Chapter 3

Risk-Driven Model

As they build successful software, software developers are choosing from alternate designs, discarding those that are doomed to fail, and preferring options with low risk of failure. 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.

Building successful software means anticipating possible failures and avoiding designs that could fail. Henry Petroski, a leading historian of engineering, says this about engineering as a whole:

> The concept of failure is central to the design process, and it is by thinking in terms of obviating failure that successful designs are achieved. ... Although often an implicit and tacit part of the methodology of design, failure considerations and proactive failure analysis are essential for achieving success. And it is precisely when such considerations and analyses are incorrect or incomplete that design errors are introduced and actual failures occur. (Petroski, 1994)

To address failure risks, the earliest software developers invented design techniques that helped them build successful software, such as domain modeling, security analyses, and encapsulation. Today, developers can choose from a huge number of design techniques. From this abundance, a hard question arises: *Which design and architecture techniques should developers use?*

If there were no deadlines then the answer would be easy: use all the techniques. But that is impractical because a hallmark of engineering is the *efficient* use of resources, including time. One of the risks developers face is that they waste too much time designing. So a related question arises: *How much design and architecture should developers do?*

There is much active debate about this question and several kinds of answers have been suggested:

- **No up-front design.** Developers should just write code. Design happens, but is coincident with coding, and happens at the keyboard rather than in advance.
- **Use a yardstick.** For example, developers should spend 10% of their time on architecture and design, 40% coding, 20% integrating, and 30% testing.
- **Build a documentation package.** Developers should employ a comprehensive set of design and documentation techniques sufficient to produce a complete written design document.
- **Ad hoc.** Developers should react to the project needs and decide on the spot how much design to do.

The ad hoc approach is perhaps the most common, but it is also subjective and provides no enduring lessons. Avoiding design altogether is impractical when failure risks are high, but so is building a complete documentation package when risks are low. Using a yardstick can help you plan how much effort designing the architecture will take, but it does not help you choose techniques.

This chapter introduces the *risk-driven model* of architectural design. The big idea is that the effort you spend on designing your software architecture should be commensurate with the risks faced by your project. When my father installed a new mailbox, he did not apply every mechanical engineering analysis and design technique he knew. Instead, he dug a hole, put in a post, and filled the hole with concrete. The risk-driven model can help you decide when to apply architecture techniques and when you can skip them.

Where a software development process orchestrates every activity from requirements to deployment, the risk-driven model only guides architectural design, and can therefore be used inside any software development process.

The risk-driven model is a reaction to a world where developers are under pressure to build high quality software quickly and at reasonable cost, yet those developers have more architecture techniques than they can afford to apply. The risk-driven model helps them answer the two questions above: how much software architecture work should they do and which techniques should they use? It is an approach that helps developers follow a middle path, one that avoids wasting time on techniques that help their projects only a little, but ensures that project-threatening risks are addressed by appropriate techniques.

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, and learn how we can balance planned design with evolutionary design. This chapter walks through the ideas that underpin the risk-driven model, but if you are the kind of person who would prefer to first see an example of it in use, you can flip ahead to Chapter 4.

3.1 What is the risk-driven model?

The *risk-driven model* guides developers to apply a minimal set of architecture techniques to reduce their most pressing risks. It suggests a relentless questioning process: "What are my risks? What are the best techniques to reduce them? Is the risk mitigated and can I start (or resume) coding?" The risk-driven model can be summarized in three steps:

1. Identify and prioritize risks
2. Select and apply a set of techniques
3. Evaluate risk reduction

Risk or feature focus. The key element of the risk-driven model is the promotion of risk to prominence. What you choose to promote has an impact. Most developers already think about risks, but they think about lots of other things too, and consequently risks can be overlooked. A recent paper described how a team that had previously done up-front architecture work switched to a purely feature-driven process. The team was so focused on delivering features that they deferred quality attribute concerns until after active development ceased and the system was in maintenance (Babar, 2009). The conclusion to draw is that teams that focus on features will pay less attention to other areas, including risks. Earlier studies have shown that even architects are less focused on risks and tradeoffs than one would expect (Clerc, Lago and van Vliet, 2007).

You do not want to waste time on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build successful systems by taking a path that spends your time most effectively. That means addressing risks by applying architecture and design techniques but only when they are motivated by risks.

Logical rationale. But what if your perception of risks differs from others' perceptions? Risk identification, risk prioritization, choice of techniques, and evaluation of risk mitigation will all vary depending on who does them. It may seem that under it all, the risk-driven model is merely improvisation.

Though different developers will perceive risks differently and consequently choose different techniques, the risk-driven model has the useful property that it yields arguments that can be evaluated. An example argument would take this form:

> We identified A, B, and C as the biggest risks, with B being primary. We spent time applying techniques X and Y because we believed they would help us reduce the risk of B. We evaluated the resulting design and decided that we had sufficiently mitigated the risk of B, so we proceeded on to coding.

You have answered the broad question, "How much software architecture should you do?" by providing a plan (i.e., the techniques to apply) based on the relevant context (i.e., the perceived risks).

Other developers might disagree with your assessment, so they could provide a differing argument with the same form, perhaps suggesting that risk D be included. A productive, engineering-based discussion of the risks and techniques can ensue because the rationale behind your opinion has been articulated and can be evaluated.

3.2 Are you risk-driven now?

Many developers believe that they already follow a risk-driven model, or something close to it. Yet there are telltale signs that many are not. One is an inability to list the risks they confront and the corresponding techniques they are applying.

Any developer can answer the question, "Which features are you working on?" but many have trouble with the question, "What are your primary failure risks and corresponding engineering techniques?" If risks were indeed primary then it would be an easy question to answer.

Technique choices should vary. Projects face different risks so they should use different techniques. Some projects will have tricky quality attribute requirements that need up-front planned design, while other projects are tweaks to existing systems and entail little risk of failure. Some development teams are distributed and so they document their designs for others to read, while other teams are co-located and can reduce this formality.

When developers fail to align their architecture activities with their risks, they will over-use or under-use architectural techniques, or both. Examining the overall context of software development suggests why this can occur. Most organizations guide developers to follow a process with some kind of documentation template or a list of design activities. These can be beneficial and effective, but they can also inadvertently steer developers astray.

Here are some examples of well-intentioned rules that guide developers to activities that may be mismatched with their project's risks.

- The team must always (or never) build full documentation for each system.
- The team must always (or never) build a class diagram, a layer diagram, etc.
- The team must spend 10% (or 0%) of the project time on architecture.

Such guidelines can be better than no guidance, but each project will face a different set of risks. It would be a great coincidence if the same set of diagrams or techniques were always the best way to mitigate a changing set of risks.

Example mismatch. Imagine a company that builds a 3-tier system. The first tier has the user interface, and is exposed to the internet. The biggest risks might be usability and security. The second and third tiers implement business rules and persistence; they are behind a firewall. The biggest risks might be throughput and scalability.

If this company followed the risk-driven model, the front-end and back-end developers would apply different architecture and design techniques to address their different risks. Instead, what often happens is that both teams follow the same company-standard process or template and produce, say, a module dependency diagram. The problem is that there is no connection between the techniques they use and the risks they face.

Standard processes or templates are not necessarily bad, but they are often used poorly. Over time, you may be able to generalize the risks on the projects at your company and devise a list of appropriate techniques. The important part is that the techniques match the risks.

The three steps to risk-driven software architecture are deceptively simple but the devil is in the details. What exactly are risks and techniques? How do you choose an appropriate set of techniques? And when do you stop architecting and start/resume building? The following sections dig into these questions in more detail.

3.3 Risks

In the context of engineering, *risk* is commonly defined as the chance of failure times the impact of that failure. Both the probability of failure and the impact are uncertain because they are difficult to measure precisely. You can sidestep the distinction between perceived risks and actual risks by bundling the concept of uncertainty into the definition of risk. The definition of risk then becomes:

$$\text{risk} = \text{perceived probability of failure} \times \text{perceived impact}$$

A result of this definition is that a risk can exist (i.e., you can perceive it) even if it does not exist. Imagine a hypothetical program that has no bugs. If you have never run the program or tested it, should you worry about it failing? That is, should you perceive a failure risk? Of course you should, but after you analyze and test it, you gain confidence in it and your perception of risk goes down. So by applying

Project management risks	Software engineering risks
"Lead developer hit by bus"	"The server may not scale to 1000 users"
"Customer needs not understood"	"Parsing of the response messages may be buggy"
"Senior VP hates our manager"	"The system is working now but if we touch anything it may fall apart"

Figure 3.1: Examples of project management and engineering risks. You should distinguish them because engineering techniques rarely solve management risks, and vice versa.

techniques you can reduce the amount of uncertainty, and therefore the amount of (perceived) risk. You can also under-appreciate or fail to perceive a risk, which we will return to shortly.

Describing risks. You can state a risk categorically, often as a quality attribute like "modifiability" or "reliability". But often this is too vague to be actionable: if you do something, are you sure that the categorical risk is actually reduced?

It is better to describe risks such that you can later test to see if they have been mitigated. Instead of just listing a quality attribute like reliability, describe each risk of failure as a testable *failure scenario*, such as "During peak loads, customers experience user interface latencies greater than five seconds."

Engineering and project management risks. Projects face many different kinds of risks so people working on a project tend to pay attention to the risks related to their specialty. For example, the sales team worries about a good sales strategy and software developers worry about a system's scalability. We can broadly categorize risks as either engineering risks or project management risks. *Engineering risks* are those risks related to the analysis, design, and implementation of the product. These engineering risks are in the domain of the engineering of the system. *Project management risks* relate to schedules, sequencing of work, delivery, team size, geography, etc. Figure 3.1 shows examples of both.

If you are a software developer, you are asked to mitigate engineering risks and you will be applying engineering techniques. The technique type must match the risk type, so only engineering techniques will mitigate engineering risks. For example, you cannot use a PERT chart (a project management technique) to reduce the chance of buffer overruns (an engineering risk), and Java will not resolve stakeholder disagreements.

Identifying risks. Experienced developers have an easy time identifying risks, but what can be done if the developer is less experienced or working in an unfamiliar

Project domain	Prototypical risks
Information Technology (IT)	Complex, poorly understood problem Unsure we're solving the real problem May choose wrong COTS software Integration with existing, poorly understood software Domain knowledge scattered across people Modifiability
Systems	Performance, reliability, size, security Concurrency Composition
Web	Security Application scalability Developer productivity / expressability

Figure 3.2: While each project can have a unique set of risks, it is possible to generalize by domain. *Prototypical risks* are ones that are common in a domain, and are a reason that software development practices vary by domain. For example, developers on Systems projects tend to use the highest performance languages.

domain? The easiest place to start is with the requirements, in whatever form they take, and look for things that seem difficult to achieve. Misunderstood or incomplete quality attribute requirements are a common risk. You can use Quality Attribute Workshops (see Section 15.6), a Taxonomy-Based Questionnaire (Carr et al., 1993), or similar, to elicit risks and produce a prioritized list of failure scenarios.

Even with diligence, you will not be able to identify every risk. When I was a child, my parents taught me to look both ways before crossing the street because they identified cars as a risk. It would have been equally bad if I had been hit by a car or by a falling meteor, but they put their attention on the foreseen and high priority risk. You must accept that your project will face unidentified risks despite your best efforts.

Prototypical risks. After you have worked in a domain for a while, you will notice *prototypical risks* that are common to most projects in that domain. For example, Systems projects usually worry more about performance than IT projects and Web projects worry about security. Prototypical risks may have been encoded as *checklists* describing historical problem areas, perhaps generated from architecture reviews. These checklists (see Section 15.6) are valuable knowledge for less experienced developers and a helpful reminder for experienced ones.

Knowing the prototypical risks in your domain is a big benefit, but even more important is realizing when your project differs from the norm so that you avoid blind spots. For example, software that runs a hospital might most closely resemble an IT project, with its integration concerns and complex domain types. However, a system that takes 10 minutes to reboot after a power failure is usually a minor risk for an IT project, but a major risk at a hospital.

Prioritizing risks. Not all risks are equally large, so they can be *prioritized*. Most development teams will prioritize risks by discussing it amongst themselves. This can be adequate, but the team's perception of risks may not be the same as the stakeholders' perception. If your team is spending enough time on software architecture for it to be noticeable in your budget, it is best to validate that time and money are being spent in accordance with stakeholder priorities.

Risks can be categorized[1] on two dimensions: their priority to stakeholders and their perceived difficulty by developers. Be aware that some technical risks, such as platform choices, cannot be easily assessed by stakeholders.

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.

3.4 Techniques

Once you know what risks you are facing, you can apply *techniques* that you expect to reduce the risk. The term technique is quite broad, so we will focus specifically on *software engineering risk reduction techniques*, but for convenience continue to use the simple name *technique*. Figure 3.3 shows a short list of software engineering techniques and techniques from other engineering branches.

Spectrum from analyses to solutions. Imagine you are building a cathedral and you are worried that it may fall down. You could build models of various design alternatives and calculate their stresses and strains. Alternately, you could apply a known solution, such as using a flying buttress. Both work, but the former approach has an analytical character while the latter has a known-good solution character.

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* (Bass, Clements and Kazman, 2003) or *patterns* (Schmidt et al., 2000; Gamma et al., 1995), and include such solutions as using a process monitor, a forwarder-receiver, or a model-view-controller.

[1] This is the same categorization technique used in ATAM to prioritize architecture drivers and quality attribute scenarios, as discussed in Section 12.11.

Software engineering	**Other engineering**
Applying design or architecture pattern	Stress calculations
Domain modeling	Breaking point test
Throughput modeling	Thermal analysis
Security analysis	Reliability testing
Prototyping	Prototyping

Figure 3.3: A few examples of engineering risk reduction techniques in software engineering and other fields. Modeling is commonplace in all engineering fields.

The risk-driven model focuses on techniques that are on the analysis-end of the spectrum, ones that are procedural and independent of the problem domain. These techniques include using models such as layer diagrams, component assembly models, and deployment models; applying analytic techniques for performance, security, and reliability; and leveraging architectural styles such as client-server and pipe-and-filter to achieve an emergent quality.

Techniques mitigate risks. Design is a mysterious process, where virtuosos can make leaps of reasoning between problems and solutions (Shaw and Garlan, 1996). For your process to be repeatable, however, you need to make explicit what the virtuosos are doing tacitly. In this case, you need to be able to explicitly state how to choose techniques in response to risks. Today, this knowledge is mostly informal, but we can aspire to creating a handbook that would help us make informed decisions. It would be filled with entries that look like this:

If you have <a risk>, consider <a technique> to reduce it.

Such a handbook would improve the repeatability of designing software architectures by encoding the knowledge of virtuoso architects as mappings between risks and techniques.

Any particular technique is good at reducing some risks but not others. 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 risks require you to invent techniques on the fly.

This frame of mind where you choose techniques based on risks helps you to work efficiently. You do not want to waste time (or other resources) on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build successful systems by taking a path that spends your time most effectively. That means only applying techniques when they are motivated by risks.

Optimal basket of techniques. To avoid wasting your time and money, you should choose techniques that best reduce your prioritized list of risks. You should seek out opportunities to kill two birds with one stone by applying a single technique to mitigate two or more risks. You might like to think of it as an *optimization problem* to choose a set of techniques that optimally mitigates your risks.

It is harder to decide which techniques should be applied than it appears at first glance. Every technique does something valuable, just not the valuable thing your project needs. For example, there are techniques for improving the usability of your user interfaces. Imagine you successfully used such techniques on your last project, so you choose it again on your current project. You find three usability flaws in your design, and fix them. Does this mean that employing the usability technique was a good idea?

Not necessarily, because 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 is inappropriate, you should spend your time analyzing or prototyping your framework choice instead of on usability. Your time is scarce, so you should choose techniques that are maximally effective at reducing your failure risks, not just somewhat effective.

Cannot eliminate engineering risk. Perhaps you are wondering why we should try to create an optimal basket of techniques when we should go all the way and eliminate engineering risk. It is tempting, since engineers hate ignoring risks, especially those they know how to address.

The downside of trying to eliminate engineering risk is *time*. As aviation pioneers, the Wright brothers spent time on mathematical and empirical investigations into aeronautical principles and thus reduced their engineering risk. But, if they had continued these investigations until risks were eliminated, their first test flight might have been in 1953 instead of 1903.

The reason you cannot afford to eliminate engineering risk is because you must balance it with non-engineering risk, which is predominantly project management risk. Consequently, a software developer does not have the option to apply every useful technique because risk reductions must be balanced against time and cost.

3.5 Guidance on choosing techniques

So far, you have been introduced to the risk-driven model and have been advised to choose techniques based on your risks. You should be wondering how to make good choices. In the future, perhaps a developer choosing techniques will act much like a mechanical engineer who chooses materials by referencing tables of properties and making quantitative decisions. For now, such tables do not exist. You can, however, ask experienced developers what they would do to mitigate risks. That is, you would choose techniques based on their experience and your own.

However, if you are curious, you would be dissatisfied either with a table or a collection of advice from software veterans. Surely there must be principles that underlie any table or any veteran's experience, principles that explain why technique X works to mitigate risk Y.

Such principles do exist and we will now take a look at some important ones. Here is a brief preview. First, sometimes you have a problem to *find* while other times you have a problem to *prove*, and your technique choice should match that need. Second, some problems can be solved with an *analogic* model while others require an *analytic* model, so you will need to differentiate these kinds of models. Third, it may only be efficient to analyze a problem using a particular type of model. And finally, some techniques have *affinities*, like pounding is suitable for nails and twisting is suitable for screws.

Problems to find and prove. In his book *How to Solve It*, George Polya identifies two distinct kinds of math problems: problems to *find* and problems to *prove* (Polya, 2004). The problem, "Is there a number that when squared equals 4?" is a problem to find, and you can test your proposed answer easily. On the other hand, "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, you can often eliminate many possible techniques because they answer the wrong kind of Polya question. Some risks are specific so they can be tested with straightforward test cases. It is easy to imagine writing a test case for "Can the database hold names up to 100 characters?" since it is a problem to find. Similarly, you may need to design a scalable website. This is also a problem to find because you only need to design (i.e., find) one solution, not demonstrate that your design is optimal.

Conversely, it is hard to imagine a small set of test cases providing persuasive evidence when you have a problem to prove. Consider, "Does the system always conform to the framework Application Programming Interface (API)?" Your tests could succeed, but there could be a case you have not yet seen, perhaps when a framework call unexpectedly passes a null reference. Another example of a problem to prove is deadlock: Any number of tests can run successfully without revealing a problem in a locking protocol.

Analytic and analogic models. Michael Jackson, crediting Russell Ackoff, distinguishes between *analogic models* and *analytic models* (Jackson, 1995; Jackson, 2000). 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 — the blip and the airplane are analogues.

Analogic models support analysis only indirectly, and usually domain knowledge and human reasoning are required. A radar screen can help you answer the question, "Are these planes on a collision course?" but to do so you are using your special purpose brainpower in the same way that an outfielder can tell if he is in position to catch a fly ball (see Section 15.6).

An analytic (what Ackoff would call *symbolic*) model, by contrast, directly supports computational analysis. Mathematical equations are examples of analytic models, as are state machines. You could imagine an analytic model of the airplanes where each is represented by a vector. Mathematics provides an analytic capability to relate the vectors, so you could quantitatively answer questions about collision courses.

When you model software, you invariably use symbols, whether they are Unified Modeling Language (UML) elements or some other notation. You must be careful because some of those symbolic models support analytic reasoning while others support analogic reasoning, even when they use the same notation. For example, two different UML models could represent airplanes as classes, one with and one without an attribute for the airplane's vector. The UML model with the vector enables you to compute a collision course, so it is an analytic model. The UML model without the vector does not, so it is an analogic model. So simply using a defined notation, like UML, does not guarantee that your models will be analytic. *Architecture description languages* (ADLs) are more constrained than UML with the intention of nudging your architecture models to be analytic ones.

Whether a given model is analytic or analogic depends on the question you want it to answer. Either of the UML models could be used to count airplanes, for example, and so could be considered analytic models.

When you know what risks you want to mitigate, you can appropriately choose an analytic or analogic model. For example, 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 to calculate response time distributions, then you will want an analytic model.

Viewtype matching. The effectiveness of some risk-technique pairings depends on the type of model or view used. *Viewtypes* are not fully discussed until Section 9.6. For now, it is sufficient to know about the three primary viewtypes. The *module viewtype* includes tangible artifacts such as source code and classes; the *runtime viewtype* includes runtime structures like objects; and the *allocation viewtype* includes allocation elements like server rooms and hardware. It is easiest to reason about modifiability from the module viewtype, performance from the runtime viewtype, and security from the deployment and module viewtypes.

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. For example, reasoning about a runtime protocol is easier with a runtime view, perhaps a state machine, than with source code. Similarly, it is easier to reason about single points of failure using an allocation view than a module view.

Despite this, developers are adaptable and will work with the resources they have, and will mentally simulate the other viewtypes. For example, 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. While a developer can make do with source code, reasoning will be easier when the risk and viewtype are matched, and the view reveals details related to the risk.

Techniques with affinities. In the physical world, tools are designed for a purpose: hammers are for pounding nails, screwdrivers are for turning screws, saws are for cutting. You may sometimes hammer a screw, or use a screwdriver as a pry bar, but the results are better when you 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 primarily helps with reliability risks, threat modeling primarily helps with security risks, and queuing theory primarily helps with performance risks (these techniques are discussed in Section 15.6).

3.6 When to stop

The beginning of this chapter posed two questions. So far, this chapter has explored the first: Which design and architecture techniques should you use? The answer is to identify risks and choose techniques to combat them. The techniques best suited to one project will not be the ones best suited to another project. But the mindset of aligning your architecture techniques, your experience, and the guidance you have learned will steer you to appropriate techniques.

We now turn our attention to the second question: How much design and architecture should you do? Time spent designing or analyzing is time that could have been spent building, testing, etc., so you want to get the balance right, not doing too much design or ignoring risks that could swamp your project.

Effort should be commensurate with risk. The risk-driven model strives to efficiently apply techniques to reduce risks, which means not over- or under-applying techniques. To achieve efficiency, the risk-driven model uses this guiding principle:

Architecture efforts should be commensurate with the risk of failure.

If you recall the story of my father and the mailbox, he was not terribly worried about the mailbox falling over, so he did not spend much time designing the solution or applying mechanical engineering analyses. He thought about the design a little bit, perhaps considering how deep the hole should be, but most of his time was spent on implementation.

When you are unconcerned about security risks, spend no time on security design. However, when performance is a project-threatening risk, work on it until you are reasonably sure that performance will be OK.

Incomplete architecture designs. When you apply the risk-driven model, you only design the areas where you perceive failure risks. Most of the time, applying a design technique means building a model of some kind, either on paper or a whiteboard. Consequently, your architecture model will likely be detailed in some areas and sketchy, or even non-existent, in others.

For example, if you have identified some performance risks and no security risks, you would build models to address the performance risks, but those models would have no security details in them. Still, not every detail about performance would be modeled and decided. Remember that models are an intermediate product and you can stop working on them once you have become convinced that your architecture is suitable for addressing your risks.

Subjective evaluation. The risk-driven model says to prioritize your risks, apply chosen techniques, then evaluate any remaining risk, which means that you must decide if the risk has been sufficiently mitigated. But what does sufficiently mitigated mean? You have prioritized your risks, but which risks make the cut and which do not?

The risk-driven model is a framework to facilitate your decision making, but it cannot make judgment calls for you. It identifies salient ideas (prioritized risks and corresponding techniques) and guides you to ask the right questions about your design work. By using the risk-driven model, you are ahead because you have identified risks, enacted corresponding techniques, and kept your effort commensurate with your risks. But eventually you must make a subjective evaluation: will the architecture you designed enable you to overcome your failure risks?

3.7 Planned and evolutionary design

You should now be prepared, at a conceptual level at least, to go out and apply software architecture on your projects. You may still have some questions about how to proceed, however, since we have not yet discussed how the risk-driven model interacts with other kinds of guidance you already know, things like planned and evolutionary design, software processes, and specifically agile software development. The remainder of the chapter shows how the risk-centric model is compatible with each of these and can be used to augment their advice.

We start by discussing three styles of design: planned, evolutionary, and minimal planned design. Planned and evolutionary are the two basic styles of design and minimal planned design is a combination of them.

Evolutionary design. *Evolutionary design* "means that the design of the system grows as the system is implemented" (Fowler, 2004). Historically, evolutionary design has been frowned upon because local and uncoordinated design decisions yield chaos, creating a hodgepodge system that is hard to maintain and evolve any further.

However, recent trends in software processes have re-invigorated evolutionary design by addressing most of its shortcomings. The agile practices of *refactoring*, *test-driven design*, and *continuous integration* work against the chaos. Refactoring (a behavior-preserving transformation of code) cleans up the uncoordinated local designs (Fowler, 1999), test-driven design ensures that changes to the system do not cause it to lose or break existing functionality, and continuous integration provides the entire team with the same codebase. Some argue that these practices are sufficiently powerful that planned design can be avoided entirely (Beck and Andres, 2004).

Of the three practices, refactoring is the workhorse that reduces the hodgepodge in evolutionary design. Refactoring replaces designs that solved older, local problems with designs that solve current, global problems. Refactoring, however, has limits. Current refactoring techniques provide little guidance for architecture scale transformations. For example, Amazon's sweeping change from a tiered, single-database architecture to a service-oriented architecture (Hoff, 2008a) is difficult to imagine resulting from small refactoring steps at the level of individual classes and methods. In addition, legacy code usually lacks sufficient test cases to confidently engage in refactoring, yet most systems have some legacy code.

Though some projects use evolutionary design recklessly, its advocates say that evolutionary design must be paired with supporting practices like refactoring, test-driven design, and continuous integration.

Planned design. At the opposite end of the spectrum from evolutionary design is *planned design*. The general idea behind planned design is that plans are worked out in great detail before construction begins. Analogies with bridge design and construction are often brought up, since bridge construction rarely begins before its design is complete.

Few people advocate[2] doing planned design for an entire software system, an approach sometimes called *Big Design Up Front (BDUF)*. However, complete planning of just the architecture is suggested by some authors (Lattanze, 2008; Bass, Clements and Kazman, 2003), since it is often hard to know on a large or complex project that

[2]Model Driven Engineering (MDE) is an exception since it needs a detailed model to generate code.

any system can satisfy the requirements. When you are not sure that any system can be built, it is best to find this out early.

Planned architecture design is also practical when an architecture is shared by many teams working in parallel, and therefore useful to know before the sub-teams start working. In this case, a planned architecture that defines the top-level components and connectors can be paired with *local design*s, where sub-teams design the internal models of the components and connectors. The architecture usually insists on some overall invariants and design decisions, such as setting up a concurrency policy, a standard set of connectors, allocating high-level responsibilities, or defining some localized quality attribute scenarios. Note that architectural modeling elements like components and connectors will be fully described in the second part of this book.

Even when following planned design, an architecture or design should rarely, if ever, be 100% complete before proceeding to prototyping or coding. With current design techniques, it is nearly impossible to perfect the design without feedback from running code.

Minimal planned design. In between evolutionary design and planned design is *minimal planned design*, or *Little Design Up Front* (Martin, 2009). Advocates of minimal planned design worry that they might design themselves into a corner if they did all evolutionary design, but they also worry that all planned design is difficult and likely to get things wrong. Martin Fowler puts estimated numbers on this, saying he does roughly 20% planned design and 80% evolutionary design (Venners, 2002).

Balancing planned and evolutionary design is possible. One way is to do some initial planned design to ensure that the architecture will handle the biggest risks. After this initial planned design, future changes to requirements can often be handled through local design, or with evolutionary design if the project also has refactoring, test-driven-design, and continuous integration practices working smoothly.

If you are concerned primarily with how well the architecture will support global or emergent qualities, you can do planned design to ensure these qualities and reserve any remaining design as evolutionary or local design. For example, if you have identified throughput as your biggest risk, you could engage in planned design to set up throughput budgets (e.g., message deliveries happen in 25ms 90% of the time). The remainder of the design, which ensured that individual components and connectors met those performance budgets, could be done as evolutionary or local design. The general idea is to perform *architecture-focused design* (see Section 2.7) to set up an architecture known to handle your biggest risks, allowing you more freedom in other design decisions.

Which is best? Regardless of which design style you prefer, you must design software before you write the code, whether it is ten minutes before or ten months before. Both design styles have devoted proponents and their debate relies on anecdotes rather than solid data, so for now opinions will vary. So if you have high confidence in your ability to do evolutionary design, you will do less planned design.

Realize that different systems will lend themselves to different styles of design. Consider the slow changes to the Apache web server over the past decade. It is suitable for planned design because its design resembles an optimization problem for a stable set of requirements (e.g., high reliability, extensibility, and performance). On the other hand, many projects have rapidly changing requirements that favor evolutionary design.

The essential tension between planned and evolutionary design is this: A long head start on architectural design yields opportunities to ensure global properties, avoid design dead ends, and coordinate sub-teams — but it comes at the expense of possibly making mistakes that would be avoided if decisions were made later. Teams with strong refactoring, test-driven development, and continuous integration practices will be able to do more evolutionary design than other teams.

The risk-driven model is compatible with evolutionary, planned, and minimal planned design. All of these design styles agree that design should happen at some point and they all allocate time for it. In planned design, that time is up-front, so applying the risk-driven model means doing up-front design until architecture risks have subsided. In evolutionary design, it means doing architecture design during development, whenever a risk looms sufficiently large. Applying it to minimal planned design is a combination of the others.

3.8 Software development process

Few developers build systems using only a design style, say evolutionary design, and a compiler. Instead, their activities are structured using a software development process that has been designed to increase their chances of successfully delivering a good system. A good software development process does more than just minimize engineering risks, since it must also factor in other business needs and risks, such as time-to-market pressures.

When you broaden your attention from pure engineering risks to the overall project risks, you find many more risks to worry about. Will the customer accept your system? Will the market have changed by the time you deliver? Will you deliver on time? Did your requirements reflect the customer's desires? Do you have the right people, are they doing the right jobs, and are they communicating effectively? Will there be lawsuits?

Software development process. A *software development process* orchestrates a team's activities with the goal of balancing both engineering and project management risks. It is tempting, but impossible, to cleanly separate engineering process from project management process. A software development process helps you prioritize risks across both engineering and project management, and perhaps to decide that even though engineering risks still exist, other risks outweigh them.

Risk as shared vocabulary. 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 does not work as desired, but he will understand the risk of its failure, and the engineer can help him assess the risk's probability and severity.

The concept of a risk is positioned in 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 during each iteration it prioritizes the risks and plans out the next step accordingly, looping until the system is delivered. In practice, some risk mitigation steps are deliberately *baked-in* to the software development process.

At a large company worried about team coordination, the process might insist on various forms of documentation at project milestones. Agile processes bake-in worries about time-to-market and customer rejecting the product, and consequently insist that the software be built and delivered 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.

Baking risk mitigation techniques into the software development process can be a blessing. It is a blessing when the process bakes-in risks that you would prioritize anyway, so it 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 its 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 $BIGCOMPANY, 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. Like a scene from a comedy, he swiveled in his chair and picked up a book describing $BIGCOMPANY'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.

If you decide to tailor your software development process to bake in risks, 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).

3.9 Understanding process variations

The descriptions offered here are coarse overviews that omit important details of each process, but are adequate background so that you can later see how the risk-driven model could be applied to any of them.

These simplified descriptions force-fit each process into a simple two-part model: An optional up-front design part with one or more iterations that follow. Not every development process here has up-front design, but all of them have at least one iteration. The development processes vary on four points:

1. Is there up-front design?
2. What is the nature of the design (planned/evolutionary; redesign allowed)?
3. How is work prioritized across iterations?
4. And how long is an iteration?

Figure 3.4 summarizes the processes and highlights some of their differences.

Two other important variation points that arise when talking about development process are: how detailed should your design models be, and how long you should hold on to your design models? None of the above processes commits to an answer for these, except for XP, which allows modeling but discourages keeping the models around past an iteration. Using this simple model of software development processes yields the following descriptions:

Waterfall. The waterfall process proceeds from beginning to end as a single long block of work that delivers the entire project (Royce, 1970). It assumes planned design work that is done in its analysis and design phases. These precede the construction phase, which can be considered a single iteration. With just one iteration, work cannot be prioritized across iterations, but it may be built incrementally within the construction phase. Applying the risk-driven model would mean doing architecture work primarily during the analysis and design phases.

Process	Up-front design	Nature of design	Prioritization of work	Iteration length
Waterfall	In analysis & design phases	Planned design; no redesign	Open	Open
Iterative	Optional	Planned or evolutionary; redesign allowed	Open, often feature-centric	Open, usually 1-8 weeks
Spiral	None	Planned or evolutionary	Riskiest work first	Open
UP / RUP	Optional; design activities front-loaded	Planned or evolutionary	Riskiest work first, then highest value	Usually 2-6 weeks
XP	None, but some do in iteration zero	Evolutionary design	Highest customer value first	Usually 2-6 weeks

Figure 3.4: Examples of software development processes and how they treat design issues. For comparison purposes, a waterfall process is treated as having a single long iteration.

Iterative. An iterative development process builds the system in multiple work blocks, called iterations (Larman and Basili, 2003). With each iteration, developers are allowed to rework existing parts of the system, so it is not just built incrementally. Iterative development optionally has up-front design work but it does not impose a prioritization across the iterations, nor does it give guidance on the nature of design work. Applying the risk-driven model would mean doing architecture work within each iteration and during the optional up-front design.

Spiral. The spiral process is a kind of iterative development, so it has many iterations, yet it is often described as having no up-front design work (Boehm, 1988). Iterations are prioritized by risk, with the first iteration handling the riskiest parts of a project. The spiral model handles both management and engineering risks. For example, it may address "personnel shortfalls" as a risk. The spiral process gives no guidance on how much architecture/design work to do, or of which architecture and design techniques to use.

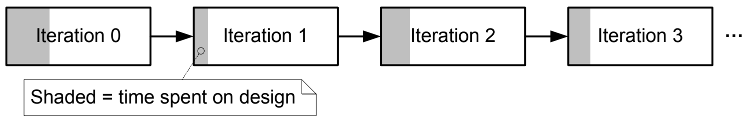

Figure 3.5: An example of how the amount of design could vary across iterations based on your perception of the risks. Based on the amount of time spent, you can infer that the most risk was perceived in iteration 0 and iteration 2.

[Rational] Unified Process (RUP). The Unified Process and its specialization, the Rational Unified Process, are iterative, spiral processes (Jacobson, Booch and Rumbaugh, 1999; Kruchten, 2003). They both highlight the importance of addressing risks early and highlight the use of architecture to address risks. The (R)UP advocates working on architecturally-relevant requirements first, in early iterations. It can accommodate either planned or evolutionary design.

Extreme Programming (XP). Extreme Programming is a specialization of an iterative and agile software development process, so it contains multiple iterations (Beck and Andres, 2004). It suggests avoiding up-front design work, though some projects add an *iteration zero* (Schuh, 2004), in which no customer-visible functionality is delivered. It guides developers to apply evolutionary design exclusively, though some projects modify it to incorporate a small amount of up-front design. Each iteration is prioritized by the customer's valuation of features, not risks.

3.10 The risk-driven model and software processes

It is possible to apply the risk-driven model to any of these software development processes while still keeping within the spirit of each. The waterfall process prescribes planned design in its analysis and design phases, but does not tell you what kind of architecture and design work to do, or how much of it. You can apply the risk-driven model during the analysis and design phases to answer those questions.

The iterative process does not have a designated place for design work, but it could be done at the beginning of each iteration. The amount of time spent on design would vary based on the risks. Figure 3.5 provides a notional example of how the amount of design could vary across iterations based on your perception of the risks.

The spiral process and the risk-driven model are cousins in that risk is primary in both. The difference is that the spiral process, being a full software development process, prioritizes both management and engineering risks and guides what happens across iterations. The risk-driven model only guides design work to mitigate engineering risks, meaning that it would help you understand which architecture techniques you should use within a specific iteration of the spiral process. Applying the risk-driven model to the spiral model or the (R)UP works the same as with an iterative process.

You will have noticed that, of the processes listed in Figure 3.4, XP (an agile process) has the most specific advice. Consequently, it is trickiest to apply the risk-driven model into the XP process (or other feature-centric agile processes). What follows is a sketch of how the risk-driven model could be applied in an agile project.

3.11 Application to an agile processes

The following description of using the risk-driven model on an agile project highlights some core issues, such as when to design and how to mix risks into a feature-driven development process. Since agile projects vary in their process, this description assumes one with a two-week iteration that plays a *planning game* to manage a *feature backlog*. On the engineering side, there are software architecture risks that you should fold into this process, which includes identification, prioritization, mitigation, and evaluation of those risks. The big challenges are: first, how to address initial engineering risks, and second, how to incorporate engineering risks that you later discover into the stack of work to do.

Risks. 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 (Commercial Off-The-Shelf) components. Some agile projects use an iteration zero to get their development environment set up, including source code control and automated build tools. You 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.

Risk backlog. At the end of an iteration, you need to evaluate how well your activities mitigated your risks. Most of the time you will have reduced a 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 can be written up as a testable feature for the system and added to the backlog.

Note that this is not an excuse to turn a nominal iteration zero into a de facto Big Design Up-Front exercise. Instead of extending the time of iteration zero, risks are pushed onto the backlog.

Prioritizing risks and features. It is challenging to fold engineering risk into a planning game to manage a backlog of features. Many agile projects divide the world

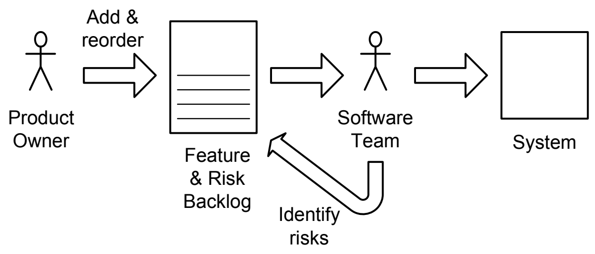

Figure 3.6: One way to incorporate risk into an agile process is to convert the feature backlog into a *feature & risk backlog*. The product owner adds features and the software team adds technical risks. The software team must help the product owner to understand the technical risks and suitably prioritize the backlog.

into product owners, who create a prioritized 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 you introduce risks, because both features and risks must be prioritized. 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 items. The question is, who prioritizes the risks?

If you give the product owner the additional responsibility to prioritize architectural risks alongside features, you can simply change the feature backlog into a feature & risk backlog, as seen in Figure 3.6. 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, you should evaluate architectural risks and feed these into the backlog.

Summary. An agile process can handle architectural risks by doing three things. Architectural risks that you know in advance can be (at least partially) handled in a time-boxed iteration zero, where no features are planned to be delivered. Small architectural risks can be handled as they arise during iterations. Large architectural risks should be promoted to be on par with features and inserted into a combined feature & risk backlog.

3.12 Risk and architecture refactoring

Over time, system developers understand increasingly well how a system should be designed. This is true regardless of which kind of process they follow (e.g., waterfall or iterative processes). In the beginning, they know and understand less. After some work (design, prototyping, iterations, etc.) they have better grounded opinions on suitable designs.

Once they recognize that their code does not represent the best design (e.g., by detecting *code smells*), they have two choices. One is to ignore the divergence, which yields *technical debt*. If allowed to accumulate, the system will become a big ball of mud (see Section 14.7). The other is to refactor the code, which keeps it maintainable. This second option is well described by Brian Foote and William Opdyke in their patterns on software lifecycle (Coplien and Schmidt, 1995).

Refactoring, by definition, means re-design and the scale of that redesign can vary. Sometimes a refactoring involves just a handful of objects or some localized code. But other times it involves more sweeping architectural changes and is called *architecture refactoring*. Since little published guidance exists for refactoring at large scale, architecture refactoring is generally performed ad hoc.

The example from the introduction where Rackspace implemented their query system three different ways (see Section 1.2) is best thought of as architecture refactoring. There, each refactoring of the architecture was precipitated by a pressing failure risk. Object-level refactorings take a negligible amount of time and therefore need little justification, so you should just go ahead and, for example, rename a variable to be more expressive of its intent. An architecture refactoring is expensive, so it requires a significant risk to justify it.

Two important lessons are apparent. First, *design does not exclusively happen up-front*. It is often reasonable to spend time up-front making the best choices you can, but it is optimistic to think you know enough to get all those design decisions right. You should anticipate spending time designing after your project's inception.

Second, *failure risk can guide architecture refactoring*. By the time it is implemented, nearly every system is out of date compared to the best thinking of its developers. That is, some technical debt exists. Perhaps, in hindsight, you wish you had chosen a different architecture. Risks can help you decide how bad it will be if you keep your current architecture.

3.13 Alternatives to the risk-driven model

The risk-driven model does two things: it helps you decide when you can stop doing architecture, and it guides you to appropriate architecture activities. It is not good at predicting how long you will spend designing, but it helps you recognize when you have done enough. There are several alternatives to the risk-driven model, with their own advantages and disadvantages.

No design. The option of not designing is a bit of a misnomer, especially if you believe that every system has an architecture, because the developers must have thought about it at some point. Perhaps they were thinking about the design (i.e., what they will code) immediately before they start typing, but they do think about the design. Such projects likely borrow heavily from presumptive architectures (see Section 2.4), where the developers pattern their system off of similar successful systems, explicitly or implicitly.

Documentation package. Some people suggest, or at least imply, that you should build a full documentation package that describes your architecture. If you follow this guidance, you will build a set of models and diagrams and write them down in such a way that someone else could read and understand the architecture, which can be quite desirable. If you need documentation, the Documenting Software Architectures book (Clements et al., 2010) will guide you to an effective set of models and diagrams to record.

However, not every project should spend the effort to create a full documentation package. For example, the “3 guys in a garage” startup probably cannot afford to write anything down.

Yardsticks. Empirical data can help you decide how much time should be spent on architecture and design. Barry Boehm has calculated the optimal amount of time to spend on the architecture for small, medium, and large projects based on a variant of his COCOMO model (Boehm and Turner, 2003). For various project sizes, he has plotted curves of architecture effort vs. total project duration. His data indicates that most projects should spend 33-37% of their total time doing architecture, with small projects spending as little as 5% and very large projects spending 40%. A yardstick like “spend 33% of your time on architecture” can be used by project managers for planning project activities and staffing requirements, yielding a time budget to spend in design.

Yardsticks, however, are little help to developers once the architecture work has started. No reasonable developer should continue design activities for additional days after the risks have been worked out, even if the yardstick provides that budget. Nor should a reasonable developer switch to coding when a major failure risk is outstanding. It is best to view such yardsticks as heuristics derived from experience combating risks, where projects of a certain size historically needed about that much time to mitigate their risks. That yardstick does not help you decide whether one more (or one less) day of architecture work is appropriate. Also, yardsticks only suggest broad categorical activities rather than guide you to particular techniques.

Ad hoc. When they choose when and how much architecture to do, most developers probably do not follow any of the alternatives above. Instead, they make a decision in the moment, based on their experience and their best understanding of the project’s needs. This may indeed be the most effective way to proceed, but it is dependent upon the skill and experience of the developer. It is not teachable, since its lessons are not explicit, nor is it particularly helpful in creating project planning estimates. It may be that, in practice, the ad hoc approach is a kind of informal risk-driven model, where developers tacitly weigh the risks and choose appropriate techniques.

3.14 Conclusion

This chapter set out to investigate two questions. First, *which design and architecture techniques should developers use?* And second, *how much design and architecture should developers do?* It reviewed existing answers, including doing no design, using yardsticks, building documentation packages, and proceeding ad hoc. It introduced the *risk-driven model* that encourages developers to: (1) prioritize the risks they face, (2) choose appropriate architecture techniques to mitigate those risks, and (3) re-evaluate remaining risks. It encourages *just enough design and architecture* by guiding developers to a prioritized subset of architecture activities. Design can happen up-front but it also happens during a project.

The risk-driven model is inspired by my father's work on his mailbox. He did not perform complex calculations — he just stuck the post in the hole then filled it with concrete. Low-risk projects can succeed without any planned architecture work, while many high-risk ones would fail without it.

The risk-driven model walks a middle path that avoids the extremes of complete architecture documentation packages and architecture avoidance. It follows the principle that your architecture efforts should be commensurate with the risk of failure. Avoiding failure is central to all engineering and you can use architecture techniques to mitigate the risks. The key element of the risk-driven model is the promotion of risk to prominence. Each project will have a different set of risks, so it likely needs a different set of techniques. To avoid wasting your time and money, you should choose techniques that best reduce your prioritized list of risks.

The question of how much software architecture work you should do has been a thorny one for a long time. The risk-driven model transforms that broad question into a narrow one: "Have your chosen techniques sufficiently reduced your failure risks?" Evaluation of risk mitigation is still subjective, but it is one that developers can have a focused conversation about.

Engineering techniques address *engineering risks*, but projects face a wide variety of risks. *Software development processes* must prioritize both management risks and engineering risks. You cannot reduce engineering risks to zero because there are also project management risks to consider, including time-to-market pressure. By applying the risk-driven model, you ensure that whatever time you devote to software architecture reduces your highest priority engineering risks and applies relevant techniques.

Agile architecture approaches often emphasize *evolutionary design* over *planned design*. Another middle path, *minimal planned design*, can be used to avoid the extremes. The essential tension is this: A long headstart on architectural design yields opportunities to ensure global properties, avoid design dead ends, and coordinate sub-teams — but it comes at the expense of possibly making mistakes that would be avoided if decisions were made later. Agile processes focusing on features can be adapted slightly to add risk to the feature backlog, with developers educating product owners on how to prioritize the feature & risk backlog.

Some readers may be frustrated that this chapter does not prescribe a list of techniques to use and a single process to follow. These are missing because the techniques that work great on one project would be inappropriate on another. And there is not yet enough data to overcome opinions about the best process to recommend. Indeed, you may not have a choice about which process you follow, but within that process you likely have the ability to use the risk-driven model. This chapter has tried to provide relevant information about how to make your own choices so that you can do just enough architecture for your projects.

3.15 Further reading

The invention of risk as a concept likely occurred quite early, with references to it in Greek antiquity, but it took on its modern, more general, idea as late as the 17th century, where it increasingly displaced the concept of *fortunes* as what drove life's outcomes (Luhmann, 1996). A few minutes after that, project managers started using risk to drive their projects. This longstanding tradition in project management has carried over into software process design, with many authors emphasizing the role of risk in software development, including Philippe Kruchten (Kruchten, 2003), Ivar Jacobson, Grady Booch, and James Rumbaugh (Jacobson, Booch and Rumbaugh, 1999), including the connection between architecture and risk.

Barry Boehm wrote about risk in the context of software development with his paper on the spiral model of software development (Boehm, 1988), which is an interesting read even if you already understand the model. The risk-driven model would on first glance appear to be quite similar to the spiral model of software development, but the spiral model applies to the entire development process, not just the design activity. A single turn through the spiral has a team analyzing, designing, developing, and testing software. The full spiral covers the project from inception to deployment. The risk-driven model, however, applies just to design, and can be incorporated into nearly any software development process. Furthermore, the spiral model guides a team to build the riskiest parts first, but does not guide them to specific design activities. Both the spiral model and the risk-driven model are in strong agreement in their promotion of risk to a position of prominence.

Barry Boehm and Richard Turner followed this up with a book on risk and agile processes (Boehm and Turner, 2003). The summary of their judgment is, “The essence of using risk to balance agility and discipline is to apply one simple question to nearly every facet of process within a project: Is it riskier for me to apply (more of) this process component or to refrain from applying it?”

Mark Denne and Jane Cleland-Huang discuss both architecture and risk in the context of software project management (Denne and Cleland-Huang, 2003). They advocate managing projects by chunking development into Minimum Marketable Features, which has the consequence of incrementally constructing your architecture.

The risk-driven model is similar to *global analysis* as described by Christine Hofmeister, Robert Nord, and Dilip Soni (Hofmeister, Nord and Soni, 2000). Global analysis consists of two steps: (1) analyzing organizational, technical, and product factors; and (2) developing strategies. Factors and strategies in global analysis map to risks and activities in the risk-driven model. Factors are broader than the technical risks in the risk-driven model, and could include, for example, headcount concerns. Both global analysis and the risk-driven model are similar in that they externalize a structured thought process of the form: I am doing X because Y might cause problems. In the published descriptions, the intention of global analysis is not to optimize the amount of effort spent on architecture, but rather to ensure that all factors have been investigated.

Two publications from the SEI can help you become more consistent and thorough in your identification and explanation of risks. Carr et al. (1993) describe a taxonomy-based method for identifying risks and Gluch (1994) introduces the condition-transition-consequence format for describing risks.

The risk-driven model advocates building limited architecture models that have detail only where you perceive risks. Similarly, authors have been advocating building minimally sufficient models for years, including Desmond D’Souza, Alan Wills, and Scott Ambler (D’Souza and Wills, 1998; Ambler, 2002). Tailoring the models built on a project to the nature of the project (greenfield, brownfield, coordination, enhancement) is discussed in Fairbanks, Bierhoff and D’Souza (2006).

The idea of cataloging techniques, or tactics, is described in the context of Attribute Driven Design in Bass, Clements and Kazman (2003). Attribute Driven Design (ADD) relies on a mapping from quality attributes to tactics (discussed in Section 11.3) , much like the risk-driven model and global analysis. The concept in this book of mapping development techniques is similar in nature. ADD guides developers to an appropriate design (a pattern), while the risk-driven model guides developers to an activity, such as performance modeling or domain analysis. The risk-driven model can be seen as taking the promotion of risk from the spiral model and adapting the tabular mapping of ADD to map risks to techniques.

Knowing what tactics or techniques to apply would be valuable knowledge to include in a software architecture handbook, and would accelerate the learning of novice developers. Such knowledge is already in the heads of *virtuosos*, as described by Mary Shaw and David Garlan (Shaw and Garlan, 1996). The better our field encodes this knowledge, the more compact it becomes and the faster the next generation of developers absorbs it and sees farther.

Though tactics and techniques were described in this chapter as tables, they could be expressed as a pattern language, as originally described by Christopher Alexander for the domain of buildings (Alexander, 1979; Alexander, 1977), and later adapted to software in the Design Patterns book (Gamma et al., 1995) by Erich Gamma and others.

Martin Fowler's essay, "Is Design Dead?" (Fowler, 2004) provides a very readable introduction to evolutionary design and the agile practices that are required to make it work.

Merging risk-based software development and agile processes is an open research area. Jaana Nyfjord's thesis (Nyfjord, 2008) proposes the creation of a Risk Management Forum to prioritize risk across products and projects in an organization. Since the goal here is to handle architecture risks that are only a subset of all project risks, a smaller change to the process may work.

This book uses risk to help you decide which techniques to use and how many of them to apply, assuming the requirements are not negotiable. Another way to use it is to help determine the scope of the projects, assuming the requirements can be changed. The quantitative technique is described in Feather and Hicks (2006), with the result being a bag of requirements that gives you the most benefit for the risk that you take on.

With many developers seeking lighter weight processes, agile development is popular. Ambler (2009) provides an overview of how architecture can be woven into agile processes, and Fowler (2004) discusses how evolutionary design can complement planned design. Boehm and Turner (2003) discuss the tension between moving fast and getting it right. A thorough treatment of a practical process for software architecture is found in (Eeles and Cripps, 2009).