

---

## Chapter 3

# Risk-Centric Architecture

---

We have now seen how to build architecture models, but seeing the number of models and the effort required raises more questions. How do I balance working on architecture with building the system? How do I know which techniques to use? How does the engineering job of architecture interact with the management job of project management? How could I use architecture techniques on agile software development projects that are focused on functionality? The short answer to these questions is to focus on risks. In this chapter, we will examine how risk reduction is central to all engineering disciplines, learn how to choose techniques to reduce risks, understand how engineering risks interact with management risks within a development process, and learn how we can use architecture-centric design to reduce risks.

Many software developers focus on the constructive, positive aspects of building software, since through their work they have created something that did not exist before. Building a successful software also means anticipating possible failures and avoiding designs that could fail. Henry Petroski, a leading historian of engineering, says, “Every solution of every design problem begins, no matter how tacitly, with a conception of how to obviate failure in all its possible and potential manifestations.” [Pet94]

As they strive to build successful software, developers are choosing from alternate designs, discarding those that are doomed to fail, and preferring options with low risk. When the risks are low it is easy to plow ahead without much thought, but, invariably, challenging design problems emerge and developers must grapple with high-risk designs, ones they are not sure will work.

Engineers use specialized techniques to help them avoid failure. If a mechanical engineer needs a beam to support a load, he can use techniques to calculate the stresses and strains, look up material properties in tables, and design a beam that will succeed. Some of these engineers are virtuosos and they can build remarkable systems that other engineers cannot. They have a deep intuition about designing systems that is partly innate and comes partly from experience. Over time, we learn what the virtuosos know intuitively, and we encode this knowledge as techniques, which enables all engineers to perform at a level that

earlier required a virtuoso [SG96].

Software developers are engineers and as such they build systems that succeed by avoiding risks of failure. Engineers in other fields have written down which techniques to mitigate risks, and ideally we would do the same in software engineering. Our field is newer and less well understood than other engineering fields, but many engineering risks have already been identified along with techniques to address them. The bad news is twofold: Each project presents unexpected or unique risks, and not every risk has a corresponding technique. Some subfields of software development, such as compilers, have become well understood and resemble mature engineering fields. For the rest, software developers learn what they can from books and hone their intuition through experience. This chapter helps explain how to apply techniques to mitigate risks..

### 3.1 Engineering risks and techniques

Software developers worry that their systems will fail: “I’m afraid that the server will not scale to 100 users”, “Parsing of the response messages may not be robust”, “It’s working now but if we touch anything it may fall apart.” These are examples of *engineering risks*, risks that are in the domain of the engineering of the system. Engineering risks are risks related to the analysis, design, and implementation of the product.

We contrast engineering risks with *project management risks*, which relate to schedules, sequencing of work, delivery, team size, geography, etc. Examples of these include: “Lead developer hit by bus”, “Customer needs not understood”, or “Senior VP hates our manager”. We must separate engineering risks from other risks because only engineering techniques will mitigate engineering risks. You cannot use a PERT chart to reduce the chance of buffer overruns, nor will UML resolve stakeholder disagreements.

In the context of engineering, a risk is the chance of failure times the impact of that failure. Both the probability of failure and the impact are difficult to know or estimate precisely, and so are uncertain. It is therefore easier to bundle the concept of uncertainty into our definition of risk, rather than talking about perceived risks. Our definition of risk then becomes the perceived probability of failure times the perceived impact.

A result of this definition is that a perceived risk can exist even if your system is flawless. Imagine a concurrent program that, by chance, has no race conditions. Since the developer does not know the program is flawless, he should be concerned about the risk of concurrency failure. Once the developer analyzes the program and discovers that it is good, his perception of the risk goes down. So by applying techniques we can reduce the amount of uncertainty, and therefore the amount of risk.

Not all risks are equally large, so they can be *prioritized*. Formal procedures exist for cataloging and prioritizing risks using risk matrices, including a US military standard MIL-STD-882D. Formal prioritization of risks is appropriate if your system, for example, handles radioactive material, but most computer systems can be less formal.

Engineering risks can be subdivided into *functional risks*, which relate to the expected functions of the system, and *quality attribute risks*, which relate to properties such as speed, modifiability, or security. The choice of a system’s software architecture will strongly influence its quality attribute risks.

*Engineering techniques* are activities performed by engineers before deployment in order to build a system that does what it should. Examples from other branches of engineering

include stress calculations, breaking point tests, thermal analysis, and reliability testing. In software engineering, examples of techniques include building models, applying design patterns, testing, and analyzing behavior. Techniques help engineers ensure that what they build solves the problem at hand.

Techniques exist on a spectrum from pure analyses, like calculating stresses, to pure solutions, like using a flying buttress on a cathedral. Other software architecture and design books have inventoried techniques on the solution-end of the spectrum, and call these techniques *tactics* [BCK03] or *patterns* [SSRB00, GHJV95], and include such solutions as using a process monitor, a forwarder-receiver, or a model-view-controller. The techniques we will discuss in this book are on the analysis-end of the spectrum, and are procedural and not problem-specific. These techniques include using models such as layer diagrams, component-and-connector models, and deployment models; analytic techniques for performance, security, and reliability; and architectural styles such as client-server and pipe-and-filter.

In a neat and orderly world, there would be a single technique to address every known risk. In practice, some risks can be mitigated by multiple techniques, while others require ad hoc thinking by developers. You should seek out opportunities to kill two birds with one stone by applying a single technique to mitigate two or more risks, or even think of it as a knapsack problem to choose a set of techniques that optimally mitigates your risks.

### 3.2 Mapping risks to techniques

Virtuosos can build extraordinary systems and succeed where others fail. There is an opportunity to improve the practice of software development by helping normal developers to perform like virtuosos. To do so, we need to make explicit what the virtuosos tacitly know and do. This includes knowing what risks to look for, knowing what techniques exist, employing techniques that are effective at reducing a given risk, and generalizing across a domain to describe its canonical risks.

Novice developers and the non-virtuosos would benefit from this knowledge, but they would not automatically become virtuosos themselves. Undoubtedly, virtuosos are extraordinarily effective for many other reasons, so making this tacit design knowledge explicit is only a partial solution, and is no *silver bullet* [Bro95].

In order to understand how we can make the tacit knowledge explicit, let's revisit the COTS component integration example from Chapter 6. There, we saw a walkthrough of integrating a third party Video Player component into a media player application.

The first bit of knowledge to make explicit is which risks to look for. In the example, we saw that architectural mismatch is a known risk when integrating components, and we have a list of the kinds of mismatch to look for. In this case, it is easy to see that the risks are engineering risks since they deal with the design itself, such as assumptions the Video Player component makes about threading and resource control. Although each project faces slightly different risks, we can generalize the kinds of risks faced by IT projects and see that they are different than those on Web projects. Section 16.5 provides sketches of the canonical risks faced in various domains. Checklists can remind developers to think about these risks (see Section 13.6).

The second bit of knowledge to make explicit is the set of engineering techniques. This book consolidates many techniques from various software architecture and design books.

These engineering techniques are described in the reference section of this book.

The third bit of knowledge to make explicit is the set of techniques that can be used to reduce the identified risks. One technique was to make the Video Player’s architectural assumptions clear to everyone, which we did with a diagram showing the Video Player component and annotations detailing its assumptions. We then created a design that applied the Adapter design pattern [GHJV95] to enable the Video Player to work within our system. To validate our design, we used a scenario to test our design on paper, then wrote prototype code to test our design in practice. Such knowledge fills in a template like this: “If you have <a risk>, a set of techniques that could reduce it is <techniques>” Section 16.3 contains a set of mappings from risks to techniques.

This approach may seem like common sense, but in practice you see developers using techniques that are inefficient or ineffective at reducing risks, or not using a technique that could help. Imagine a project whose primary worry is about buffer overruns leading to security problems. The developers follow the process decided by their corporation, and produce a module dependency diagram because that is what their corporation’s design document template says to do. There is no clear connection between the technique they have employed and the risk they face.

It should be clear that the techniques used should match the risks faced. Looking up a technique in a table may be effective, but it is intellectually unsatisfactory. The following sections describe some ideas that provide a conceptual foundation for the risk-technique table, and some generalizations that would help us know what kinds of techniques would be reasonable choices even before we consulted the table.

### Problems to find and prove

In his book *How to Solve It*, George Polya [Pol04] identifies two distinct kinds of math problems: problems to find and problems to prove. The problem, “Is there a number that when squared equals 4?” is a problem to find, and you can test your proposed answer easily. “Is the set of prime numbers infinite?” is a problem to prove. Finding things tends to be easier than proving things because for proofs you need to demonstrate something is true in all possible cases.

When searching for a technique to address a risk, we can often eliminate many possible techniques because they answer the wrong kind of Polya’s question. Some risks are specific; they can be tested with straightforward test cases. It is easy to imagine writing a test case for “Will the system support 100 simultaneous connections?” It is hard to imagine a small set of test cases providing persuasive evidence of “Does the system always conform to the framework APIs?” because there could be a case we have not yet seen, perhaps when a framework call unexpectedly passes us a null reference. Another example is deadlock: Any number of tests can run successfully without revealing a problem in the locking protocol.

### Analytic and analogic models

Michael Jackson, crediting Russel Ackoff, identifies *analogic models* and *analytic models* [Jac95, Jac00]. In an analogic model, each model element has an analogue in the domain of interest. A radar screen is an analogic model of some terrain, where blips on the screen correspond to airplanes – they are analogues. Analogic models support analysis only indi-

rectly, and usually domain knowledge and human reasoning are required. With the radar screen, you can ask “Are these planes on a collision course?” but to answer the question you are using your special purpose brainpower (see Section 4.3) in the same way that an outfielder can tell if he is in position to catch a fly ball.

An analytic (what Ackoff would call *symbolic*) model, by contrast, directly supports analysis of the domain. Mathematical equations are examples of analytic models, as are state machines. We could imagine an analytic model of the airplanes where each is represented by a vector. With a mathematical model we could quantitatively answer questions about collision courses.

When we model software, we invariably use symbols, whether they are UML elements or some other notation. We must be careful because some of our symbolic models support analytic reasoning while others support analogic reasoning, even though they use the same notation. A UML model of airplanes could represent them as classes with an attribute for the airplane’s vector, or it could omit this attribute. With the vector, we can reason symbolically, so it is an analytic model. Without the vector, it is an analogic model. So simply using a defined notation, like UML, does not guarantee that our models will be analytic. Academic *architecture description languages* (ADLs) are more constrained than UML with the intention of encouraging your architecture models to be analytic.

When you know what risks you want to mitigate, you can appropriately choose an analytic or analogic model. If you are concerned that your engineers may not understand the relationships between domain entities, you may build an analogic model in UML and confirm with domain experts that it is correct. Conversely, if you need a particular analysis, such as calculating response times, then you will want an analytic model.

### **Viewtype matching**

The effectiveness of some risk-technique pairings depends on the viewpoint. It is easiest to reason about modifiability from the module viewpoint, performance from the runtime viewpoint, and security from the deployment and module viewpoints.

Each view reveals selected details of a system. Reasoning about a risk works best when the view being used reveals details relevant to that risk. Despite this, developers are adaptable and will work with the resources they have, and will mentally simulate the other viewpoints. Developers usually have access to the source code, so they are quite adept at imagining the runtime behavior of the code, and where it will be deployed. Reasoning about a system’s state is easier with a state machine from the runtime viewpoint than it is from source code in the module viewpoint.

While a developer can make do with source code, reasoning will be easier when the risk and viewpoint are matched, and the view reveals details related to the risk.

### **Techniques with affinities**

In the physical world, we design tools for a purpose: hammers are for pounding nails, screwdrivers are for turning screws, saws are for cutting. We sometimes hammer a screw, or use a screwdriver as a pry bar, but the results are better when we use the tool that matches the job.

In software architecture, some techniques only go with particular risks because they were designed that way, and it is difficult to use them for another purpose. For example,

Rate Monotonic Analysis (see Section 13.2) primarily helps with reliability risks, Threat Modeling (see Section 13.1) primarily helps with security risks, and Queuing Theory (see Section 13.3) primarily helps with performance risks.

### 3.3 Risk and process

We now switch our attention from choosing engineering techniques to how to apply those techniques in the context of a software development process. As we broaden our focus from pure engineering to the development process, we find many more risks to worry about. Will the customer accept our system? Will the market have changed by the time we deliver? Will we deliver on time? Did our requirements reflect the customer's desires? Do we have the right people, are they doing the right jobs, and are they communicating effectively? Will there be lawsuits? And we worry that there are risks still unknown to us.

A *software development process* must balance both engineering and project management risks. It is tempting, but impossible, to cleanly separate engineering process from project management process. If the space shuttle were a pure engineering project, freed from project management pressure to deliver a final product, it would never launch because there would always be another risk to reduce through additional design and testing. It is the responsibility of the software development process to prioritize risks across both engineering and project management, and to decide that even though engineering risks still exist, other risks outweigh them.

Risks are the shared vocabulary between engineers and project managers. A manager's job is to understand tradeoffs and make decisions across the risks on a project. A manager may not be technical enough to understand why a module may not work as desired, but he will understand the risk of its failure, and the engineer can help him assess its probability and severity. The concept of a risk is the common ground between the world of engineering and the world of project management. Engineers may choose to ignore office politics and marketing meetings, and managers may choose to ignore the database schema and performance estimates, but in the idea of risks they find common ground to make decisions about the system.

#### Baked-in risks

If you had never seen a software development process before, you might imagine it was like a control loop in a program, where each iteration prioritizes the risks and plans out the next step accordingly, looping until the system is delivered. In practice, some risks are *baked-in* to the software development process rather than being evaluated continuously. The process at a large company worried about team coordination might insist on various forms of documentation at project milestones. Agile processes bake-in worries that the customer will reject the product, and consequently insist that the software be built in short iterations. IT-specific processes often face risks associated with unknown and complex domains, so their processes may bake in constructing domain models. Whenever I leave the house, I pat my pockets to ensure that I have my wallet and keys because it is enough of a risk to bake into my habits rather than pondering what I could be forgetting each time.

Baking risks into the software development process can be a blessing. It is a blessing when the process bakes-in risks that you would prioritize anyway, and saves you the time of every day deciding that, for example, you should stick to two-week iterations rather than

slipping the schedule. It is an efficient means of conveying expertise from experienced software developers, because they can point to successful results of following a process, rather than explaining their philosophy on software development that was baked-in. In an agile method such as XP, a team following the process can succeed even if they do not understand why XP chose that particular set of techniques.

Baking risks into the software development process can be a curse when you get it wrong. Many years ago, I interviewed with a tiny startup company. The project manager, formerly with IBM, asked me what I thought about process and I told him that it needed to be appropriate for the project, the domain, and the team. Above all else, I said, applying a process from a book, unaltered, was unlikely to work. He swiveled in his chair and picked up a book describing IBM's development process and said, "This is the process we will be following." Needless to say, I did not end up working there, but I wish I could have seen the five co-located engineers producing detailed design documents and other bureaucracy that are baked-in to processes for large, distributed teams. Some important features to consider include project complexity (big, small), team size (big, small), location (distributed, co-located), domain (IT, finance, systems, embedded, safety-critical, etc.), and kind of customer (internal, external, shrink-wrapped).

### Sketch of use in agile process

Existing books describe case studies of how architecture was used and documented on large software projects [BCK03, CBB<sup>+</sup>02]. This is not a book on software development process, but developers on smaller projects deserve a sketch of how architecture techniques could work on their projects.

Since agile projects vary in their process, let's assume one with a two-week *iteration* that plays a *planning game* to manage the *feature backlog*. On the engineering side, we have software architecture risks that we need to fold into this process, which includes identification, prioritization, mitigation, and evaluation of those risks. The big challenges are: how to address initial engineering risks, and how to incorporate engineering risks into the stack of work to do.

You will have identified some risks at the beginning of the project, such as the initial choices for architectural style, choice of frameworks, and choice of other COTS components. Some agile projects use an *iteration zero* [Sch04] to get their development environment set up, including source code control and automated build tools. No customer-visible functionality is delivered in iteration zero. We can piggyback here to start mitigating the identified risks. Developers could have a simple whiteboard meeting to ensure everyone agrees on an architectural style, or come up with a short list of styles to investigate. If performance characteristics of COTS components are unknown but important, some quick prototyping can be done to provide approximate speed or throughput numbers.

At the end of iteration zero, you need to evaluate how well your activities mitigated your risks. Most of the time you will have reduced the risk sufficiently that it drops off your radar, but sometimes not. Imagine that at the end of the iteration you have learned that prototyping shows that your preferred database will run too slowly. This is the beginning of a *risk backlog*. This risk must be written up as a testable feature for the system and added to the backlog.

It is challenging to fold engineering risk into a planning game to manage a backlog of features. Many agile projects divide the world into product owners, who create a priori-

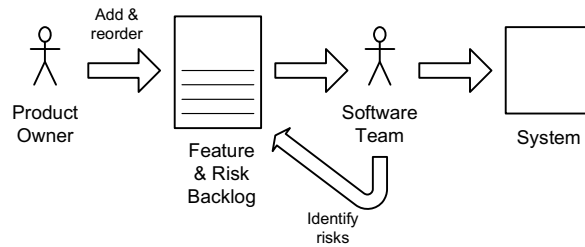


Figure 3.1: Feature and risk backlog

tized list of features called the backlog, and developers, who take features from the top of the backlog and build them. The world becomes more complex once we introduce risks, because we need to prioritize both features and risks. Some risks are small enough that they can be handled as they arise during an iteration, but larger risks will need to be scheduled just like features are. Whenever possible, risks should be written up as testable features.

Merging risk-based software development and agile processes is an open research area. Jaana Nyfjord's thesis [Nyf08] proposes the creation of a Risk Management Forum to prioritize risk across products and projects in an organization. Since our goal here is to handle architecture risks that are only a subset of all project risks, a smaller change to the process will likely work. If we give the product owner the additional responsibility to prioritize architectural risks alongside features, we can simply change the feature backlog into a feature & risk backlog, as seen in Figure 3.1. Software developers may see a feature low in the backlog asking for security. It is their job to educate the product owners that if they ever want to have a secure application, they need to address that risk early, since it will be difficult or impossible to add later. As part of the reflection at the end of each iteration, we need to evaluate architectural risks and feed these into the backlog.

In summary, we can handle architectural risks in an agile process by doing two things. Architectural risks that we know up-front can be handled in iteration zero, where no features are planned to be delivered. Small architectural risks can be handled as they arise during iterations, but large architectural risks must be promoted to be on par with features, and inserted into a combined feature & risk backlog.

### Rational forward progress

Imagine a software developer whose only concern is to minimize engineering risks. He would rationally choose to apply every applicable technique to minimize those engineering risks to build the best possible system. If the Wright brothers had minimized engineering risks, their first test flight might have been in 1953 instead of 1903.

The reason we do not always minimize engineering risks is because we must balance them with non-engineering risks, which are predominantly project management risks. If we minimize project management risks, we would seek out systems that were already delivered and did not cost anything. As a consequence, a software developer does not have the option to apply every useful technique because the time and cost to do so must be balanced against the reduction in risk.

We can phrase this intention as a goal: We want to make *rational forward progress*.

We want to be rational, in that we can justify why we do what we do. We do not want to waste time on unnecessary techniques, nor do we want to ignore project-threatening risks. We want to build successful systems by taking a path that spends our time most effectively. Our general rule is to insist that our engineering efforts are commensurate with the risk of failure. That means only applying techniques when they are motivated by risks.

It is harder to be rational than it appears at first glance. Let's say you successfully used the technique of domain modeling on your last project, so you decide to use it on this project. You find three flaws in your design, and fix them. You might become convinced that employing that technique was a good idea, because you otherwise would not have found the three flaws. But such reasoning ignores the opportunity cost. The fair comparison is against the other techniques you could have used. If your biggest risk is that your chosen framework might be inappropriate, you should spend your time analyzing that first. Your time is scarce, so you should choose techniques that are maximally effective at reducing the risk of failure, not just somewhat effective.

### **3.4 Reducing risk with architecture**

So far we have seen how individual risks can be mitigated with architectural techniques, but software architecture can also play a larger role. Every software system has an architecture, whether its designers intentionally chose it or not, and this architecture will strongly influence how well the system will scale up, how easy it will be to modify, and if it can be secured.

It is useful to consider the metaphor of the software architecture as the skeleton of a system. People walk on two legs because of their skeleton, while horses walk on four. A skeleton will support some loads and not others, and there are tradeoffs. Both a person and a horse can carry your apples to the market, but while horses are better for speed and heavy loads, they have difficulty climbing trees to pick apples. Note that there is not one choice that is best overall, but better or worse choices given what you want to do.

We often think of the software architecture as enabling or inhibiting qualities other than simply the required functionality. The skeletons of the person and the horse both support the main function, taking apples to market, but differ in their speed, throughput, and flexibility. These qualities that are not functions are called quality attributes, and we will discuss them again in Chapter 2.

It is important to recognize that we can mix-and-match architecture and functionality. That is, we could change a system's architecture yet keep its functionality, or use the same architecture on a system with different functionality. Architecture need not be something that just happens while we design the system to support its functionality. Architecture results from our design decisions, and these decisions have consequences.

Problems arise when the skeleton is a poor fit for the requirements. As we add more functionality, we may discover that the skeleton is failing us. When we focus exclusively on functionality, we can find ourselves fighting the architecture. Examples of this include trying to add security to a system that originally lacked it, or supporting exceptional behavior when the original system focused on the success path only.

Conversely, we can carefully choose or design the skeleton to achieve benefits that would be difficult to achieve otherwise. If we cannot anticipate how users will want to reconfigure our system, an architecture similar to Unix pipes and filters works well, since

users can pipe data through a novel configuration of filters of their choosing. The system can even be extended with new kinds of pipes and filters. This choice of architecture is aligned with our needs.

A flexible architecture based on pipes and filters is not always the best choice. Consider a weather forecasting system. You may have various weather models that you can write as filters and flexibly combine them to produce forecasts. But weather forecasts are only useful before the weather arrives. If your flexible system produces forecasts too slowly, then trading off performance for flexibility is a bad idea.

Each software project proceeds slightly differently, but we can generalize software developers' approach to architecture into two camps. The first is called *architecture-indifferent design*, where either the architecture simply emerges without deliberate choice, or the architecture is not chosen to help reduce risks of failure. The second is called *architecture-centric design*, where the architecture is chosen deliberately to aid in reducing failure risks and achieving desired functionality and quality. These approaches are a technical choice, and are mostly independent of software development process.

### **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. Every system has an architecture, and that architecture will influence risks, features, and qualities, but the developers may have been indifferent to it and simply copied the architecture from their previous project, used the standard architecture in their domain, or followed a corporate standard. Indifference to the architecture does not mean that the choice is inappropriate, only that an opportunity to choose a suitable architecture was passed up. Any benefit provided by the architecture is accidental, not intentional. It is possible that the architecture is mismatched with the project needs and the developers will struggle against it.

Following this path is not a complete rejection of architecture, and developers may employ architectural techniques from time to time. However, reasoning is done from concrete items such as objects and classes, rather than architectural abstractions such as components and connectors. Developers will not realize that they can change the architecture to achieve their goals and instead focus on changing localized parts of the system.

While it may seem doomed, an architecture-indifferent design can work well in some cases. Standalone systems with few challenging requirements are relatively low risk, surprisingly common, and easy to accomplish without focusing on architecture. Also, some domains have presumptive architectures, such as 3-tier systems in IT or communicating processes in operating systems, that are historically shown to match the risks from the domain. Developers that pay little attention to architecture but follow precedent often arrive at suitable architectures. The use of mature off-the-shelf connectors and components, such as service busses and relational databases, is helpful because they may 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 explicit up-front architectural design.

Even if your system is fortunate enough to be covered by the special cases above, there is reason to worry about failure. 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.

When the developers do need to analyze the system for performance, security, modifiability, or some other quality, it is more difficult to look only at objects instead of larger abstractions such as components. Analysis works best when a model is simple. Reasoning about performance, security, or protocols is easier with a handful of components than with hundreds of objects. The intention of analysis is often to show that the system will never get into a bad state, i.e., one of Polya's problems to prove. However, when the system is too complex to analyze, testing is used as a stand-in, 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.

### Architecture-centric design

The defining characteristic of architecture-indifferent 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.

We saw an example of architecture-centric design in Chapter 6 when we wanted to ensure that a crash of the third-party media player would not crash the whole system. We decided to run it in a separate process, which may have reduced performance somewhat, but made it possible to isolate the failure.

Most successful developers follow architecture-centric design even if they have not realized 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 to be 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. But abstractions also enable us to reason about the essence of a problem more clearly, for example: Components running in their own threads require thread-safe connectors, or distributed components cannot assume references will be in the same memory space.

Using architecture to reduce risks, achieve features, or ensure qualities means you must be on the lookout for requirements that will impact the architecture, and these requirements will not always be clearly stated. They may be hidden in a cryptic statement from a stakeholder or 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.

This book generally refers to software engineers as *developers* but within that broad category there are a few more specific roles. Developers who are responsible for a single application may be called *application architects* and those responsible for many applica-

tions may be called *enterprise architects*. Application architects can be successful in using an architecture-indifferent approach because they design an application's functionality in addition to its architecture. It is possible for them to understand and manage thousands of objects that comprise their application. Enterprise architects, however, do not control the functionality of any one application, and design an ecosystem in which individual applications contribute to the overall enterprise. Their choices will help or hinder the enterprise to integrate applications, enable variability across regions or markets, and standardize deployment environments.

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 projects must choose programming languages and frameworks early as well. Some qualities will be obvious from the start, like security if you are working at a bank. Others can be discovered during design reviews and quality attribute workshops. You have the option of doing Big Design Up Front, or you could learn by *planning to throw one away* as Fred Brooks recommends [Bro95], 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.

Your system will always have an architecture, and when you choose the architecture-centric design approach, you choose 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, not documenting the architecture is a much lower priority.

The kind of architecture documentation to use is a decision that sits on the junction between engineering and project management. The architecture document is technical in nature and helps to reduce the engineering risk that the members of the development team move in differing directions, and so it can be seen as engineering. But co-locating the development team and doing pair programming are alternate ways to ensure cohesive team actions, and these are closer to project management choices than engineering ones.

### 3.5 How much architecture?

An important question in software architecture, and one that we still do not have a good answer for is, "How much architecture is enough?" or simply, "When do we stop?" Time spent modeling or analyzing is time that could have been spent building, so we want to get the balance right. Ideally, we would have an objective, quantitative decision procedure, but we will settle for a well understood qualitative one.

We have empirical data to guide our decision. Barry Boehm has calculated the optimal amount of architecture for small, medium, and large projects based on a variant of his COCOMO model [BT03]. His data indicates that the largest projects should spend more time on architecture, as much as a third of the total time. Joseph Maranzano also has empirical data from telecommunications showing that architecture reviews save millions of dollars [Mar05].

This book suggests that, like other engineering fields, software engineering should fo-

cus on mitigating risks. We now know how to identify risks and have techniques ready to combat many of them. We are tempted work until all engineering risks have been eliminated, but the time spent on that must be balanced with other, non-engineering risks, such as pleasing the customer and delivering products. And since systems are large and complex, how do we really know that a risk has been eliminated?

So if focusing on risks cannot give us a qualitative, let alone quantitative decision procedure, have we gained anything? Yes, for two reasons. First, because we now have a shared vocabulary with project management so that engineering risks can be prioritized alongside other risks. Second, our evaluation ability has been refined. If you need directions to a restaurant, you'd rather have a map than not, even if the map is missing information like road construction and traffic delays. Knowing the risks facing your project and how well your design addresses them is much better than simply guessing you should spend 20% of your time on architecture. The broad question of "How much software architecture is enough?" has been transformed into "Have my chosen techniques sufficiently reduced my risk of failure?"

### 3.6 Summary

In this chapter we set out to understand how focusing on risks could help us to understand how to productively use software architecture. Avoiding failure is central to all engineering and we can use architecture techniques to mitigate the risks we identify. Not every risk has a corresponding architecture or design technique, but many do, and they are listed in the reference section in Chapter 16. The reasons why certain risks are mitigated by certain techniques become clear when we understand problems to find or prove, analytic and analogic models, the matching of viewtypes to risks, and the affinities of particular techniques. By making the connection between risks and techniques explicit we enable developers to rival the performance of virtuosos.

Engineering techniques address engineering risks, but projects face a wide variety of risks. Software development processes must mesh with our management process, and must prioritize both management risks and engineering risks. Development processes often bake-in certain risks that are common in a given domain or organization, which works well when the project indeed faces those risks, and works poorly when mismatched. Agile development processes can be adapted slightly to incorporate stories regarding architecture in addition to the normal user stories.

We cannot reduce engineering risks to zero, because there are other project management risks to consider, including time-to-market pressure. By applying the principle of rational forward progress, we ensure that we work on the highest priority risks and apply relevant techniques. Although this does not give us a simple answer about when to stop, it converts a "finger in the wind" kind of estimation technique into a qualitative evaluation of risk.

Two different systems can have the same functionality but different architectures, and the chosen architecture will influence how easy it is to achieve that functionality. Many projects can afford to be architecture-indifferent because simply choosing the domain's standard architecture style will result in an adequate architecture. More challenging risks will require architecture-centric design where the architecture is deliberately chosen to reduce risks.