
Chapter 3

Risk-Centric 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:

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. (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-centric model*, inspired by Attribute Driven Design (ADD) and the Spiral model. Risks are central, so developers: (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. Like ADD, and unlike the Spiral model, the risk-centric model is not a full software development process and can instead be used inside a process such as XP or RUP.

The risk-centric model is a reaction to a world where developers are under pressure to build high quality software — but quickly and at reasonable cost — and developers have many architecture techniques to choose from. The risk-centric model helps you answer the two questions above: it provides guidance on how much software architecture to do and on choosing appropriate techniques. This chapter walks through the ideas behind the risk-centric 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-centric model?

The *risk-centric 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 coding?” The risk-centric model can be summarized in three steps:

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

The key element of the risk-centric 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 forgotten. A recent paper described how a team that had previously done up-front architecture work switched to

a feature-driven process and ended up deferring quality attribute concerns. In fact, these concerns were deferred until active development ceased and the system was in maintenance (Babar, 2009). The conclusion to draw is that a team that has promoted features to prominence will indeed pay less attention to other areas, including risks.

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.

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-centric model is merely improvisation.

Though different developers will perceive risks differently and consequently choose different techniques, the risk-centric 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 (techniques to apply) based on the relevant context (perceived risks).

Other developers might disagree, 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-centric now?

Most developers believe that they already follow a risk-centric 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.

Instead, in practice you see developers over-using techniques that are inefficient at reducing relevant risks and under-using techniques that could help. Examining the overall context of software development reveals why this can occur. Most organizations guide developers to follow a process with some kind of documentation template. These processes and templates can most certainly 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 a full documentation package for each system.
- The team must always (or never) build a class diagram, a layer diagram, etc.
- The team must spend 10% of the project time on design or architecture.

Each project will face a different set of risks. Consequently, choosing in advance the set of architecture techniques, or the amount of time to spend on those techniques, will be inefficient. It would be a great coincidence that the same set of diagrams or techniques is always the best way to mitigate a changing set of risks.

Enabling variation

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 they followed the risk-centric 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 they both follow the same company-standard process and produce, say, a module dependency diagram because that is what their corporation's design document template says to do. Yet there is no clear connection between the techniques they use and the risks they face.

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.

The risk-centric model advises developers to use techniques corresponding to project risks, but to do so, developers must explicitly identify and evaluate risks. The three steps to risk-centric software architecture are deceptively simple because 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 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. It is therefore easier to bundle the concept of uncertainty into the definition of risk, rather than talking about perceived risks versus actual risks. The definition of risk then becomes:

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

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 not be robust”
“Senior VP hates our manager”	“It’s working now but if we touch anything it may fall apart”

Table 3.1: Examples of project management and engineering risks

A result of this definition is that a risk can exist (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 the risk goes down. So by applying techniques you can reduce the amount of uncertainty, and therefore the amount of risk.

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, how can you tell if the risk is actually reduced? Another way of saying this is that you must describe risks well enough so that you can later test to see if it has been mitigated. Instead of just listing a quality attribute like reliability, a better way is to describe each risk of failure as a testable *failure scenario*, such as “The program de-references a null pointer, and crashes.”

Engineering and project management risks

Software developers worry that their systems will fail. *Engineering risks* are those risks related to the analysis, design, and implementation of the product. These engineering risks¹ are in the domain of the engineering of the system. Engineering risks are contrasted with *project management risks*, which relate to schedules, sequencing of work, delivery, team size, geography, etc. Table 3.1 shows examples of both.

It is important to distinguish these two kinds of risks because the technique type must match the risk type. Only engineering techniques will mitigate engineering risks, so for example you cannot use a PERT chart (a project management technique) to reduce the chance of buffer overruns (an engineering risk), nor will UML 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 domain? The easiest place to start is

¹Engineering risks can be further 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. Your architecture choices will usually impact quality attribute risks than functional risks.

with the requirements, in whatever form they take, and look for things that seem difficult to achieve. Stakeholders often fail to clearly articulate quality attribute requirements, so you can elicit them formally or informally to find challenging requirements. Quality Attribute Workshops (Barbacci et al., 2003) can be adapted to generate failure scenarios and produce a prioritized list of failure scenarios.

Additionally, each domain has a set of *prototypical risks* that is different from other domains. For example, Systems projects usually worry more about performance than IT projects. Individual organizations may have created *checklists* describing historical problem areas, perhaps generated from architecture reviews. These checklists (see Section 13.1) are valuable knowledge for less experienced developers and a helpful reminder for experienced ones.

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 forseen and high priority risk. You must accept that unforeseen risks will exist.

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 is often 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 the time and money are being spent in accordance with stakeholder priorities.

Risks can be categorized² on two dimensions: their priority to stakeholders and their perceived difficulty by developers. Be aware that some technical risks 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*. Table 3.2 shows a short list of software engineering techniques and techniques from other engineering branches.

²This is the same categorization technique used in ATAM to prioritize architecture drivers and quality attribute scenarios, as discussed in Section 11.11.

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

Table 3.2: Examples of engineering risk reduction techniques in software engineering and other fields

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.

This book focuses on techniques that are on the analysis-end of the spectrum, procedural, and independent of the problem domain. These techniques include using models such as layer diagrams, component assembly models, and deployment models; analytic techniques for performance, security, and reliability; and architectural styles such as client-server and pipe-and-filter.

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. Right now 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>, techniques that could reduce it include <techniques>.

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

A 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 encourages efficient work to reduce risks. 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 only applying techniques when they are motivated by risks.

Cannot eliminate engineering risk

Imagine a software developer whose only concern is to minimize engineering risks. He would choose to apply every relevant technique to minimize those engineering risks to build the best possible system. The engineering risk reduction comes at a cost: time. The Wright brothers could have continued their mathematical and empirical investigations into aeronautical principles and thus minimized engineering risk. But then their first test flight might have been in 1953 instead of 1903.

The reason you cannot afford to eliminate engineering risks is because you must balance them with non-engineering risks, which are predominantly project management risks. Consequently, 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.

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, or even think of it as a *knapsack 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. You might become convinced that employing that usability 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 is inappropriate, you should spend your time analyzing the 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.

Put another way, if you hear that a project is applying a set of techniques, then it is impossible to know that the project has chosen wisely unless you also know what risks it faces.

3.5 Guidance on choosing techniques

In the future, perhaps a developer choosing techniques will act much like a mechanical engineer choosing materials by referencing tables of properties and making informed decisions. For now, you must use your experience. However, even if you could, looking up techniques in a table would be intellectually unsatisfactory because there are reasons behind the table that explain why those techniques work.

The following sections describe some ideas that provide a conceptual foundation for the risk-technique mapping, and some generalizations that help you know what kinds of techniques are reasonable choices.

Problems to find and prove

In his book *How to Solve It*, George Polya (Polya, 2004) 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, 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 “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 API’s (Application Programming Interface)?” because there could be a case you have not yet seen, perhaps when a framework call unexpectedly passes 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* (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 — they are analogues. Analogic models support analysis only indirectly, 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 13.3) 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. You could imagine an analytic model of the airplanes where each is represented by a vector. Mathematics provides an analytic model to evaluate the vectors, so you could quantitatively answer questions about collision courses.

When you model software, you invariably use symbols, whether they are 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 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, you 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 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.

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 viewpoint. Viewtypes are not fully discussed until Section 8.4. For now, it is sufficient to know that the module viewpoint includes tangible artifacts such as source code and classes; the runtime viewpoint includes runtime structures like objects; and the allocation viewpoint includes allocation elements like server rooms and hardware. 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, 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 primar-

ily helps with security risks, and Queuing Theory primarily helps with performance risks (these techniques are discussed in Section 13.1) .

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 developers should use. You now know how to identify risks and choose techniques to combat them. We now turn our attention to the second question: how much design and architecture developers should do. Time spent designing or analyzing is time that could have been spent building, so you want to get the balance right.

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

Architecture efforts should be commensurate with the risk of failure.

If you are not worried about security failure scenarios, then do not do any security design. However, if poor performance is a project-threatening risk, then work on it until you are reasonably sure that performance will be OK, or until project management risks (such as the risk of not delivering the project on time) overshadow it.

This does not mean that every detail about performance has been modeled and decided, but rather that you have become convinced that your architecture is suitable for overcoming your top-priority risks. After that, you can shift gears and let design occur locally or use evolutionary design to tweak the architecture.

3.7 Planned and evolutionary design

The decisions about how much architecture work to do and which techniques to use are not made in a vacuum. The quantity and type of architecture work depends on the style of design and the overall process.

Some developers prepare their designs substantially in advance, called planned design, while others prepare designs that cover only a short time into the future, called evolutionary design. Planned design is useful to ensure that your system will be able to satisfy challenging requirements, such as lots of transactions per second. Evolutionary design has the potential to be more time-efficient so long as developers, using short-term thinking, do not “paint themselves into a corner.”

Additionally, design and construction happens within a software development process. That process does more than just minimize engineering risks, as it must also factor in other business needs and risks, such as time-to-market pressures. This section discusses three styles of design — evolutionary, planned, and minimal planned — and the next section discusses software development processes.

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. Continuous integration provides the entire team with the same codebase, test-driven design ensures that changes to the system do not cause it to lose or break existing functionality, and refactoring (a behavior-preserving transformation of code (Fowler, 1999)) cleans up the uncoordinated local designs. 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. While theoretically possible, the current refactoring techniques have a difficult time with 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. In addition, legacy code usually lacks sufficient test cases to confidently engage in refactoring, yet most systems have some legacy code.

It is worth reinforcing that every advocate of evolutionary design says it is a bad idea unless it is 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.

However, almost no one advocates³ 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 *any* system can satisfy the requirements. When you are not sure any system would work, it is best to find this out early.

Planned architecture design is also practical when a shared, central architecture is shared by many sub-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 designs*, where sub-teams design the

³Model Driven Engineering (MDE) is an exception since it needs a detailed model to generate code.

internal models of the components and connectors. The architecture usually decides some overall invariants and design decisions, such as setting up a concurrency policy, a standard set of connectors, allocating high-level responsibilities, and defining some localized quality attribute scenarios.

Even when following planned design, an architecture or design should rarely, if ever, be 100% complete before proceeding to prototyping or coding. It is nearly impossible to get the design perfect without getting 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 nominal numbers on this, saying he does 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 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-centric design* (see Section 2.8) to set up an architecture known to handle your biggest risks, allowing process freedom in the rest of design.

Choosing

While there are clear differences, both planned design and evolutionary design are kinds of design. Developers must design software before they write the code, whether it is ten minutes before or ten months before.

The current thinking on planned and evolutionary design has devoted proponents and the debate centers around anecdotes rather than adequate data, so for now opinions will vary. If you have high confidence in your ability to do evolutionary design, you will do less planned design, and vice versa.

The essential tension 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 they 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-centric 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 a time slot for it. In planned design, that slot is up-front, so applying the risk-centric model means doing up-front design until architecture risks have subsided. In evolutionary design, it means doing architecture design ad hoc during development, whenever a risk looms sufficiently large. Applying it to minimal planned design is a combination of the others.

3.8 Risk and process

So far this chapter has only discussed how you can use risk to choose architecture and design techniques. When you have your head down in the engineering, you worry about risks like server scalability and component modifiability. But when you broaden your attention from pure engineering process to the overall software development process, 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?

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 engineering. 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 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 during each iteration it 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. 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 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 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. Like a scene from a comedy, 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).

Understanding process variations

Before describing how risk-centric software architecture fits into software development processes, you need to have a basic understanding of a representative set of development processes. The descriptions offered here are coarse overviews, but are adequate background so that you can later see how the risk-centric model could be applied to any of them.

The descriptions fit each process to a simple two-part model: 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:

Process	Up-front design	Nature of design	Prioritization of work	Iteration length
Waterfall	In analysis & design phases	Planned design; no redesign	Incremental construction	Open
Iterative	Optional	Planned or evolutionary; redesign allowed	Open	Open
Spiral	None	Planned or evolutionary	Riskiest work first	Open
XP	None, but some do in iteration zero	Evolutionary design	Highest customer value first	Often 2-6 weeks

Table 3.3: Examples of software development processes and how they treat design issues

Is there up-front design? What is the nature of the design (planned/evolutionary; redesign allowed)? How is work prioritized across iterations? And how long is an iteration? Table 3.3 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 (Royce, 1970) proceeds from beginning to end as a single long block of work which delivers the entire project. 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 build incrementally within the construction phase. Applying the risk-centric model would mean doing architecture work primarily during the analysis and design phases.

Iterative. An iterative development process (Larman and Basili, 2003) builds the system in multiple work blocks, called iterations. 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-centric model would mean doing architecture work within each iteration and during the optional up-front design.

Spiral. The spiral process (Boehm, 1988) is a kind of iterative development, so it has many iterations, yet it is often described as having no up-front design work. Iterations are prioritized by risk, with the first iteration handling the riskiest parts of a project. The

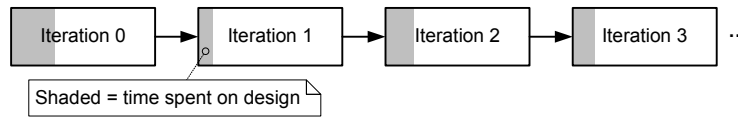


Figure 3.1: Example of how the amount of design could vary across iterations based on your perception of the risks

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 the nature of design work, or of which architecture and design techniques to use.

Extreme Programming (XP). Extreme Programming (Beck and Andres, 2004) is a specialization of an iterative and Agile software development process, so it contains multiple iterations. 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.

How the risk-centric model fits in

It is possible to apply the risk-centric 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-centric 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 3.1 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-centric 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-centric 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-centric model to the spiral model works the same as with an iterative process.

You will have noticed that, of the processes listed in Table 3.3, XP has the most specific advice. Consequently, it is trickiest to apply the risk-centric model into the XP process (or other feature-centric agile processes). What follows is a sketch of how the risk-centric model could be applied in an agile project.

3.9 Sketch: Applying the risk-centric model to agile processes

There is not enough detail here for you to use this as a process but 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 the *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 as they come up 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 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 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 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 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 you must 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. 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 3.2. 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

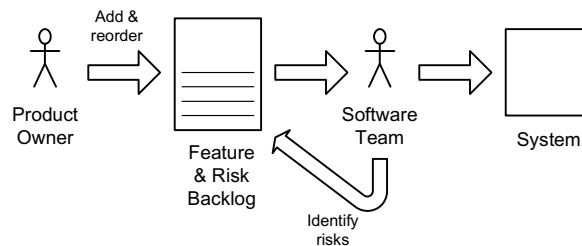


Figure 3.2: One way to incorporate risk into an agile process is to convert the feature backlog into a feature & risk backlog. The product owner adds features and the software team adds technical risks. The software team must help the product owner to understand the technical risks and suitably prioritize the backlog.

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.

In summary, an agile process can handle architectural risks by doing two things. Architectural risks that you know in advance can be (at least partially) handled in a timeboxed 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 should be promoted to be on par with features, and inserted into a combined feature & risk backlog.

3.10 Alternatives to the risk-centric model

The risk-centric 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 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, so 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 2.5), 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 always 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., 2002) 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-centric model, where developers tacitly weigh the risks and choose appropriate techniques.

3.11 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-centric 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.

The risk-centric 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-centric 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-centric 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-centric 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 risk-centric software architecture, you ensure that whatever time you devote to software architecture reduces 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 they 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. Instead, 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.

3.12 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).

The idea of focusing on risk is not a new one, either in engineering as a whole or in

software architecture specifically. 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-centric 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-centric 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-centric model are in strong agreement in their promotion of risk to a position of prominence.

He followed this up with a recent book on risk and agile processes (Boehm and Turner, 2003). The summary of his 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?”

The risk-centric 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-centric model. Factors are broader than the technical risks in the risk-centric model, and could include, for example, headcount concerns. Both global analysis and the risk-centric 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 minimize the amount of effort spent on architecture, but rather to ensure that all factors have been investigated.

Authors have been advocating building minimally sufficient models for years, including Desmond D’Souza (D’Souza and Wills, 1998; D’Souza, 2006) and Scott Ambler (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 10.3), much like the risk-centric 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-centric model guides developers to an activity, such as performance modeling or domain analysis. The risk-centric 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 have so far been thought of 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 unchangeable. 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 compliment planned design. (Boehm and Turner, 2003) discuss the tension between moving fast and getting it right.