The Risk-Centric Model of Software Architecture*

George Fairbanks
Rhino Research
http://rhinoresearch.com

ABSTRACT

Developers are faced with a smorgasbord of architecture activities but few models telling them which activities to use. Alternatives include the *documentation package* model, which advocates a complete architectural description from many perspectives, and the *evolutionary design* model, which advocates no up-front architectural work. This paper 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" 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.

1. INTRODUCTION

Software engineering is a relatively new engineering discipline, yet, over the past few decades, knowledge has built up about how to reliably develop software and avoid pitfalls. In fact, so much knowledge has built up that no project can possibly apply every known software engineering technique, useful as those techniques are. Consequently, various software development processes provide guidance to developers about choosing and applying techniques.

A similar situation is beginning to occur in the sub-field of software architecture. Our knowledge about software architecture has been greatly enhanced over the past few years [5, 10, 9, 25, 22] and this year a thorough textbook on the subject has been produced [26]. As a result, developers have more architectural techniques than they can economically apply on a project. Yet, unlike the overall field of software development, developers have relatively little guidance on how to choose and apply software architecture techniques.

To understand the guidance that has been offered, it is helpful to set up a strawman dichotomy. On the one hand we have developers who are creation-centric and who justifiably revel in their uniquely human ability to create. Their thoughts are only on how to create the next bit of functionality. On the other hand of our strawman dichotomy, we have the developers who have seen soaring hopes crash down and who remind us that our creations all too often fail. Their thoughts are only on how to safeguard against the next failure.

In this strawman dichotomy, the creation-centric developers would evolve the architecture ad hoc and follow an *evolutionary design* model [15, 6]. Evolutionary design does not suggest that architecture is unimportant, but rather advises that it is hard to get correct up-front. They also suggest that a squeaky clean codebase, frequently refactored, is amenable to architectural rework. So the right choice is to make architecture decisions as late as possible.

The cautious developers from the strawman would work out architectures in advance and follow a *documentation package* model [10]. Our systems will have challenging requirements, and our architecture choices will make those requirements easier or harder to achieve. We should find out in advance what those *architecture drivers* are and create a full architecture description to avoid failure.

Each of us is a mixture of those two strawmen in the dichotomy. Parts of us delight at what lines of code can create; other parts worry how our creations will fail. Indeed, every developer understands that failures must be avoided because failure avoidance is central to any engineering discipline. Henry Petroski, a leading historian of engineering, says 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. [23]

Notice that Petroski describes solutions to design problems, not the overall engineering process or how to obtain accurate requirements. He says simply that to design something that works, you must consider how it can go wrong and avoid those possibilities.

2. RISK-CENTRIC MODEL

So we are faced with a dilemma. We must balance our desire to euphorically race forward and create with our need to assiduously investigate and avoid failures. Developers want to succeed yet they must economize time and money. Software architecture techniques could help them succeed, but they are expensive.

*This paper contains excerpts from the forthcoming book, Risk-Centric Software Architecutre by George Fairbanks, to appear 2010, published by Taylor and Francis, Alan Apt editor.

Unquestionably, many development teams have struck a balance on their own projects using their own criteria. What we need, however, is a repeatable and generic way to strike that balance. This paper introduces the risk-centric model for choosing a subset of architecture techniques that correspond to the risks facing the project.

2.1 What it is

The *risk-centric model* guides developers to apply a minimal set of architecture techniques to reduce their most pressing risks. The risk-centric model advises 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

It is similar to the spiral model of software development [7] in that both focus on risk and work on the highest risk items first. However, the risk-centric model applies only to architecture, while the spiral model applies to overall software development. The spiral model is applied once to a project and ends when the software has been completed. In contrast, the risk-centric model will be used at the beginning of projects as well as time-to-time during the project. It ends when architecture risks have been mitigated.

The risk-centric model also adapts an important feature from Attribute Driven Design (ADD) [5]. ADD uses a mapping from quality attributes to tactics to address them. For example, ADD uses a mapping from the quality attribute "availability" to the tactic "ping/echo", which can be used to achieve availability. Similarly, the risk-centric model uses a mapping from risks to architecture techniques that address them. For example, it can use the mapping from the risk "protocol may deadlock" to the technique "analyze protocol using FSP".

The key element of the risk-centric model is the promotion of risk to prominence. What we choose to promote has an impact. Most developers already think about risks, but they think about lots of other things too. A recent paper described how a team that had previously done up-front architecture work switched to a feature-driven agile process and ended up deferring quality attribute concerns — in fact they were deferred until active development ceased and the system was in maintenance [4]. The conclusion to draw is not that agile processes cannot handle quality attribute requirements, but that a team that has promoted features to prominence will indeed pay less attention to other areas, including risks.

We do not want to waste time on low-impact techniques, nor do we want to ignore project-threatening risks. We want to build successful systems by taking a path that spends our time most effectively. That means only applying techniques when they are motivated by risks.

2.2 What it is not

Most developers believe that they already follow a risk-centric model, or something close to it. However, in practice you see developers using techniques that are inefficient or ineffective at reducing risks, or not using a technique that could help. Examining the overall context of software development reveals why this can occur. Most organizations guide developers to follow a process and often provide some kind of documentation template. Developers are guided to activities by the process or template, rather than being guided by risks particular to their project.

Each project will face a different set of risks. Consequently, choosing the set of architecture techniques, or choosing a fixed 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.

Another telltale sign that a project is not using a risk-centric model is an inability to list the risks they confront and the corresponding techniques they are applying. For a team using the risk-centric model, this should be equally easy as listing features for a team following a feature-driven model.

2.3 Enabling variation

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 takeaway point is that the risk-centric model advises developers to use techniques corresponding to project risks, yet too often one finds developers over- or under-applying techniques since they are not explicitly evaluating 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 we choose an appropriate set of techniques? And when do we stop architecting and start building? The following sections will dig into these questions in more detail.

3. RISKS

Since both the probability of failure and the impact are uncertain, we bundle the concept of uncertainty into our definition of risk, rather than talking about perceived risks versus actual risks. Our definition of risk then becomes:

$$risk = perceivedProbabililtyOfFailure \times perceivedImpact$$

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

3.1 Describing risks

You can state a risk categorically, for example "modifiability" or "throughput". But often this is too vague to be actionable: if we do something, how can we tell if the risk is actually reduced? Another way of saying this is that we must describe risks well enough so that we can later test to see if it has been mitigated. 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."

3.2 Engineering and non-engineering risks

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

We contrast engineering risks with *project management risks*, which relate to schedules, sequencing of work, delivery, team size,

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

Table 1: Examples of engineering risk reduction techniques in software architecture and other fields

geography, etc. Examples of these include: "Lead developer hit by bus", "Customer needs not understood", or "Senior VP hates our manager".

The technique type must match the risk type. We separate engineering risks from other risks because only engineering techniques will mitigate engineering risks. You cannot use a PERT chart to reduce the chance of buffer overruns, nor will UML resolve stakeholder disagreements.

3.3 Identifying risks

Experienced developers have an easy time identifying risks but what can be done if we are less experienced or working in an unfamiliar domain? The easiest place to start is with the requirements, whatever form they take, and look for things that seem difficult to achieve. Stakeholders often fail to clearly articulate quality attribute requirements, so we can elicit them formally or informally to find challenging ones.

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 are valuable knowledge for less experienced developers and a helpful reminder for experienced ones.

4. TECHNIQUES

Once we know what risks we are facing, we can apply *techniques* that we expect to reduce the risk. The term technique is quite broad, so we will focus specifically on software architecture risk reduction techniques, but for convenience continue to use the simple name *technique*. Table 1 shows a short list of software engineering techniques and techniques from other engineering branches.

4.1 Analyses and 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 one approach has an analytical character while the other 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* [5] or *patterns* [24, 16], and include such solutions as using a process monitor, a forwarder-receiver, or a model-view-controller.

4.2 Techniques mitigate risks

Design is a mysterious process, where virtuosos can make leaps of reasoning between problems and solutions [25]. To make a repeatable process, however, we need to make explicit what the virtuosos are doing tacitly. In this case, we need to make explicit 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>.*

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 require developers to invent techniques on the fly. We can improve the repeatability of designing software architectures by encoding the knowledge of virtuoso architects as mappings between risks and techniques.

4.3 Cannot eliminate engineering risk

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

The reason we cannot eliminate engineering risks is because we 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.

4.4 Optimal basket of techniques

To avoid wasting our time and money, we must choose techniques that best reduce our prioritized list of risks. We 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. Imagine you successfully used the technique of domain modeling on your last project, so you choose it for this project. You find three flaws in your design, and fix them. You might become convinced that employing that technique was a good idea, because you otherwise would not have found the three flaws. But such reasoning ignores the *opportunity cost*. The fair comparison is against the other techniques you could have used. If your biggest risk is that your chosen framework is inappropriate, you should spend your time analyzing the framework choice instead of domain modeling. 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.

5. WHEN TO STOP

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

Like other engineering fields, software engineering should focus on mitigating risks. We now know how to identify risks and have

techniques ready to combat many of them. We are tempted work until all engineering risks have been eliminated, but the time spent on that must be balanced with other, non-engineering risks, such as pleasing the customer and delivering products.

When deciding how much architecture to do, our guiding principle is: *Architecture efforts should be commensurate with the risk of failure.* If you are not worried about security failure scenarios, do not do any security design. However, if performance is an architecture driver, work on it until you are reasonably sure that performance will be OK, or until the risk of not delivering the project on time overshadows it. This does not mean that every detail has been decided, but rather that the architecture will support or enable you to achieve the architecture drivers. After that, you can shift gears and let design occur locally or use evolutionary design (discussed in Section **??**) to tweak the architecture.

After we have applied our chosen techniques, we must decide if the risk has been sufficiently mitigated, or if we need to continue our architecture and design activities. Where do we draw the line? This is primarily a subjective decision based on a developer's conviction that the current architecture or design will support the architecture drivers, but we do have some yardsticks to guide us.

5.1 Yardsticks

We have empirical data that suggests how much time should be spent on architecture and design. Barry Boehm has calculated the optimal amount of architecture for small, medium, and large projects based on a variant of his COCOMO model [8]. His data indicates that the largest projects should spend more time on architecture, as much as a third of the total time.

A yardstick like "spend 20% of your time on design" can be used for rough planning of activities, yielding a time budget to spend in design. However, 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 just start 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.

When you are in the trenches, though, you do not use that yardstick to measure your progress, but instead measure it by how well your design activities have mitigated your identified risks. Using risk to help decide when to stop is therefore an improvement over simple yardsticks. And, unlike yardsticks that only provide categorical time budgets, it guides us to appropriate kinds of architecture and design activities.

5.2 Subjectivity

Unfortunately, the risk-centric model is riddled with subjectivity: risk identification, variation in priority, choice of techniques, and evaluation of risk mitigation will all vary depending on the who does them. Experienced developers will likely perform better than novices.

Despite the subjectivity, we are way ahead because a risk-centric approach yields arguments that can be evaluated. An example argument would take the form: we identified A, B, and C as the biggest risks, with B being an architecture driver, so we spent time applying a techniques X and Y, evaluated the resulting design, and decided the risk of B was sufficiently low, so prototyping or development could continue. For any of the subjective points, another developer could provide a differing evaluation, perhaps suggesting that risk D be included, and a rational discussion would ensue.

It is hard to imagine an approach that could eliminate subjectivity, for example one that could say definitively that a project contained no performance risks, or definitively that a security risk had been eliminated. Instead, the broad question of "How much software architecture should we do?" has been transformed into a narrow one, "Have our chosen techniques sufficiently reduced our failure risks?"

6. STYLES OF DESIGN

Software architecture takes place in a larger design context that includes design styles and software development processes. This section discusses three styles of design — evolutionary, planned, and minimal planned — and the next section discusses software development processes.

6.1 Evolutionary design

Evolutionary design "means that the design of the system grows as the system is implemented" [15]. Chaos usually results because all design decisions are made locally and without coordination, creating a hodgepodge system that is hard to maintain and evolve any further.

Recent trends in software processes have re-invigorated evolutionary design by avoiding 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 [14]) cleans up the uncoordinated local designs. Some argue that these practices are sufficiently powerful that planned design can be avoided entirely [6].

Of the three practices, refactoring is the workhorse that enables 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 evolution from a tiered architecture to a service-oriented architecture [17] 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 remembering 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.

6.2 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. However, almost no one advocates doing planned design for the entire system, sometimes called *Big Design Up Front (BDUF)*, except for Model Driven Engineering (MDE) folks who generate code from their models. However, planning just the architecture is advocated [18, 5], since it is often hard to know on a large or complex project that *any* system can satisfy the requirements.

Planned architecture design is also helpful when a shared, central architecture is shared by many sub-teams working in parallel, and therefore useful to know from the beginning. 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 decides on some properties that must hold, such as setting up a concurrency policy, allocating high-level responsibilities, and defining some localized quality attribute scenarios.

6.3 Minimal planned design

In between evolutionary design and planned design is *minimal planned design*, or *Little Design Up Front* [20]. 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 [27].

One way to strike the balance between the two kinds of design is to ensure that the architecture will support its architecture drivers during some initial planned design. 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.

In special cases we have better guidance on how to balance planned and evolutionary design. When we are concerned primarily with how well the architecture will support global or emergent qualities, we can do planned design to ensure these and reserve any remaining design as evolutionary or local design. For example, if we have identified throughput as our architecture driver, we 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* [12] to set up an architecture known to achieve our architecture drivers, allowing freedom in the rest of design.

6.4 Using the risk-centric model

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

While there are clear differences, planned design and evolutionary design are both kinds of design. Developers must design software before they write the code, whether it is ten minutes before or ten months before. The essential tension is this: a long head-start on architectural design yields opportunities to avoid design dead-ends, ensure global properties, and coordinate sub-teams 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. Each of these design styles says that design should happen but leaves open how it should proceed. Applying the risk-centric model to planned design means doing up-front design until architecture risks have subsided. Applying it to evolutionary design means doing architecture design ad hoc during development, whenever a risk looms sufficiently large. Applying it to minimal planned design is just a combination of the two.

7. SKETCH: ADDING RISK TO AGILE PROCESSES

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

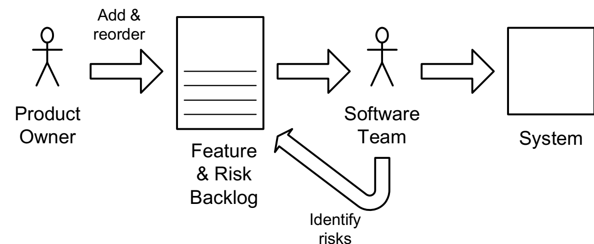

Figure 1: One way to incorporate risk into an agile process is to convert the feature backlog into a feature & risk backlog. The product owner adds features and the software team adds technical risks. The software team must help the product owner to understand the technical risks and suitably prioritize the backlog.

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. We can piggyback here to start mitigating the identified risks. Developers could have a simple whiteboard meeting to ensure everyone agrees on an architectural style, or come up with a short list of styles to investigate. If performance characteristics of COTS components are unknown but important, some quick prototyping can be done to provide approximate speed or throughput numbers.

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

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 we introduce risks, because we need to prioritize both features and risks. Some risks are small enough that they can be handled as they arise during an iteration, but larger risks will need to be scheduled just like features are. Whenever possible, risks should be written up as testable features.

If we give the product owner the additional responsibility to prioritize architectural risks alongside features, we can simply change the feature backlog into a feature & risk backlog, as seen in Figure 1. Software developers may see a feature low in the backlog asking for security. It is their job to educate the product owners that if they ever want to have a secure application, they need to address that risk early, since it will be difficult or impossible to add later. As part of the reflection at the end of each iteration, we need to evaluate architectural risks and feed these into the backlog.

In summary, we can handle architectural risks in an agile process

by doing two things. Architectural risks that we know in advance can be 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 must be promoted to be on par with features, and inserted into a combined feature & risk backlog.

8. CONCLUSION

In this paper, we set out to understand how focusing on risks could help us to efficiently use software architecture. We wanted to walk a middle path that avoided the extremes of complete architecture documentation packages and total architecture avoidance. To compromise, we followed the principle that our architecture efforts should be commensurate with the risk of failure. Avoiding failure is central to all engineering and we can use architecture techniques to mitigate the risks we identify. The key element of 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 our time and money, we must choose techniques that best reduce our prioritized list of risks.

The question of how much software architecture work we should do has been a thorny one for a long time. The risk-centric model transforms that broad question into a narrow one, "Have our chosen techniques sufficiently reduced our 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. We 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, we ensure that whatever time we 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 avoid design dead-ends, ensure global properties, and coordinate sub-teams 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.

9. RELATED WORK

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 [19].

The idea of focusing on risk is not a new one, either in engineering as a whole or in software engineering specifically. Barry Boehm wrote about risk in the context of software development with his paper on the spiral model of software development [7], which is an interesting read even if you already understand the model. He followed this up with a recent book on risk and agile processes [8]. 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?"

Authors have been advocating building minimally sufficient models for years, including Desmond D'Souza [11] and Scott Ambler [3]. Tailoring the models built on a project to the nature of the project (greenfield, brownfield, coordination, enhancement) is discussed in [13].

The idea of cataloging techniques, or tactics, is described in the context of Attribute Driven Design in [5]. They develop a table of tactics that are mapped to quality attributes they help achieve. The concept in this book of mapping development techniques is similar in nature. 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.

Though tactics and techniques have so far been expressed as tables, they could be expressed as a pattern language, as originally described by Christopher Alexander for the domain of buildings [2, 1], and later adapted to software in the Design Patterns book [16] by Erich Gamma and others.

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 [25]. 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.

Martin Fowler's essay, "Is Design Dead?" [15] 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 [21] proposes the creation of a Risk Management Forum to prioritize risk across products and projects in an organization. Since our goal here is to handle architecture risks that are only a subset of all project risks, a smaller change to the process may work.

10. REFERENCES

[1] C. Alexander. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, USA, 1977.

[2] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] S. Ambler. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, 1st edition, 3 2002.

[4] M. A. Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. *Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009*, 2009.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition edition, 2003.

[6] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2 edition, 11 2004.

[7] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[8] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, 1 edition, 8 2003.

[9] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach (ACM Press)*. Addison-Wesley Professional, 5 2000.

[10] P. Clements, F. Bachmann, L. Bass, D. Garlan, JamesIvers,

R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[11] D. F. D'Souza and A. C. Wills. *Objects,Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[12] G. Fairbanks. *Risk-Centric Software Architecture*. Taylor and Francis, To appear 2010.

[13] G. Fairbanks, K. Bierhoff, and D. D'Souza. Software architecture at a large financial firm. *Proceedings of ACM SIGPLAN Conference on Object Oriented Programs, Systems, Languages, and Applications (OOPSLA) 2006*, 2006.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 7 1999.

[15] M. Fowler. Is design dead? 2004. http://martinfowler.com/articles/designDead.html.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-WesleyProfessional Computing Series)*. Addison-Wesley Professional, 1995.

[17] T. Hoff. Amazon architcture, 2008. http://highscalability.com/amazon-architecture.

[18] A. J. Lattanze. *Architecting Software Intensive Systems: A Practitioners Guide*. Auerbach Publications, 1 edition, 11 2008.

[19] N. Luhmann. Modern society shocked by its risks. *Social Sciences Research Centre: Occasional Papers*, 1996.

[20] R. Martin. The scatology of agile architecture, April 2009. http://blog.objectmentor.com/articles/2009/04/25/the-scatology-of-agile-architecture.

[21] J. Nyfjord. *Towards Integrating Agile Development and Risk Management*. PhD thesis, Stockholm University, 2008.

[22] D. E. Perry and A. L. Wolf. Foundation for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[23] H. Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, 5 1994.

[24] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 1 edition, 9 2000.

[25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[26] R. Taylor, N. Medvidović, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 11 2008.

[27] B. Venners. A conversation with martin fowler, part iii. *Artima Developer*, 2002. http://www.artima.com/intv/evolutionP.html.