Risk-First

Software Development De-Risked

Volume 1: The Menagerie

Rob Moffat

Risk-First: The Menagerie

By Rob Moffat

ISBN: 9781717491855

Credits

tbd

Cover Images: Biodiversity Heritage Library. Biologia Centrali-Americana. Insecta. Rhynchota. Hemiptera-Homoptera. Volume 1 (1881-1905)

Cover Design By P. Moffat (`peter@petermoffat.com`)

Thanks to:

Books In The Series

- **Risk-First: The Menagerie:** Book one of the Risk-First series argues the case for viewing *all* of the activities on a software project through the lens of *managing risk*. It introduces the menagerie of different risks you're likely to meet on a software project, naming and classifying them so that we can try to understand them better.
- **Risk-First: Tools and Practices:** Book two of the Risk-First series explores the relationship between software project risks and the tools and practices we use to mitigate them. Due for publication in 2020.

Online

Material for the books is freely available to read, drawn from `risk-first.org`.

Published By

`Kite9 Ltd.`
`14 Manor Close`

Colchester
CO6 4AR

Contents

Contents iii

Preface v

Executive Summary xi

I Introduction 1

1 A Simple Scenario 3

2 Development Process 7

3 Meeting Reality 15

4 All Risk Management 23

5 Evaluating Risk 29

6 Cadence 43

7 De Risking 47

8 A Conversation 53

II Risk 57

III Application 59

9 Coming Next 61

10 Estimates 63

Preface

Welcome to Risk-First!

Let's cover some of the big questions up-front: The why, what, who, how and where of *The Menagerie*.

Why

> Scrum, Waterfall, Lean, Prince2: what do they all have in common?

I've started this because, on my career journey, I've noticed that the way I do things doesn't seem to match up with the way the books *say* it should be done. And, I found this odd and wanted to explore it further. Hopefully, you, the reader, will find something of use in this.

I started with this observation: *Development Teams* put a lot of faith in methodology. Sometimes, this faith is often so strong it borders on religion. (Which in itself is a concern.) For some, this is Prince2. For others, it might be Lean or Agile.

Developers put a lot of faith in *particular tools* too. Some developers are pro-or-anti-Java, others are pro-or-anti-XML. All of them have their views coloured by their *experiences* (or lack of) with these tools. Was this because their past projects *succeeded* or *failed* because of them?

As time went by, I came to see that the choice of methodology, process or tool was contingent on the problem being solved, and the person solving the problem. We don't face a shortage of tools in IT, or a shortage of methodologies, or a shortage of practices. Essentially, that all the tools and methodologies that the industry had supplied were there to help *minimize the risk of my project failing*.

This book considers that perspective: that building software is all about *managing risk*, and that these methodologies are acknowledgements of this

fact, and they differ because they have *different ideas* about which are the most important *risks to manage*.

What This Is

Hopefully, after reading this, you'll come away with:

- An appreciation of how risk underpins everything we do as developers, whether we want it to or not.
- A framework for evaluating methodologies, tools and practices and choosing the right one for the task-at-hand.
- A recontextualization of the software process as being an exercise in mitigating different kinds of risk.
- The tools to help you decide when a methodology or tool is *letting you down*, and the vocabulary to argue for when it's a good idea to deviate from it.

This is not intended to be a rigorously scientific work: I don't believe it's possible to objectively analyze a field like software development in any meaningful, statistically significant way. (For one, things just change too fast.)

> "I have this Pattern"
>
> —Attributed to Ward Cunningham, *Have This Pattern, C2 Wiki*[1]

Does that diminish it? If you have visited the TVTropes[2] website, you'll know that it's a set of web-pages describing *common patterns* of narrative, production, character design etc. to do with fiction. For example:

> "Sometimes, at the end of a Dream Sequence or an All Just a Dream episode, after the character in question has woken up and demonstrated any [lesson] that the dream might have been communicating, there's some small hint that it wasn't a dream after all, even though it quite obviously was. . . right?."
>
> —Or Was It a Dream?, *TVTropes*[3]

[1] `http://c2.com/ppr/wiki/WikiPagesAboutWhatArePatterns/HaveThisPattern.html`

[2] `https://tvtropes.org`

[3] `https://tvtropes.org/pmwiki/pmwiki.php/Main/OrWasItADream`

Is it scientific? No. Is it correct? Almost certainly. TVTropes is a set of *empirical patterns* for how stories on TV and other media work. It's really useful, and a lot of fun. (Warning: it's also incredibly addictive).

In the same way, "Design Patterns: Elements of Reusable Object-Oriented Software[4]", is a book detailing patterns of *structure* within Object-Oriented programming, such as:

> "[The] Adapter [pattern] allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class. . . "
>
> —Design Patterns, *Wikipedia*[5]

Patterns For Practitioners

Design Patterns aimed to be a set of *useful* patterns which practitioners could use in their software to achieve certain goals. "I have this pattern" was a phrase used to describe how they had seen a certain set of constraints before, and how they had solved it in software.

This book was a set of experts handing down their battle-tested practices for other developers to use, and, whether you like patterns or not, knowing them is an important part of being a software developer, as you will see them used everywhere you go and probably use them yourself.

In the same way, Risk-First aims to be a set of *Patterns for Software Risk*. Hopefully after reading this book, you will see where risk hides in software projects, and have a name for it when you see it.

Towards a "Periodic Table"

In the latter chapters of "The Menagerie" we try to assemble these risk patterns into a cohesive whole. Projects fail because of risks, and risks arise from predictable sources.

What This is Not

This is not intended to be a rigorously scientific work: I don't believe it's possible to objectively analyze a field like software development in any meaningful, statistically significant way. (For one, things just change too fast.)

[4] http://amzn.eu/d/3cOwTkH

[5] https://en.wikipedia.org/wiki/Design_Patterns

Neither is this site isn't going to be an exhaustive guide of every possible software development practice and methodology. That would just be too long and tedious.

Neither is this really a practitioner's guide to using any particular methodology: If you've come here to learn the best way to do Retrospectives, then you're in the wrong place. There are plenty of places you can find that information already. Where possible, this site will link to or reference concepts on Wikipedia or the wider internet for further reading on each subject.

Who

This work is intended to be read by people who work on software projects, and especially those who are involved in managing software projects.

If you work collaboratively with other people in a software process, you should find Risk-First a useful lexicon of terms to help describe the risks you face.

But here's a warning: This is going to be a depressing book to read. It is book one of a two-book series, but in **Book One** you only get to meet the bad guy.

While **Book Two** is all about *how to succeed*, This book is all about how projects *fail*. In it, we're going to try and put together a framework for understanding the risk of failure, in order that we can reconstruct our understanding of our activities on a project based on avoiding it.

So, if you are interested in *avoiding your project failing*, this is probably going to be useful knowledge.

For Developers

Risk-First is a tool you can deploy to immediately improve your ability to plan your work.

Frequently, as developers we find software methodologies "done to us" from above. Risk-First is a toolkit to help *take apart* methodologies like Scrum, Lean and Prince2, and understand them. Methodologies are *bicycles*, rather than *religions*. Rather than simply *believing*, we can take them apart and see how they work.

For Project Managers and Team Leads

All too often, Project Managers don't have a full grasp of the technical details of their projects. And this is perfectly normal, as the specialization belongs

below them. However, projects fail because risks materialize, and risks materialize because the devil is in those details.

This seems like a lost cause, but there is hope: the ways in which risks materialize on technical projects is the same every time. With Risk-First we are attempting to name each of these types of risk, which allows for a dialog with developers about which risks they face, and the order they should be tackled.

Risk-First allows a project manager to pry open the black box of development and talk with developers about their work, and how it will affect the project. It is another tool in the (limited) arsenal of techniques a project manager can bring to bear on the task of delivering a successful project.

How

One of the original proponents of the Agile Manifesto, Kent Beck, begins his book Extreme Programming by stating:

> "It's all about risk"
>
> —Kent Beck, *Extreme Programming Explained*[6]

This is a promising start. From there, he introduces his methodology, Extreme Programming, and explains how you can adopt it in your team, the features to observe and the characteristics of success and failure. However, while *Risk* has clearly driven the conception of Extreme Programming, there is no clear model of software risk underpinning the work, and the relationship between the practices he espouses and the risks he is avoiding are hidden.

In this book, we are going to introduce a model of software project risk. This means that in **Book Two** (Risk-First: Tools and Practices), we can properly analyse Extreme Programming (and Scrum, Waterfall, Lean and all the others) and *understand* what drives them. Since they are designed to deliver successful software projects, they must be about mitigate risks, and we will uncover *exactly which risks are mitigated* and *how they do it*.

[6] http://amzn.eu/d/gUQjnbF

Where

All of the material for this book is available Open Source on github.com[7], and at the risk-first.org[8] website. Please visit, your feedback is appreciated.

There is no compulsion to buy a print or digital version of the book, but we'd really appreciate the support. So, if you've read this and enjoyed it, how about buying a copy for someone else to read?

A Note on References

Where possible, references are to the Wikipedia[9] website. Wikipedia is not perfect. There is a case for linking to the original articles and papers, but by using Wikipedia references are free and easy for everyone to access, and hopefully will exist for a long time into the future.

On to The Executive Summary

[7] https://github.com

[8] https://risk-first.org

[9] https://wikipedia.org

Executive Summary

1. There are Lots of Ways of Running Software Projects

There are lots of different ways to look at a project. For example, metrics such as "number of open tickets", "story points", "code coverage" or "release cadence" give us a numerical feel for how things are going and what needs to happen next. We also judge the health of projects by the practices used on them - Continuous Integration, Unit Testing or Pair Programming, for example.

Software methodologies, then, are collections of tools and practices: "Agile", "Waterfall", "Lean" or "Phased Delivery" (for example) all suggest different approaches to running a project, and are opinionated about the way they think projects should be done and the tools that should be used.

None of these is necessarily more "right" than another- they are suitable on different projects at different times.

A key question then is: **how do we select the right tools for the job?**

2. We can Look at Projects in Terms of Risks

One way to examine a project in-flight is by looking at the risks it faces.

Commonly, tools such as RAID logs and RAG status reporting are used. These techniques should be familiar to project managers and developers everywhere.

However, the Risk-First view is that we can go much further: that each item of work being done on the project is mitigating a particular risk. Risk isn't something that just appears in a report, it actually drives *everything we do*.

For example:

- A story about improving the user login screen can be seen as reducing *the risk of users not signing up*.

- A task about improving the health indicators could be seen as mitigating *the risk of the application failing and no-one reacting to it*.
- Even a task as basic as implementing a new function in the application is mitigating *the risk that users are dissatisfied and go elsewhere*.

One assertion of Risk-First therefore, is that every action you take on a project is to mitigate some risk.

3. We Can Break Down Risks on a Project Methodically

Although risk is usually complicated and messy, other industries have found value in breaking down the types of risks that affect them and addressing them individually.

For example:

- In manufacturing, *tolerances* allow for calculating the likelihood of defects in production.
- In finance, reserves are commonly set aside for the risks of stock-market crashes, and teams are structured around monitoring these different risks.
- The insurance industry is founded on identifying particular risks and providing financial safety-nets for when they occur, such as death, injury, accident and so on.

Software risks are difficult to quantify, and mostly, the effort involved in doing so *exactly* would outweigh the benefit. Nevertheless, there is value in spending time building *classifications of risk for software*. That's what Risk-First does: describes the set of *risk patterns* we see every day on software projects.

With this in place, we can:

- Talk about the types of risks we face on our projects, using an appropriate language.
- Expose Hidden Risks that we hadn't considered before.
- Weigh the risks against each other, and decide which order to tackle them.

4. We Can Analyse Tools and Techniques in Terms of how they Mitigate Risk

If we accept the assertion above that *all* the actions we take on a project are about mitigating risks, then it stands to reason that the tools and techniques available to us on a project are there for mitigating different types of risks.

For example:

- If we do a Code Review, we are partly trying to mitigate the risks of bugs slipping through into production, and also mitigate the Key-Man Risk of knowledge not being widely-enough shared.
- If we write Unit Tests, we're also mitigating the risk of bugs going to production, but we're also mitigating against future changes breaking our existing functionality.
- If we enter into a contract with a supplier, we are mitigating the risk of the supplier vanishing and leaving us exposed. With the contract in place, we have legal recourse against this risk.

Different tools are appropriate for mitigating different types of risks.

5. Different Methodologies for Different Risk Profiles

In the same way that our tools and techniques are appropriate to dealing with different risks, the same is true of the methodologies we use on our projects. We can use a Risk-First approach to examine the different methodologies, and see which risks they address.

For example:

- **Agile** methodologies prioritise mitigating the risk that requirements capture is complicated, error-prone and that requirements change easily.
- **Waterfall** takes the view that coding effort is an expensive risk, and that we should build plans up-front to avoid it.
- **Lean** takes the view that risk lies in incomplete work and wasted work, and aims to minimize that.

Although many developers have a methodology-of-choice, the argument here is that there are tradeoffs with all of these choices. Methodologies are like *bicycles*, rather than *religions*. Rather than simply *believing*, we can take them apart and see how they work.

We can place methodologies within a framework, and show how choice of methodology is contingent on the risks faced.

6. Driving Development With a Risk-First Perspective

We have described a model of risk within software projects, looking something like this:

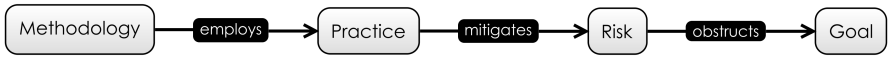


Figure 1: Methdologies, Risks, Practices

How do we take this further?

The first idea we explore is that of the Risk Landscape: Although the software team can't remove risk from their project, they can take actions that move them to a place in the Risk Landscape where the risks on the project are more favourable than where they started.

From there, we examine basic risk archetypes you will encounter on the software project, to build up a Taxonomy of Software Risk, and look at which specific tools you can use to mitigate each kind of risk.

Then, we look at different software practices, and how they mitigate various risks. Beyond this we examine the question: *how can a Risk-First approach inform the use of this technique?*

For example:

- If we are introducing a **Sign-Off** in our process, we have to balance the risks it *mitigates* (coordination of effort, quality control, information sharing) with the risks it *introduces* (delays and process bottlenecks).
- If we have **Redundant Systems**, this mitigates the risk of a *single point of failure*, but introduces risks around *synchronizing data* and *communication* between the systems.
- If we introduce **Process**, this may make it easier to *coordinate as a team* and *measure performance* but may lead to bureaucracy, focusing on the wrong goals or over-rigid interfaces to those processes.

Risk-First aims to provide a framework in which we can *analyse these choices* and weigh up *accepting* versus *mitigating* risks.

Still interested? Then dive into reading the introduction.

Part I

Introduction

CHAPTER 1

A Simple Scenario

In this chapter, I'm going to introduce some terms for thinking about risk.

Lets for a moment forget about software completely, and think about *any endeavour at all* in life. It could be passing a test, mowing the lawn or going on holiday. Choose something now. I'll discuss from the point of view of "cooking a meal for some friends", but you can play along with your own example.

1.1 Goal In Mind

Now, in this endeavour, we want to be successful. That is to say, we have a **Goal In Mind**: we want our friends to go home satisfied after a decent meal, and not to feel hungry. As a bonus, we might also want to spend time talking with them before and during the meal. So, now to achieve our Goal In Mind we *probably* have to do some tasks.

Since our goal only exists *in our head*, we can say it is part of our **Internal Model** of the world. That is, the model we have of reality. This model extends to *predicting what will happen*.

If we do nothing, our friends will turn up and maybe there's nothing in the house for them to eat. Or maybe, the thing that you're going to cook is going to take hours and they'll have to sit around and wait for you to cook it and they'll leave before it's ready. Maybe you'll be some ingredients short, or maybe you're not confident of the steps to prepare the meal and you're worried about messing it all up.

1.2 Attendant Risk

These *nagging doubts* that are going through your head I'll call the Attendant Risks: they're the ones that will occur to you as you start to think about what will happen.

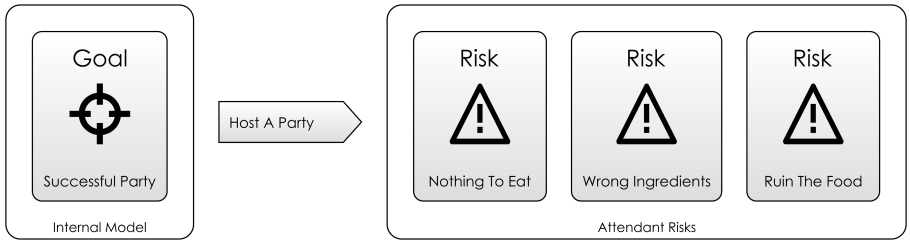


Figure 1.1: Goal In Mind, with the risks you know about

When we go about preparing this wonderful evening, we can choose to deal with these risks: shop for the ingredients in advance, prepare parts of the meal, maybe practice the cooking in advance. Or, we can wing it, and sometimes we'll get lucky.

How much effort we expend on these Attendant Risks depends on how big we think they are. For example, if you know there's a 24-hour shop, you'll probably not worry too much about getting the ingredients well in advance (although, the shop *could still be closed*).

1.3 Hidden Risks

There are also **Hidden Risks** that you *don't* know about: if you're poaching eggs for dinner, perhaps you don't know that fresh eggs poach best. The difference is, Attendant Risks are risks you are aware of, but can't be sure of the amount they will impact you. Hidden Risks are ones you are unaware of.

Donald Rumsfeld[1] famously called these "Unknown Unknowns".

Different people evaluate risks differently, and they'll also *know* about different risks. What is an Attendant Risk for one person is a Hidden Risk for another.

Which risks we know about depends on our **knowledge** and **experience**, then. And that varies from person to person (or team to team).

[1] `https://en.wikipedia.org/wiki/There_are_known_knowns`

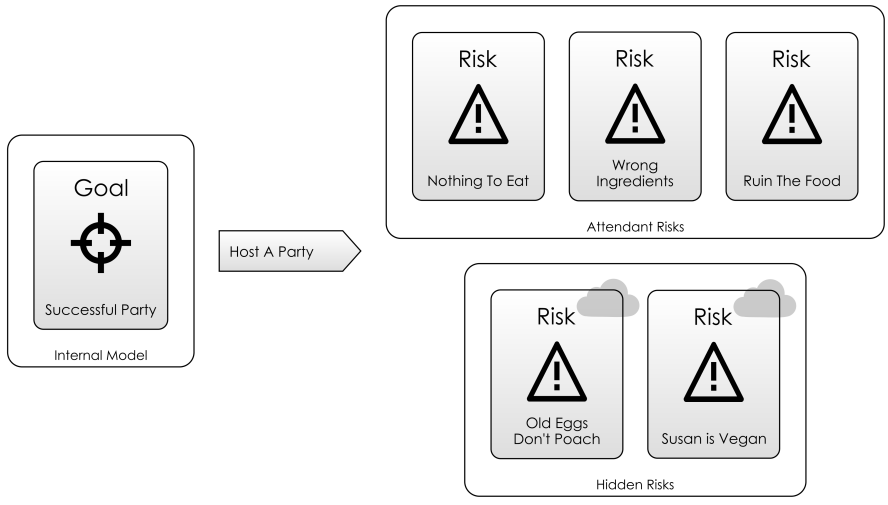


Figure 1.2: Goal In Mind, the risks you know about and the ones you don't

1.4 Meeting Reality

As the dinner party gets closer, we make our preparations, and the inadequacies of the Internal Model become apparent. We learn what we didn't know and the Hidden Risks reveal themselves. Other things we were worried about don't materialise. Things we thought would be minor risks turn out to be greater.

Our model is forced to Meet Reality, and the model changes, forcing us to deal with these risks, as shown in this diagram. Whenever we try to *do something* about a risk, it is called Taking Action. Taking Action *changes* reality, and with it your Internal Model of the risks you're facing. That's because it's only by interacting with the world that we add knowledge to our Internal Model about what works and what doesn't. Even something as passive as *checking the shop opening times* is an action, and it improves on our Internal Model of the world.

If we had a good Internal Model, and took the right actions, we should see positive outcomes. If we failed to manage the risks, or took inappropriate actions, we'll probably see negative outcomes.

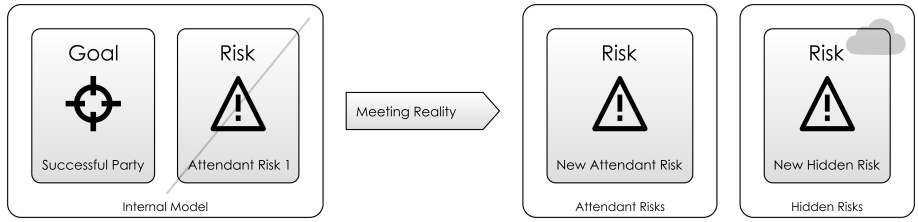


Figure 1.3: How Taking Action affects Reality, and also changes your Internal Model

1.5 On To Software

Here, we've introduced some new terms that we're going to use a lot: Meet Reality, Attendant Risk, Hidden Risk, Internal Model, Taking Action and Goal In Mind. And, we've applied them in a simple scenario.

But Risk-First is about understanding risk in software development, so let's examine the scenario of a new software project, and expand on the simple model being outlined above: instead of a single person, we are likely to have a team, and our model will not just exist in our heads, but in the code we write.

On to Development Process. . .

CHAPTER 2

Development Process

In the previous chapter we introduced some terms for talking about risk (such as Attendant Risk, Hidden Risk and Internal Model) via a simple scenario.

Now, let's look at the everyday process of developing *a new feature* on a software project, and see how our risk model informs it.

2.1 An Example Process

Let's ignore for now the specifics of what methodology is being used - we'll come to that later. Let's say your team have settled for a process something like the following:

1. **Specification**: A new feature is requested somehow, and a business analyst works to specify it.
2. **Code And Unit Test**: A developer writes some code, and some unit tests.
3. **Integration**: They integrate their code into the code base.
4. **UAT**: They put the code into a User Acceptance Test (UAT) environment, and user(s) test it.
5. . . . All being well, the code is **Released to Production**.

Now, the *methodology* being used might be Waterfall, it might be Agile. We're not going to commit to specifics at this stage. Also we don't need to consider whether this is particularly a *good* process: you could add code review, a pilot phase, integration testing, whatever. It's probably not perfect, but let's just assume that *it works for this project* and everyone is reasonably happy with it.

We're just doing some analysis of *what process gives us*.

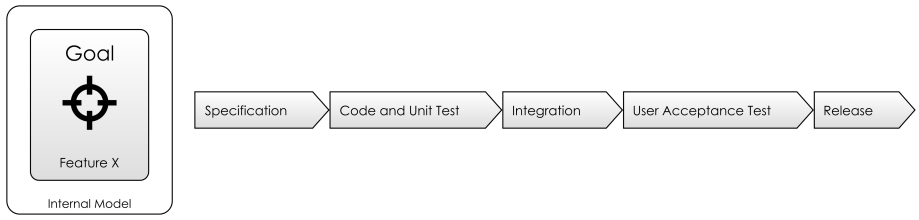


Figure 2.1: A Simple Development Process

2.2 Minimizing Risks - Overview

I am going to argue that this entire process is *informed by software risk*:

1. We have *a business analyst* who talks to users and fleshes out the details of the feature properly. This is to minimize the risk of **building the wrong thing**.
2. We *write unit tests* to minimize the risk that our code **isn't doing what we expected, and that it matches the specifications**.
3. We *integrate our code* to minimize the risk that it's **inconsistent with the other, existing code on the project**.
4. We have *acceptance testing* and quality gates generally to **minimize the risk of breaking production**, somehow.

We could skip all those steps above and just do this:

1. Developer gets wind of new idea from user, logs onto production and changes some code directly.

We can all see this would be a disaster, but why?

Two reasons:

1. You're Meeting Reality all-in-one-go: All of these risks materialize at the same time, and you have to deal with them all at once.
2. Because of this, at the point you put code into the hands of your users, your Internal Model is at its least-developed. All the Hidden Risks now need to be dealt with at the same time, in production.

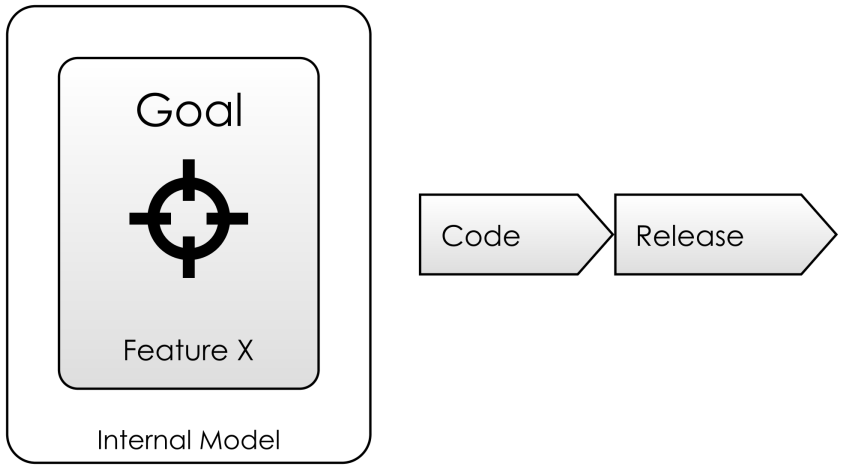


Figure 2.2: A Dangerous Development Process

2.3 Applying the Model

Let's look at how our process should act to prevent these risks materializing by considering an unhappy path, one where at the outset, we have lots of Hidden Risks. Let's say a particularly vocal user rings up someone in the office and asks for new **Feature X** to be added to the software. It's logged as a new feature request, but:

- Unfortunately, this feature once programmed will break an existing **Feature Y**.
- Implementing the feature will use some api in a library, which contains bugs and have to be coded around.
- It's going to get misunderstood by the developer too, who is new on the project and doesn't understand how the software is used.
- Actually, this functionality is mainly served by **Feature Z**. . .
- which is already there but hard to find.

The diagram below shows how this plays out.

This is a slightly contrived example, as you'll see. But let's follow our feature through the process and see how it meets reality slowly, and the Hidden Risks are discovered:

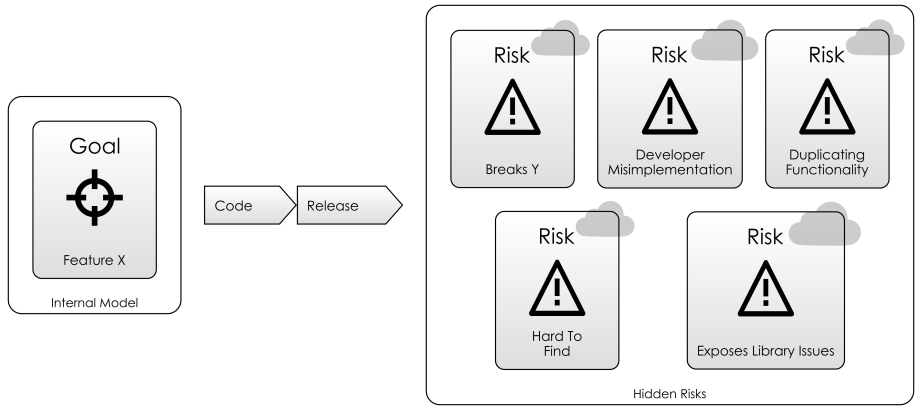


Figure 2.3: Development Process - Exposing Hidden Risks

Specification

The first stage of the journey for the feature is that it meets the Business Analyst (BA). The *purpose* of the BA is to examine new goals for the project and try to integrate them with *reality as they understands it*. A good BA might take a feature request and vet it against his Internal Model, saying something like:

- "This feature doesn't belong on the User screen, it belongs on the New Account screen"
- "90% of this functionality is already present in the Document Merge Process"
- "We need a control on the form that allows the user to select between Internal and External projects"

In the process of doing this, the BA is turning the simple feature request *idea* into a more consistent, well-explained *specification* or *requirement* which the developer can pick up. But why is this a useful step in our simple methodology? From the perspective of our Internal Model, we can say that the BA is responsible for:

- Trying to surface Hidden Risks
- Trying to evaluate Attendant Risks and make them clear to everyone on the project.

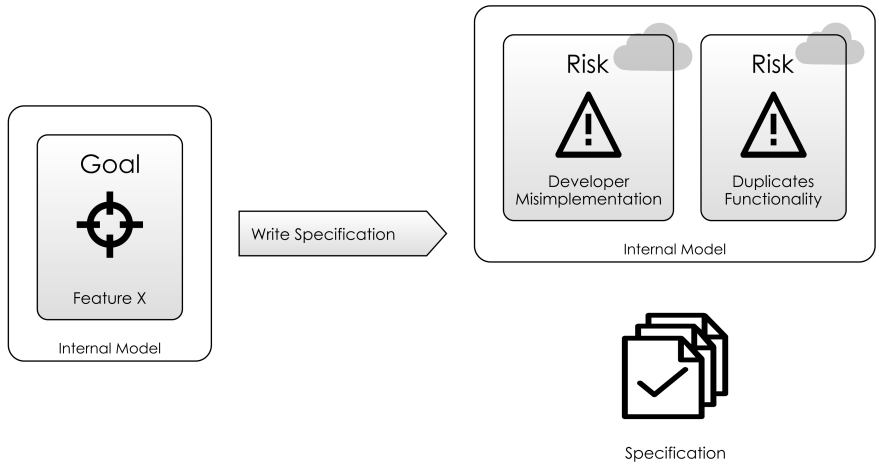


Figure 2.4: BA Specification: exposing Hidden Risks as soon as possible

In surfacing these risks, there is another outcome: while **Feature X** might be flawed as originally presented, the BA can "evolve" it into a specification, and tie it down sufficiently to reduce the risks. The BA does all this by simply *thinking about it*, *talking to people* and *writing stuff down*.

This process of evolving the feature request into a requirement is the BA's job. From our Risk-First perspective, it is *taking an idea and making it Meet Reality*. Not the *full reality* of production (yet), but something more limited.

Code And Unit Test

The next stage for our feature, **Feature X** is that it gets coded and some tests get written. Let's look at how our Goal In Mind meets a new reality: this time it's the reality of a pre-existing codebase, which has it's own internal logic.

As the developer begins coding the feature in the software, she will start with an Internal Model of the software, and how the code fits into it. But, in the process of implementing it, she is likely to learn about the codebase, and her Internal Model will develop.

At this point, let's stop and discuss the visual grammar of the Risk-First Diagrams we've been looking at. A Risk-First diagram shows what you

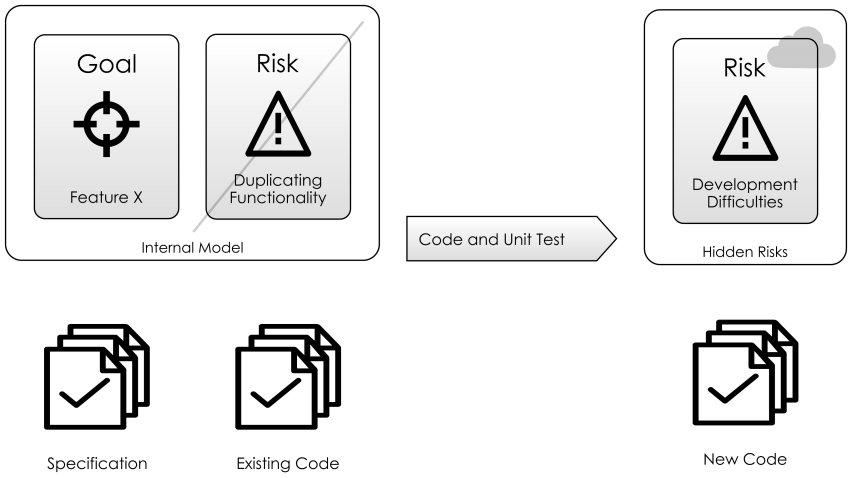


Figure 2.5: Coding Process: exposing more hidden risks as you code

expect to happen when you Take Action. The action itself is represented by the shaded, sign-post-shaped box in the middle. On the left, we have the current state of the world, on the right is the anticipated state *after* taking the action.

The round-cornered rectangles represent our Internal Model, and these contain our view of Risk, whether the risks we face right now, or the Attendant Risks expected after taking the action. In the diagram above, taking the action of "coding and unit testing" is expected to mitigate the risks of "Developer Misimplementation" and "Duplicating Functionality".

Beneath the internal models, we are also showing real-world tangible artifacts. That is, the physical change we would expect to see as a result of taking action. In this diagram, the action will result in "New Code" being added to the project, needed for the next steps of the development process.

Integration

Integration is where we run *all* the tests on the project, and compile *all* the code in a clean environment, collecting together the work from the whole development team.

So, this stage is about meeting a new reality: the clean build.

At this stage, we might discover the Hidden Risk that we'd break **Feature Y**

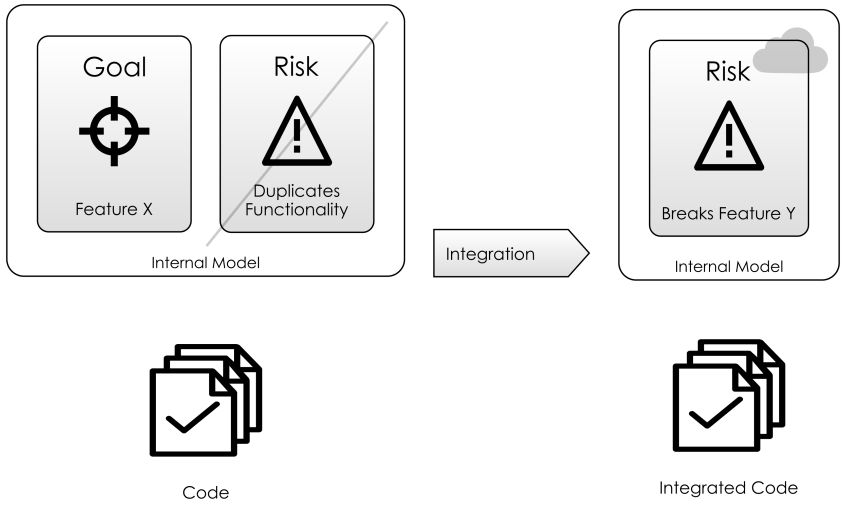


Figure 2.6: Integration testing exposes Hidden Risks before you get to production

User Acceptance Test

Next, User Acceptance Testing (UAT) is where our new feature meets another reality: *actual users*. I think you can see how the process works by now. We're just flushing out yet more Hidden Risks.

- Taking Action is the *only* way to create change in the world.
- It's also the only way we can *learn* about the world, adding to our Internal Model.
- In this case, we discover a Hidden Risk: the user's difficulty in finding the feature. (The cloud obscuring the risk shows that it is hidden).
- In return, we can *expect* the process of performing the UAT to delay our release (this is an attendant schedule risk).

2.4 Observations

First, the people setting up the development process *didn't know* about these *exact* risks, but they knew the *shape that the risks take*. The process builds "nets" for the different kinds of Hidden Risks without knowing exactly what they are.

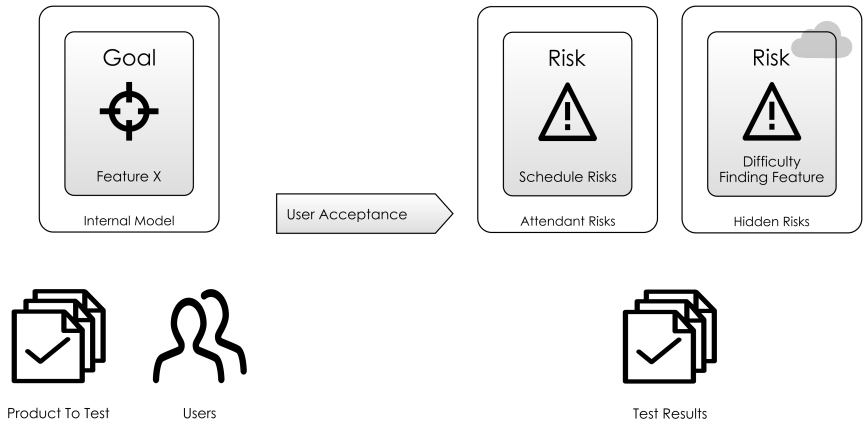


Figure 2.7: UAT - putting tame users in front of your software is better than real ones, where the risk is higher

Second, are these really risks, or are they *problems we just didn't know about*? I am using the terms interchangeably, to a certain extent. Even when you know you have a problem, it's still a risk to your deadline until it's solved. So, when does a risk become a problem? Is a problem still just a schedule-risk, or cost-risk? We'll come back to this question presently.

Third, the real take-away from this is that all these risks exist because we don't know 100% how reality is. We don't (and can't) have a perfect view of the universe and how it'll develop. Reality is reality, *the risks just exist in our head*.

Fourth, hopefully you can see from the above that really *all this work is risk management*, and *all work is testing ideas against reality*.

In the next chapter, we're going to look at the concept of Meeting Reality in a bit more depth.

CHAPTER 3

Meeting Reality

In this chapter, we will look at how exposing your Internal Model to reality is in itself a good risk management technique.

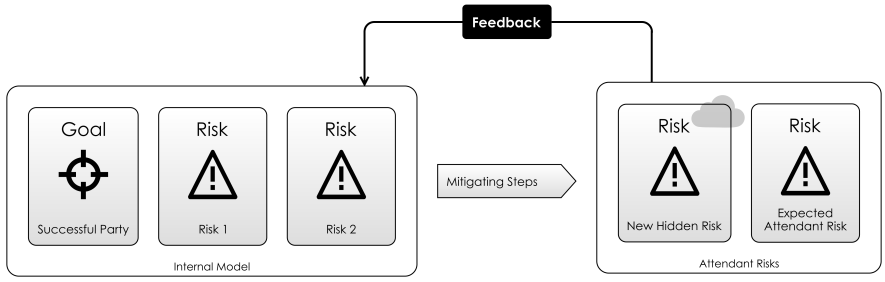
3.1 Revisiting the Model

In A Simple Scenario, we looked at a basic model for how **Reality** and our Internal Model interacted with each other: we take action based on out Internal Model, hoping to **change Reality** with some positive outcome.

And, in Development Process we looked at how we can meet with reality in *different forms*: Analysis, Testing, Integration and so on, and saw how the model could work in each stage of a project.

It should be no surprise to see that there is a *recursive* nature about this: The actions we take each day have consequences: they expose new hidden risks**, which inform our Internal Model, and at the same time, they change reality in some way. As a result, we then have to take *new actions* to deal with these new risks.

So, let's see how this kind of recursion looks on our model.



.

3.2 "Navigating the Risk Landscape"

The above diagram shows *just one possible action*, in reality, you'll have choices. We often have multiple ways of achieving a Goal In Mind.

What's the best way?

I would argue that the best way is the one which mitigates the most existing risk while accruing the least attendant risk to get it done.

Ideally, when you take an action, you are trading off a big risk for a smaller one. Take Unit Testing for example. Clearly, writing Unit Tests adds to the amount of development work, so on it's own, it adds Schedule Risk. However, if you write *just enough* of the right Unit Tests, you should be short-cutting the time spent finding issues in the User Acceptance Testing (UAT) stage, so you're hopefully trading off a larger Schedule Risk from UAT and adding a smaller Schedule Risk to Development. There are other benefits of Unit Testing too: once written, a suite of unit tests is almost cost-free to run repeatedly, whereas repeating a UAT is costly as it involves people's time.

You can think of Taking Action as moving your project on a "Risk Landscape": ideally, when you take an action, you move to some place with worse risk to somewhere more favourable.

Sometimes, you can end up somewhere *worse*: the actions you take to manage a risk will leave you with worse Attendant Risks afterwards. Almost certainly, this will have been a Hidden Risk when you embarked on the action, otherwise you'd not have chosen it.

An Example: Automation

For example, *automating processes* is very tempting: it *should* save time, and reduce the amount of boring, repetitive work on a project. But sometimes, it

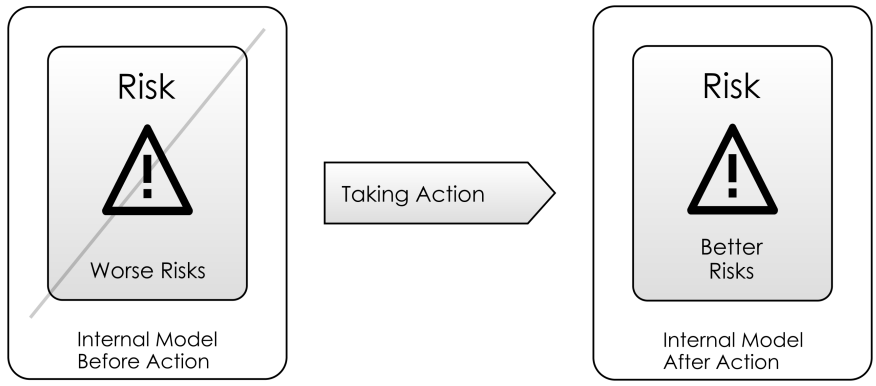


Figure 3.1: Navigating The Risk Landscape

turns into an industry in itself, and consumes more effort than it's worth.

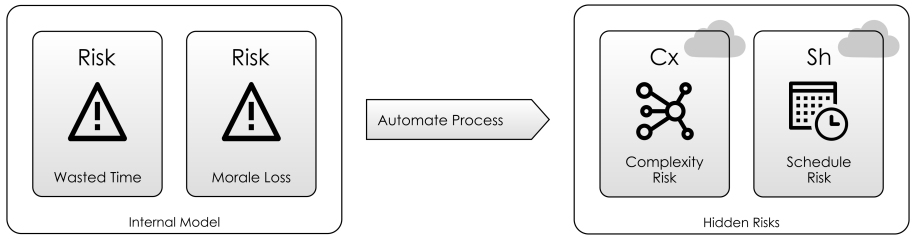


Figure 3.2: Hidden Risks of Automation

Another Example: MongoDB

On a recent project in a Bank, we had a requirement to store a modest amount of data and we needed to be able to retrieve it fast. The developer chose to use MongoDB[1] for this. At the time, others pointed out that other teams in the bank had had lots of difficulty deploying MongoDB internally, due to licensing issues and other factors internal to the bank.

Other options were available, but the developer chose MongoDB because of their *existing familiarity* with it: therefore, they felt that the Hidden Risks of MongoDB were *lower* than the other options, and disregarded the others' opinions.

[1] `https://www.mongodb.com`

This turned out to be a mistake: The internal bureacracy eventually proved too great, and MongoDB had to be abandoned after much investment of time.

This is not a criticism of MongoDB: it's simply a demonstration that sometimes, the cure is worse than the disease. Successful projects are *always* trying to *reduce* Attendant Risks.

3.3 Pay-Off

We can't know in advance how well any action we take will work out. Therefore, Taking Action is a lot like placing a bet.

Pay Off then is our judgement about whether we expect an action to be worthwhile: Are the risks we escape *worth* the attendant risks we will encounter? We should be able to *weigh these separate risks in our hands* and figure out whether the (Glossary#pay-off) makes a given Action worthwhile.

The fruits of this gambling are revealed when we meet reality, and we can see whether our bets were worthwhile.

3.4 The Cost Of Meeting Reality

Meeting reality *in full* is costly. For example, going to production can look like this:

- Releasing software
- Training users
- Getting users to use your system
- Gathering feedback

All of these steps take a lot of effort and time. But you don't have to meet the whole of reality in one go. But we can meet it in a limited way which is less expensive.

In all, to de-risk, you should try and meet reality:

- **Sooner**, so you have time to mitigate the hidden risks it uncovers
- **More Frequently**: so the hidden risks don't hit you all at once
- **In Smaller Chunks**: so you're not over-burdened by hidden risks all in one go.
- **With Feedback**: if you don't collect feedback from the experience of meeting reality, hidden risks *stay hidden*.

In Development Process, we performed a UAT in order to Meet Reality more cheaply and sooner. The *cost* of this is that we delayed the release to do it, adding risk to the schedule.

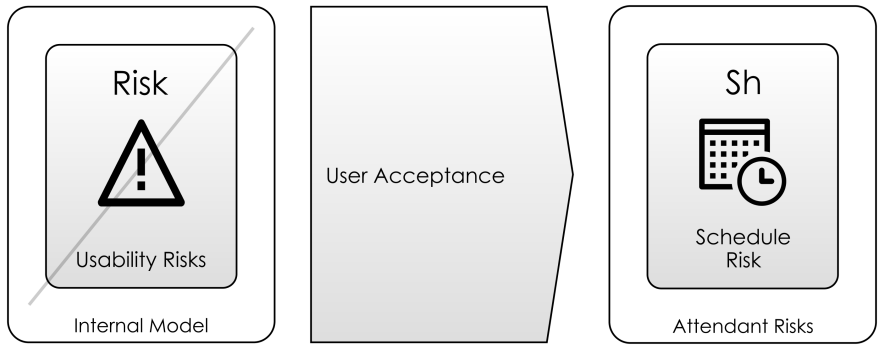


Figure 3.3: Testing flushes out Hidden Risk, but increases Schedule Risk

3.5 Practice 1: YAGNI

As a flavour of what's to come, let's look at YAGNI, an acronym for You Aren't Gonna Need It:

> YAGNI originally is an acronym that stands for "You Aren't Gonna Need It". It is a mantra from Extreme Programming that's often used generally in agile software teams. It's a statement that some capability we presume our software needs in the future should not be built now because "you aren't gonna need it". - YAGNI, *Martin Fowler*[2]

The idea makes sense: if you take on extra work that you don't need, *of course* you'll be accreting Attendant Risks.

But, there is always the opposite opinion: You *Are* Gonna Need It[3]. As a simple example, we often add log statements in our code as we write it (so we can trace what happened when things go wrong), though strictly following YAGNI strictly says we shouldn't.

[2] `https://www.martinfowler.com/bliki/Yagni.html`

[3] `http://wiki.c2.com/?YouAreGonnaNeedIt`

Which is right?

Now, we can say: do the work *if there is a worthwhile Pay-Off*.

- Logging statements are *good*, because otherwise, you're increasing the risk that in production, no one will be able to understand *how the software went wrong*.
- However, adding them takes time, which might introduce Schedule Risk.

So, it's a trade-off: continue adding logging statements so long as you feel that overall, the activity pays-off reducing overall risk.

3.6 Practice 2: Do The Simplest Thing That Could Possibly Work

Another mantra from Kent Beck (originator of the Extreme Programming[4] methodology), is "Do The Simplest Thing That Could Possibly Work", which is closely related to YAGNI and is about looking for solutions which are simple. Our risk-centric view of this strategy would be:

- Every action you take on a project has it's own Attendant Risks.
- The bigger or more complex the action, the more Attendant Risk it'll have.
- The reason you're taking action *at all* is because you're trying to reduce risk elsewhere on the project
- Therefore, the biggest Pay-Off is likely to be the one with the least Attendant Risk.
- So, usually this is going to be the simplest thing.

So, "Do The Simplest Thing That Could Possibly Work" is really a helpful guideline for Navigating the Risk Landscape, but this analysis shows clearly where it's left wanting:

- *Don't* do the simplest thing if there are other things with a better Pay-Off available.

[4] https://en.wikipedia.org/wiki/Extreme_programming

3.7 Summary

So, here we've looked at Meeting Reality, which basically boils down to taking actions to manage risk and seeing how it turns out:

- Each Action you take is a step on the Risk Landscape
- Each Action exposes new Hidden Risks, changing your Internal Model.
- Ideally, each action should reduce the overall Attendant Risk on the project (that is, puts it in a better place on the Risk Landscape

Could it be that *everything* you do on a software project is risk management? This is an idea explored in the next chapter.

CHAPTER 4

All Risk Management

In this chapter, I am going to propose the idea that everything you do on a software project is Risk Management.

In the last chapter, we observed that all the activities in a simple methodology had a part to play in exposing different risks. They worked to manage risk prior to them creating bigger problems in production.

Here, we'll look at one of the tools in the Project Manager's tool-box, the RAID Log[1], and observe how risk-centric it is.

4.1 RAID Log

Many project managers will be familiar with the RAID Log. It's simply four columns on a spreadsheet: **Risks**, **Actions**, **Issues** and **Decisions**.

Let's try and put the following Risk into the RAID Log:

> Debbie needs to visit the client to get them to choose the logo to use on the product, otherwise we can't size the screen areas exactly.

- So, is this an **action**? Certainly. There's definitely something for Debbie to do here.
- Is it an **issue**? Yes, because it's holding up the screen-areas sizing thing.
- Is it a **decision**? Well, clearly, it's a decision for someone.
- Is it a **risk**? Probably: Debbie might go to the client and they *still* don't make a decision. What then?

[1] `http://pmtips.net/blog-new/raid-logs-introduction`

4.2 Let's Go Again

This is a completely made-up example, deliberately chosen to be hard to categorise. Normally, items are more one thing than another. But often, you'll have to make a choice between two categories, if not all four.

This *hints* at the fact that at some level it's all about risk:

4.3 Every Action Attempts to Mitigate Risk

The reason you are *taking* an action is to mitigate a risk. For example:

- If you're coding up new features in the software, this is mitigating Feature Risk (which we'll explore in more detail later).
- If you're getting a business sign-off for something, this is mitigating the risk of everone not agreeing on a course of action (a Coordination Risk).
- If you're writing a specification, that's mitigating the type of "Developer Misimplementation Risk" we saw in the last chapter.

4.4 Every Action Has Attendant Risk.

- How do you know if the action will get completed?
- Will it overrun, or be on time?
- Will it lead to yet more actions?
- What Hidden Risk will it uncover?

Consider *coding a feature* (as we did in the earlier Development Process chapter). We saw here how the whole process of coding was an exercise in learning what we didn't know about the world, uncovering problems and improving our Internal Model. That is, flushing out the Attendant Risk of the Goal In Mind.

And, as we saw in the Introduction, even something *mundane* like the Dinner Party had risks.

4.5 An Issue is Just A Type of Risk

- Because issues need to be fixed. . .
- And fixing an issue is an action. . .
- Which, as we just saw also carry risk.

One retort to this might be to say: "An issue is a problem I have now, whereas a risk is a problem that *might* occur." I am going to try and break that mind-set in the coming pages, but I'll just start with this:

- Do you know *exactly* how much damage this issue will do?
- Can you be sure that the issue might not somehow go away?

Issues then, just seem more "definite" and "now" than *risks*, right? This classification is arbitrary: they're all just part of the same spectrum, they all have inherent uncertainty, so there should be no need to agonise over which column to put them in.

4.6 Goals Are Risks Too

In the previous chapters, we've introduced something of a "diagram language" of risk. Let's review it:

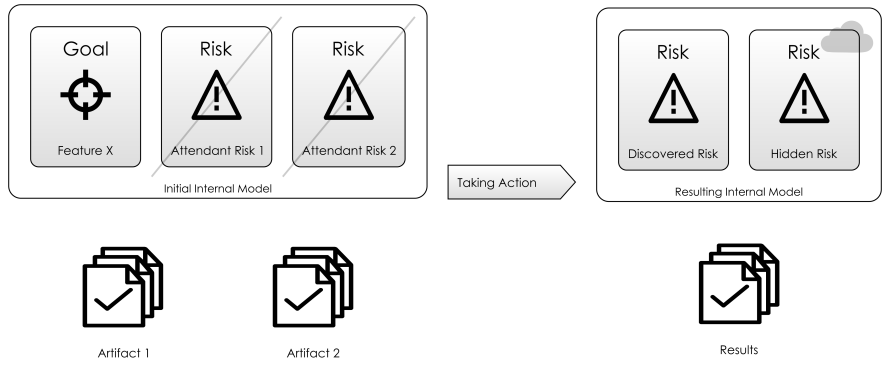


Figure 4.1: Risk-First Diagram Language

Goals live inside our Internal Model, just like Risks. It turns out, that functionally, Goals and Risks are equivalent. For example, The Goal of "Implementing Feature X" is equivalent to mitigating "Risk of Feature X not being present".

Let's try and back up that assertion with a few more examples:

Goal	Restated As A Risk
Build a Wall	Mitigate the risk of something getting in / out

Goal	Restated As A Risk
Land a man on the moon	Mitigate the risk of looking technically inferior during the cold war
Move House	Mitigate the risks/problems of where you currently live

There is a certain "interplay" between the concepts of risks, actions and goals. After all, on the Risk Landscape they correspond to a starting point, a movement, and a destination. From a redundancy perspective, any one of these can be determined by knowing the other two.

Psychologically, humans are very goal-driven: they like to know where they're going, and are good at organising around a goal. However, by focusing on goals ("solutionizing") it's easy to ignore alternatives. By focusing on "Risk-First", we don't ignore the reasons we're doing something.

4.7 Every Decision is About Pay-Off.

- By the very nature of having to make a decision, there's the risk you'll decide wrongly.
- And, making a decision takes time, which could add risk to your schedule.
- And what's the risk if the decision doesn't get made?

Sometimes, there will be multiple moves available on the Risk Landscape and you have to choose. Let's take a hypothetical example: You're on a project and you're faced with the decision - release now or do more testing?

Obviously, in the ideal world, we want to get to the place on the Risk Landscape where we have a tested, bug-free system in production. But we're not there yet, and we have funding pressure to get the software into the hands of some paying customers. The table below shows an example:

Risk Managed	Action	Attendant Risk	Pay-Off
Funding Risk	Go Live	Reputational Risk, Operational Risk	MEDIUM
Implementation Risk	User Acceptance Test	Worse Funding Risk, Operational Risk	LOW

This is (a simplification of) the dilemma on lots of software projects - *test further*, to reduce the risk of users discovering bugs (Implementation Risk) which would cause us reputational damage, or *get the release done* and reduce our Funding Risk by getting paying clients sooner.

In the above table, it *appears* to be better to do the "Go Live" action, as there is a greater Pay Off. The problem is, actions are not *commutative*, i.e. the order you do them in counts.

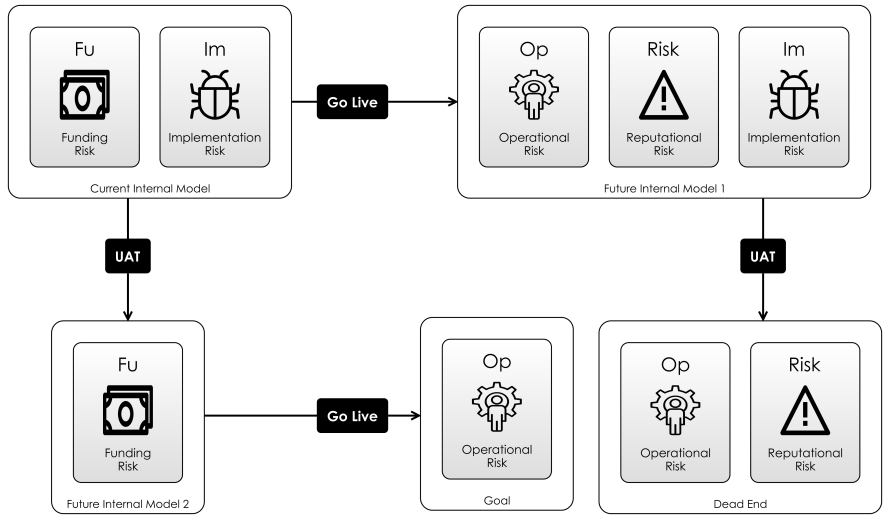


Figure 4.2: UAT or Go Live: Where will you end up?

The above diagram shows our decision as *moves on the Risk Landscape*. Whether you "Go Live" first, or "UAT" first makes a difference to where you will end up. Is there a further action you can take to get you from the "Dead End" to the "Goal"? Perhaps.

Failure

So, when we talk about a project "failing", what do we mean?

Usually, we mean we've failed to achieve a goal, and since *goals are risks*, it is simply the scenario where we are overwhelmed by Attendant Risks: there is *no* action to take that has a good-enough Pay Off to get us out of our hole.

4.8 What To Do?

It makes it much easier to tackle the RAID log if there's only one list. But you still have to choose a *strategy*: Do you tackle the *most important* risk on the list, or the *most urgent*, or take the action with the biggest Pay Off and deal with it?

In the next chapter, Evaluating Risk we'll look at some approaches to choosing what to do.

CHAPTER 5

Evaluating Risk

Here, I am going to re-cap on some pre-existing risk management theory in order to set the scene for the next chapter which heads back to looking at risk on software projects.

5.1 Risk Registers

Most developers are familiar with recording issues in an issue tracker. For all of the same reasons, it's good practice to record the risks you face running a project or an operation in a Risk Register[1]. Typically, this will include for each risk:

- The **name** of the risk, or other identifier.
- A **categories** to which the risk belongs (this is the focus of the Risk Landscape chapter in Part 2).
- A **brief description** or name of the risk to make the risk easy to discuss
- Some estimate for the **Impact**, **Probability** or **Risk Score** of the risk.
- Proposed actions and a log of the progress made to manage the risk.

Some points about this description:

A Continuum of Formality

Remember back to the Dinner Party example at the start: the Risk Register happened *entirely in your head*. There is a continuum all the way from "in your head" through "using a spreadsheet" to dedicated Risk Management software.

[1] https://en.wikipedia.org/wiki/Risk_register

It's also going to be useful *in conversation*, and this is where the value of the Risk-First approach is: providing a vocabulary to *talk about risks* with your team.

Probability And Impact

Probability is how likely something is to happen, whilst **Impact** is the cost (usually financial) when it does happen.

In a financial context (or a gambling one), we can consider the overall **Risk Score** as being the sum of the **Impact** of each outcome multiplied by it's **Probability**. For example, if you buy a 1-Euro ticket in a raffle, there are two outcomes: win or lose. The impact of *winning* would be (say) a hundred Euros, but the **probability** might be 1 in 200. The impact of *losing* would be the loss of 1 Euro, with

Outcome	Impact	Probabilty	Risk Score
Win	+ 99 EUR	1 in 200	.5 EUR
Lose	- 1 EUR	199 in 200	-.99 EUR

Risk Management in the finance industry *starts* here, and gets more complex, but often (especially on a software project), it's better to skip all this, and just figure out a Risk Score. This is because if you think about "impact", it implies a definite, discrete event occurring, or not occurring, and asks you then to consider the probability of that occurring.

Risk-First takes a view that risks are a continuous quantity, more like *money* or *water*: by taking an action before delivering a project you might add a degree of Schedule Risk, but decrease the Operational Risk later on by a greater amount.

5.2 Risk Matrix

A risk matrix presents a graphical view on where risks exist. Here is an example, showing the risks from the dinner party in the A Simple Scenario chapter:

This type of graphic is *helpful* in deciding what to do next, although alternatively, you can graph the overall **Risk Score** against the Pay-Off. Easily mitigated risk (on the right), and worse risks (at the top) can therefore be dealt with first (hopefully).

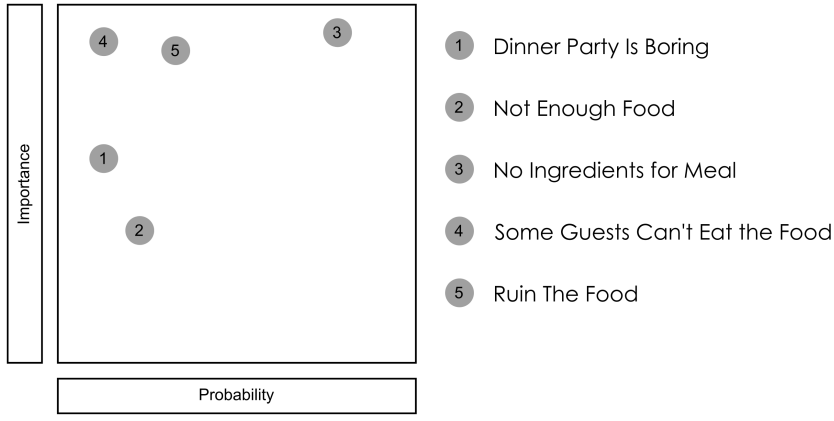


Figure 5.1: Risk Register of Dinner Party Risks

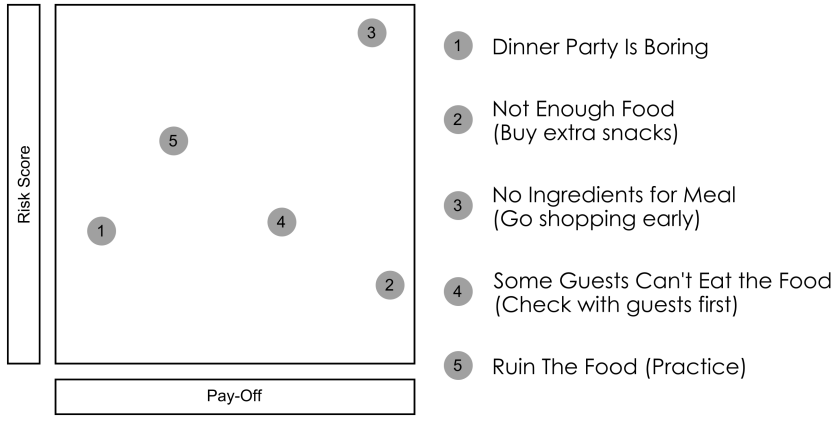


Figure 5.2: Risk Register of Dinner Party Risks, Considering Pay-Off

5.3 Unknown Unknowns

One of the criticisms of the Risk Register approach is that of mistaking the map for the territory. That is, mistakenly believing that what's on the Risk Register *is all there is*.

In the preceding discussions, I have been careful to point out the existence of Hidden Risks for that very reason. Or, to put another way:

> What we don't know is what usually gets us killed - Petyr Baelish, *Game of Thrones*

Donald Rumsfeld's famous Known Knowns is also a helpful conceptualisation:

- **A *known* unknown** is an Attendant Risk. i.e. something you are aware of, but where the precise degree of threat can't be established.
- **An *unknown* unknown** is a Hidden Risk. i.e a risk you haven't even thought to exist yet.

5.4 Risk And Uncertainty

Arguably, this site uses the term 'Risk' wrongly: most literature suggests risk can be measured[2] whereas uncertainty represents things that cannot.

I am using **risk** everywhere because later we will talk about specific risks (e.g. Boundary Risk or Complexity Risk), and it doesn't feel grammatically correct to talk about those as **uncertainties**, especially given the pre-existing usage in Banking of terms like Operational Risk[3] or Reputational risk[4] which are also not really a-priori measurable.

5.5 The Opposite Of Risk Management

Let's look at the classic description of Risk Management:

> Risk Management is the process of thinking out corrective actions before a problem occurs, while it's still an abstraction. The opposite of risk management is crisis management, trying to figure

[2] https://keydifferences.com/difference-between-risk-and-uncertainty.html

[3] https://en.wikipedia.org/wiki/Operational_risk

[4] https://www.investopedia.com/terms/r/reputational-risk.asp

> out what to do about the problem after it happens. - Waltzing With Bears, *Tom De Marco & Tim Lister*[5]

This is not how Risk-First sees it:

First, we have the notion that Risks are discrete events, again. Some risks *are* (like gambling on a horse race), but most *aren't*. In the Dinner Party, for example, bad preparation is going to mean a *worse* time for everyone, but how good a time you're having is a spectrum, it doesn't divide neatly into just "good" or "bad".

Second, the opposite of "Risk Management" (or trying to minimise the "Down-side") is either "Upside Risk Management", (trying to maximise the good things happening), or it's trying to make as many bad things happen as possible.

Third, Crisis Management is *still just Risk Management*: the crisis (Earthquake, whatever) has *happened*. You can't manage it because it's in the past. All you can do is Risk Manage the future (minimize further casualties and human suffering, for example).

Yes, it's fine to say "we're in crisis", but to assume there is a different strategy for dealing with it is a mistake: this is the Fallacy of Sunk Costs[6].

5.6 Invariances #1: Panic Invariance

You would expect then, that any methods for managing software delivery should be *invariant* to the level of crisis in the project. If, for example, a project proceeds using Scrum[7] for eight months, and then the deadline looms and everyone agrees to throw Scrum out of the window and start hacking, then *this implies there is a problem with Scrum*, and that it is not *Panic Invariant*. In fact, many tools like Scrum don't consider this:

- If there is a production outage during the working week, we don't wait for the next Scrum Sprint to fix it.
- Although a 40-hour work-week *is a great idea*, this goes out of the window if the databases all crash on a Saturday morning.

In these cases, we (hopefully calmly) *evaluate the risks and Take Action*.

This is **Panic Invariance**: your methodology shouldn't need to change given the amount of pressure or importance on the table.

[5] http://amzn.eu/d/i0IDFA2

[6] https://en.wikipedia.org/wiki/Escalation_of_commitment

[7] https://en.wikipedia.org/wiki/Scrum_(software_development)

5.7 Invariances #2: Scale Invariance

Another test of a methodology is that it shouldn't fall down when applied at different *scales*. Because, if it does, this implies that there is something wrong with the methodology. The same is true of physical laws: if they don't apply under all circumstances, then that implies something is wrong. For example, Newton's Laws of Motion fail to calculate the orbital period of Mercury, which led to Einstein trying to improve on them with the Theory of Relativity[8].

Some methodologies are designed for certain scales: Extreme Programming is designed for small, co-located teams. And, that's useful. But the fact it doesn't scale tells us something about it: chiefly, that it considers certain *kinds* of risk, while ignoring others. At small scales, that works ok, but at larger scales, other risks (such as team Coordination Risk) increase too fast for it to work.

So ideally, a methodology should be applicable at *any* scale:

- A single class or function
- A collection of functions, or a library
- A project team
- A department
- An entire organisation

If the methodology *fails at a particular scale*, this tells you something about the risks that the methodology isn't addressing. It's fine to have methodologies that work at different scales, and on different problems. One of the things Risk-First explores is trying to place methodologies and practices within a framework to say *when* they are applicable.

5.8 Value vs Speed

"Upside Risk"

"Upside Risk" isn't a commonly used term: industry tends to prefer "value", as in "Is this a value-add project?". There is plenty of theory surrounding **Value**, such as Porter's Value Chain[9] and Net Present Value[10]. This is all fine so long as we remember:

[8] https://en.wikipedia.org/wiki/Theory_of_relativity

[9] https://en.wikipedia.org/wiki/Value_chain

[10] https://en.wikipedia.org/wiki/Net_present_value

- **The probability of Pay-Off is risky**: Since the value is created in the future, we can't be certain about it happening - we should never consider it a done-deal. **Future Value** is always at risk. In finance, for example, we account for this in our future cash-flows by discounting them according to the risk of default.
- **The Pay-Off amount is risky**: Additionally, whereas in a financial transaction (like a loan, say), we might know the size of a future payment, in IT projects we can rarely be sure that they will deliver a certain return. On some fixed-contract projects this sometimes is not true: there may be a date when the payment-for-delivery gets made, but mostly we'll be expecting an uncertain pay-off.
- Humans tend to be optimists (especially when there are lots of Hidden Risks), hence our focus on Downside Risk. Sometimes though, it's good to stand back and look at a scenario and think: am I capturing all the Upside Risk here?

Speed

For example, in Rapid Development[11] by Steve McConnell we have the following diagram:

And, this is *fine*, McConnell is structuring the process from the perspective of *delivering as quickly as possible*. However, here, I want to turn this on it's head. Software Development from a risk-first perspective is an under-explored technique, and I believe it offers some useful insights. So the aim here is to present the case for viewing software development like this:

As we will see, *Speed* (or Schedule Risk as we will term it) is one risk amongst others that need to be considered from a risk-management perspective. There's no point in prioritising *speed* if the software fails in production due to Operational Risk issues and damages trust in the product.

Eisenhower's Box

Eisenhower's Box is a simple model allowing us to consider *two* aspects of risk at the same time:

- How valuable the work is (Importance, Value).
- How soon it is needed (Urgency, Time).

The problem is, we now need to take a call on whether to do something that is *urgent* or something that is *important*.

[11] `http://a.co/d/ddWGTB2`

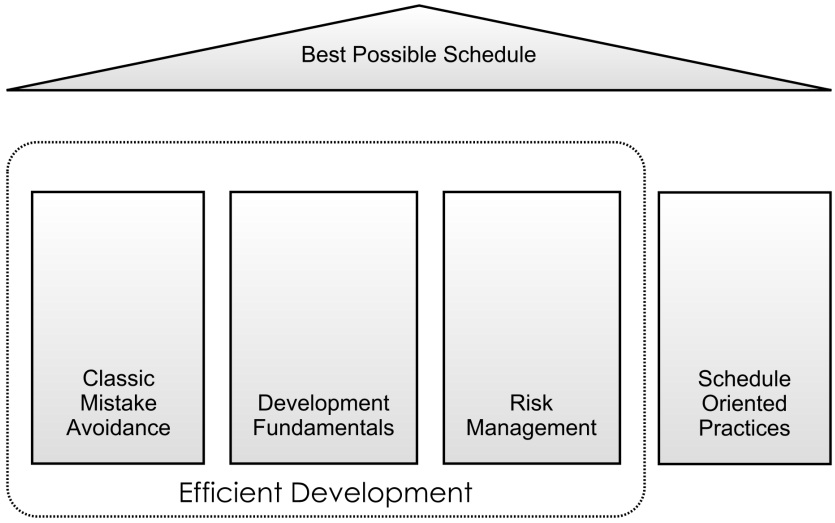


Figure 5.3: Pillars, From Rapid Development By Steve McConnell

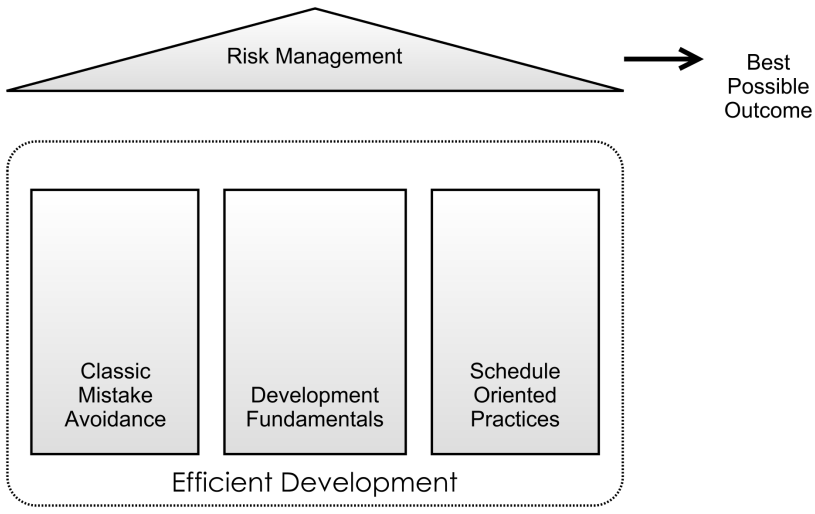


Figure 5.4: Pillars, re-arranged

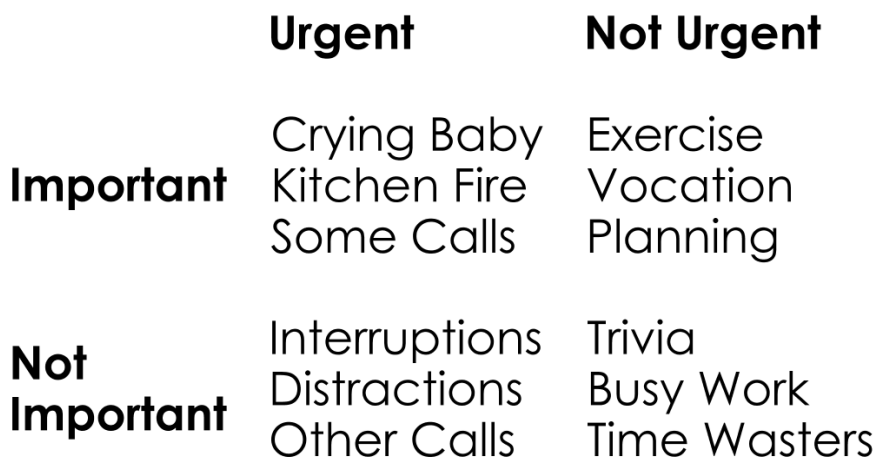


***Figure 5.5: A basic "Eisenhower box" to help evaluate urgency and importance. Items may be placed at more precise points within each quadrant. - Adapted From Time Management,* Wikipedia[12]**

5.9 Discounting

Net Present Value allows us to discount value in the future, which offers us a way to reconcile these two variables. The further in the future the value is realised, the bigger the discount. This is done because payment *now* is better than payment in the future: there is the risk that something will happen to prevent that future payment. This is why we have *interest rates* on loan payments.

In the diagram, you can see two future payments, Payment **A** of č100 due in one year, and Payment **B** of č150 due in 10 years. By discounting at a given rate (here at a high rate of 20% per year) we can compare their worth *now*. At this discount rate, Payment **A**, - arriving next year - has a far greater value.

Can we do the same thing with risk? Let's introduce the concept of Net Present Risk, or NPR:

> Net Present Risk is the *Impact* of a Future risk, discounted to a common level of *Urgency*.

Let's look at a quick example to see how this could work out. Let's say you had the following risks:

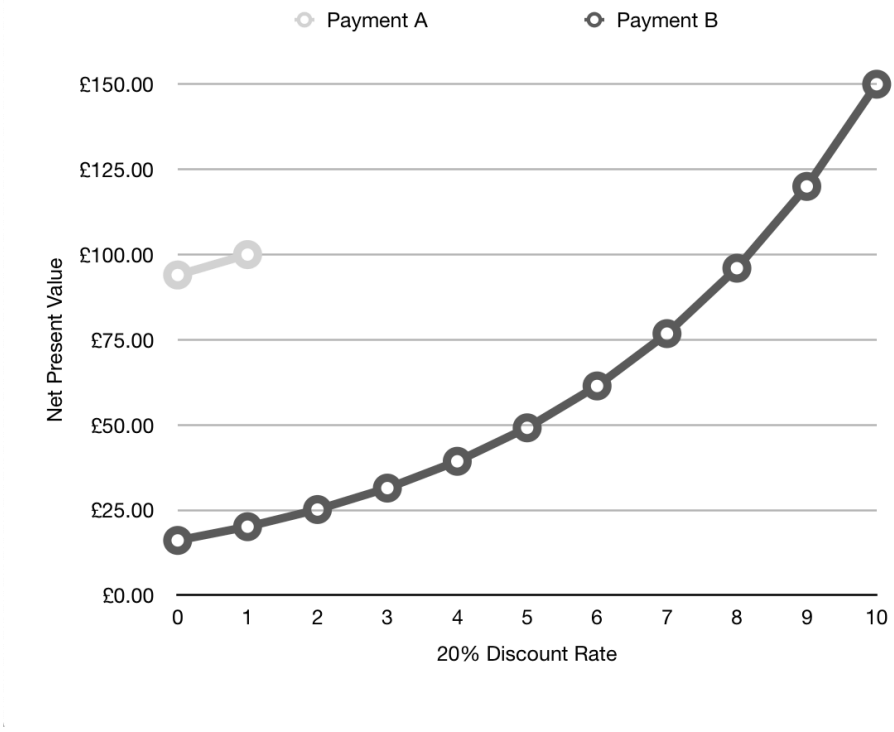


Figure 5.6: Net Present Value Discounting

- Risk **A**, which will cost you č50 in 5 day's time.
- Risk **B**, which will cost you č70 in 8 day's time.

Which has the biggest NPR? Well, it depends on the discount rate that you apply. Let's assume we are discounting at 6% per *day*. A graph of the discounted risks looks like this:

On this basis, the biggest NPR is **B**, at about č45. If we increase the discount factor to 20%, we get a different result:

Now, risk **A** is bigger.

Because this is *Net* Present Risk, we can also use it to make decisions about whether or not to mitigate risks. Let's assume the cost of mitigating any risk *right now* is č40. Under the 6% regime, only Risk **B** is worth mitigating today, because you spend č40 today to get rid of č45 of risk (today).

Under the 20% regime, neither are worth mitigating. The 20% Discount Rate may reflect that sometimes, future risks just don't materialise.

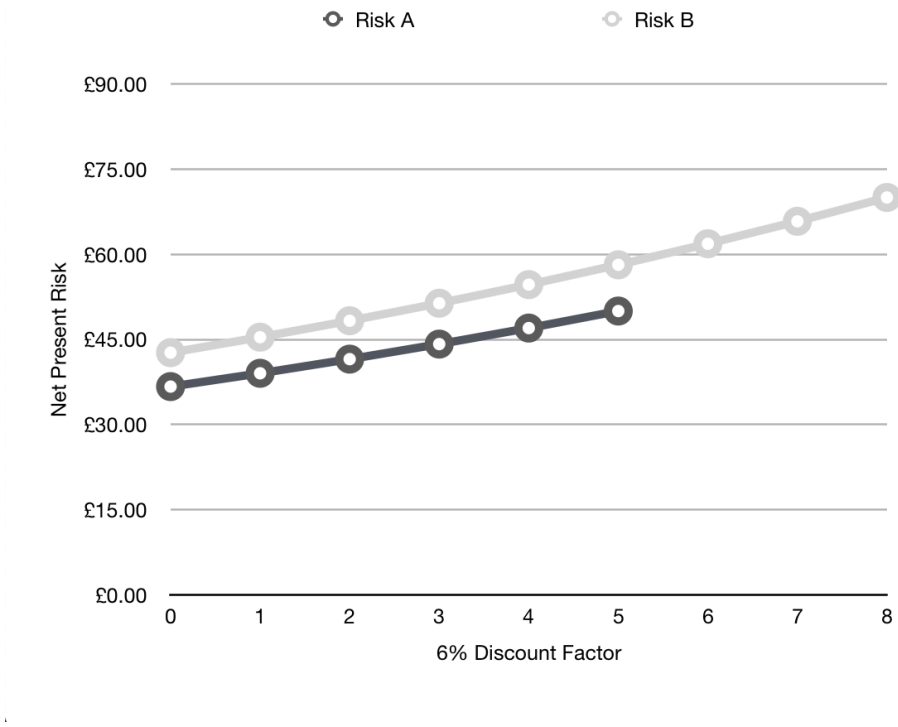


Figure 5.7: Net Present Risk, 6% Discount Rate

Discounting the Future To Zero

I have worked in teams sometimes where the blinkers go down, and the only thing that matters is *now*. Anything with a horizon over a week is irrelevant. Regimes of such hyper-inflation[13] are a sure sign that something has *really broken down* within a project. Consider in this case a Discount Factor of 60% per day, and the following risks:

- Risk A: č10 cost, happening *tomorrow*
- Risk B: č70 cost, happening in *5 days*.

Risk B is almost irrelevant under this regime, as this graph shows:

Why do things like this happen? Often, the people involved are under incredible job-stress: usually they are threatened on a daily basis, and therefore feel they have to react. In a similar way, publicly-listed companies also often

[13] `https://en.wikipedia.org/wiki/Hyperinflation`

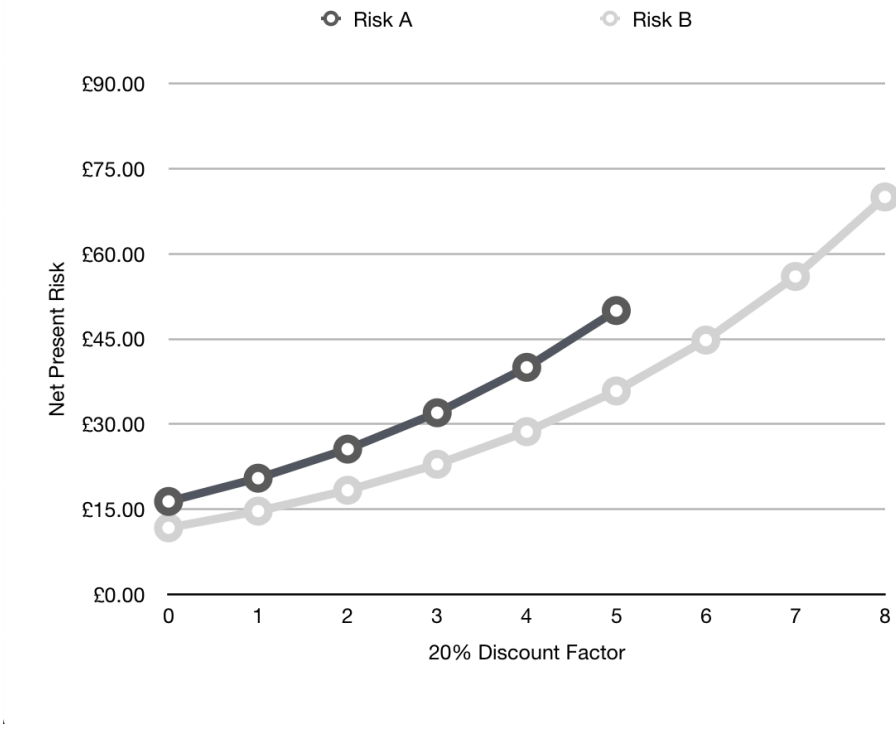


Figure 5.8: Net Present Risk, 20% Discount Rate

apply short-term focus, because they only care about the *next annual report*, which limits their horizons and ability to consider future risk.

Under these circumstances, we often see *Pooh-Bear Procrastination*:

> "Here is Edward Bear coming downstairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way. . . if only he could stop bumping for a moment and think of it!"
>
> —A. A. Milne, *Winne-the-Pooh*[14]

5.10 Is This Scientific?

Enough with the numbers and the theory: Risk-First is an attempt to provide a practical framework, rather than a scientifically rigorous analysis. For

[14] http://amzn.eu/d/acJ5a2j

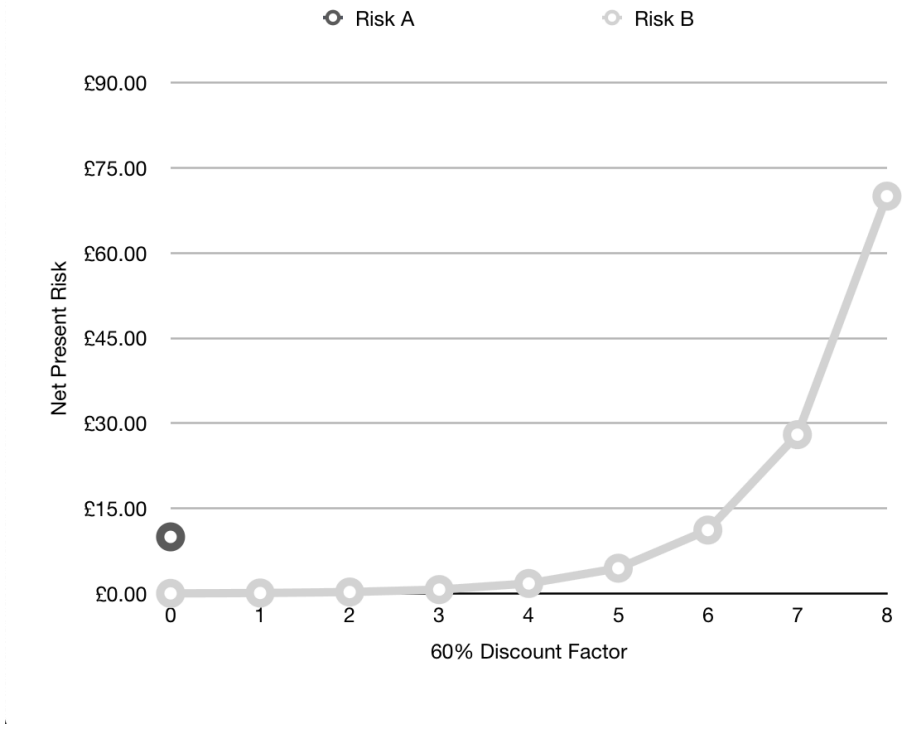


Figure 5.9: Net Present Risk, 60% Discount Rate

software development, you should probably *give up* on trying to compute risk numerically. You *can't* work out how long a software project will take based purely on an analysis of (say) *function points*. (Whatever you define them to be).

- First, there isn't enough scientific evidence for an approach like this. We *can* look at collected data about IT projects, but techniques and tools advance rapidly.
- Second, IT projects have too many confounding factors, such as experience of the teams, technologies used etc. That is, the risks faced by IT projects are *too diverse* and *hard to quantify* to allow for meaningful comparison from one to the next.
- Third, as soon as you *publish a date* it changes the expectations of the project (see Student Syndrome).
- Fourth, metrics get misused and gamed (as we will see in a later chapter).

Reality is messy. Dressing it up with numbers doesn't change that and you

risk fooling yourself. If this is the case, is there any hope at all in what we're doing? Yes: *forget precision*. You should, with experience be able to hold up two separate risks and answer the question, "is this one bigger than this one?"

With that in mind, let's look at how we can meet reality as fast and often as possible.

CHAPTER 6

Cadence

Let's go back to the model again, introduced in Meeting Reality:

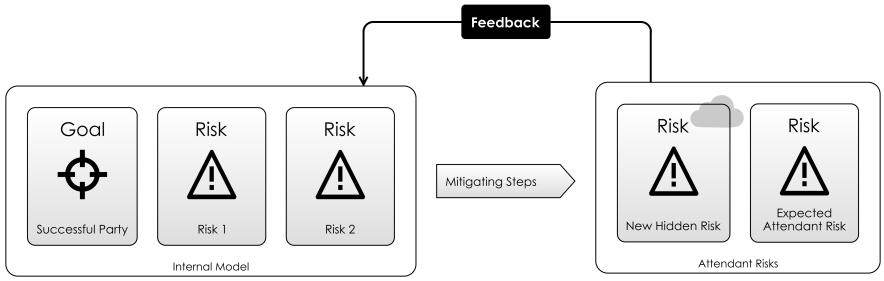


Figure 6.1: Meeting Reality: reality is changed and so is your internal model.

As you can see, it's an idealized **Feedback Loop**.

How *fast* should we go round this loop? The longer you leave your goal in mind, the longer it'll be before you find out how it really stacks up against reality.

Testing your goals in mind against reality early and safely is how you'll manage risk effectively, and to do this, you need to set up **Feedback Loops**. e.g.

- **Bug Reports and Feature Requests** tell you how the users are getting on with the software.
- **Monitoring Tools and Logs** allow you to find out how your software is doing in reality.

- **Dog-Fooding** i.e using the software you write yourself might be faster than talking to users.
- **Continuous Delivery**[1] is about putting software into production as soon as it's written.
- **Integration Testing** is a faster way of meeting *some* reality than continually deploying code and re-testing it manually.
- **Unit Testing** is a faster feedback loop than Integration Testing.
- **Compilation** warns you about logical inconsistencies in your code.

.. and so on.

Time / Reality Trade-Off

This list is arranged so that at the top, we have the most visceral, most *real* feedback loop, but at the same time, the slowest.

At the bottom, a good IDE can inform you about errors in your Internal Model in real time, by way of highlighting compilation errors . So, this is the fastest loop, but it's the most *limited* reality.

Imagine for a second that you had a special time-travelling machine. With it, you could make a change to your software, and get back a report from the future listing out all the issues people had faced using it over its lifetime, instantly.

That'd be neat, eh? If you did have this, would there be any point at all in a compiler? Probably not, right?

The whole *reason* we have tools like compilers is because they give us a shortcut way to get some limited experience of reality *faster* than would otherwise be possible. Because, cadence is really important: the faster we test our ideas, the more quickly we'll find out if they're correct or not.

Development Cycle Time

Developers often ignore the fast feedback loops at the bottom of the list above, because the ones nearer the top *will do*. In the worst cases, changing two lines of code, running the build script, deploying and then manually testing out a feature. And then repeating.

If you're doing it over and over, this is a terrible waste of time. And, you get none of the benefit of a permanent suite of tests to run again in the future.

The Testing Pyramid[2] hints at this truth:

[1] `https://en.wikipedia.org/wiki/Continuous_delivery`

[2] `http://www.agilenutshell.com/episodes/41-testing-pyramid`

- **Unit Tests** have a *fast feedback loop*, so have *lots of them*.
- **Integration Tests** have a slightly *slower feedback loop*, so have *few of them*. Use them when you can't write unit tests (at the application boundaries).
- **Manual Tests** have a *very slow feedback loop*, so have *even fewer of them*. Use them as a last resort.

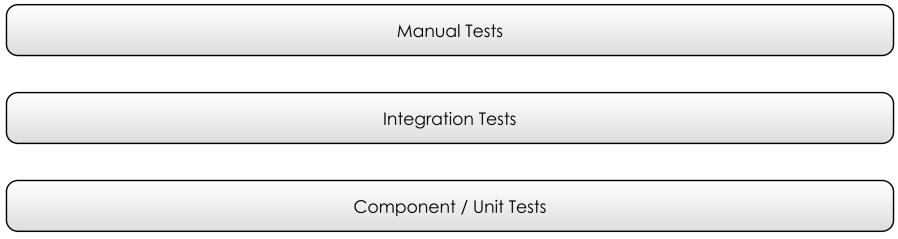


Figure 6.2: The Testing Pyramid

Production

You could take this chapter to mean that Continuous Delivery (CD) is always and everywhere a good idea. That's not a bad take-away, but it's clearly more nuanced than that.

Yes, CD will give you faster feedback loops, but even getting things into production is not the whole story: the feedback loop isn't complete until people have used the code, and reported back to the development team.

The right answer is to use multiple feedback loops:

In the next chapter De-Risking we're going to introduce a few more useful terms for thinking about risk.

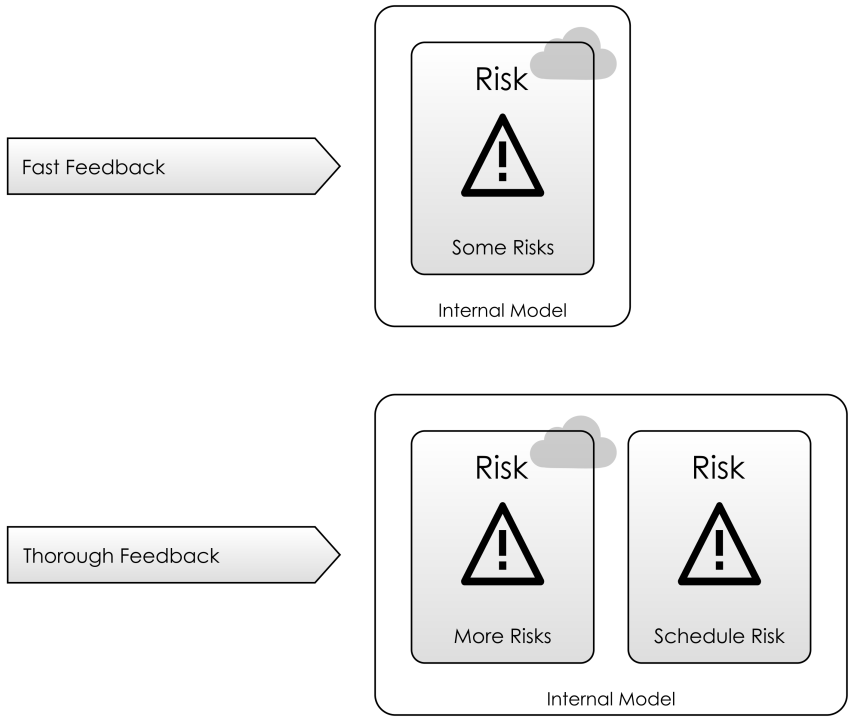


Figure 6.3: Different actions have different feedback loops

CHAPTER 7

De Risking

As we saw in A Conversation, it's important not only to consider the Attendant Risks you're trying to mitigate, but the ones you're likely to pick up in the process. This means picking a careful path through the Risk Landscape. This is the essence of *De-Risking*.

> "To take steps to make (something) less risky or less likely to involve a financial loss."
>
> —De-Risk,*OxfordDictionaries.com*[1]

Some simple examples of this might be:

- **Safety-nets and ropes** De-Risk climbing. But, the activity of climbing itself is otherwise much unchanged.
- **Backups and Source-Control** De-Risk the development process by reducing the impact of computer failure. Our process is changed *slightly* by this imposition, but we're not massively inconvenienced.
- **Insurance** De-Risks owning a house, or going on holiday or driving a car. Usually, the payment is small enough not to impact us too much.
- **The National Health Service (NHS)** De-Risks medical expense by pooling health-care costs across the entire population. If you were struck down with a debilitating illness, then at least you wouldn't also have to pay to get better.

Let's look at some common strategies for De-Risking.

[1] https://en.oxforddictionaries.com/definition/de-risk

7.1 Mitigate

Mitigating the risk is taking steps towards minimising either it's likelihood or impact (as we discussed in the Evaluating Risk chapter). This is the main approach we will be looking at in Part 2. We'll break down risk into its different types and look at the general mitigations for each. The examples above of De-Risking were all mitigations. (Safety-nets, for example, mitigate the impact of hitting the ground.)

7.2 Avoid

Avoiding a risk, means taking a route on the Risk Landscape *around* the risk. For example, if you are working in a team which has no experience of relational databases, then *storing data in files* might be a way to avoid the Learning-Curve Risk associated with this technology.

Of course, you may pick up other, more serious Attendant Risks as a result: Relational Databases are software solutions to many kinds of Coordination Risk problem.

Not launching an online service *avoids* the Operational Risk involved in running one. Although you avoid the upsides too.

7.3 Transfer

Transferring risk means *making it someone else's problem*. For example, when I buy home insurance, the impact of my house burning down is reduced. It hasn't gone away completely, but at least the financial element of it is handled by the insurance company.

In part 2, we'll see how **Transfer** of risk is an essential feature of Software as a Service. Inside organisations, **Transfer** of risk can become a political game:

> "... ownership results in 'one throat to choke' for audit functions [and] from ownership comes responsibility. A lot of the political footwork in an enterprise revolves around trying to not own technologies. Who wants to be responsible for Java usage across a technology function of dozens of thousands of staff, any of whom might be doing crazy stuff? You first, mate."
>
> —Why Are Enterprises So Slow?, *zwischenzugs.com*[2]

[2] https://zwischenzugs.com/2018/10/02/why-are-enterprises-so-slow/

7.4 Ignore / Accept

Accepting a risk is to deal with it when it arises. One example is the Key-Man Risk involved in having a super-star programmer on the team. Although there would be fallout if they left, they are often mitigating more risk than they cause.

Another example is using particular software dependencies: Building a mobile application which requires a Facebook account to log in might give rise to the risk that people without Facebook accounts can't log in, but might simplify the software to such an extent that it's worthwhile.

Whereas **Accepting** a risk seems to imply an eyes-wide-open examination, **Ignoring** seems to imply that either the risk is so insignificant it doesn't warrant evaluation, or so daunting that it can't be stared down. Either way, **Ignoring** a risk amounts to the same thing as **Accepting** it, since you're not doing anything about it.

Accepting a risk has to occur *before* we can **Mitigate** it.

A Nice Problem To Have

Ignoring or **Accepting** risks is a lot less work than **Mitigating** them, and sometimes it can feel negligent to just add them to the backlog or risk-register without doing anything immediately about them. One useful test I have found is whether "This would be a nice problem to have". For example:

> Running out of space in the database would be a nice problem to have, because it would mean we have lots of users

> Users complaining about lacking function X would be a nice problem to have, because it would mean they were using the system

Applying this kind of logic at the start of a project leads you towards building a Minimum Viable Product[3].

Learned Helplessness

Sometimes, risks just go away on their own. Learned Helplessness[4] on the other hand, is where we *could* do something about the risk, but fail to see that as an option:

[3] https://en.wikipedia.org/wiki/Minimum_viable_product

[4] https://en.wikipedia.org/wiki/Learned_helplessness

> "Learned helplessness is behaviour typical of animals, and in rare cases humans, that occurs when the subject endures repeatedly painful or otherwise aversive stimuli which it is unable to escape or avoid. After such experience, the organism often fails to learn or accept"escape" or "avoidance" in new situations where such behavior would likely be effective. "
>
> —Learned Helplessness, *Wikipedia*[5]

7.5 Contain

Containing risks means setting aside sufficient time or money to deal with them if they occur. This is an excellent approach for Hidden Risk or entire sets of minor Attendant Risks.

Whenever a project-manager builds slack into a project plan, this is **Containment**. "Time-Boxing" is also containment: this is where you give a piece of work a week (say) to prove itself. If it can't be done in this time, we move on and try a different approach.

In the chapter on Schedule Risk we are going to look in detail at how this works.

7.6 Exploit

Exploiting as a strategy usually means taking advantage of the upside of a risk. For example, ensuring enough stock is available to mitigate the risk of a rush on sales over the Christmas period, or ensuring your website has enough bandwidth to capture all the traffic headed towards it after it's featured on television.

Going back to the example of home insurance, the Insurance company is **exploiting** the risk of my house burning down by selling me insurance. This is a common pattern: wherever there is risk, there is likely to be a way to profit from it.

Later, in the chapter on Process Risk we'll be looking at how **exploiting risk** can happen organically within a company.

7.7 Re-cap

Let's look at the journey so far:

[5] https://en.wikipedia.org/wiki/Learned_helplessness

- In A Simple Scenario we looked at how risk pervades every goal we have in life, big or small. We saw that risk stems from the fact that our Internal Model of the world couldn't capture everything about reality, and so some things were down to chance.
- In the Development Process we looked at how common software engineering conventions like Unit Testing, User Acceptance Testing and Integration could help us manage the risk of taking an idea to production, by *gradually* introducing it to reality in stages.
- Then, generalizing the lessons of the Development Process article, we examined the idea that Meeting Reality frequently helps flush out Hidden Risks and improve your Internal Model.
- In It's All Risk Management we took a leap of faith: Could *everything* we do just be risk management? And we looked at the RAID log and thought that maybe it could be.
- Next, in A Software Project Scenario we looked at how you could treat the project-as-a-whole as a risk management exercise, and treat the goals from one day to the next as activities to mitigate risk.
- Evaluating Risk was an aside, looking at some terminology and the useful concept of a Risk Register.
- We looked at Cadence, and how feedback loops allow you Navigate the Risk Landscape more effectively, by showing you more quickly when you're going wrong.

What this has been building towards is supplying us with a vocabulary with which to communicate to our team-mates about which Risks are important to us, which actions we believe are the right ones, and which tools we should use.

In the next chapter we will see an example of this in action.

CHAPTER 8

A Conversation

After so much theory, it seems like it's time to look at how we can apply these principles in the real world.

The following is based the summary of an issue from just a few weeks ago. It's heavily edited and anonymized, and I've tried to add the Risk-First vocabulary along the way, but otherwise, it's real.

Some background: **Synergy** is an online service with an app-store, and **Eve** and **Bob** are developers working for **Large Corporation LTD**, which wants to have an application accepted into Synergy's app-store.

Synergy's release means that the app-store refresh will happen in a few weeks, so this is something of a hard deadline: if we miss it, the next release will be four months away.

8.1 A Risk Conversation

Eve: We've got a problem with the Synergy security review.

Bob: Tell me.

Eve: Well, you know Synergy did their review and asked us to upgrade our Web Server to only allow TLS version 1.1 and greater?

Bob: Yes, I remember: We discussed it as a team and thought the simplest thing would be to change the security settings on the Web Server, but we all felt it was pretty risky. We decided that in order to flush out Hidden Risk, we'd upgrade our entire production site to use it *now*, rather than wait for the app launch.

Eve: Right, and it *did* flush out Hidden Risk: some people using Windows 7, downloading Excel spreadsheets on the site, couldn't download them: for

some reason, that combination didn't support anything greater than TLS version 1.0. So, we had to back it out.

Bob: Ok, well I guess it's good we found out *now*. It would have been a disaster to discover this after the go-live.

Eve: Yes. So, what's our next-best action to mitigate this?

Bob: Well, we could go back to Synergy and ask them for a reprieve, but I think it'd be better to mitigate this risk now if we can... they'll definitely want it changed at some point.

Eve: How about we run two web-servers? One for the existing content, and one for our new Synergy app? We'd have to get a new external IP address, handle DNS setup, change the firewalls, and then deploy a new version of the Web Server software on the production boxes.

Bob: This feels like there'd be a lot of Attendant Risk: and all of this needs to be handled by the Networking Team, so we're picking up a lot of Bureaucracy Risk. I'm also worried that there are too many steps here, and we're going to discover loads of Hidden Risks as we go.

Eve: Well, you're correct on the first one. But, I've done this before not that long ago for a Chinese project, so I know the process - we shouldn't run into any new Hidden Risk.

Bob: OK, fair enough. But isn't there something simpler we can do? Maybe some settings in the Web Server?

Eve: Well, if we were using Apache, yes, it would be easy to do this. But, we're using Baroque Web Server, and it *might* support it, but the documentation isn't very clear.

Bob: OK, and upgrading it is a *big* risk, right? We'd have to migrate all of our configuration...

Eve: Yes, let's not go there. But if we changing the settings on Baroque, we have the Attendant Risk that it's not supported by the software and we're back where we started. Also, if we isolate the Synergy app stuff now, we can mess around with it at any point in future, which is a big win in case there are other Hidden Risks with the security changes that we don't know about yet.

Bob: OK, I can see that buys us something, but time is really short and we have holidays coming up.

Eve: Yes. How about for now, we go with the isolated server, and review next week? If it's working out, then great, we continue with it. Otherwise, if we're not making progress next week, then it'll be because our isolation

solution is meeting more risk than we originally thought. We can try the settings change in that case.

Bob: Fair enough, it sounds like we're managing the risk properly, and because we can hand off a lot of this to the Networking Team, we can get on with mitigating our biggest risk on the project, the authentication problem, in the meantime.

Eve: Right. I'll check in with the Networking Team each day and make sure it doesn't get forgotten.

8.2 Aftermath

Hopefully, this type of conversation will feel familiar. It should. There's nothing ground-breaking at all in what we've covered so far; it's more-or-less just Risk Management theory.

If you can now apply it in conversation, like we did above, then that's one extra tool you have for delivering software.

So with the groundwork out of the way, let's get on to Part 2 and investigate The Risk Landscape.

8.3 Isn't It Obvious?

At this point, you might be wondering what all the fuss is about. This stuff is all obvious! It's what we do anyway! Perhaps. Risk management *is* what we do anyway:

> "We've survived 200,000 years as humans. Don't you think there's a reason why we survived? We're good at risk management."
>
> —Nassim Nicholas Taleb, author of *The Black Swan*[1]

The problem is that although all this *is* obvious, it appears to have largely escaped codification within the literature, practices and methodologies of software development. Further, while it is obvious, there is a huge hole: Successful De-Risking depends heavily on individual experience and talent.

If there are three take-aways from Risk-First, it is these:

- Concentrate on Risks, not Goals (goals are risks in disguise, anyway)

[1] https://www.zerohedge.com/news/2018-03-13/taleb-best-thing-society-bankruptcy-goldman-sachs

- Every action you take is De-Risking *something* (or should be)
- tbd.

Risk-First has stressed the existence of Hidden Risk over and again, but *it's only hidden if you're not expecting it*. In part 2, we are going to look at all the places where Hidden Risk lives.

Part II

Risk

Part III

Application

CHAPTER 9

Coming Next

- preview of what's to come in part 3.

Bad to leave on the failure notes, let's talk about some successes.

3. What's To Come

- risk based debugging.
- risk based coding.

CHAPTER 10

Estimates

In this chapter, we're going to put a Risk-First spin on the process of Estimating. But, in order to get there, we first need to start with understanding *why* we estimate. We're going to look at some "Old Saws" of software estimation and what we can learn from them. Finally, we'll bring our Risk-First menagerie to bear on de-risking the estimation process.

10.1 The Purpose Of Estimating

Why bother estimating at all? There are two reasons why estimates are useful:

1. **To allow for the creation of *events*.** As we saw in Deadline Risk, if we can put a date on something, we can mitigate lots of Coordination Risk. Having a *release date* for a product allows whole teams of people to coordinate their activities in ways that hugely reduce the need for Communication. Much like "attack at dawn" allows disparate units of an army to avoid lots of the Coordination Risk inherent in "attack on my signal". This is a *good reason for estimating*, because by using events you are mitigating risk. This is often called a *hard deadline*.
2. **To allow for the estimation of the Pay-Off of an action.** This is a *bad reason for estimating*, as we will discuss in detail below. But briefly, the main issue is that Pay-Off isn't just about figuring out Schedule Risk - you should be looking at all the other Attendant Risks of the action too.

10.2 How Estimates Fail

Estimates are a huge source of contention in the software world:

> "Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time."
>
> —Software Development Effort Estimation, *Wikipedia*[1].

Why is it so bad? The problem with a developer answering a question such as:

> How long will it take to deliver X?

Is the following:

- The developer and the client likely don't agree on exactly what X is, and any description of it is inadequate anyway (Invisibility Risk).
- The developer has a less-than-complete understanding of the environment he will be delivering X in (Complexity Risk and Map And Territory Risk.
- The developer has some vague ideas about how to do X, but he'll need to try out various approaches until he finds exactly the right one (Boundary Risk and Learning-Curve Risk).
- The developer has no idea what Hidden Risk will surface when he starts work on it.
- The developer has no idea what will happen if he takes too long and misses the date by a day/week/month/year (Schedule Risk).

. . . and so on.

The reason the estimate of *time* is wrong is because All Activity Is About Mitigating Risk and the estimate of *risk* is wrong.

It's a problem as old as software itself, and in deference to that, let's examine the estimating problem via some "Old Saws".

10.3 Old Saw #1: The "10X Developer"

> "A 10X developer is an individual who is thought to be as productive as 10 others in his or her field. The 10X developer would produce 10 times the outcomes of other colleagues, in a production, engineering or software design environment."
>
> —10X Developer, *Techopedia*[2]

[1] https://en.m.wikipedia.org/wiki/Software_development_effort_estimation

[2] https://www.techopedia.com/definition/31673/10X-developer

Let's try and pull this apart:

- How do we measure this "productivity"? In Risk-First terms, this is about taking action to *transform* our current position on the Risk Landscape to a position of more favourable risk. A "10X Developer" then must be able to take actions that have much higher Pay-Off than a "1X Developer". That is mitigating more Initial Risk, and generating less Attendant Risk.
- It stands to reason then, that someone taking action *faster* will leaving us with less Schedule Risk.
- However, if they are *more expensive*, they may leave us with greater Funding Risk afterwards.
- But, Schedule Risk isn't the only risk being transformed: The result might be bugs, expensive new dependencies or spaghetti-code complexity.
- The "10X" developer *must* also leave behind less of these kind of risks too.
- That means that the "10X Developer" isn't merely faster, but *taking different actions*. They are able to use their talent and experience to see actions with greater pay-off than the 1X Developer.

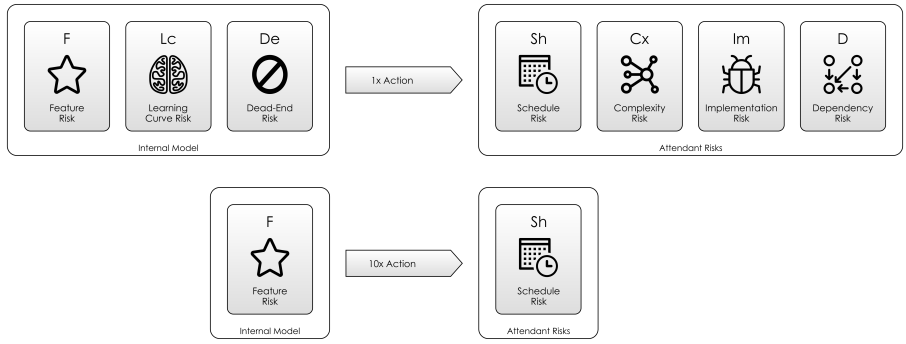


Figure 10.1: 1x Task vs 10X Task

Debate rages as to whether the "10X Developer" even exists. Crucially, it would seem that such a thing would be predicated on the existence of the "1X Developer", who gets "1x" worth of work done each day. It's not clear that there is any such thing as an average developer who is mitigating risk at an average rate.

Even good developers have bad days, weeks or projects. Taking Action is like placing a bet. Sometimes you lose and the Pay-Off doesn't appear:

- The Open-Source software you're trying to apply to a problem doesn't solve it in the way you need.
- A crucial use-case of the problem turns out to change the shape of the solution entirely, leading to lots of rework.
- An assumption about how network security is configured turns out to be wrong, leading to a lengthy engagement with the infrastructure team.

How to Be a "10X Developer"

The easiest way to be a "10X developer" is to have *done the job before*. If you're coding in a familiar language, with familiar libraries and tools, delivering a cookie-cutter solution to a problem in the same manner you've done several times before, then you will be a "10X developer" compared to *you doing it the first time*: - There's no Learning Curve Risk, because you already learnt everything. - There's no Dead End Risk because you already know all the right choices to make.

10.4 Old Saw #2: Quality, Speed, Cost: Pick Any Two

> The Project Management Triangle (called also the Triple Constraint, Iron Triangle and "Project Triangle
>
> - The quality of work is constrained by the project's budget, deadlines and scope (features).
> - The project manager can trade between constraints.
> - Changes in one constraint necessitate changes in others to compensate or quality will suffer."
>   - Project Management Triangle, *Wikipedia*[3]

From a Risk-First perspective, we can now see that this is an over-simplification. If *quality* is a Feature Fit metric, *deadlines* is Schedule Risk and *budget* refers to Funding Risk then that leaves us with a lot of risks unaccounted for:

- I can deliver a project in very short order by building a bunch of screens that *do nothing* (accruing _stunning levels of Implementation Risk as I go).

[3] https://en.wikipedia.org/wiki/Project_management_triangle

- Or, by relying on a lottery win, project's budget is fine. (Although I would have *huge* Funding Risk because *what are the chances of winning the lottery?*. (You can bring in *any* project at *any* time by accepting crazy levels of risk.
- And Brooks' Law contradicts this by saying you can't trade budget for deadlines:

> "Brooks' law is an observation about software project management according to which"adding human resources to a late software project makes it later".
>
> —Brooks Law, *Wikipedia*[4]

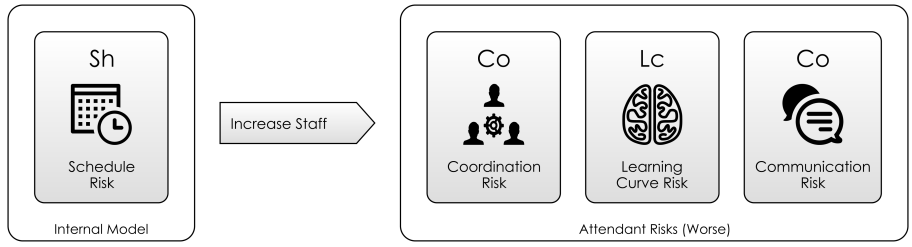


Figure 10.2: Brooks' Law, Risk-First Style

Focusing on just these three variables isn't enough. You can game these variables by sacrificing others: we need to be looking at the project's risk *holistically*:

- There's no point in calling a project complete if the dependencies you are using are unreliable or undergoing rapid change
- There's no point in delivering the project on time if it's an Operational Risk nightmare, and requires constant round-the-clock support and will cost a fortune to *run*. (Working on a project that "hits it's delivery date" but is nonetheless a broken mess once in production is too common a sight.)
- There's no point in delivering a project on-budget if the market has moved on and needs different features.

[4] https://en.wikipedia.org/wiki/Brooks_law

Old Saw #3: Parkinson's Law

We've already looked at Parkinson's Law in Agency Risk, but lets recap:

> Parkinson's law is the adage that 'work expands so as to fill the time available for its completion'.

Let's leave aside the Agency Risk concerns this time. Instead, let's consider this from a Risk-First perspective. *Of course* work would expand to fill the time available: *Time available* is an *absence of Schedule Risk*, it's always going to be sensible to accept Schedule Risk as a trade-off for other more serious risks.

This is why projects will *always* take at least as long as is budgeted for them.

A Case Study

Let's look at a quick example of this in action, taken from Rapid Development by Steve McConnell[5]. At the point of this excerpt, Carl (the Project Manager) is discussing the schedule with Bill, the project sponsor:

> I think it will take about 9 months, but that's just a rough estimate at this point," Carl said. "That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. We absolutely need to bring that system in within 6 months. Can you do it in 6?

Later in the story, the schedule has slipped twice and is about to slip again:

> . . . At the 9-month mark, the team had completed detailed design, but coding still hadn't begun on some modules. It was clear that Carl couldn't make the 10-month schedule either. He announced the third schedule slip number— to 12 months. Bill's face turned red when Carl announced the slip, and the pressure from him became more intense. (2)

At point (2), Carl's tries to mitigate Feature Risk by increasing Schedule Risk, although he knows that Bill will trust him less for doing this, as shown below:

[5] http://amzn.eu/d/eTWKOsK

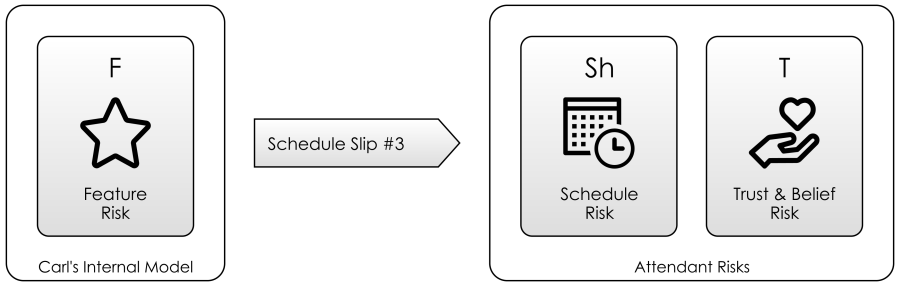


Figure 10.3: Carl's Schedule Slip increases Trust and Belief Risks

> Carl began to feel that his job was on the line. Coding proceeded fairly well, but a few areas needed redesign and reimplementation. The team hadn't coordinated design details in those areas well, and some of their implementations conflicted. At the 11-month oversight-committee meeting, Carl announced the fourth schedule slip— to 13 months. Bill became livid. "Do you have any idea what you're doing?" he yelled. "You obviously don't have any idea! You obviously don't have any idea when the project is going to be done! I'll tell you when it's going to be done! It's going to be done by the 13-month mark, or you're going to be out of a job! I'm tired of being jerked around by you software guys! You and your team are going to work 60 hours a week until you deliver!" (3)

At point (3), after the schedule slips again, Bill threatens Carl's job. Why does he do this? Because *he doesn't trust Carl's evaluation of the Schedule Risk*. By telling Carl that it's his job on the line, he makes sure Carl appreciates the Schedule Risk. However, forcing staff to do overtime is a dangerous ploy: it could disenfranchise the staff, or cause corners to be cut:

> Carl felt his blood pressure rise, especially since Bill had backed him into an unrealistic schedule in the first place. But he knew that with four schedule slips under his belt, he had no credibility left. He felt that he had to knuckle under to the mandatory overtime or he would lose his job. Carl told his team about the meeting. They worked hard and managed to deliver the software in just over 13 months. Additional implementation uncovered additional design flaws, but with everyone working 60 hours a week,

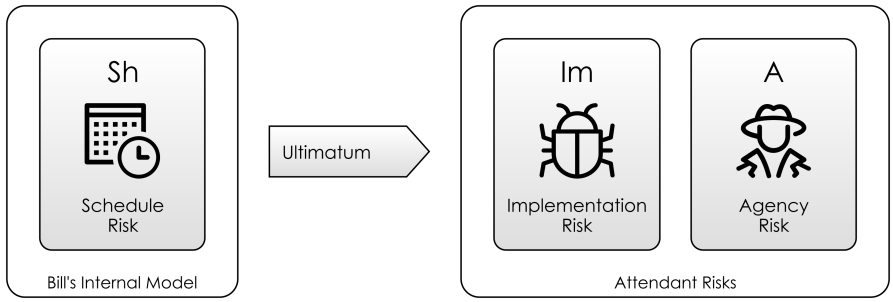


Figure 10.4: Bill's Ultimatum

they delivered the product through sweat and sheer willpower. "
(4) - McConnell, Steve, *Rapid Development*

At point (4), we see that Bill's gamble worked (for him at least): the project was delivered on time by the team working overtime for two months. This was lucky - it seems unlikely that no-one quit and that the code didn't descend into a mess in that time.

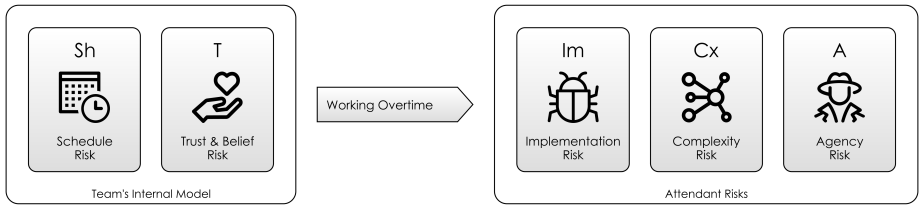


Figure 10.5: Team Response

Despite this being a fictional (or fictionalised) example, it rings true for many projects. What *should* have happened at point (1)? Both Carl and Bill estimated incorrectly... Or did they?

10.5 Agile Estimation

One alternative approach, must espoused in DevOps/Agile is to pick a short-enough period of time (say, two days or two weeks), and figure out what the most meaningful step towards achieving an objective would be in that time.

By fixing the time period, we remove Schedule Risk from the equation, don't we?

Well, no. First, how to choose the time period? Schedule Risk tends to creep back in, in the form of something like Man-Hours[6] or Story Points[7]:

> "Story points rate the relative effort of work in a Fibonacci-like format: 0, 0.5, 1, 2, 3, 5, 8, 13, 20, 40, 100. It may sound counter-intuitive, but that abstraction is actually helpful because it pushes the team to make tougher decisions around the difficulty of work."
>
> —Story Points, *Atlassian*[8]

Second, the strategy of picking the two-day action with the greatest Pay-Off is *often good*. After all, this is just Gradient Descent[9], and that's a perfectly good way for training Machine Learning[10] systems. However, just like following a river downhill from the top of a mountain will *often* get you to the sea, it probably won't take the shortest path, and sometimes you'll get stuck at a lake.

The choice of using gradient descent means that you have given up on Goals: Essentially, we have here the difference between "Walking towards a destination" and "Walking downhill". Or, if you like, a planned economy and a market economy. But, we don't live in *either*: everyone lives in some mixture of the two: our governments *have plans* for big things like roads and hospitals, and taxes. Other stuff, they leave to the whims of supply and demand. A project always ends up being the same.

10.6 Risk-First Estimating

Let's figure out what we can take away from the above experiences:

- The 10X developer saw, and the difference made by experience implies that a lot of the effort on a project comes from Learning Curve Risk and Dead End Risk.

[6] https://en.wikipedia.org/wiki/Man-hour

[7] https://www.atlassian.com/agile/project-management/estimation

[8] https://www.atlassian.com/agile/project-management/estimation

[9] https://en.wikipedia.org/wiki/Gradient_descent

[10] https://en.wikipedia.org/wiki/Machine_learning

- The lesson from "Quality, Speed, Cost" is that actually, we need to be considering *all* risks, not just some arbitrary milestones on a project plan. Project plans can always be gamed, and you can always leave risks unaccounted for in order to hit the goals (good old Map and Territory Risk strikes again).
- The lesson from the Parkinson's Law was that by giving people a *time budget*, you absolve them from Schedule Risk... at least until they realise they're going to overrun. This gives them one less dimension of risk to worry about, but means they end up taking all the time you give them, because they are optimising over the remaining risks.
- Finally, the lesson from Agile Estimation is that *just iterating* is sometimes not as efficient as *using your intuition and experience* to find a more optimal path.

How can we synthesise this knowledge, along with what we've learned into something that makes more sense?

Tip #1: Estimating Should be About *Estimating Pay Off*

For a given action / road-map / business strategy, what Attendant Risks are we going to have when we get there? Yes, we'll all be older (there *will be* Schedule Risk), but it's also about:

- What bets are we making about where the market will be?
- What Communication Risk will we face explaining our product to people?
- What Feature Fit risks are we likely to have when we get there?
- What Complexity Risks will we face building our software? How can we avoid it ending up as a Big Ball Of Mud?
- Where are we likely to face Boundary Risks and Dead End Risks

Instead of the Agile Estimation being about picking out a story-point number based on some idealised amount of typing that needs to be done, it should be about surfacing and weighing up risks. e.g:

- "I think this task is problematic because it's going to massively increase our Dependency Risk to add a new database here."
- "I don't think we should have component A interacting with component B because it'll introduce extra Communication Risk which we will always be tripping over."

- "I worry we might not understand what the sales team want and are facing Implementation Risk. How about we try and get agreement on a specification?"

Tip #2: The Risk Landscape is Increasingly Complex: Utilise This

Figure 10.6: Journey via the Central Line

If you were travelling across London from Ealing (in the West) to Stratford (in the East) the *fastest* route might be to take the Central Line. You could do it via the A406 road, which would take a *bit* longer. It would *feel* like you're mainly going in completely the wrong direction doing that, but it's much faster than cutting straight through London and you don't pay the congestion charge.

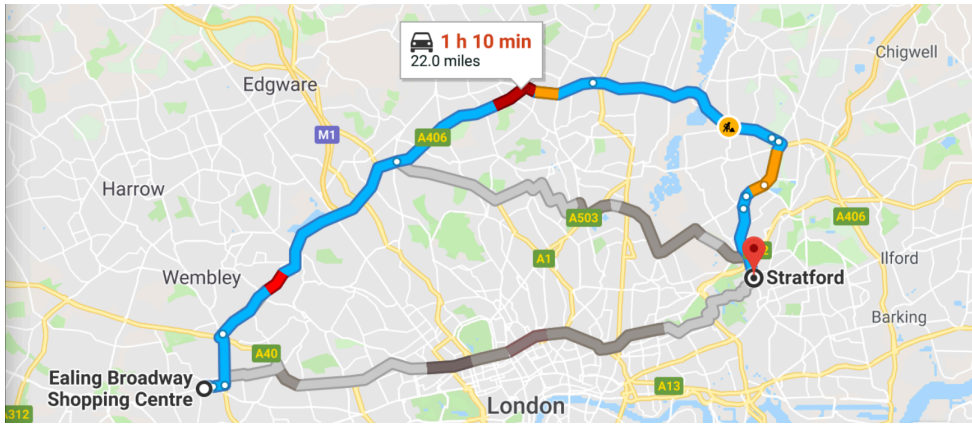


Figure 10.7: Journey by Car

In terms of risk, they all have different profiles. You're often delayed in the car, by some amount. The tube is *generally* reliable, but when it breaks down

or is being repaired it might end up quicker to walk.

If you were doing this same journey on foot, it's a very direct route, but would take five times longer. However, if you were making this journey a hundred years ago, that might be the way you chose (horseback might be a bit faster).

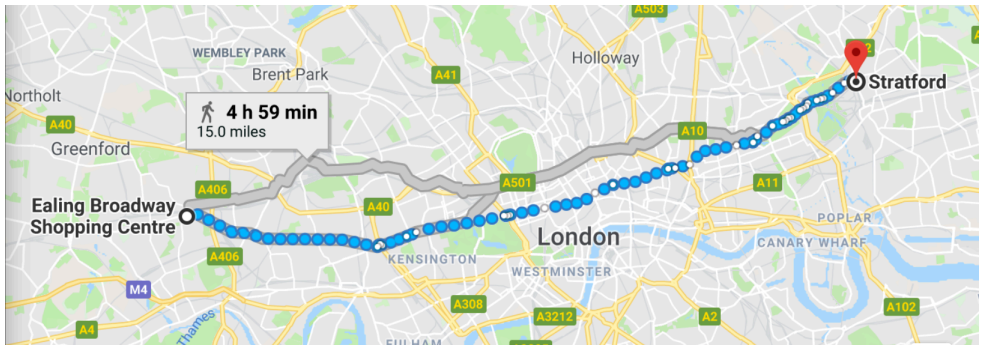


Figure 10.8: Journey on Foot

In the software development past, *building it yourself* was the only way to get anything done. It was like London *before road and rail*. Nowadays, you are bombarded with choices. It's actually *worse than London* because it's not even a two-dimensional geographic space and there are multitudes of different routes and acceptable destinations. Journey planning on the software Risk Landscape is an optimisation problem *par excellence*.

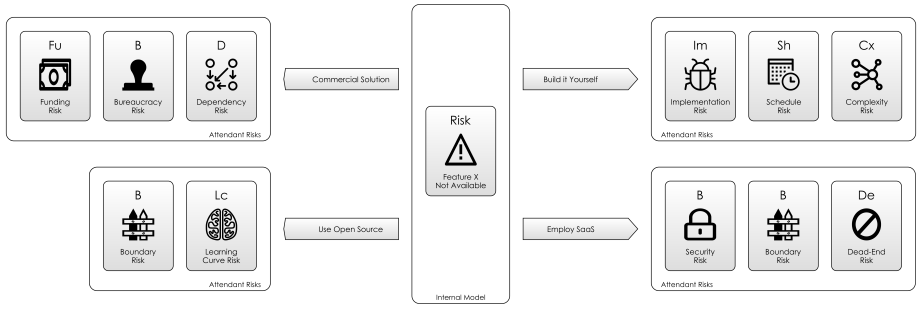


Figure 10.9: Possible Moves On The Risk Landscape

Because the modern Risk Landscape is so complex:

- There can be orders of magnitude difference in *time*, with very little difference in destination.
- If it's Schedule Risk you're worried about, *Code Yourself* isn't a great solution (for the whole thing, anyway). "Take the tube" and at least

partly use something someone built already. There are probably multiple alternatives you can consider.
- If no one has built something similar already, then why is that? Have you formulated the problem properly?
- Going the wrong way is *so much easier*.
- Dead-Ends (like a broken Central Line) are much more likely to trip you up.
- You need to keep up with developments in your field. Read widely.

Tip #3: Meet Reality Early on the Biggest Risks

In getting from A to B on the Risk Landscape, imagine that all the Attendant Risks are the stages of a journey. Some might be on foot, train, car and so on. In order for your course of action to work, all the stages in the journey have to succeed.

Although you might have to make the steps of a journey in some order, you can still mitigate risk in a different order. For example, checking the trains are running, making sure your bike is working, booking tickets and taxis, and so on.

The *sensible* approach would be to test the steps *in order from weakest to strongest*. This means working out how to meet reality for each risk in turn, in order from biggest risk to smallest.

Often, a *strategy* will be broken up into multiple actions. *Which are the riskiest actions?* Figure this out, using the Risk-First vocabulary and the best experience you can bring to bear, then, perform the actions which Pay Off the biggest risks first.

As we saw from the "10X Developer" saw, Learning Curve Risk and Dead End Risk, are likely to be the biggest risks. How can we front-load this and tackle these earlier?

- *Having a vocabulary* (like the one Risk-First provides) allows us to *at least talk about these*. e.g. "I believe there is a Dead End Risk that we might not be able to get this software to run on Linux."
- Build mock-ups:
  - UI wireframes allow us to bottom out the Communication Risk of the interfaces we build.
  - Spike Solutions allow us to de-risk algorithms and approaches before making them part of the main development.

- Don't pick delivery dates far in the future. Collectively work out the biggest risks with your clients, and then arrange the next possible date to demonstrate the mitigation.
- Do actions *early* that are *simple* but are nevertheless show-stoppers. They are as much a source of Hidden Risk as more obviously tricky actions.

Tip #4: Talk Frankly About All The Risks

Let's get back to Bill and Carl. What went wrong between points `(1)` and `(2)`? Let's break it down:

- **Bill *wants* the system in 3-4 months.** It doesn't happen.
- **He says it "must be delivered in 6 months", but this doesn't happen either.** However, the world (and the project) doesn't end: *it carries on*. What does this mean about the truth of his statement? Was he deliberately lying, or just espousing his view on the Schedule Risk?
- **Carl's original estimate was 9 months.** Was he working to this all along? Did the initial brow-beating over deadlines at point `(1)` contribute to Agency Risk in a way that *didn't* happen at point `(2)`?
- **Why *did* Bill get so angry?** His understanding of the Schedule Risk was, if anything, *worse* than Carl's. It's not stated in the account, but it's likely the Trust Risk moved upwards: Did his superiors stop trusting him? Was his job at stake?
- **How could including this risk in the discussion have improved the planning process?** Could the conversation have started like this instead?

> I think it will take about 9 months, but that's just a rough estimate at this point," Carl said. "That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. I need to show the board something by then or I'm worried they will lose confidence in me and this project
>
> OK," said Carl. "But I'm really concerned we have huge Feature Fit Risk. The task of understanding the requirements and doing the design is massive.
>
> Well, in my head it's actually pretty simple, " said Bill. "Maybe I don't have the full picture, or maybe your idea of what to build is more complex than I think it needs to be. That's a massive risk

right there and I think we should try and mitigate it right now before things progress. Maybe I'll need to go back to the board if it's worse than I think.

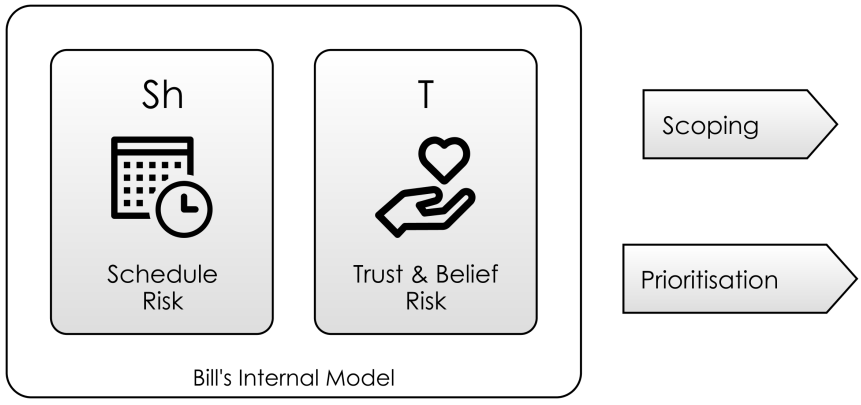


Figure 10.10: Identifying The Action

Tip #5: Picture Worrying Futures

The Bill/Carl problem is somewhat trivial (not to mention likely fictional). How about one from real life? On a project I was working on in November some years ago, we had two pieces of functionality we needed: Bulk Uploads and Spock Integration. (It doesn't really matter what these are). The bulk uploads would be useful *now*. But, the Spock Integration wasn't due until January. In the Spock estimation meeting I wrote the following note:

> Spock estimates were 4, 11 and 22 until we broke it down into tasks. Now, estimates are above 55 for the whole piece. And worryingly, we probably don't have all the tasks. We know we need bulk uploads in November. Spock is January. So, do bulk uploads?

The team *wanted* to start Bulk Uploads work. After all, from these estimates it looked like Spock could easily be completed in January. However, the question should have been:

> If it was February now, and we'd *got nothing done*, what would our biggest risk be?

Missing Bulk Uploads wouldn't be a show-stopper, but missing Spock would be a huge regulatory problem. *Start work on the things you can't miss.*

This is the essence of De-Risking.