
Chapter 2

Software Architecture & Modeling

Engineers use abstraction and models to solve large and complex problems. However, the use of models does not mean that you must design everything up front or put architects in corner offices. You can separately decide job roles, process, and the engineering artifacts.

Software architecture is about the design of your system and the impact it has on the system's qualities, things like performance, security, and modifiability. Pinning the definition down further is difficult, but we know that architecture acts as the skeleton of a system, influences system qualities (called quality attributes), and is independent of the system's functionality.

Sometimes any architecture you choose will work out just fine; other times it is not obvious that a solution is even possible. The harder the problem is, the more you will need to pay attention to your architecture choices.

This chapter discusses three levels of engagement with architecture. In architecture-indifferent design, you pay little attention to the architecture, perhaps using a presumptive architecture, such as a 3-tier system on an IT project. In architecture-centric design, you design the architecture so that it helps the system achieve a desired property. And with architecture hoisting, you build source code that manages a property at runtime.

2.1 Abstraction and modeling

When I was in high school, I asked my father for help with my calculus homework. I was shocked to learn that despite his working as an engineer since college, his calculus knowledge was rusty and he rarely used it. Yet he also told me that his company exclusively hired engineers — not because they needed to apply calculus on the job, but because the engineering training that included calculus gave them the right kind of problem solving skills.

Only simple problems can be solved without abstraction. Instead, engineers must map complex real-world problems into an abstract model (such as a calculus equation), solve the problem within the model, then translate that solution back into a real world solution. For engineers, it is the ability to solve problems using abstract models that is essential. Calculus is just one kind of abstract model.

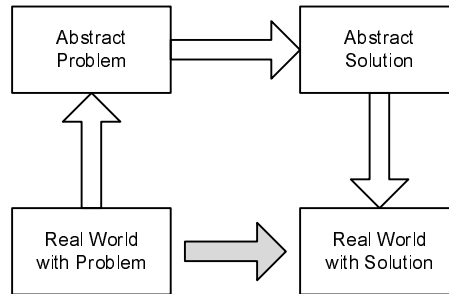


Figure 2.1: A commuting diagram, popularized in software engineering by Mary Shaw. Simple problems can be solved directly (gray arrow). Complex problems are solved by using abstractions.

Figure 2.1 shows the process of solving problems using models as a commuting diagram. An engineer's goal is to move from a real world problem to a real world solution. Easy problems can be solved directly, without abstraction, and can move directly across the gray arrow. The problems that engineers get paid to solve, however, are harder and require the longer path via abstractions.

Scale and complexity require abstraction

Software developers naturally reach for abstractions when a system is large or complex. Today, they naturally use classes and design patterns as appropriate. When a system has just 50 classes, it makes sense to just reason about the classes in it. When it has 500 classes, objects and patterns can be used to reason about it. But when a system has 5000 classes, developers reach for larger abstractions to make sense of it. It is not a question of being forced into new abstractions, it is a matter of using the ones suited for the scale or complexity of the problem.

Abstractions can also be more a more efficient way to learn about a system than direct inspection of source code. Imagine one developer wants to explain a system he already understands to another developer. If they had a long time, they could both read and discuss lots of source code, perhaps hundreds of thousands of lines. But if they only had a few hours then sketching out a model of the system would be more effective.

Abstractions provide insight and leverage

You will surely recall from your math classes a story problem such as the following:

Two trains are 3000m apart and headed towards each other on the same track.
One is traveling 10m/s, the other 20m/s. When will they meet?

When your teacher introduced such problems, you already knew some algebra, so you could have solved a problem if it were stated as $10x + 20x = 3000$. The teacher's intent with the story problem was for you to learn how to map the story problem into an abstract algebraic model, and then back to the story domain. To solve the problem, you had to learn to build a model that included the details that were relevant to the question being asked. The model provided insight into the essential problem and algebra provided leverage to solve it. The domain of trains gave you no particular insight or leverage, but an algebraic model did.

Ideally, software architecture would be solvable and universal just like algebra. Architecture modeling is rarely as simple as the train problem, but architecture models can provide insight and leverage. With an appropriate model, you can do things like find possible intrusion vectors, identify bottlenecks, and estimate latency. This is important because developers need to reason about more than just a system's features; they need to reason about its qualities too.

Reasoning about system qualities

I recently attended a lecture about building scalable websites. The presenter discussed technologies X and Y, his inability to make technology X run quickly, and his successful switch to technology Y. He described the compactness of the new language, the improvement in the interfaces, its extensibility, and finally presented the much improved throughput numbers for his website.

Under all these details, there was a nugget of insight to be found, which was that technology X stored data hierarchically and technology Y stored it flat. Both used relational databases, but a web page request in technology X required, on average, twenty database queries to retrieve the hierarchical data, while technology Y required just one. Substantially all of the throughput differences between technology X and Y could be traced to that single design difference: hierarchical vs. flat data storage. When evaluating throughput, every other feature of technology Y was "nice to have" except for the data representation choice. But how can you arrive at that conclusion?

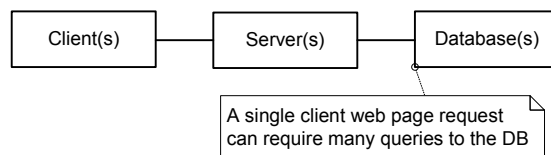


Figure 2.2: An informal sketch (a "cartoon") of the web system. Sometimes you will keep models like this in your head, other times you will sketch them on paper or a whiteboard. The second part of this book discusses standard models and notations for building architecture models.

To reason about system properties, you must have a model in your head that makes sense of the details, something like the sketch (i.e., cartoon) in Figure 2.2. That model is so simplified that it works for both technology X and Y, but it is sufficient to sort out the details that are known. Every web request that comes in will require some processing on the server and some number of queries to the database. If you assign some plausible numbers to these variables, such as 10ms for the server processing time and 50ms for the time to query the database once, it is clear that twenty database queries is going to slow the system down. Of course this is a very simple model and it ignores important factors like caching and queueing, but it demonstrates that details by themselves are insufficient: a model is needed to make sense of them.

Architecture models are a good way to understand and address thorny issues because they can cut through the extraneous detail and allow you to focus on the essential parts and relationships, to make predictions, and to evaluate alternatives. If you were running a website on technology X, code tweaking would not fix the throughput problem. You could only have succeeded by recognizing that, regardless of other benefits a hierarchical data structure might provide, it was unsuitable for achieving the throughput you needed.

Models elide details

When you reasoned about when two trains will meet, you safely elided the color of the trains and many other details. When you reasoned about the performance of a website, you elided details like the programming language. Models, by their nature, elide details. To create a useful model, you must choose to include the right details. Over-including irrelevant details is tolerable but it may clutter the model, making it harder to reason about.

Recall the problem from the introduction about modeling a path from New York to Los Angeles using highways. Some roads have signposts made of wood, others concrete, and others metal — this choice can be safely abstracted when you consider the shortest path. However, the shortest path from New York to Los Angeles may include a road smaller than a highway.

If your model only includes highways, your solution may be sub-optimal. On the other hand, if your model includes every vaguely flat and driveable surface (parking lots, front yards, fire roads) then your model will be huge and consequently the problem becomes much harder to solve.

You should be aware that the use of abstraction may include a tradeoff: The problem is too difficult to reason about without abstraction, but reasoning with the abstraction is imperfect.

Models can amplify reasoning

Models are used by different people for different purposes. There are three basic levels of modeling skill: reading models, writing models, and amplifying reasoning with models. As shown graphically in Figure 2.3, the ability to accurately *read a model* is the most common, and it is a prerequisite for the others. For example, the buyers of a custom-designed house

need to be able to read the house blueprints so that they can speak up when the designs do not match their desires. An analogous situation occurs with custom software development between the stakeholders and the software developers.

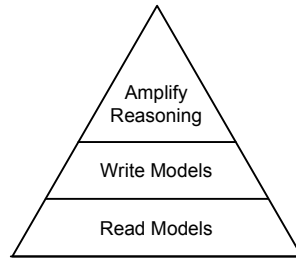


Figure 2.3: Levels of modeling competence

Fewer people must be able to *write a model*. The designer of a house uses blueprints to communicate a design that originates in his head to the various people who need to agree to the plans and to build the house. The designer can minimize the chances of miscommunication if he uses a standard notation¹ for his models.

But beyond this, the house designer uses the blueprints in a way that help him *amplify his reasoning*. Does opening a door block a cabinet? How much drywall is needed in the bedroom? The designer can create models that help him to answer such questions. Similarly, experienced software designers know how to build models to enable analysis, to make errors easier to detect, and to discover truths that are not immediately obvious.

2.2 Architects architecting architectures

It is tempting to jump from the idea that abstractions and modeling are helpful in engineering to the conclusion that software architects should design the architecture in advance. Before deciding when you should do what, it is best to disentangle job roles, processes, and artifacts.

Three ideas are often jumbled together when talking about software architecture: the *job title* called architect, the *process* of architecting a system, and the *engineering artifact* called the software architecture.

- **The job role: architect.** One possible job title in an organization is that of a software architect. Some architects sit in corner offices and make pronouncements that are disconnected from the engineering reality and other architects are intimately involved in the ongoing construction of the software. Either way, the title and the office are not intrinsic to the work of designing or building software. All software developers should understand the software architecture, not just the architects.

¹You should avoid getting stuck at the syntactic level of model creation. UML and other notations are useful tools but mastering the detailed syntax of a modeling language is a means to an end. Instead, focus on how that tool can amplify your reasoning.

- **The process: architecting.** There is no software at the beginning of a project, but by the end of the project there is a running system. In between, the team performs activities (i.e., they follow a process) to construct the system. Some teams design up-front and other teams design as they build. The process that the team follows is separable from the design that emerges. A team could follow any number of different processes and produce, for example, a 3-tier system. Or put another way, it is nearly impossible to tell what process a team followed by looking only at the finished software.
- **The engineering artifact: the architecture.** If you look at an automobile, you can tell what type of car it is, perhaps an all-electric car, a hybrid car, or an internal combustion car. That characteristic of the car is distinct from the process followed to design it, and distinct from the job titles used by its designers. The car's design is an engineering artifact. Different choices about process and job titles could still result in them creating, for example, a hybrid car. Software is similar. If you look at a finished software system, you can distinguish various designs; for example: collaborating peer-to-peer nodes in a voice over IP network, multiple tiers in information technology systems, or parallelized MapReduce compute nodes in internet systems. Every software system has an architecture just as every car has a design. Some software is cobbled together without a regular process, yet its architecture is still visible.

This book focuses on software architecture as an engineering artifact. It touches briefly on roles and process later in this chapter, and Chapter 3 covers the idea of using risk to answer the question “How much architecture should you do?” The rest of the book treats architecture as an engineering artifact, something to be understood and designed so that you can be a better software engineer.

2.3 What is software architecture?

A software system's *design* consists of the decisions and intentions that are in the heads of the developers. Sometimes the designs are written down, sometimes not. In contrast, when people talk about *specifications* they generally mean designs that are written down with the idea that someone else will read them and write source code that is consistent with those specifications.

If a design consists of the decisions and intentions of software developers, then what is the *architecture*? The architecture of a software system is a part of its design. The concept of a design is quite broad, so it is useful to have a different term to describe the decisions and intentions that will have a big impact on the system's features and qualities. Otherwise you could not distinguish the important decisions and intentions from the mundane ones. While many people agree that important decisions exist, they often disagree about what they are, and therefore about how to distinguish architecture from design.

Definition

A great variety of definitions of software architecture have been offered, indicating a lack of consensus. However, there is reasonable agreement that architecture deals with macroscopic, sweeping issues in software design. One of the most popular definitions is from the Software Engineering Institute (Bass, Clements and Kazman, 2003):

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

This definition highlights topics that are important: structure, elements, externally visible properties, and relationships. By not mentioning the internal design of the software elements, the definition implies a distinction between software architecture and software design, suggesting that architecture focuses on the biggest elements and their relationships, while design focuses on their insides.

If architecture is the macroscopic, sweeping issues of software design, then how is it different than *high-level design*? High-level design is an old term, and has been a long-standing concern. It is present in Fred Brooks' writings about IBM's operating system designs in the 1960's (Brooks, 1995) and clearly stated in the 1968's NATO conference (Naur and Randell, 1969). Academic writers, in an effort to differentiate new design ideas from earlier high-level design, called these new ideas software architecture (Perry and Wolf, 1992; Shaw and Garlan, 1996). These ideas included a focus on runtime views and system qualities, and improved abstractions. When people use the term *high-level design*, they could be referring to the older ideas that emphasize compile-time modules and functionality, or they could be using it synonymously with the term *architecture*.

Intentionality

Defining the architecture as the macroscopic parts and detailed design as everything else works OK. However, there are examples of architectural details that are not limited to the macroscopic parts. For example, the original Java Beans spec required a naming pattern for exposed Bean properties because, under the covers, it would use reflection to convert methods like `getTargetVelocity` into an exposed property called `TargetVelocity`. The naming pattern for methods is about as low-level a decision as possible, yet it is architecturally significant for Java Beans. Similarly, an architecture may prohibit threads, require a method to complete within 100ms, require computation to be divided into jobs, or other details that are down deep in the code.

It is unsatisfactory to conclude that architecture concerns the macroscopic parts of design, except sometimes when it does not. In such a definition, who would decide what counts as architecture? Perhaps the designers of houses and skyscrapers can explain the difference between architecture and design. Like software, house designers have architectural designs and detailed designs, but while software is only about a half century old, houses have been built for millennia.

My brother builds skyscrapers and he tells me that, in his field, the architect will usually specify some low-level details, but leave others to be decided by the construction company. The architect includes details in the architecture if they contribute to an overall quality of the building, such as its watertightness, aesthetic appeal, or constructability. Otherwise a detail is considered non-architectural. On a recent job of my brother's, the architect insisted that the gap between windows be quite small because this detail was important for the architect's intention about how the building would look.

To distinguish architectural details from other details, it would appear that intentionality is the key: there is a *chain of intentionality* from a few high-level intentions or decisions of the architect that reaches down into a few low-level details. Most of the details are left open to any reasonable implementation, but some are constrained, via a chain back to the top-level intentions of the architect. Based on that insight, here is an alternate definition of software architecture:

The architecture of a software system is the design decisions made by its developers to achieve their top-level intentions, and the designs that follow directly from those decisions.

The definition does not insist that the architecture contain all of the highest level elements and relationships in a system, so perhaps an architecture could consist of some high-level elements and some lower-level details, but leave undecided other details, including the possibility of additional high-level elements.

This definition is good in that it captures the chain of intentionality. Its main problem is that it excludes purely unintentional architectures, ones that just happen without conscious thought of their developers. Though this definition is over-constrained, it does help explain two things. First, architecture is not just the top-level elements or design decisions. Second, low-level details are considered architectural when they are connected via a chain of intentionality, not because of the whim of an architect.

There are many questions involving architecture and design. Is architecture part of design or are they separate? Where does one stop and the other begin? What is the difference between high-level design and architecture? In practice, the architecture is part of the design. As with skyscrapers, if a detail is important to achieve an overall quality of the system, it is probably architectural. Ultimately, it is the ideas themselves that count, not the names, so it is not worth obsessing about fine distinctions between design and architecture.

2.4 Why is software architecture important?

These definitions of software architecture are not sufficient to stop arguments about where architecture stops and design begins². Despite differing opinions about the definition of software architecture, it is possible to characterize it:

²If you find yourself in a heated argument over the definition so that you can compartmentalize roles in your company, then you likely have bigger problems to work out. On the other hand, if you build software and are simply looking for techniques that help you build it better and reduce the risks of failure, then you are likely tolerant of some fuzziness surrounding the definitions.

- **Architecture acts as the skeleton of a system.** Every system has an architecture, whether its developers consciously chose it or not. There is no single right architecture, but there are more or less suitable skeletons for the job.
- **Architecture influences quality attributes.** Quality attributes are externally visible properties, such as security, usability, latency, or modifiability. Metaphorically speaking, different skeletons are better or worse at handling different burdens, so choosing the right architecture can make achieving desired qualities easier.
- **Architecture is orthogonal to functionality.** It is possible to build the same banking system as a 3-tier architecture or as a peer-to-peer system. When the architecture is poorly matched to the functionality, however, developers will struggle against it.

And, perhaps most importantly:

- **Architecture is about constraints.** Architecture is the art of imposing just enough constraints so that the system has the properties you want. For example, a design that ensures scalability may require some components to be stateless in order to achieve that scalability.

The following sections look at each of these ideas in turn.

Architecture as skeleton

It is useful to consider the imperfect metaphor of architecture as a skeleton. The skeleton provides the overall structure for an animal and influences what it can do. Birds are good at flying and kangaroos are good at jumping largely because of their skeletons. Most fast animals have four legs but while the two-legged ones are slower, they might be able to use tools easier.

You cannot say that one skeleton is better than another unless you can say that jumping is better than flying. You can, however, say whether a skeleton is well suited to its function or not, since it would take a lot of work to make a kangaroo skeleton fly.

Software is largely similar. A 3-tier architecture enables information technology (IT) systems to localize changes and handle transactional loads. A cooperating processes architecture is well suited to operating systems because it isolates faults. It is hard to imagine a distributed VOIP network like Skype using anything other than a peer-to-peer architecture.

The skeleton metaphor is imperfect, however, because an architecture is more than the big visible parts, and the invisible parts (such as constraints) are often the more important ones. For example, your locking policy, memory management strategy, or technique for integrating third party components can all be part of the architecture, yet each of these is invisible in a running system or in its source code.

Architecture influences qualities

Developers must pay attention to what the software does, that is, its *functionality*. Accounting software that fails to account, or animation software that fails to animate is not

particularly useful. But systems also have additional requirements that are not related to their function, referred to as *extra-functional* or *non-functional* requirements³. Developers must pay attention to extra-functional requirements too, since accounting software that lets bad guys read secret accounts or animation software that runs too slowly is not particularly useful either.

Beyond supporting the required functionality, a system's architecture enables or inhibits qualities such as security or performance. The skeletons of a person and a horse both support the function of taking apples to market, but consider how each differs in throughput or accommodates varying tasks. It is usually possible to make any architecture choice work, but some choices make the qualities easy to achieve, while other choices make them hard. These system qualities are called *quality attributes*, and are discussed in depth in Chapter 6.

Functional requirements that evolve over time are a challenge to any system, but evolving quality attributes can force drastic changes. A system that was designed to support a hundred users may be impossible to scale up to a hundred thousand without a rewrite. You often see successive generations or waves of an application as it outgrows its old architecture, somewhat like a crab outgrowing its shell. There are options for handling this, discussed in Section 3.7, including getting the quality attributes understood up-front versus getting them correct enough to move forward.

Architecture independent of functionality

There is no single best architecture, but architectures, like skeletons, are more suited to some tasks than others. A kangaroo with hollow bones might be too fragile and a bird with strong legs would fly like an ostrich. On the other hand, considering whales and bats shows you how far a skeleton can be pushed.

It is important to recognize that you can mix-and-match architecture and functionality. That is, you could change a system's architecture yet keep its functionality, or reuse the same architecture on a system with different functionality.

Although a system's architecture is an independent choice from its functionality, a poor architecture choice can make functionality and quality attributes difficult to achieve. With enough effort you probably could build any system using any architecture, but developers trying to make a 3-tier operating system would be pulling their hair out.

Architecture is about constraints

Constraints are essential in the construction of a system, in its ability to perform its job, and in the ability to maintain it over time. What a system *does not do* is as important as what it does. To ensure that a system possesses certain qualities, you must constrain it so that you know what it will *not* do. For example, a secure system will not exchange data with

³Extra-functional is the preferred term since "extra" is more etymologically accurate than "non" in this context — these requirements go beyond functional requirements, not negate them. Most people would interpret a sign saying "non-functional" hanging on a water fountain to mean that it is broken, not that it is high throughput.

untrusted parties, and a usable system will not start long-running computations without providing a cancel option.

You voluntarily choose to constrain your design in order to achieve qualities such as performance or security. For example, a train is severely constrained by its tracks, and consequently lacks flexibility in its destinations. But this constraint specifically enables other qualities, such as low rolling friction leading to efficiency. Security is another benefit, as it is impractical to hijack a train. An unconstrained design can, by definition, do anything, so if you are to have any hope of analyzing it, you must constrain it.

Constraints are a tool to be used by engineers to produce systems. Like tools used by artists or craftsmen, they serve the skill of those who wield them. A hammer and chisel in the hands of a skilled artist can create a beautiful sculpture but in the hands of a vandal they can break the nose from one. Appropriately applied, you can gain many benefits through constraints:

- **Embody judgment.** Constraints are a means to transfer wisdom or understanding from one developer to another. Senior engineers have a detailed and nuanced understanding of the domain they work in, and it takes time to convey this knowledge to others. Through constraints on the design, they can guide other engineers to acceptable solutions without fully transferring their knowledge.
- **Promote conceptual integrity.** Fred Brooks argues that conceptual integrity of a system an important goal of system design, and that a single good idea consistently applied is better than several brilliant ideas scattered across a system (Brooks, 1995). Desmond D'Souza taps into the same idea when he argues that architectural constraints reduce needless creativity of developers, enabling them to use that creativity in places where it is needed (D'Souza and Wills, 1998).
- **Reduce complexity.** Constraints can factor out complexity so that a system's underlying principles are evident. You have no doubt at some point spent time scrutinizing a small segment of code. That level of concentration simply will not scale to millions of lines of code, or even tens of thousands, without constraints to simplify your analysis.⁴
- **Understand runtime behavior.** Source code can be inspected directly, yet it can be difficult to understand how it will behave at runtime. You can write tricky code whose runtime behavior is nearly impossible to understand, or you can constrain it so that its runtime behavior is evident.

⁴Several years ago, I was making changes to an unfamiliar codebase and was making good progress on the problem until I looked into a setter method, like `SETFOO()`, and found that it was performing complex logic, including sending notifications to other parts of the code. I was particularly surprised to learn that it sometimes did not even set `foo` at all. At that point I realized that my assumptions about the code's constraints were false, and that the task would take much longer, since a method called `LAUNCHSPACESHUTTLE()` might in fact be doing something else entirely. Understanding a codebase is much easier with constraints, for example my that setter methods do indeed set the variable and only have local effects.

Any skepticism about imposing constraints originates not from the enlightened way that *you* would use them, but from the coarse and ignorant way that *others* use them. At some point you will have chafed at constraints placed by others on your system. Though constraints are sometimes used poorly, you cannot design without them because constraints impose organization on chaos, the engineer's mortal enemy. You must use them responsibly, like a sharp tool that can just as easily remove a finger as cut a board, rather than reject them outright. Designing a system's architecture requires reasoned choices about what is allowed and what is not.

2.5 When is architecture important?

When you are building the software equivalent of a doghouse, say a website to collect registrations for a family picnic, you are unlikely to spend much time thinking about the architecture. Conversely, you would hope that the developers of hospital software are paying attention to it. So how do you decide when architecture is important?

Architecture is likely to be important in systems with large scale or complexity. Here are three specific cases with high architecture risk.

- **Small solution space.** Architecture risk is high when the solution space is small or it is hard to design any acceptable solution. Consider the difficulty of creating a human powered airplane compared to creating a faster car. The airplane will require that everything is just right, including low weight and high efficiency. On the other hand, making a faster car is often no harder than adding a bigger engine.
- **High failure risk.** Anytime the failure risks are high it is probably important to get the architecture right. You would not want your hospital system failing because people might die, nor do you want the Mars rovers failing and wasting lots of money.
- **Difficult quality attributes.** Architecture influences your ability to satisfy quality attributes, so while making another email system seems easy, making one with quick performance that supports millions of users is hard.

The overarching answer is to look at how bad it would be to get the architecture wrong. When the architecture risk is small compared to the project risk, you are unlikely to pay much attention to it. Amdahl's law says that speeding up one part of a system has an impact proportional to that part's contribution. Similarly, the benefit to getting the architecture right is proportional to its contribution to overall system risk.

Presumptive architectures

People used to say that nobody ever got fired for buying IBM, meaning that since IBM mainframe systems dominated the market, choosing those systems was assumed to be reasonable. Many domains have an architecture that dominates the same way that IBM mainframes once did. For example, IT systems are often designed in tiers, with one tier handling

the user interface, another handling the business processing logic, and another storing data to a database. Another example is the use of cooperating processes in an operating system.

These are examples of *presumptive architectures* because developers in those domains may not even seriously consider other architectures so long as their project appears similar to previous ones.

Presumptive architectures work out because the architecture is a good match for the risks commonly found in the domain. IT systems often face concurrent access to shared data, shifting business rules, and long-lived data. An N-tiered system is a good match for those problems.

Presumptive architectures are similar to reference architectures. A *reference architecture* is a specification that describes an architectural solution to a problem. You can find reference architectures for high-reliability embedded systems or for using a particular vendor's technology to build web-based systems.

2.6 How should software architecture be used?

Corresponding to how important architecture is to your project, you can choose from three approaches to architecture. These three approaches cover a gradient of how much attention you pay to the architecture. These approaches are technical choices that are mostly independent of software development process.

- **Architecture-indifferent design.** If architecture is not much of a concern, you may apply architecture-indifferent design, where either the architecture simply emerges without deliberate choice, or a presumptive architecture is used.
- **Architecture-centric design.** If architecture is a concern, you may apply architecture-centric design, where you choose the architecture deliberately to aid in reducing failure risks and achieving desired functionality and quality.
- **Architecture hoisting.** You can go a step further and apply architecture hoisting, where the architecture directly owns, manages, or guarantees a feature, property, or quality attribute.

If you have not yet already, you probably want to be moving away from architecture-indifferent design and towards architecture-centric design. However, stopping at architecture-centric design is the right choice for many projects. Architecture hoisting is a great technique but is not necessary on every project. The following sections discuss each approach in more detail.

2.7 Architecture-indifferent design

The defining characteristic of *architecture-indifferent design* is that the developers *do not* consciously depend on the architecture to reduce risks, achieve features, or ensure qualities. The architecture will influence risks, features, and qualities, but the developers may have

been indifferent to it, simply copied the architecture from their previous project, used the presumptive architecture in their domain, or followed a corporate standard.

Every system has an architecture, whether it is deliberately chosen or not. Following this approach still yields an architecture, just not one that has been chosen to line up with the project's risks. Developers may employ architectural techniques from time to time, but reasoning is generally done from concrete items such as objects and classes, rather than architectural abstractions such as components and connectors. Developers generally will not consider changing the architecture to achieve their goals and instead change localized parts of the system. An architecture is there but it is mostly ignored.

Indifference to the architecture does not mean that the architecture is unsuitable, only that an opportunity to choose a suitable architecture was passed up. Benefits provided by the architecture are accidental. When the architecture is unsuitable, the developers will struggle against it but they may succeed if they are diligent and resourceful.

Architecture-indifferent design is acceptable when the architecture risk is low. Standalone systems with few challenging requirements are relatively low risk, surprisingly common, and easy to build without focusing on architecture. Systems that follow presumptive architectures usually succeed.

Mature and powerful off-the-shelf connectors and components, such as service buses and relational databases, mitigate against the risks that accompany architecture indifferent design. They handle difficult problems such as concurrency or scalability that would otherwise require architectural planning by developers. These same factors also contribute to the ability of developers to evolve a system without more careful architectural design.

Architecture indifferent design has several drawbacks. A system that starts out with a suitable architecture can be evolved into an unsuitable one by a team of developers lacking a shared architectural vision. For example, developers may try to speed up the system through various local and unprincipled changes. Over time, the complexity of the system will rise, perhaps beyond the ability of the developers to effectively maintain it.

Difficulties may arise later when the developers want to analyze the system for performance, security, modifiability, or some other quality. Analysis works best when a model is simple, and an architecture-indifferent approach may yield a complex system with lots of local exceptions to rules. Furthermore, the architecture was not chosen deliberately so it may not lend itself to any particular analysis. Testing is used as a stand-in to analysis, but it cannot be used to convince you that all responses happen in 10ms, data never leaks across insecure channels, or your protocol cannot become stuck.

2.8 Architecture-centric design

The defining characteristic of *architecture-centric design* is that the developers consciously depend on the architecture to reduce risks, achieve features, or ensure qualities. The mildest form of this is choosing an architecture that is compatible with what the system must do and the qualities it must possess. A higher standard is to design the architecture such that it ensures a desired property holds, or is known to be achievable or analyzable. All software

architecture books assume that you are using this approach.

Many successful developers follow architecture-centric design even if they do not realize it. If your system needs to acquire locks, you probably follow an ordering convention to avoid deadlocks. If your system has no garbage collection yet needs to avoid memory leaks, it may have a standard for how memory is freed to prevent leaks, such as freeing memory based on module scope. If your system uses a cache, it probably has access restrictions to ensure that the cache coherency is maintained. These are all invariants across your system that are designed to achieve architectural qualities.

Architecture-centric design usually means embracing architectural abstractions like components and connectors, if only because they yield a model that is easier to reason about because it is smaller. Abstractions also reveal the essence of a problem more clearly. For example, it is evident that components running in their own threads require thread-safe connectors, and distributed components cannot assume references will be in the same memory space, but these observations would be obscured if you were only looking at the source code.

Architecture-centric design means you must be on the lookout for requirements that will impact the architecture, and these requirements are rarely stated clearly. They may be hidden in a cryptic statement from a stakeholder or be common to other systems in your domain. When you recognize one of these, you should be asking yourself how your system will do that, and if it is something to be pushed into the architecture.

Your system will always have an architecture, and when you choose architecture-centric design, you are choosing to pay attention to it. Paying attention to the architecture does not necessarily mean documenting it. In big projects, not documenting the architecture is a big risk. In a startup company where all three developers live in the same garage, documenting the architecture is a much lower priority.

Architecture-centric design is compatible with any software development process. There is a temptation to assume a waterfall process with architecture design up front, but design of the architecture is just another engineering task like designing modules, objects, or data structures. It is easier if you know early what the architecture needs to do, because it may be hard to change written code, but this is also true of the choice in programming languages, interfaces, and frameworks.

Some features or qualities will be obvious from the start, like security if you are working at a bank. Others can be discovered as design and implementation progresses. You have the option of doing Big Design Up Front, or you could learn by *planning to throw one away* as Fred Brooks recommends (Brooks, 1995), or you could look for *architecture smells* that signal problems to be refactored in the next iteration. Regardless of which development process you choose, you will need to decide how much attention to place on architecture.

2.9 Architecture hoisting

Following architecture-centric design means that you are using the architecture to ensure a property or to achieve a quality attribute. Those properties or qualities often emerge from

the architecture, meaning you cannot put your finger on any one part and say that it is responsible. Looking at the source code, architecture-centric design can be noticed only indirectly since there is no one part you can point to that localizes, for example, scalability or concurrency.

Architecture hoisting is a kind of architecture-centric design where the architecture directly owns, manages, or guarantees a feature, property, or quality attribute. With architecture hoisting you can point to code that localizes scalability or concurrency. This usually entails writing the application code separately from the architecture code, so the architecture has a tangible existence in the source code.

Hoisting features is commonplace. Imagine that your system must log all access to a resource. You could require that everyone using the resource should log their access, but if someone forgets then the invariant is broken. Alternately, you could encapsulate the resource behind an API (Application Programming Interface) that logs all accesses. Doing so ensures that the logging invariant is satisfied and the invariant has been hoisted into the architecture.

Hoisting properties or quality attributes is less common but there are some mainstream examples. An application server, such as those used for web applications, is software that handles runtime qualities of an application. The application server handles running many copies of an application on a single machine (hoisting concurrency) or even spreading out the copies across multiple machines (hoisting scaling). An Enterprise Java Bean (EJB) application server hoists concurrency, scalability, and persistence. In fact, an EJB application server can be seen as an improved Servlet application server that is more effective at hoisting concurrency problems. The Eclipse framework hoists many features, properties, and qualities, such as resource management, concurrency, and platform independence.

When properties or quality attributes are hoisted, the application must adhere to some constraints in order to work within the architecture. For example, EJB disallows applications from starting their own threads or writing to local disk. These restrictions make sense, since it would be difficult for EJB to handle concurrency when applications created their own threads, or move applications between servers when they have data on a local disk.

Architecture hoisting usually involves tradeoffs. Automatic garbage collection hoists memory management but can make achieving performance targets difficult. Domain-specific concurrency patterns may be more efficient than a hoisted general-purpose mechanism.

Architecture hoisting can be seen as a kind of tyranny over developers, burdening them with additional constraints and bureaucracy, or it can be seen as liberation for developers, freeing them to focus on functionality instead of quality attributes. Hoisting is just a mechanism and can be used appropriately or not. It is effective when the system design requires quality attributes but achieving them would be a burden to developers. Often developers may be experts in the domain but not on how to ensure a quality like security or performance, so hoisting can enable experts to work within their specialty.

2.10 Architecture in large organizations

This book is *not* about what software development process to follow, how to be an architect, or how to structure the software development roles inside your organization. Consequently, it refers to software engineers as *developers*, not differentiating architects from programmers.

However, software development within large organizations brings its own challenges as a result of scale. Large companies and organizations divide themselves into divisions, departments, and teams. They introduce roles and assign responsibilities. While there are better and worse ways to organize a company, none is perfect. You should be aware that any way of dividing a company will solve some problems while creating others. Furthermore, you should understand why, for example, enterprise architects cannot apply architecture-indifferent design.

A common organization pattern in large companies is to give one group responsibility for cultivating the architecture that spans applications. This job is often called *enterprise architecture*. It gives rise to two job roles: *enterprise architects* and *application architects*.

- **Enterprise architect.** Enterprise architects are developers who are responsible for many applications. Enterprise architects do not control the functionality of any one application. Instead, they design an ecosystem inside which individual applications contribute to the overall enterprise. How well the enterprise architects cultivate the ecosystem will help or hinder the enterprise to achieve its goals, usually things like integrating applications, enabling variability across regions or markets, and standardizing deployment environments. Enterprise architects are like *movie producers* in that they influence the outcome only indirectly. Since they cannot directly influence qualities in the software, i.e., they cannot write code or design individual applications, enterprise architects apply architecture-centric design or architecture hoisting. Enterprise architects constrain the application architects by choosing architectures and constraints with the intent of achieving their desired qualities and goals.
- **Application architect.** Application architects are developers who are responsible for a single application. It is possible for them to understand and manage thousands of objects that comprise their application. Application architects are like movie directors whose daily actions create the shape of the product. Application architects can be successful in using an architecture-indifferent approach because they design an application's functionality in addition to its architecture. They can also apply architecture-centric design for their application, or architecture hoisting.

The separation of enterprise architecture from application architecture helps a company avoid the heterogeneity and the resulting chaos that would invariably happen without a deliberate effort to standardize. This benefit comes with some costs, including:

- **Reporting structure.** Programmers and enterprise architects rarely report to the same boss, which means their priorities will be different. Conflicts invariably arise, often regarding schedules, integration, architectural constraints, and platforms.

- **Working in silos.** Enterprise architects may over-constrain the architecture because they do not fully understand the needs of individual applications. Programmers may undervalue the benefit of standardizing across applications, believing their application should be exempted from onerous enterprise architecture constraints.

Since no organization structure is without flaws, the best you can hope for is to understand the tradeoffs and anticipate the problems. Knowing why enterprise architecture groups exist separately from development and knowing the kinds of trouble that can arise means that everyone can watch out for early warning signs and work to mitigate them.

Technically, the best strategy is to spread architecture knowledge around, as will be discussed more in Section 5.8. Having a separate enterprise architect group is not a bad idea, but its chance of success is higher if all developers understand core architecture principles, understand that architectural constraints exist in order to achieve goals and qualities, and understand the specific architecture they are building on.

2.11 Conclusion

Software architecture is a kind of design that deals with the large scale decisions, but drawing a clear boundary between architecture and design is difficult. Every system has an architecture, just as every system has a design, but that architecture may or may not have been deliberately chosen.

Treating software architecture as a skeleton can be a helpful, but imperfect, metaphor. The architecture enables or inhibits the system from achieving desired functionality or qualities in the same way that skeletons dictate how animals can behave. Birds are good at flying because their skeleton enables it, yet deer are surprisingly good jumpers despite their weight. Similarly, hardworking developers can push an inappropriate architecture to handle a load it is ill suited to support, but they will have an easier time when the architecture is suitable.

Architecture is just one of the many tasks competing for your scarce attention, so it would be good to know how much attention it deserves. When your systems are small or low-risk, architecture is less important because the chance or impact of failure is low. However, architecture is important on systems where the solution space is small, the risk of failure is high, or the desired quality attributes are difficult to achieve. Generally speaking, you should pay as much attention to architecture as it contributes risk to the overall project, since if there is little architecture risk then optimizing it only helps a little.

When architecture is less important to you, you can follow *architecture-indifferent design*, meaning that you focus on local changes to achieve the goals of your system. You do not ask the architecture to shoulder any burdens and may use a *presumptive architecture* by default. In contrast, when the architecture is more important to you, you can follow *architecture-centric design*. You would deliberately choose an architecture that is suited to the demands of your project, perhaps an architecture that makes scalability or modifiability easier to achieve. You can go further and *hoist* problems into the architecture, for example,

letting an application server handle concurrency problems, or a garbage collector handle memory management.

2.12 Further reading

The idea of using models to solve problems is central to all engineering. Our use of the commuting diagram to emphasize this is from Mary Shaw (Shaw and Garlan, 1996).

Some people asked to build an architecture model will have an easier time of it than others, perhaps like some people have a particular facility for art or mathematics. My experience with architectural skills transfer shows a wide range of aptitude across software developers (Fairbanks, 2003). If you find yourself overseeing the training or skills transfer involving models, keep these three levels in mind when choosing a curriculum.

The term *architecture hoisting* originated with NASA Mission Data Systems (MDS) developers, including Daniel Dvorak, Kirk Reinholtz, Nicholas Rouquette, and Kenny Meyer (Meyer, 2009). Their use of the term was meant to emphasize how existing space systems code could obscure details about, for example, the spacecraft position or velocity. In their usage, architecture hoisting was making important things visible in the architecture, including essential state variables and emergent behavior such as scheduling. Over time, I have come to adopt the definition presented in this chapter, which is consistent with their original intent.

This chapter refers to one of computer science's famous laws, Amdahl's law (Amdahl, 1967). Other famous laws include Brooks' law, "adding manpower to a late software project makes it later" (Brooks, 1995) and Conway's law, "any organization that designs a system ... will inevitably produce a design whose structure is a copy of the organization's communication structure." (Conway, 1968).

The term *software architecture* has been defined many times. There are several flavors of definitions, but you should know about the two most popular ones. An example of the first flavor is the one presented in the chapter from the SEI, which says architecture is about the structure of the elements and their relationships (Bass, Clements and Kazman, 2003). The second flavor is discussed by Martin Fowler and Ralph Johnson, who say "Architecture is the set of design decisions that must be made early in a project" (Fowler, 2003b). This is also known informally as the "stuff that's hard to change later" definition. Notice that this definition does not constrain what the decisions or stuff is, so it could include things like your choice of programming language.

This book avoids discussing how architects must work within organizations because other books already do a good job at this, including (Bass, Clements and Kazman, 2003) and (Lattanze, 2008). For a business management view on what software architecture provides to the bottom line, (Ross, Weill and Robertson, 2006) discusses a conceptual framework for how architecture strategy should be aligned with business strategy. The financial benefits of architecture are discussed in (Maranzano, 2005) and (Boehm and Turner, 2003).

Enterprise architecture is a large field into itself. Jeanne Ross, Peter Weill, and David Robertson do a good job of showing how business strategy should coordinate with software

architecture (Ross, Weill and Robertson, 2006). Martin Fowler's book is the best place to look for the standard patterns of enterprise architecture (Fowler, 2002). Several conceptual models exist for enterprise architecture, often called *enterprise architecture frameworks*, including The Open Group Architecture Framework (TOGAF) (The Open Group, 2008), Department of Defense Architecture Framework (DoDAF) (Wisnosky, 2004), and the Zachman Framework (Zachman, 1987).

Academic results on software architecture are generally reported in conferences and workshops. Ones to watch include:

- **WICSA / ECSA**: Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA, 2009)
- **SHARK**: SHARing and Reusing architectural Knowledge (SHARK, 2009)
- **ICSE**: International Conference on Software Engineering (ICSE, 2009)
- **OOPSLA**: International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA, 2009).

Additionally, the Software Engineering Institute (SEI) website frequently publishes technical reports (SEI Library, 2009).