Just Enough Architecture: The Risk-Driven Model

George Fairbanks
Rhino Research

Abstract: Developers have access to more architectural design techniques than they can afford to apply. The Risk-Driven Model guides developers to do just enough architecture by identifying their project's most pressing risks and applying only architecture and design techniques that mitigate them. The key element of the Risk-Driven Model is the promotion of risk to prominence. It is possible to apply the Risk-Driven Model to essentially any software development process, such as waterfall or agile, while still keeping within its spirit.

If you knew nothing about software development, you might imagine that the best developers were the ones who spent the most time writing code. Yet it has been clear for a long time that judicious application of other activities—such as analysis, design and testing—will result in better software. However, at some point, doing more analysis, design or testing becomes counterproductive since it steals time and resources away from other, more productive activities.

Designing an appropriate software architecture is one of those non-coding activities that can improve the quality of a system, but if developers spend too much effort on it, they will be stealing from other activities. Consequently, an important question is raised: How much design and architecture should developers do? Any realistic answer must balance design and architecture effort against other activities.

This article introduces the Risk-Driven Model of architectural design. It guides developers to apply effort to their software architecture commensurate with the risks faced by their project. That is, low-risk and highly precedented systems should skimp on architecture, while high-risk and novel systems should pay more attention to it.

This might seem like common sense, but it is not what happens today on most projects. Most commonly, a project's software development process dictates both the amount of effort and the specific architecture techniques. Unless this process is tuned to risk, it results in too much or too little effort spent on architecture.

How Architecture is Done Today

There is active debate about how much architecture work developers should do and several answers have been proposed:

>> No architectural 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.

Any of these answers could be appropriate, but it depends on the project. The problem with these answers is that they do not help developers find a balance—they instead prescribe that balance in advance. What developers need is a way to decide which architecture techniques they should apply and which they should skip.

The Risk-Driven Model

The Risk-Driven Model helps developers decide how much architecture work to do. The essence of the Risk-Driven Model is these three steps:

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

It 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.

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.

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

Technique Choices Should Vary

Most organizations guide developers to follow a process with some kind of documentation template or a list of design activities. Templates 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 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% (or 0%) of the project time on design or architecture.

It would be a great coincidence if an unchanging set of diagrams or techniques was always the best way to mitigate a changing set of risks. Standard processes or templates can be helpful, but they are often used poorly. Over time, you may be able to generalize the risks on the projects at your organization and devise a list of appropriate techniques. The important part is that the techniques match the risks.

Example Mismatch

Imagine an organization that builds a three-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.

What often happens is that both teams follow the same company-standard process or template and produce, say, a module dependency diagram. The diagram is probably somewhat helpful, but it takes the space of more helpful techniques better matched to the project risks.

Following the Risk-Driven Model, the front-end and back-end teams would apply different techniques. For example, the front-end developers might create user interface mockups and analyze their design for intrusion vectors. The back-end developers might do performance modeling and impose constraints to enable scalability.

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 sign 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, it would be an easy question to answer. Another sign is that all developers use the same techniques.

Most architecture templates have a section on risks, but that is not the same as using risks to decide which techniques to use. To be risk-driven in your architectural decision making, you need to have a rationale that ties your actions (i.e., use of architectural techniques) back to your risks.

Logical Rationale

The Risk-Driven Model has the useful property of yielding arguments that can be evaluated. An example argument would take this form:

We identified A, B and C as 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.

Other developers might disagree with this 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.

Incomplete Architecture Designs

When developers apply the Risk-Driven Model, they only design the areas where they 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, the architecture model will likely be detailed in some areas and sketchy, or even non-existent, in others.

For example, if developers have identified some performance risks and no security risks, they 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 developers can stop working on them once they have become convinced that the architecture is suitable for addressing the risks.

Software Processes

Since the Risk-Driven Model applies only to architecture design and is not a full software development process, it is possible to apply it within essentially any process. A 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 to do. You can apply the Risk-Driven Model during the analysis and design phases to answer those questions.

Today, many developers follow iterative processes, and it goes against the grain to bolt-on a several-month architecture design phase. An iterative process does not have a designated place for design work, but architecture design could be done at the beginning of each iteration. The amount of time spent on design would vary based on the risks. Figure 1 provides a notional example of how the amount of design could vary across iterations based on your perception of the risks.

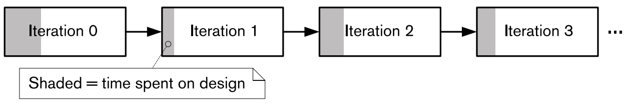

Figure 1: An example of how the amount of design could vary across iterations based on your perception of the risks. In this example, more risk was perceived in iterations 0 and 2.

ADDITIONAL READING

Barry Boehm wrote about risk in the context of software development with his paper on the Spiral Model of software development [1]. 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.

The Unified Process and its specialization, the Rational Unified Process, are iterative, spiral processes [2, 3]. They highlight both the importance of addressing risks early and the use of architecture to address risks. The (R)UP advocates working on architecturally relevant requirements first, in early iterations.

Barry Boehm and Richard Turner discuss risk and agile processes [4] and 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?"

The Risk-Driven Model is similar to global analysis as described by Christine Hofmeister, Robert Nord and Dilip Soni [5]. 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.

This article is excerpted from a chapter in the book Just Enough Software Architecture: A Risk-Driven Approach and the full chapter is available for download [6]. It additionally discusses engineering versus management risks, and details on application of the Risk-Driven Model to various software development processes.

Agile processes are usually special cases of iterative processes, but they have the additional difficulty of fitting architectural work into a backlog. If the backlog must now contain both user features and technical risks, it may be difficult for business stakeholders to prioritize it.

A 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 a Spiral process or the (Rational) Unified Process works the same as with an iterative process.

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.

However, there are principles that underlie any table or any veteran's experience, principles that explain why technique X works to mitigate risk Y. 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. And third, 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 [7]. 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 proof, 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?" 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 [8]. 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 or 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.

An analytic 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 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 it with domain experts. Conversely, if you need to calculate response time distributions, then you will want an analytic model.

Techniques With Affinities

In the physical world, tools are designed for a purpose: hammers are for pounding nails, screwdrivers are for turning screws and 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.

Conclusion

This article introduces 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 software architecture by guiding developers to a prioritized subset of architecture activities. Design can happen up-front but it also happens during a project. Low-risk projects can succeed without any planned architecture work, while many high-risk projects would fail without it.

The Risk-Driven Model walks a middle path that avoids the extremes of complete architecture documentation packages and complete architecture avoidance. It follows the principle that your architecture efforts should be commensurate with the risk of failure. 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 each will need a different set of techniques. To avoid wasting your time and money, you should choose architecture techniques that best reduce your prioritized list of risks.

ABOUT THE AUTHOR

George Fairbanks is the president of Rhino Research, a software architecture training and consulting company. His new book, Just Enough Software Architecture: A Risk-Driven Approach, has been widely praised by academics and practicing software developers. His Ph.D. work at Carnegie Mellon University introduced design fragments, a new way to specify and assure the correct use of frameworks through static analysis. He has been teaching software architecture and object-oriented design for over a decade. He has written code for telephone switches, the Eclipse IDE, Android apps and his own web dot-com startup.

Rhino Research
george.fairbanks@rhinoresearch.com
2445 7th St, Boulder CO 80304
(303) 834-7760

REFERENCES

1. Boehm, Barry. "A Spiral Model of Software Development and Enhancement." 21.5 *IEEE Computer* (1988): 61–72.
2. Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.
3. Kruchten, Philippe. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2003.
4. Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, 2003.
5. Hofmeister, Christine, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
6. Fairbanks, George. *Just Enough Software Architecture: A Risk-Driven Approach* <http://RhinoResearch.com/book>. Marshall & Brainerd, 2010.
7. Polya, George. *How To Solve It: A New Aspect of Mathematical Method*. Princeton University Press. 2004.
8. Jackson, Michael. *Software Requirements and Specifications*. Addison-Wesley. 1995.