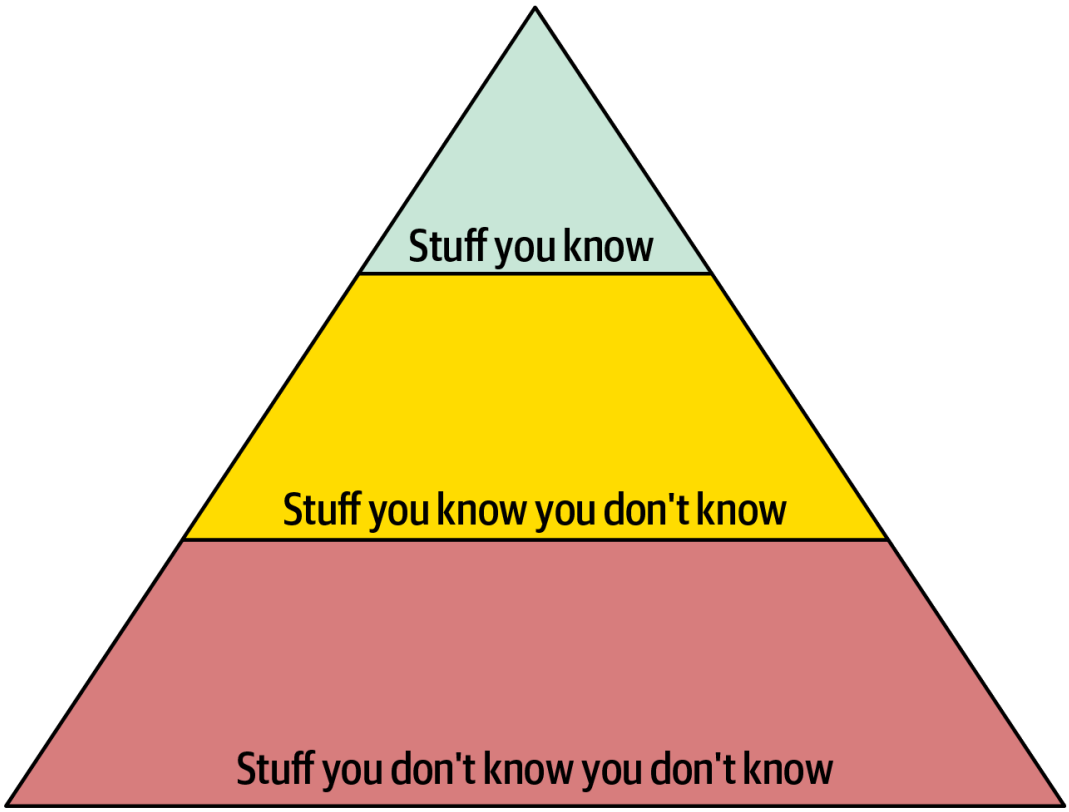

Figure 2-2. The pyramid representing all knowledge

Stuff you know includes the technologies, frameworks, languages, and tools technologists use on a daily basis to perform their jobs (such as a Java programmer knowing Java). They're good, or even expert, at all these things. Notice that this level of knowledge (represented by the top part of the pyramid) is the smallest and contains the fewest things. This is because most technologists have to pick and choose the areas in which to develop expertise—no one can be an expert at everything.

Stuff you know you don't know (the middle part of the pyramid) includes things a technologist knows a little about or has heard of but in which they have little or no experience or expertise. For example, most technologists have *heard* of Clojure and know it's a programming language based on Lisp, but can't write source code in Clojure. This level of knowledge is much bigger than the top level. This is because people can become familiar with many more things than they can develop expertise in.

Stuff you don't know you don't know is the largest part of the knowledge pyramid. It includes the entire host of technologies, tools, frameworks, and languages that would be the perfect solution to a problem, if only the technologist trying to solve the problem knew that these solutions exist. The goal in any individual's career should be to move things from the *stuff you don't know you don't know* into the second area of the pyramid, the *stuff you know you don't know*—and, when expertise becomes necessary, to move things from the middle part of the pyramid to the top: the *stuff you know*.

Early on in a developer's career, expanding the top of the pyramid (Figure 2-3) means gaining valuable expertise. However, the *stuff you know* is also 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. Keeping things at the top of the pyramid requires a time investment to maintain expertise. This top part represents the individual's *technical depth*: the things they know really well.

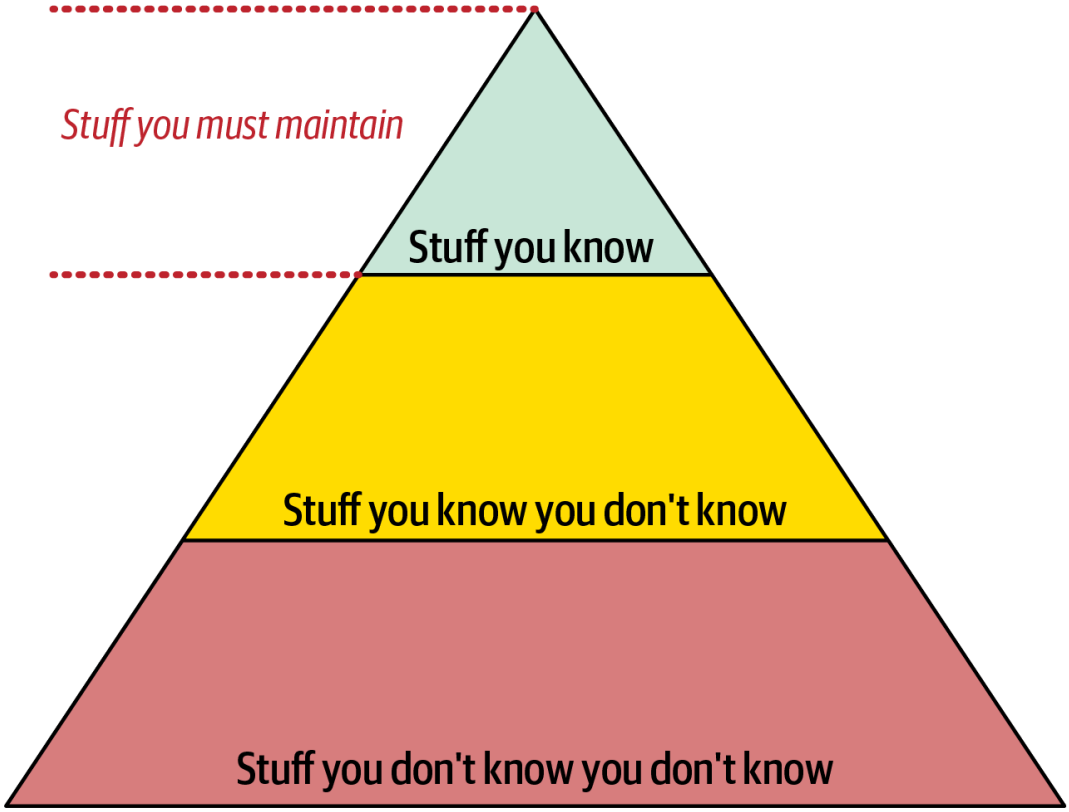

Figure 2-3. Developers must maintain expertise to retain it

However, the nature of knowledge changes as developers transition into the architect role. A large part of the value of an architect is that they have a *broad* understanding of technology and how to use it to solve particular problems. For example, it's better for an architect 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 2-4.

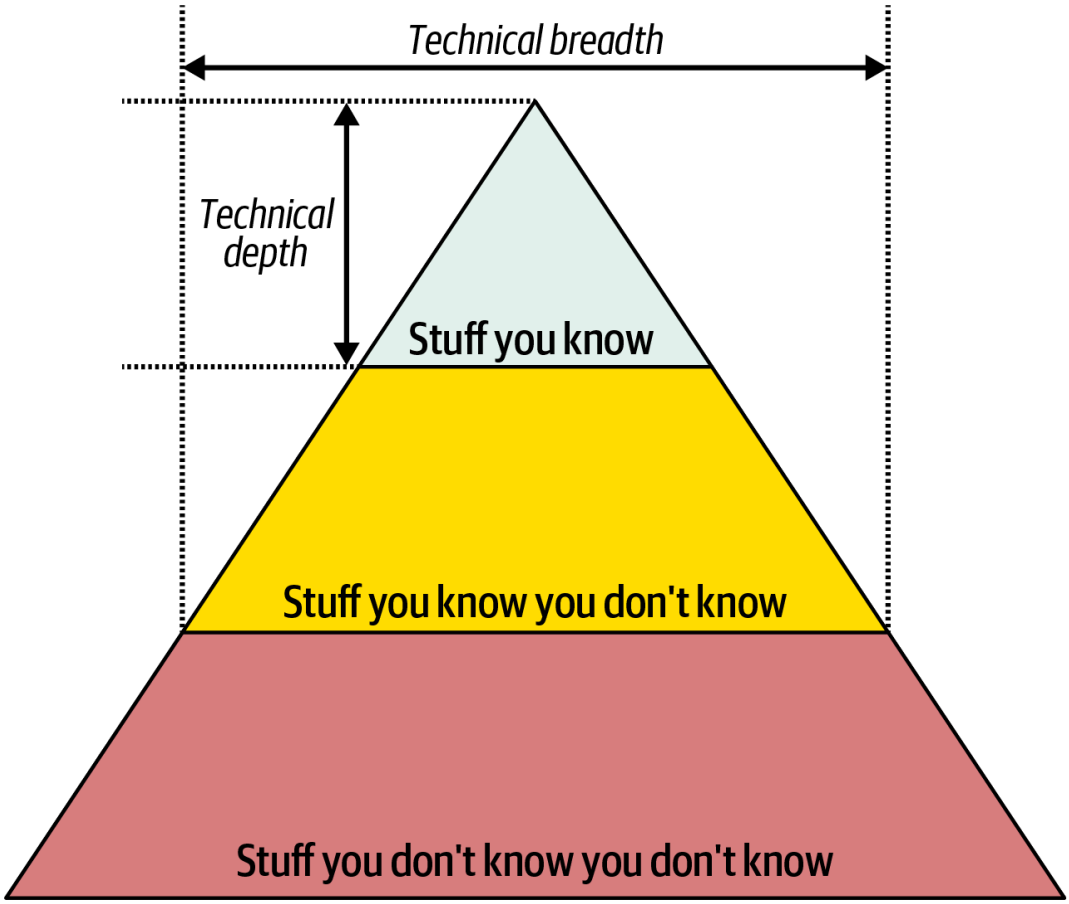

Figure 2-4. How much someone knows about a topic is technical depth, and how many topics someone knows is technical breadth

For 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 2-5. Some areas of expertise will remain, probably in particularly enjoyable technology areas, while others usefully atrophy.

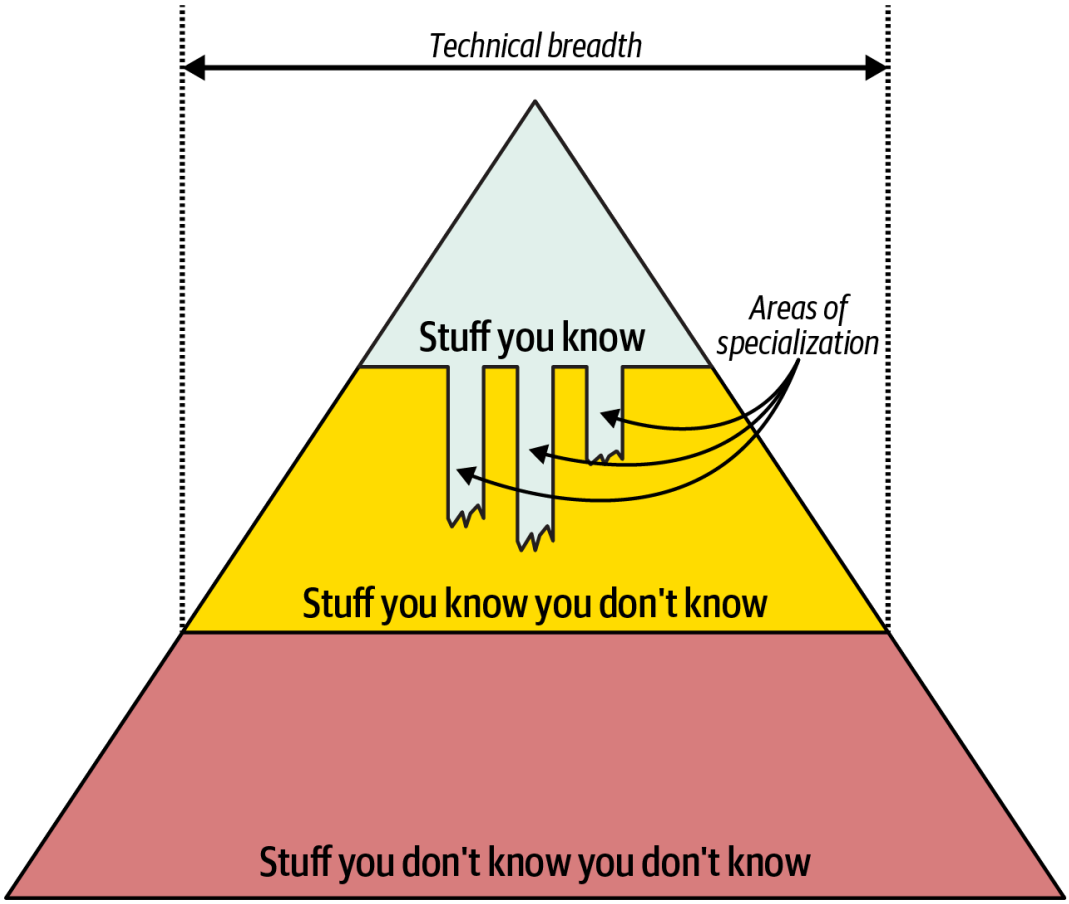

Figure 2-5. Enhanced breadth and shrinking depth for the architect role

Our knowledge pyramid illustrates how fundamentally different the roles of *architect* and *developer* are. Developers spend their whole careers honing expertise. 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 Antipattern").

Frozen Caveman Antipattern

An *antipattern* is what programmer Andrew Koenig defines as something that seems like a good idea when you begin, but leads you into trouble. A behavioral antipattern commonly observed in the wild, the Frozen Caveman antipattern, describes architects who revert to their pet irrational concern for every architecture. For example, one of Neal's colleagues worked on a system that featured a centralized architecture. 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 the client's headquarters from communicating with its stores in Italy, causing great inconvenience. While the chances of this recurring were extremely small, the architects had become obsessed with this particular architectural characteristic.

Generally, this antipattern manifests in architects who have been burned in the past by a poor decision or unexpected occurrence, making them particularly

cautious about anything related. While risk assessment is important, it should be realistic as well. Understanding the difference between genuine and perceived technical risk is part of the ongoing learning process. Thinking like an architect requires overcoming these Frozen Caveman ideas and experiences, seeing other solutions, and asking more relevant questions.

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 the depth and the breadth of their portfolio of knowledge is something every developer should consider throughout their career. But how does an architect gain technical breadth? The next sections provide a few techniques that will help you uncover the "stuff you don't know that you don't know."

The 20-Minute Rule

As illustrated in Figure 2-5, technical breadth is more important to architects than technical depth. But how do you stay current on all the latest trends and buzzwords while also working full-time, developing your career, spending time with friends, and taking care of your family?

One technique we use is the *20-minute rule*. The idea is to devote *at least* 20 minutes a day to learning something new or diving deeper into a specific topic. Figure 2-6 illustrates some good places to spend your 20 minutes, such as InfoQ, DZone Refcardz, and the Thoughtworks Technology Radar. You can learn more about unfamiliar buzzwords by looking them up on the internet, promoting that knowledge from "the stuff you don't know you don't know" into the "stuff you know you don't know." You could even spend that time reading a book like this one. The point is to carve some time out of your busy day to focus on developing your technical breadth and thus your career.

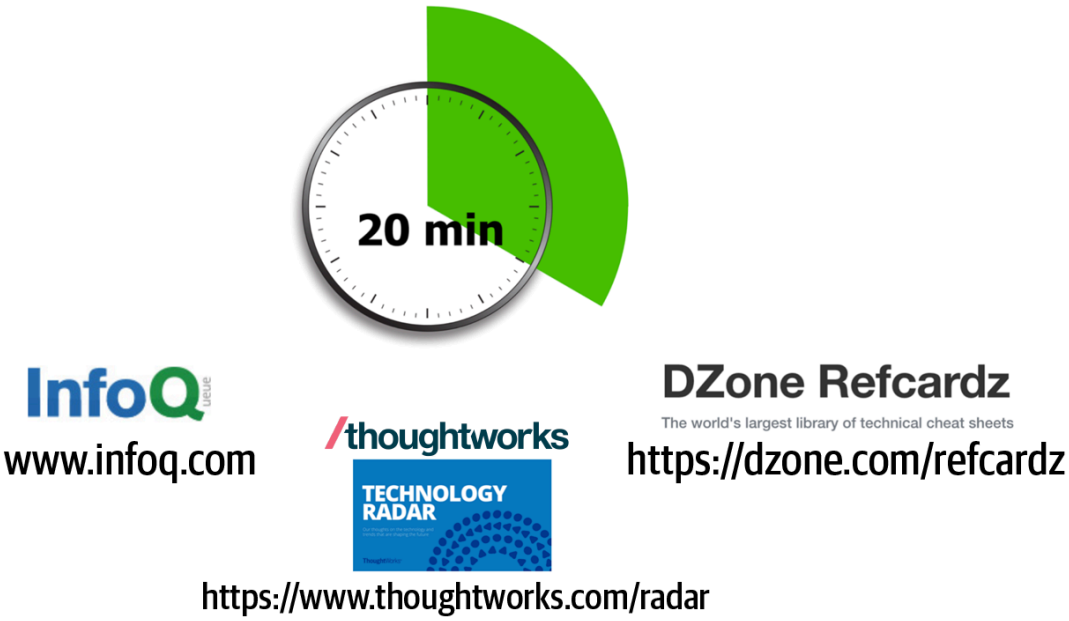

Figure 2-6. The 20-minute rule

Many technologists, when they first embrace this concept, plan their 20 minutes for lunch or after work; however, in our experience, these time periods rarely work. It's easy to start using lunchtimes to catch up on work rather than take a break, and evenings are even worse, with social plans, family time, and more after a long day. Instead, we strongly recommend taking your 20 minutes first thing in the morning—right after grabbing a cup of coffee or tea and, importantly, *before*

you check your email. Once you check email, your morning is over and your day has started. Get your 20 minutes in while your mind is fresh and before distractions take over.

Following the 20-minute rule will increase your technical breadth and help you develop and maintain the knowledge that will make you an effective software architect.

Developing a Personal Radar

For most of the ’90s and the beginning of the ’00s, one of your authors was the CTO of a small training and consulting company. When he started there, the primary platform was Clipper, a rapid-application development tool for building DOS applications atop dBASE files. Until one day it vanished. The company had noticed the rise of Windows, but the business market was still DOS…until it abruptly wasn’t. One coworker lamented that they couldn’t take their enormous body of now-useless Clipper knowledge and replace it with something else. Has any group in history, the coworker wondered, learned and thrown away so much detailed knowledge within their lifetimes as software developers do? The experience left a lasting impression: *ignore the march of technology at your peril*.

It also taught us an important lesson about technology bubbles. When developers, architects, and other technologists become heavily invested in a specific technology, pouring our work and thought into it, we tend to live in a memetic bubble. Inside the bubble, which also serves as an echo chamber, everyone knows and cares about that technology as much as we do. We might not even see honest appraisals from outside the bubble, especially if the bubble was created by a technology vendor in the first place. And when the bubble begins to collapse, there’s no warning until it’s too late.

What we lack in our bubble is a *technology radar*: a living document to help us assess the risks and rewards of existing and nascent technologies. The radar concept comes from Thoughtworks, where Neal serves as a director and software architect. In this section, we’ll describe how this concept came to be and then show you how to create a personal radar.

The Thoughtworks Technology Radar

The Technology Advisory Board (TAB) is a group of senior technology leaders within Thoughtworks that assists the CTO in making decisions about technology directions and strategies for the company and its clients. To stay current, this group began producing what’s now a biannual Technology Radar.

This had unexpected side effects. When Neal spoke at conferences, attendees began seeking him out to thank him for helping produce the Radar, often adding that their company had started producing its own version. Neal also realized that this was the answer to a question constantly asked at conference speaker panels: “How do you keep up with technology? How do you figure out what to pursue next?” The answer, of course, is that the speakers all have some form of internal radar.

Parts

The Thoughtworks Radar consists of four quadrants that attempt to cover most of the software-development landscape:

Tools

Everything from development tools like IDEs to enterprise-grade integration tools

Languages and frameworks

Computer languages, libraries, and frameworks, typically open source

Techniques

Any practice that assists software development overall, including processes, engineering practices, and advice

Platforms

Technology platforms, including databases, cloud vendors, and operating systems

Rings

The Radar has four rings, listed here from outer to inner:

Hold

The original meaning of the Hold ring was “hold off for now,” to represent technologies that were too new to reasonably assess yet—technologies that were getting lots of buzz but weren’t yet proven. The Hold ring has evolved, and now indicates something more like “don’t start anything new with this technology.” There’s no harm in using it on existing projects, but think twice about using it for new development.

Assess

The Assess ring indicates that a technology is worth exploring (such as through development spikes, research projects, or conference sessions) to see how it will affect the organization. For example, when mobile browsers became prominent, many large companies visibly went through this phase when formulating a mobile strategy.

Trial

The Trial ring is for technologies worth pursuing. If a capability is in this ring, it is important to understand how to build it. Now is the time to pilot a low-risk project.

Adopt

Thoughtworks feels strongly that the industry should adopt the items listed in the Adopt ring.

In the example view of the Radar in Figure 2-7, each “blip” represents a different technology or technique. While Thoughtworks uses the radar to broadcast its collective opinions about the software world, many developers and architects also use it as a way of structuring their technology assessment process and organizing their thinking about what to invest time in. For personal use, we suggest altering the meanings of the quadrants to the following:

Hold

This can include not only technologies and techniques to avoid, but also habits you are trying to break. For example, if you're an architect from the .NET world, you might be accustomed to reading the latest news and gossip on forums about team internals. While entertaining, this may be a low-value information stream. Placing it in the Hold ring serves as a reminder of what you want to avoid.

Assess

Use the Assess ring for promising technologies that you've heard good things about but haven't had time to assess for yourself yet. This ring forms a staging area for more serious future research.

Trial

The Trial ring indicates active research and development, such as performing spike experiments within a larger code base. These are technologies worth spending time on to understand more deeply so that you can include them effectively in trade-off analysis.

Adopt

Your personal Adopt ring represents the new things you're most excited about and best practices for solving particular problems.

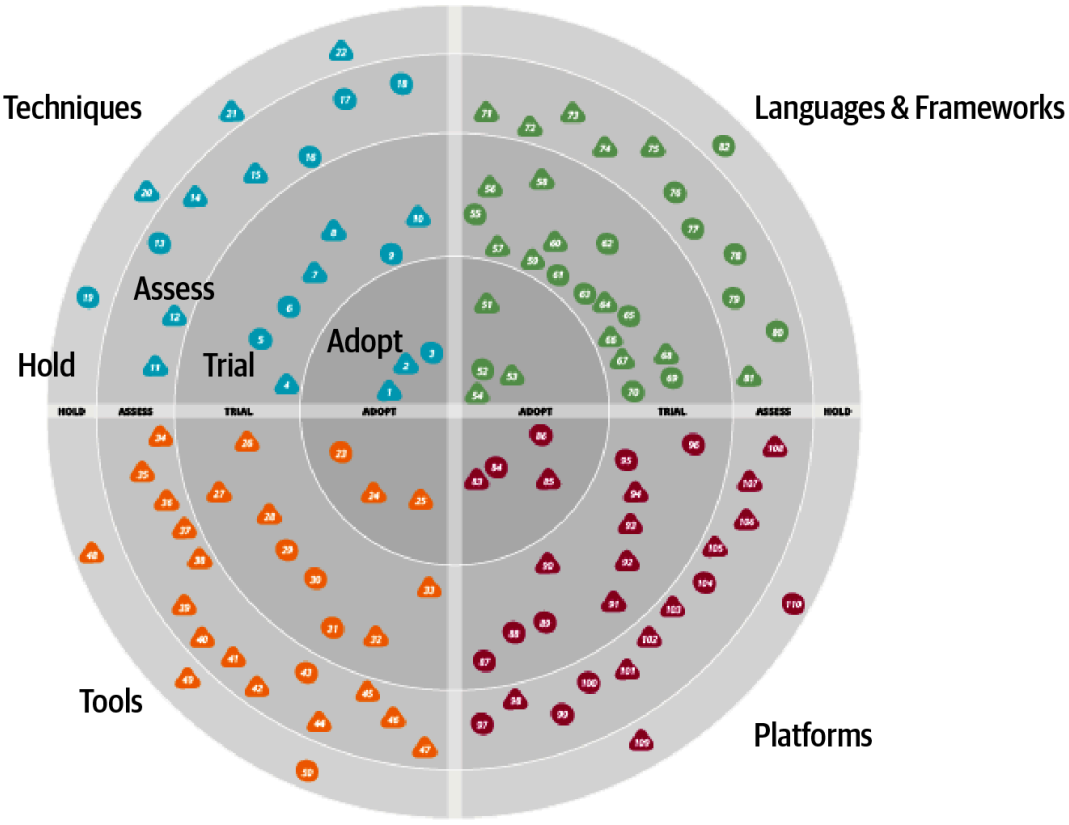

Figure 2-7. A sample Thoughtworks Technology Radar

Most technologists pick technologies on a more or less ad hoc basis, based on what's cool or what their employers are using—but it's dangerous to your career to adopt a laissez-faire attitude toward your technology portfolio. Creating a technology radar helps you formalize your thinking about technology and balance opposing decision criteria. (For example, it might be harder to get a new job focused on the "cooler" technology, whereas a more established technology might have a huge job market but offer less interesting work.)

Treat your technology portfolio like a financial portfolio: diversify! Choose some technologies and/or skills that are widely in demand, and track that demand. But you might also want to try some technology gambits, like generative AI or embedded IOT devices. Anecdotes abound about developers who freed themselves from cubicle-dwelling servitude by working late at night on open source projects that became popular and, eventually, purchasable. This is yet another reason to focus on breadth rather than depth.

Building a personal radar provides a good scaffolding for broadening your technology portfolio—but ultimately, the exercise is more important than the outcome. Creating the radar visualization gives you an excuse to carve out time from your busy schedule to think about these things, which is often the only way to accomplish this kind of thinking.

While the most important part of building your personal radar is the conversations it generates, it also produces some very useful visualizations. After much popular demand from technologists building their own radar visualizations, Thoughtworks released a tool called Build Your Own Radar. With a Google spreadsheet as input, it generates radar visualization showing your personal radar. We encourage every technologist to leverage it.

Analyzing Trade-Offs

Thinking like an architect is about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine the best solution. The reason this is one of the critical activities of an architect (and hence part of architectural thinking) is exemplified through the following quote by Mark (one of your authors):

> Architecture is the stuff you can't Google or ask an LLM about.
>
> Mark Richards

Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is "It depends." While this answer might be annoying, it is unfortunately true. You cannot Google the answer or ask an AI engine or large language model (LLM) whether REST or messaging would be better for your system or whether microservices is the right architecture style for your new product, because the answer *does* depend. It depends on the deployment environment, business drivers, company culture, budgets, time frames, developer skill set, and dozens of other factors. Everyone's environment, situation, and problem will be different. That's why architecture is so hard. To quote Neal, your other author:

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

For example, consider an item auction system (Figure 2-8) where online bidders bid on items up for auction. 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.

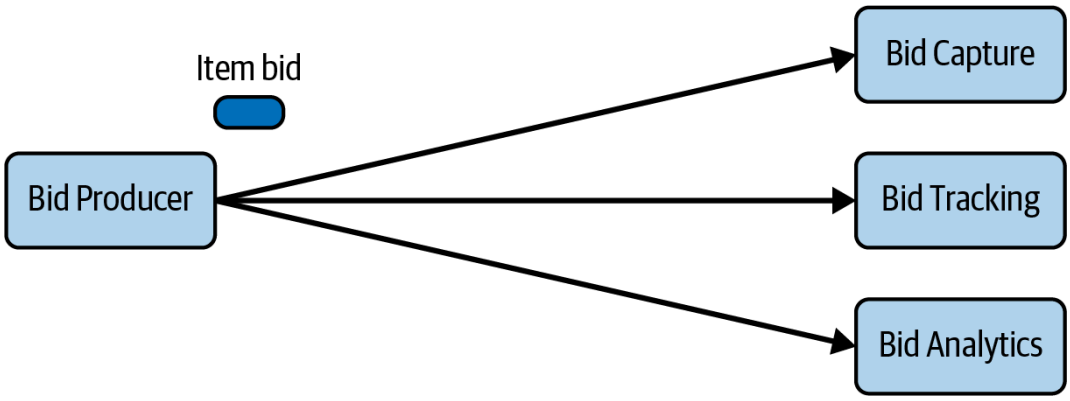

Figure 2-8. Auction system example of a trade-off—queues or topics?

For the asynchronous behavior in this system, an architect could use queues in a point-to-point messaging fashion, or use a topic in a publish-and-subscribe messaging fashion. Which one should they choose? They can't Google the answer. Architectural thinking requires them to analyze the trade-offs associated with each option and select the best (or least worst) option given the specific situation.

The two messaging options for the item auction system are shown in Figure 2-9, which illustrates using topics in a publish-and-subscribe messaging model, and Figure 2-10, which depicts using queues in a point-to-point messaging model.

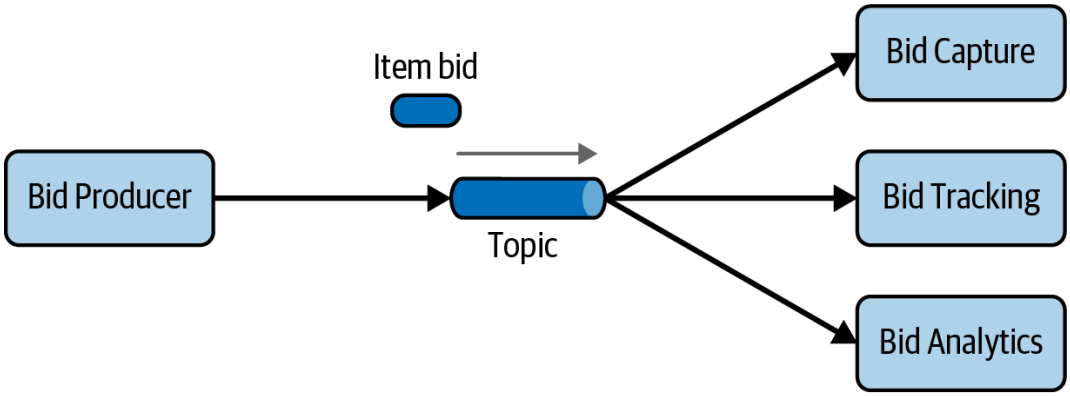

Figure 2-9. Using a topic for communication between services

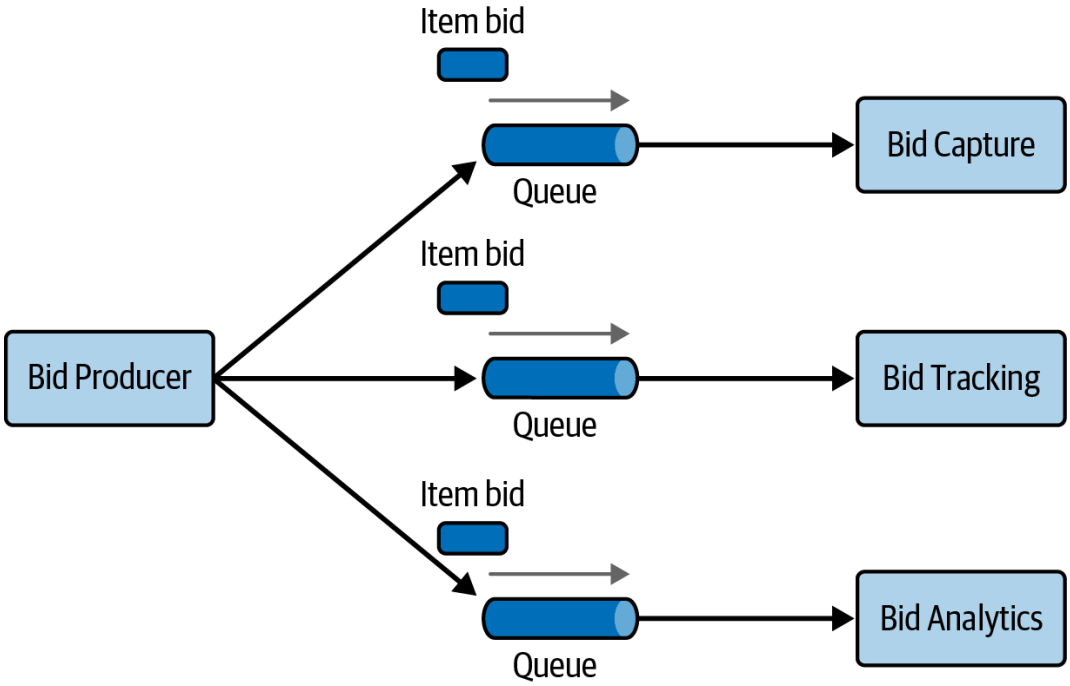

Figure 2-10. Using queues for communication between services

The clear advantage (and seemingly obvious solution) to this problem in Figure 2-9 is *architectural extensibility*. The `Bid Producer` service only requires a single connection to a topic. Compare that to 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 (to provide each bidder with a history of all of their bids), no changes would be needed to the existing services or infrastructure. The new `Bid History` service could simply subscribe to the topic that already contains the bid information.

However, with the queue option shown in Figure 2-10, the `Bid History` service would require a new queue, 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 means that adding new bidding functionality requires significant changes to the services and infrastructure, whereas with the topic approach, no changes to the existing infrastructure are needed. Also, the `Bid Producer` is less coupled in the topic option, where 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 will be used (and by whom), and hence is more coupled to the system.

So far, this trade-off analysis seems to make it 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.
>
> Rich Hickey

Thinking architecturally means not only looking at the benefits of a given solution, but also analyzing the associated negatives, or trade-offs. Continuing with the auction system example, a software architect would analyze the negatives of the topic solution as well as the positives. In Figure 2-9, you can see that with a topic, *anyone* can access bidding data, which introduces a possible issue with data access and data security. However, 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 was to listen in on a queue, the corresponding service would not receive those bids, and a notification would immediately be sent about the loss of data (and possible security breach). In other words, it is very easy to wiretap 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 that receive the bidding data must accept the same data contract and set of bidding data. In the queue option, 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, and thus a separate contract that does not impact any other service.

Another disadvantage of the topic model is that it does not support monitoring the number of messages in the topic, so it can't support autoscaling capabilities. However, with the queue option, each queue can be monitored individually and programmatic load balancing can be applied to each bidding consumer so that each can be automatically scaled independently. Note that this trade-off is technology specific: 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 fuller trade-off analysis, which is the better option?

It depends! Table 2-1 summarizes these trade-offs.

Table 2-1. Trade-offs for topics

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

Again, *everything* in software architecture has trade-offs: advantages and disadvantages. Thinking like an architect means analyzing these trade-offs, then asking questions like: "Which is more important: extensibility or security?" An architect's choice will always depend on the business drivers, environment, and a host of other factors.

Understanding Business Drivers

Thinking like an architect also means understanding the business drivers 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 business-domain knowledge and healthy, collaborative relationships with key business stakeholders. We've devoted four chapters in the book to this specific topic: in Chapter 4, we define various architecture characteristics. In Chapter 5, we describe ways to identify and qualify architecture characteristics. In Chapter 6, we describe how to measure each characteristic to ensure the business needs of the system are met. And finally, in Chapter 7 we discuss the scope of architectural characteristics and how they relate to coupling.

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 should 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.

Our first tip for anyone striving to balance hands-on coding with being a software architect is avoiding the *Bottleneck Trap*. The Bottleneck Trap antipattern occurs when an architect takes ownership of code within the critical path of a system (usually the underlying framework code or some of the more complicated parts) and becomes a bottleneck to the team. This happens because the architect is not a full-time developer and therefore must balance development work (like writing and testing source code) with the architect role (drawing diagrams, attending meetings, and well, attending more meetings).

One way to avoid the Bottleneck Trap is for the architect to delegate the critical parts of the system to others on the development team and then focus on coding a minor piece of business functionality (such as a service or UI screen) one to three iterations down the road. This has three positive effects. First, the architect gains hands-on experience by writing production code while avoiding becoming a bottleneck on the team. Second, the critical path and framework code are distributed to the development team (where they belong), giving the team 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. This helps them better identify with the development team’s pain points around processes, procedures, and the development environment (and hopefully work to improve those things).

Suppose, however, that the architect can’t develop code with the development team. How can they still remain hands-on and maintain some level of technical depth? The following are some tips and techniques for architects who want to continue deepening their technical abilities:

Frequent proofs-of-concept

> Doing frequent proofs-of-concept (POCs) requires the architect to write source code; it also helps validate an architecture decision by taking the implementation details into account. For example, suppose an architect gets stuck trying to choose 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 firsthand the implementation details and the amount of effort required to develop the full solution. It also allows them to better compare architectural characteristics between the different caching solutions, such as scalability, performance, and overall fault tolerance.
>
> 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 any architect would want is for their sloppy throwaway code to be treated as representing their typical quality of work. Second, writing production-quality proof-of-concept code means getting

practice at writing quality, well-structured code, rather than continually developing bad coding practices by creating quick, sloppy POCs.

Tackle technical debt

Another way architects can remain hands-on is to tackle some technical debt, freeing the development team to work on the critical functional user stories. Tech debt is usually low priority, so if the architect doesn't get the chance to complete a technical debt task within a given iteration, it's not the end of the world and generally won't impact the success of the iteration.

Fix bugs

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

Automate

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 them. They'll 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 in the form of architectural analysis and fitness functions to ensure the vitality and compliance of the architecture is another great way to stay hands-on. For example, an architect can write Java code in ArchUnit in the Java platform to automate architectural compliance, or custom fitness functions to ensure architectural compliance while gaining hands-on experience. We talk about these techniques in Chapter 6.

Do code reviews

A final technique to remain hands-on as an architect is to do frequent code reviews. While the architect is not actually writing code, this at least keeps them *involved* in the source code. Further benefits of code reviews include ensuring compliance with the architecture and identifying mentoring and coaching opportunities on the team.

There's More to Architectural Thinking

This chapter forms the foundational aspects of beginning to think like an architect. However, there's much more to thinking like an architect than what is described in this chapter. Thinking like an architect involves understanding the overall structure of a system (we cover that topic next in Chapter 3), understanding business concerns and translating those concerns to architectural characteristics (we have four chapters on that topic), and finally, seeing a system through its logical components—the building blocks of a system (we discuss that topic in Chapter 8).