

RISK-FIRST

SOFTWARE DEVELOPMENT

Volume 1: The Menagerie



ROB MOFFAT

Risk-First: The Menagerie

By Rob Moffat

Copyright 2018 Kite9 Ltd.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed "Attention: Permissions Coordinator," at the address below.

ISBN: 9781717491855

Credits

tbd

Cover Images: Biodiversity Heritage Library. Biologia Centrali-Americanana. 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
9 Risk Landscape	59
10 Feature Risk	65
11 Complexity Risk	75

12 Communication Risk	93
III Application	115
13 Coming Next	117
14 Estimates	119

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 TVTropes² 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 Software⁴”, 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, Wikipedia⁵

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/3c0wTkH>

⁵https://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.com⁷, and at the risk-first.org⁸ 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 Wikipedia⁹ 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 in-flight. 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, such as Continuous Integration, Unit Testing or Pair Programming.

Software methodologies, then, are collections of tools and practices: “Agile”, “Waterfall”, “Lean” or “Phased Delivery” all prescribe 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 the 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 to manage 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 is that **every action you take on a project is to manage a 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, projects and teams are structured around monitoring risks like *credit risk*, *market risk* and *liquidity risk*.
- *Insurance* 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: it describes a 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.
- Anticipate 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 Manage 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 minimise the risks of bugs slipping through into production, and also manage the Key-Man Risk of knowledge not being widely-enough shared.
- If we write Unit Tests, we're addressing the risk of bugs going to production, but we're also mitigating against the risk of *regression*, and 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.

From the above examples, it's clear that **different tools are appropriate for managing different types of risks**.

5. Different Methodologies are 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 the risk that requirements capture is complicated, error-prone and that requirements change easily.
- **Waterfall** takes the view that development effort is an expensive risk, and that we should build plans up-front to avoid re-work.
- **Lean** takes the view that risk lies in incomplete work and wasted work, and aims to minimise 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. ”

6. We can Drive Development With a Risk-First Perspective

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

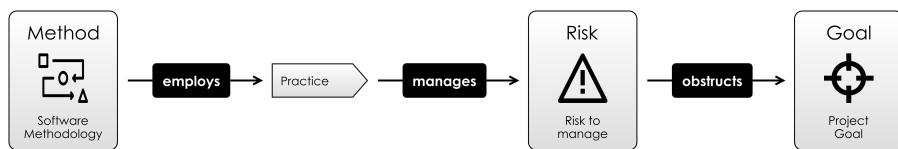


Figure 1: Methodologies, Risks, Practices

How do we take this further?

One idea explored is 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 manage various risks. Beyond this we examine the question: *how can a Risk-First approach inform the use of this practice?*

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 build in **Redundancy**, 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 actions* 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.

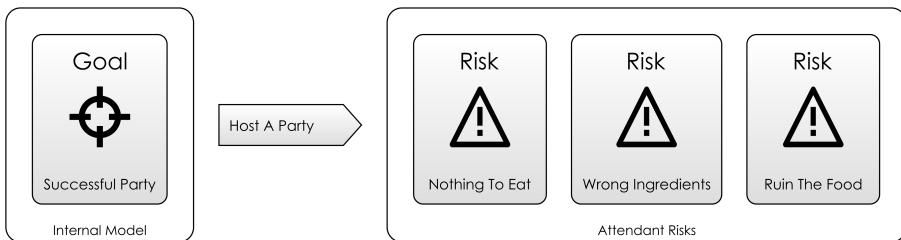


Figure 1.1: Goal In Mind, with the risks you know about

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 Rumsfeld¹ 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).

¹https://en.wikipedia.org/wiki/There_are_known_unknowns

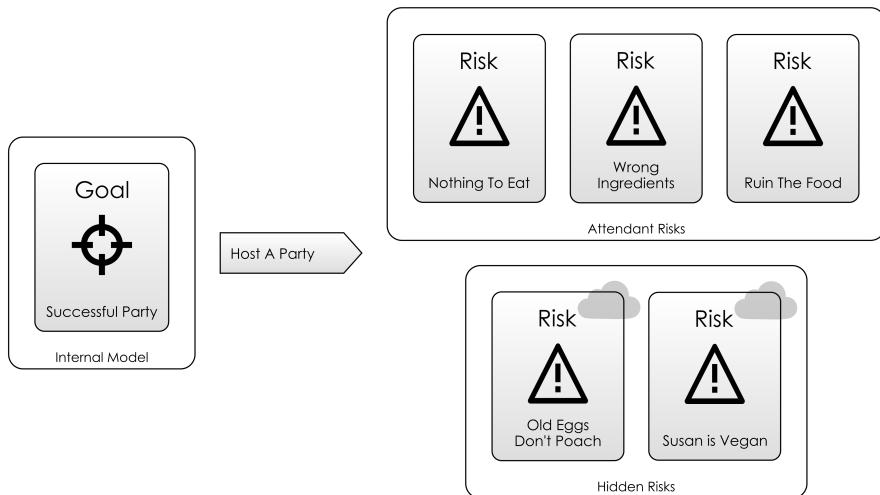


Figure 1.2: Goal In Mind, the risks you know about and the ones you don't

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.

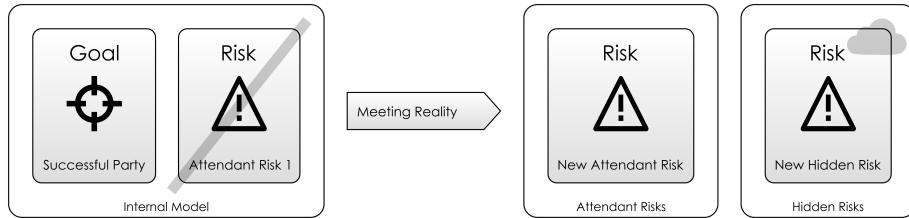


Figure 1.3: How Taking Action affects Reality, and also changes your Internal Model

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...

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*.

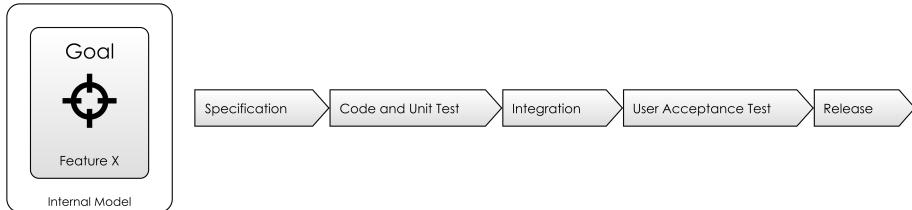


Figure 2.1: A Simple Development Process

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.

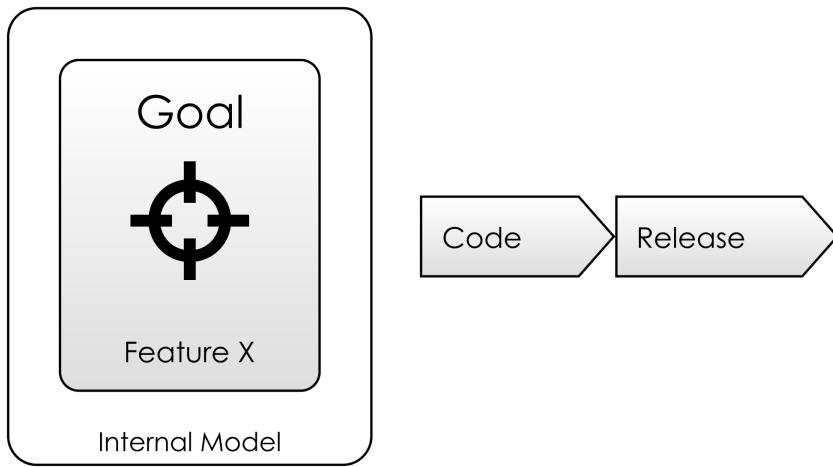


Figure 2.2: A Dangerous Development Process

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:

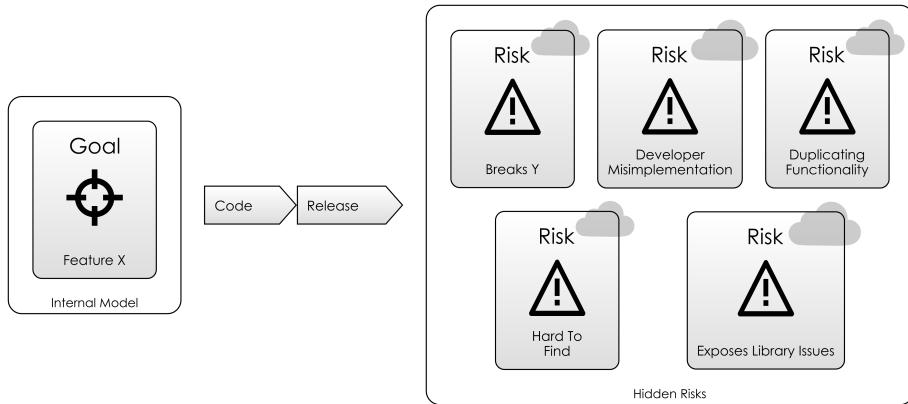


Figure 2.3: Development Process - Exposing Hidden Risks

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 understand 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.

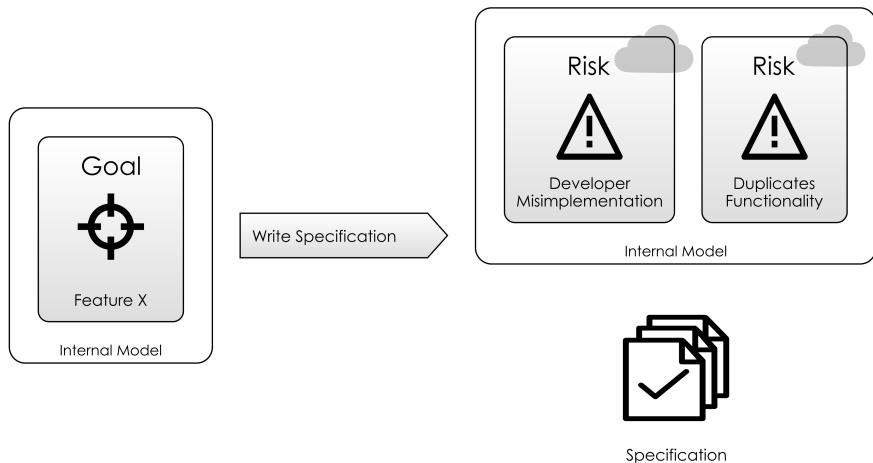


Figure 2.4: BA Specification: exposing Hidden Risks as soon as possible

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 its 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

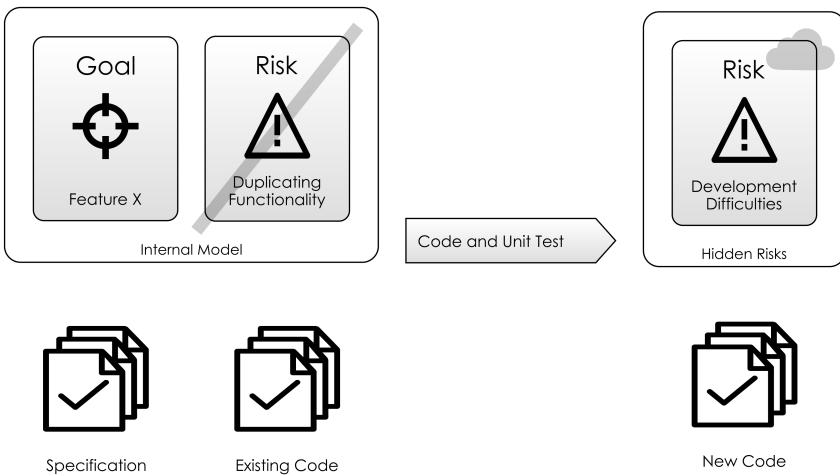


Figure 2.5: Coding Process: exposing more hidden risks as you code

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**

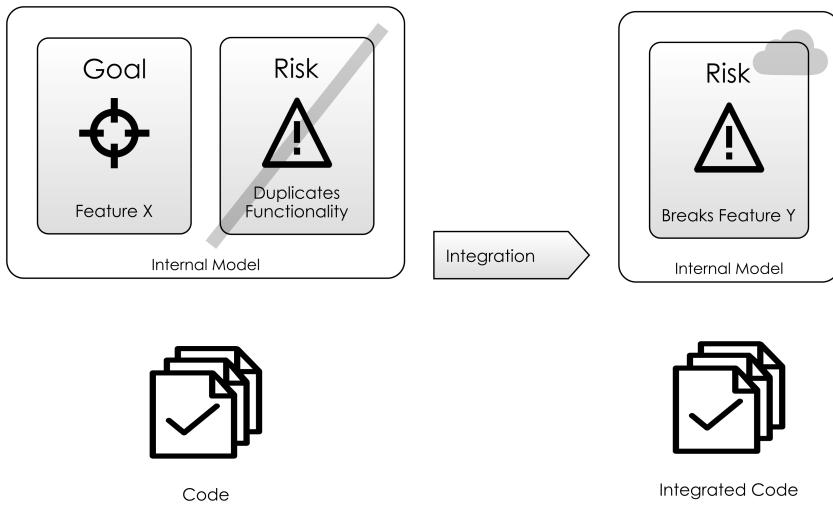


Figure 2.6: Integration testing exposes Hidden Risks before you get to production

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.

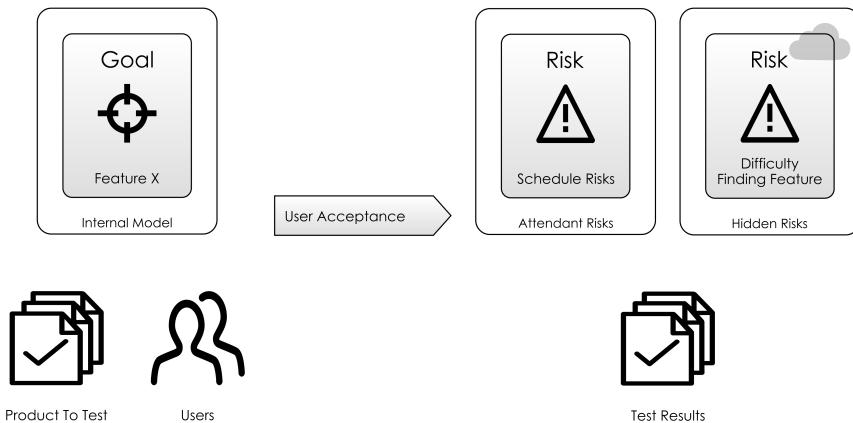


Figure 2.7: UAT - putting tame users in front of your software is better than real ones, where the risk is higher

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.

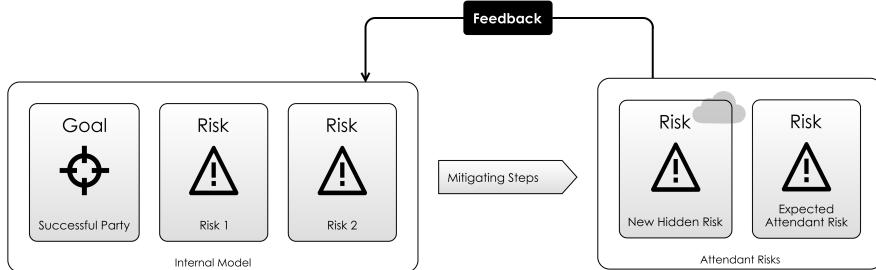
3.1 Revisiting the Model

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 our 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 its 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

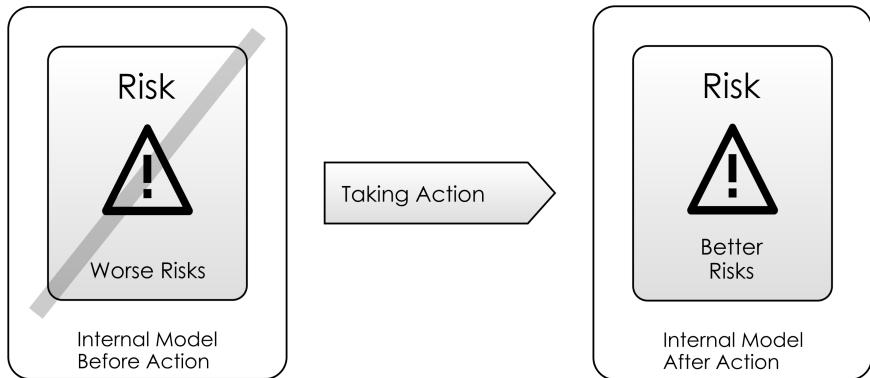


Figure 3.1: Navigating The Risk Landscape

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

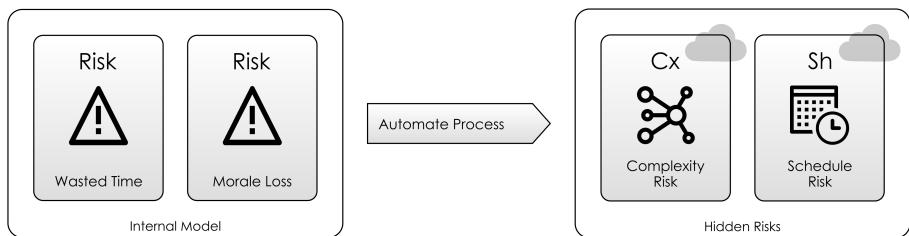


Figure 3.2: Hidden Risks of Automation

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 MongoDB¹ 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 bureaucracy 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.

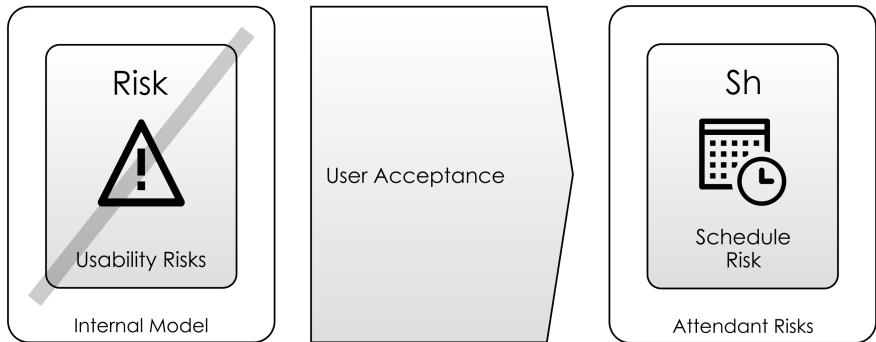


Figure 3.3: Testing flushes out Hidden Risk, but increases Schedule Risk

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 Fowler²

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 It³. 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 Programming⁴ 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 its 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.

⁴https://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.

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 Log¹, 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 everyone 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:

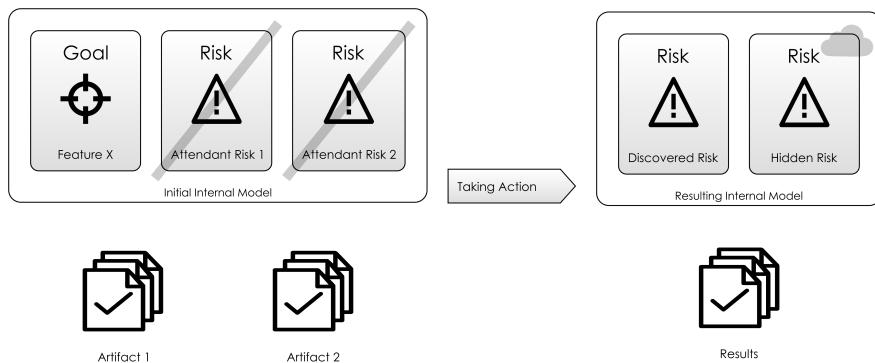


Figure 4.1: Risk-First Diagram Language

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.

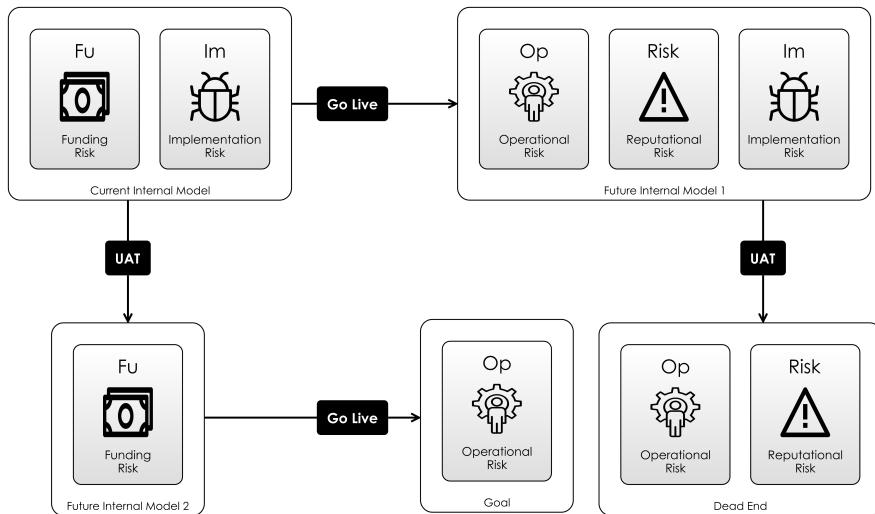


Figure 4.2: UAT or Go Live: Where will you end up?

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.

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 Register¹. 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.

¹https://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 its **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	Probability	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).

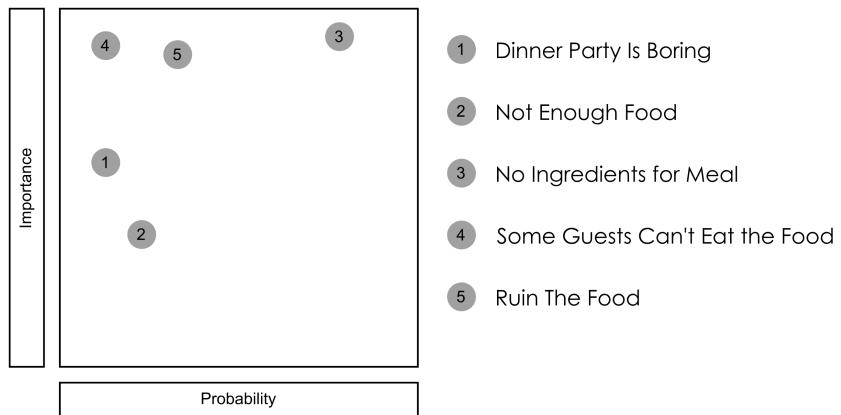


Figure 5.1: Risk Register of Dinner Party Risks

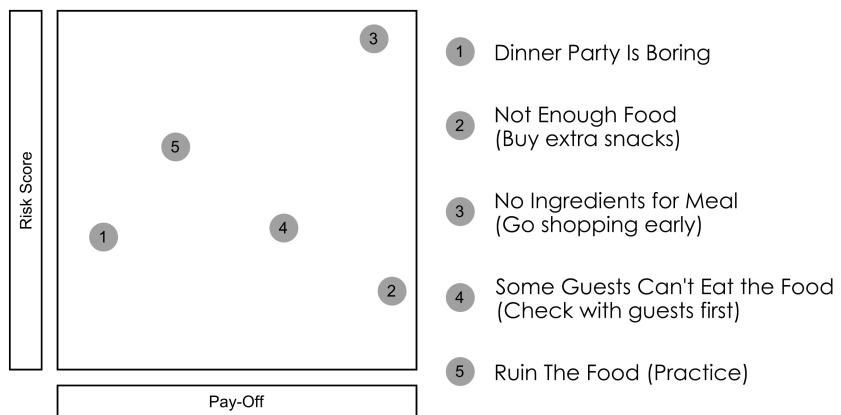


Figure 5.2: Risk Register of Dinner Party Risks, Considering Pay-Off

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 measured² 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 Risk³ or Reputational risk⁴ 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>

³https://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 Lister⁵

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 Costs⁶.

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 Scrum⁷ 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>

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

⁷[https://en.wikipedia.org/wiki/Scrum_\(software_development\)](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 Relativity⁸.

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 Chain⁹ and Net Present Value¹⁰. This is all fine so long as we remember:

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

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

¹⁰https://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 Development¹¹ 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 its 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>

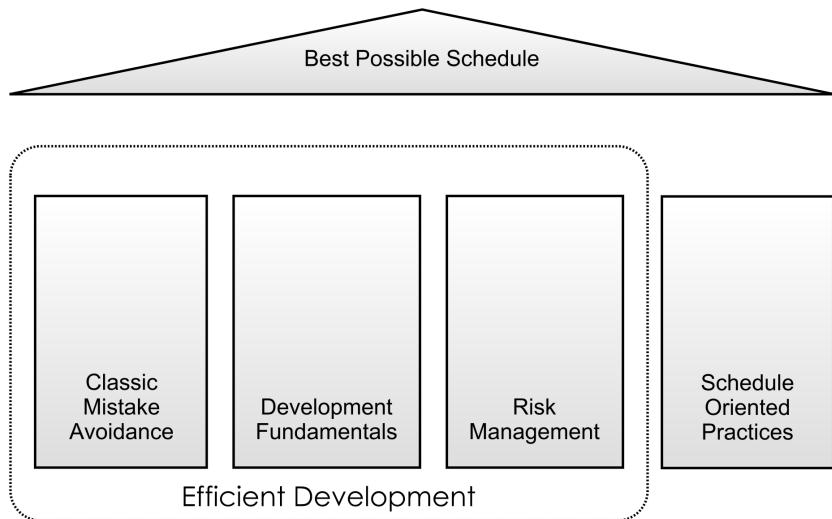


Figure 5.3: Pillars, From Rapid Development By Steve McConnell

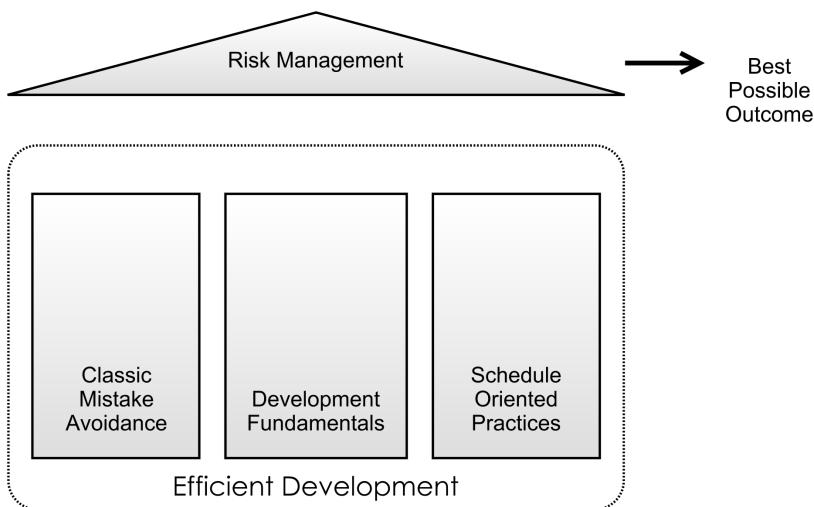


Figure 5.4: Pillars, re-arranged

	Urgent	Not Urgent
Important	Crying Baby Kitchen Fire Some Calls	Exercise Vocation Planning
Not Important	Interruptions Distractions Other Calls	Trivia Busy Work Time Wasters

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, Wikipedia¹²

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:

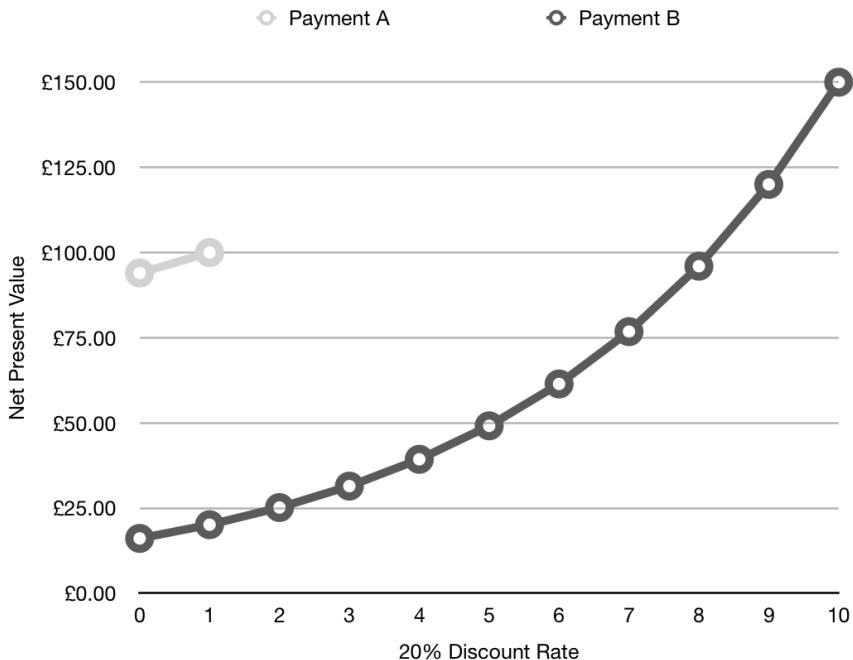


Figure 5.6: Net Present Value Discounting

- Risk A, which will cost you £50 in 5 day's time.
- Risk B, which will cost you £70 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 £45. 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 £40. Under the 6% regime, only Risk B is worth mitigating today, because you spend £40 today to get rid of £45 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.

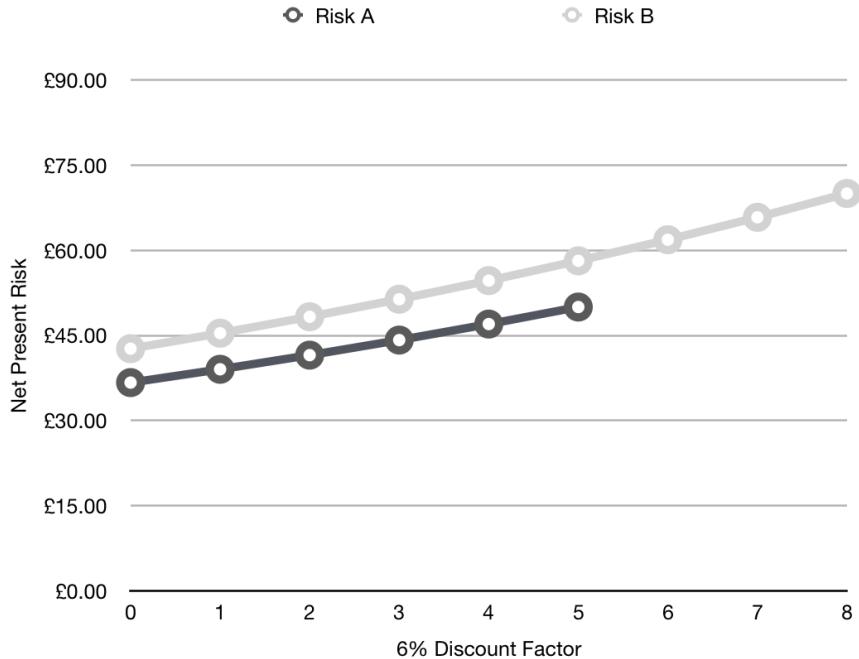


Figure 5.7: Net Present Risk, 6% Discount Rate

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-inflation¹³ 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: £10 cost, happening *tomorrow*
- Risk B: £70 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>

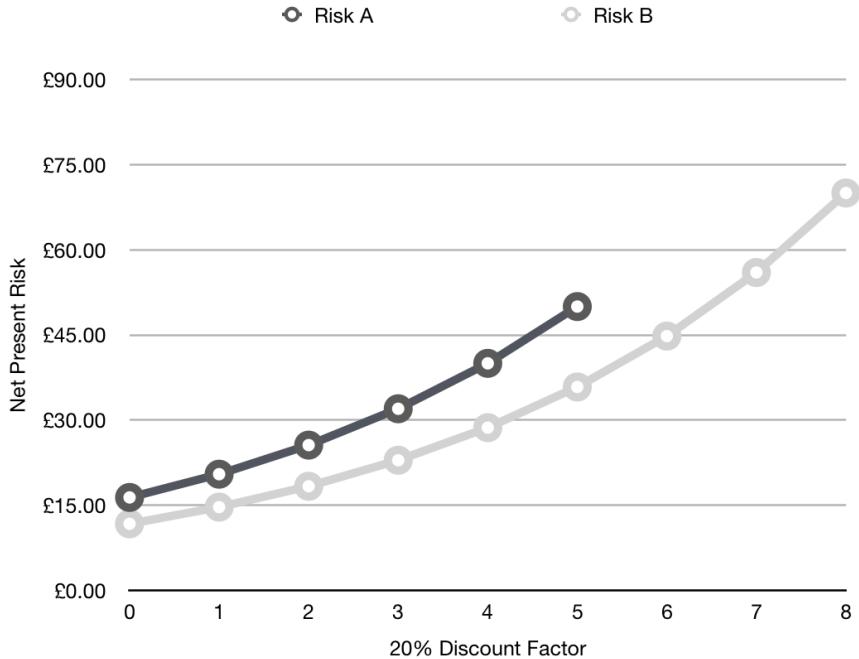


Figure 5.8: Net Present Risk, 20% Discount Rate

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, *Winnie-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>

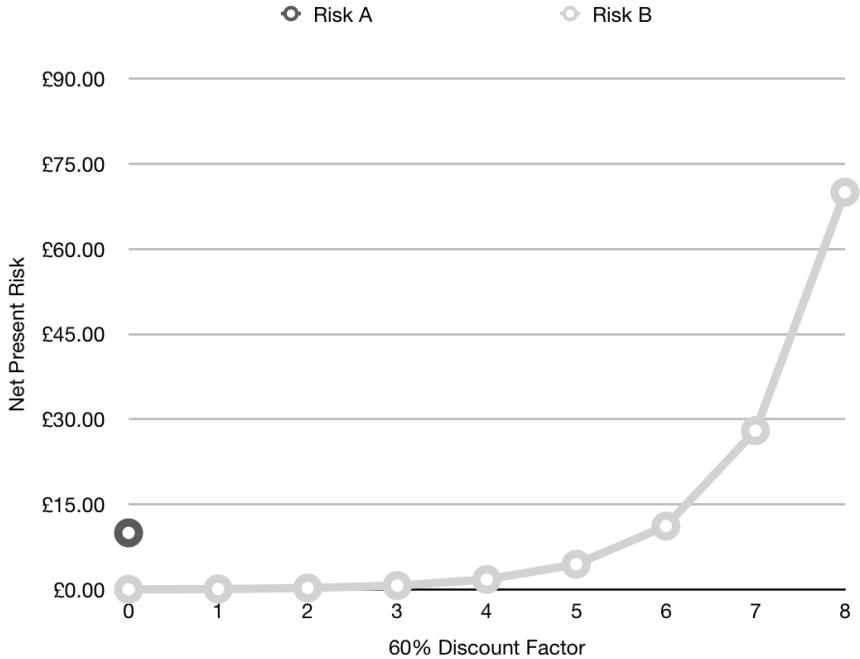


Figure 5.9: Net Present Risk, 60% Discount Rate

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.

Cadence

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

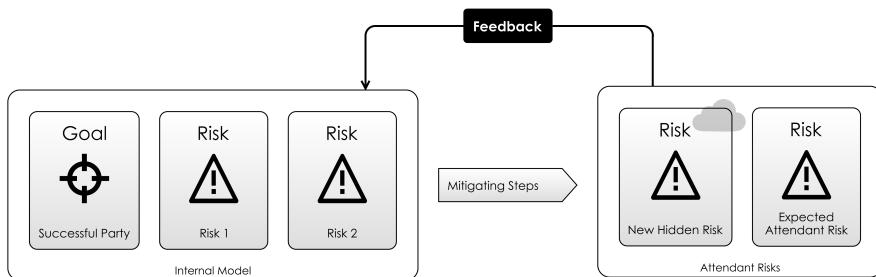


Figure 6.1: Meeting Reality: reality is changed and so is your internal model.

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 short-cut 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 Pyramid² hints at this truth:

¹https://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.

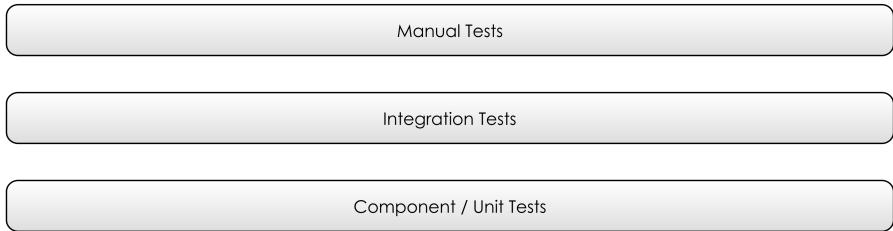


Figure 6.2: The Testing Pyramid

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.

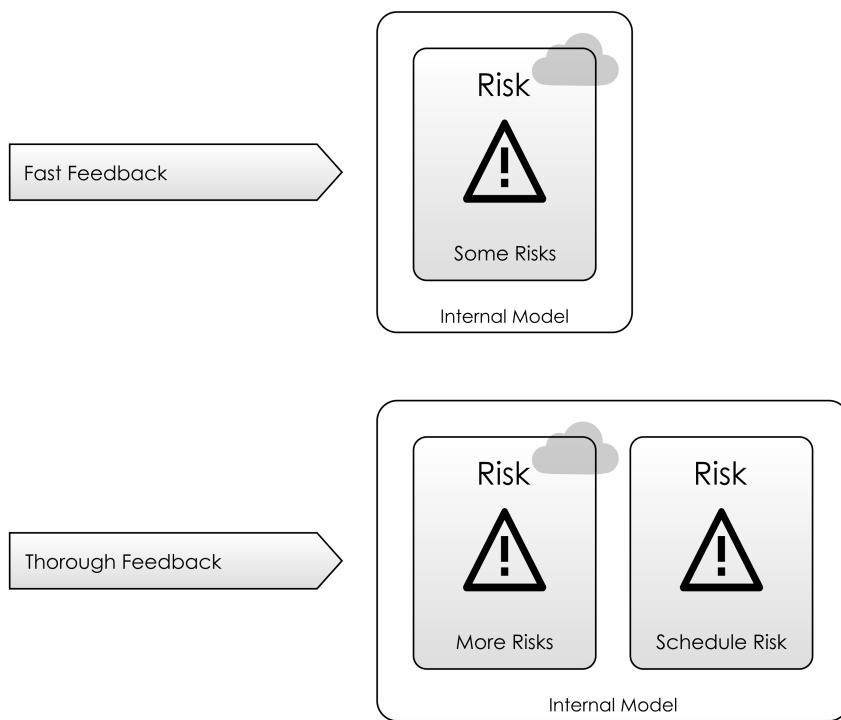


Figure 6.3: Different actions have different feedback loops

De Risking

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.com¹

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, 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 Product³.

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

Learned Helplessness

Sometimes, risks just go away on their own. Learned Helplessness⁴ on the other hand, is where we *could* do something about the risk, but fail to see that as an option:

“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, Wikipedia⁵

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.

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

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

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:

- 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.

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 a real issue around the time of writing. 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 process means that the app-store submission must happen in a few weeks, so this is something of a hard deadline: if we miss it, the next opportunity for 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 of our existing software broke on Windows 7, which sadly we still need to support. 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 app had gone live on Synergy's app-store.

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: we're adding Complexity Risk to our estate, 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 to Apache is a *big* risk, right? We'd have to migrate all of our configuration...

Eve: Yes, let's not go there. So, *changing* the settings on Baroque, we have the 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. In that case, we can attempt the settings change instead.

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 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.

In the next chapter, we are going to briefly look at software methodology, and how it comes up short in when addressing risk.

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

Part II

Risk

Risk Landscape

9.1 Why Should We Categorise The Risks?

This is a “spotters’ guide” to risks, not an in-depth encyclopedia.

If we were studying insects, this might be a guide giving you a description and a picture of each insect, telling you where to find it and what it does. That doesn’t mean that this is *all* there is to know. Just as a scientist could spend her entire life studying a particular species of bee, each of the risks we’ll look at really has a whole sub-discipline of Computer Science attached to it, which we can’t possibly hope to cover all of.

As software developers, we can’t hope to know the detailed specifics of the whole discipline of Complexity Theory¹, or Concurrency Theory². But, we’re still required to operate in a world where these things exist. So, we may as well get used to them, and ensure that we respect their primacy. We are operating in *their* world, so we need to know the rules.

Risk is messy. It’s not always easy to tease apart the different components of risk and look at them individually. Let’s look at a high-profile recent example to see why.

9.2 The Financial Crisis

In the Financial Services³ industry, whole *departments* exist to calculate things like:

¹https://en.wikipedia.org/wiki/Complexity_theory

²[https://en.wikipedia.org/wiki/Concurrency_\(computer_science\)](https://en.wikipedia.org/wiki/Concurrency_(computer_science))

³https://en.wikipedia.org/wiki/Financial_services

- Market Risk⁴: the risk that the amount some asset you hold/borrow/have loaned is going to change in value.
- Credit Risk⁵: the risk that someone who owes you a payment at a specific point in time might not pay it back.
- Liquidity Risk⁶: the risk that you can't find a market to sell/buy something, usually leading to a shortage of ready cash.

... and so on. But, we don't need to know the details exactly to understand this story.

They get expressed in ways like this:

“we have a 95% chance that today we'll lose less than £100”

In the financial crisis, though, these models of risk didn't turn out to be much use. Although there are lots of conflicting explanations of what happened, one way to look at it is this:

- Liquidity difficulties (i.e. amount of cash you have for day-to-day running of the bank) caused some banks to not be able to cover their interest payments.
- This caused credit defaults (the thing that Credit Risk measures were meant to guard against) even though the banks *technically* were solvent.
- That meant that, in time, banks got bailed out, share prices crashed and there was lots of Quantitative Easing⁷.
- All of which had massive impacts on the markets in ways that none of the Market Risk models foresaw.

All the Risks were correlated⁸. That is, they were affected by the *same underlying events, or each other*.

9.3 The Risk Landscape Again

It's like this with software risks, too, sadly.

⁴https://en.wikipedia.org/wiki/Market_risk

⁵https://en.wikipedia.org/wiki/Credit_risk

⁶https://en.wikipedia.org/wiki/Liquidity_risk

⁷https://en.wikipedia.org/wiki/Quantitative_easing

⁸<https://www.investopedia.com/terms/c/correlation.asp>

In Meeting Reality, we looked at the concept of the Risk Landscape, and how a software project tries to *navigate* across this landscape, testing the way as it goes, and trying to get to a position of *more favourable risk*.

It's tempting to think of our Risk Landscape as being like a Fitness Landscape⁹. That is, you have a "cost function" which is your height above the landscape, and you try and optimise by moving downhill in a Gradient Descent¹⁰ fashion.

However, there's a problem with this: As we said in Risk Theory, we don't have a cost function. We can only guess at what risks there are. And, we have to go on our *experience*. For this reason, I prefer to think of the Risk Landscape as a terrain which contains *fauna* and *obstacles* (or, specifically Boundaries).

I am going to try and show you some of the fauna of the Risk Landscape. We know every project is different, so every Risk Landscape is also different. But, just as I can tell you that the landscape outside your window will probably will have some roads, trees, fields, forests, buildings, and that the buildings are likely to be joined together by roads, I can tell you some general things about risks too.

In fact, we're going to try and categorise the kinds of things we see on this Risk Landscape. But, this isn't going to be perfect:

- One risk can "blend" into another just like sometimes a "field" is also a "car-park" or a building might contain some trees (but isn't a forest).
- There is *correlation* between different risks: one risk may cause another, or two risks may be due to the same underlying cause.
- As we saw in Part 1, mitigating one risk can give rise to another, so risks are often *inversely correlated*.

9.4 We're all Naturalists Now

This is a new adventure. There's a long way to go. Just as naturalists are able to head out and find new species of insects and plants, we should expect to do the same. This is by no means a complete picture - it's barely a sketch.

It's a big, crazy, evolving world of software. Help to fill in the details. Report back what you find.

⁹https://en.wikipedia.org/wiki/Fitness_landscape

¹⁰https://en.wikipedia.org/wiki/Gradient_descent

9.5 Our Tour Itinerary

Below is a table outlining the different risks we'll see. There *is* an order to this: the later risks are written assuming a familiarity with the earlier ones. Hopefully, you'll stay to the end and see everything, but you're free to choose your own tour if you want to.

Risk	Description
Feature Risk	When you haven't built features the market needs, or they contain bugs, or the market changes underneath you.
Communication Risk	Risks associated with getting messages heard and understood.
Complexity Risk	Your software is so complex it makes it hard to change, understand or run.
Dependency Risk	Risks of depending on other people, products, software, functions, etc. This is a general look at dependencies, before diving into specifics like...
Scarcity Risk	Risks associated with having a dependency on time, money or some other limited resource.
Software Dependency Risk	When you choose to depend on a software library, service or function.
Process Risk	When you depend on a business process, or human process to give you something you need.
Boundary Risk	Risks due to making decisions that limit your choices later on. Sometimes, you go the wrong way on the Risk Landscape and it's hard to get back to where you want to be.
Agency Risk	Risks that staff have their own Goals, which might not align with those of the project or team.
Coordination Risk	Risks due to the fact that systems contain multiple agents, which need to work together.
Map And Territory Risk	Risks due to the fact that people don't see the world as it really is. (After all, they're working off different, imperfect Internal Models.)

Risk	Description
Operational Risk	Software is embedded in a system containing people, buildings, machines and other services. Operational risk considers this wider picture of risk associated with running a software service or business in the real world.

On each page we'll start by looking at the category of the risk *in general*, and then break this down into some specific sub-types. At the end, in Staging and Classifying we'll have a recap about what we've seen and make some guesses about how things fit together.

So, let's get started with Feature Risk.

Feature Risk

Feature Risks are risks to do with functionality that you need to have in your software.

As a simple example, if your needs are served perfectly by Microsoft Excel, then you don't have any Feature Risk. However, the day you find Microsoft Excel wanting, and decide to build an Add-On is the day when you first appreciate some Feature Risk.

In a way, Feature Risk is very fundamental: if there were *no* feature risk, the job would be done already, either by you, or by another product, and the product would be perfect!

Not considering Feature Risk means that you might be building the wrong functionality, for the wrong audience or at the wrong time. And eventually, this will come down to lost money, business, acclaim, or whatever else reason you are doing your project for. So let's unpack this concept into some of its variations.

10.1 Feature Fit Risk

This is the one we've just discussed above - the feature that you (or your clients) want to use in the software *isn't there*:

- This might manifest itself as complete *absence* of something you need, e.g "Why is there no word count in this editor?"
- It could be that the implementation isn't complete enough, e.g "why can't I add really long numbers in this calculator?"

Feature risks are mitigated by talking to clients and building things, which leads on to...

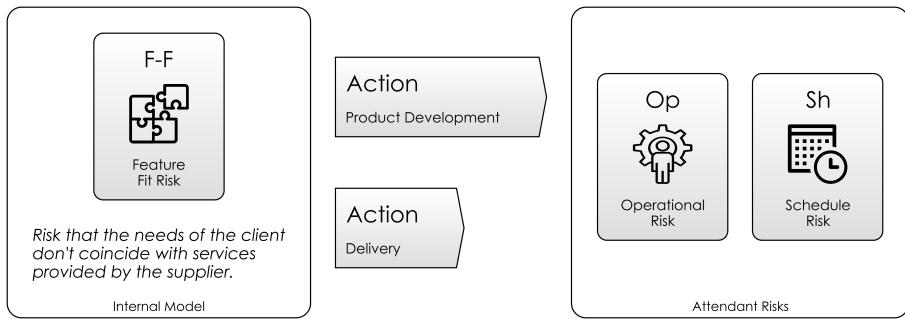


Figure 10.1: Feature Risk

10.2 Implementation Risk

Feature Risk also includes things that don't work as expected: That is to say, bugs¹. Although the distinction between “a missing feature” and “a broken feature” might be worth making in the development team, we can consider these both the same kind of risk: *the software doesn't do what the user expects*.

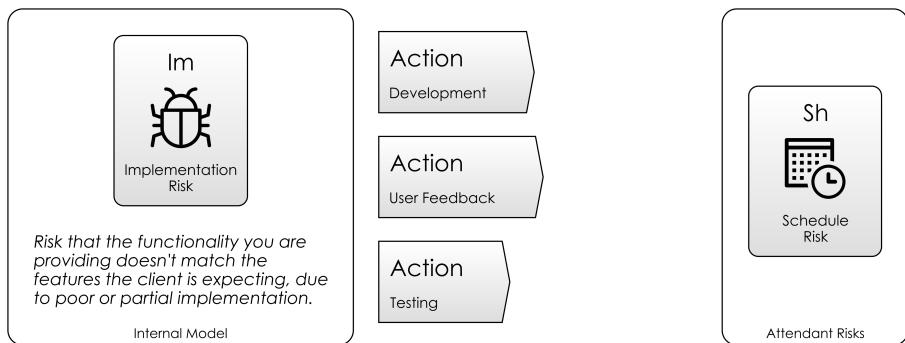


Figure 10.2: Implementation Risk

At this point, it's worth pointing out that sometimes, *the user expects the wrong thing*. This is a different but related risk, which could be down to training, documentation or simply a poor user interface. We'll look at that more in Communication Risk.

¹https://en.wikipedia.org/wiki/Software_bug

10.3 Regression Risk

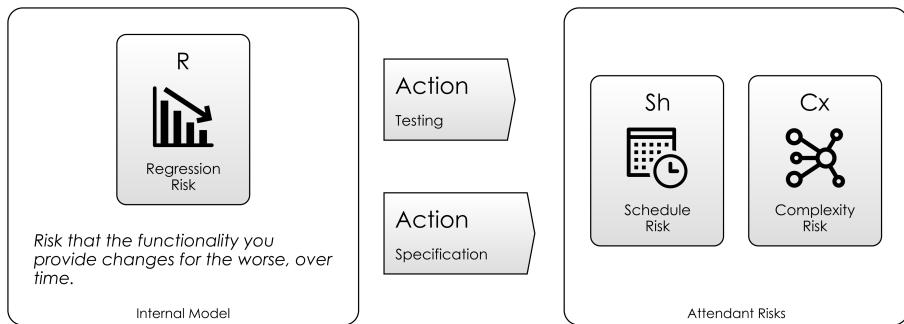


Figure 10.3: Regression Risk

Regression Risk is the risk of breaking existing features in your software when you add new ones. As with the previous risks, the eventual result is the same; customers don't have the features they expect. This can become a problem as your code-base gains Complexity, as it becomes impossible to keep a complete Internal Model of the whole thing in your head.

Delivering new features can delight your customers, breaking existing ones will annoy them. This is something we'll come back to in Reputation Risk.

10.4 Conceptual Integrity Risk

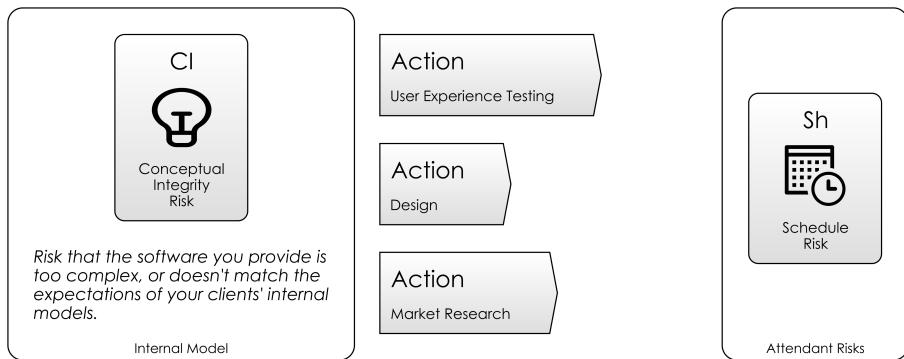


Figure 10.4: Conceptual Integrity Risk

Sometimes, users *swear blind* that they need some feature or other, but it runs at odds with the design of the system, and plain *doesn't make sense*. Often, the development team can spot this kind of conceptual failure as soon as it enters the Backlog. Usually, it's in coding that this becomes apparent.

Sometimes, it can go for a lot longer. I once worked on some software that was built as a score-board within a chat application. However, after we'd added much-asked-for commenting and reply features to our score-board, we realised we'd implemented a chat application *within a chat application*, and had wasted our time enormously.

Feature Phones² are a real-life example: although it *seemed* like the market wanted more and more features added to their phones, Apple's iPhone³ was able to steal huge market share by presenting a much more enjoyable, more coherent user experience, despite being more expensive and having fewer features. Feature Phones had been drowning in increasing Conceptual Integrity Risk without realising it.

This is a particularly pernicious kind of Feature Risk which can only be mitigated by good Design. Human needs are fractal in nature: the more you examine them, the more complexity you can find. The aim of a product is to capture some needs at a *general* level: you can't hope to anticipate everything.

Conceptual Integrity Risk is the risk that chasing after features leaves the product making no sense, and therefore pleasing no-one.

10.5 Feature Access Risk

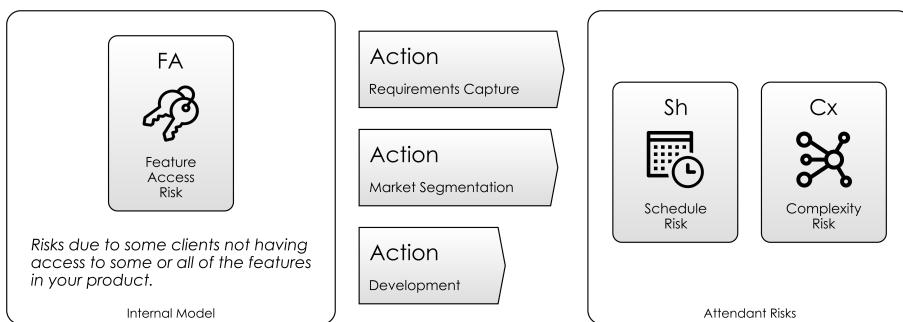


Figure 10.5: Feature Access Risk

²https://en.wikipedia.org/wiki/Feature_phone

³<https://en.wikipedia.org/wiki/IPhone>

Sometimes, features can work for some people and not others: this could be down to Accessibility⁴ issues, language barriers or localisation.

You could argue that the choice of *platform* is also going to limit access: writing code for XBox-only leaves PlayStation owners out in the cold. This is *largely* Feature Access Risk, though Dependency Risk is related here.

In Marketing, minimising Feature Access Risk is all about Segmentation⁵: trying to work out *who* your product is for, and tailoring it to that particular market. For developers, increasing Feature Access means increasing complexity: you have to deliver the software on more platforms, localised in more languages, with different configurations of features. Mitigating Feature Access Risk therefore means increased effort and complexity (which we'll come to later).

Market Risk

Feature Access Risk is related to Market Risk, which I introduced on the Risk Landscape page as being the value that the market places on a particular asset.

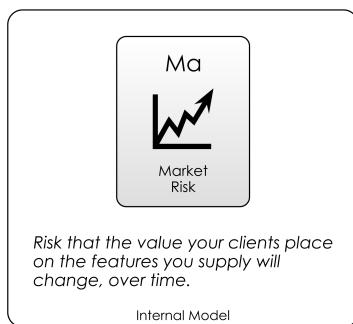


Figure 10.6: Market Risk

“Market risk is the risk of losses in positions arising from movements in market prices.”

—Market Risk, Wikipedia⁶

⁴<https://en.wikipedia.org/wiki/Accessibility>

⁵https://en.wikipedia.org/wiki/Market_segmentation

⁶https://en.wikipedia.org/wiki/Market_risk

I face market risk when I own (i.e. have a *position* in) some Apple⁷ stock. Apple's stock price will decline if a competitor brings out an amazing product, or if fashions change and people don't want their products any more.

Since the product you are building is your asset, it makes sense that you'll face Market Risk on it: the *market* decides what it is prepared to pay for this, and it tends to be outside your control.

10.6 Feature Drift Risk

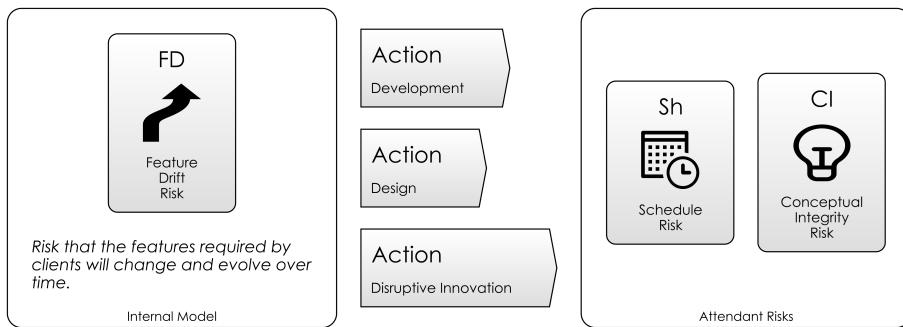


Figure 10.7: Feature Drift Risk

Feature Drift is the tendency that the features people need *change over time*. For example, at one point in time, supporting IE6 was right up there for website developers, but it's not really relevant anymore. Although that change took *many* years to materialize, other changes are more rapid.

The point is: Requirements captured *today* might not make it to *tomorrow*, especially in the fast-paced world of IT. This is partly because the market *evolves* and becomes more discerning. This happens in several ways:

- Features present in competitor's versions of the software become *the baseline*, and they're expected to be available in your version.
- Certain ways of interacting become the norm (e.g. qwerty⁸ keyboards, or the control layout in cars: these don't change with time).
- Features decline in usefulness: *Printing* is less important now than it was, for example.

⁷<http://apple.com>

⁸<https://en.wikipedia.org/wiki/QWERTY>

As we will see later in Boundary Risk, Feature Drift Risk is often a source of Complexity Risk, since you often need to add new features, while not dismantling old features as some users still need them.

Feature Drift Risk is *not the same thing as Requirements Drift*, which is the tendency projects have to expand in scope as they go along. There are lots of reasons they do that, a key one being the Hidden Risks uncovered on the project as it progresses.

Fashion

Fashion plays a big part in IT, as this infographic on website design shows⁹. True, web-sites have got easier to use as time has gone by, and users now expect this. Also, bandwidth is greater now, which means we can afford more media and code on the client side. However, *fashion* has a part to play in this.

By being *fashionable*, web-sites are communicating: *this is a new thing, this is relevant, this is not terrible*: all of which is mitigating a Communication Risk. Users are all-too-aware that the Internet is awash with terrible, abandonware sites that are going to waste their time. How can you communicate that you're not one of them to your users?

10.7 Delight

If this breakdown of Feature Risk seems reductive, then try not to think of it that way: the aim *of course* should be to delight users, and turn them into fans. That's a laudable Goal, but should be treated in the usual Risk-First way: *pick the biggest risk you can mitigate next*.

Consider Feature Risk from both the down-side and the up-side:

- What are we missing?
- How can we be *even better*?

Hopefully, this has given you some ideas about what Feature Risk involves. Hopefully, you might be able to identify a few more specific varieties. But, it's time to move on and look in more detail at Complexity Risk and how it affects what we build.

⁹<https://designers.hubspot.com/blog/the-history-of-web-design-infographic>

10.8 Analysis

At this point, it would be easy to stop and say, look, here are a bunch of Feature Risk issues that you could face. But, it turns out that we're going to be relying heavily on Feature Risk as we go on in order to build our understanding of other risks, so it's probably worth spending a bit of time up front to classify what we've found.

The Feature Risks identified here basically exist in a 3-dimensional space:

- **Fit:** How well the features fit for a particular client.
- **Audience:** The range of clients (the *market*) that may be able to use this feature.
- **Evolution:** The way the fit and the audience changes and evolves as time goes by.

Fit

“This preservation, during the battle for life, of varieties which possess any advantage in structure, constitution, or instinct, I have called Natural Selection; and Mr. Herbert Spencer has well expressed the same idea by the Survival of the Fittest”

—Charles Darwin (Survival of the Fittest), *Wikipedia*¹⁰.

Darwin's conception of fitness was not one of athletic prowess, but how well an organism worked within the landscape, with the goal of reproducing itself.

Feature Fit Risk, Conceptual Integrity Risk and Implementation Risk all hint at different aspects of this “fitness”. We can conceive of them as the gaps between the following entities:

- Perceived Need: What the user *thinks* they want
- Expectation: What the user *expects*
- Reality: What they actually *get*.

For further reading, you can check out The Service Quality Model¹¹, which this model is derived from. This model analyses the types of *quality gaps* in services, and how consumer expectations and perceptions of a service arise. In the Staging And Classifying chapter, we'll come back and build on this model further.

¹⁰https://en.wikipedia.org/wiki/Survival_of_the_fittest

¹¹<http://en.wikipedia.org/SERVQUAL>

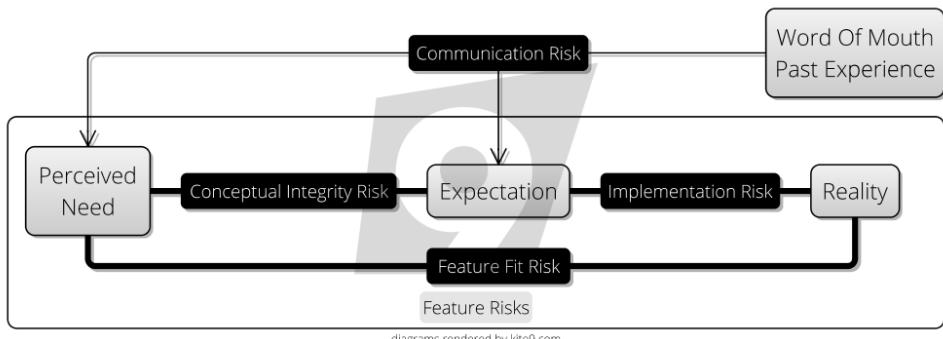


Figure 10.8: Feature Risks Assembled: Fit Risks

Fit and Audience

Two risks, Feature Access Risk and Market Risk considers *Fit* for a whole **Audience** of users. They are different: just as it's possible to have a small audience, but a large revenue, it's possible to have a product which has low Feature Access Risk (i.e lots of users can access it without difficulty) but high Market Risk (i.e. the market is highly volatile or capricious in its demands). *Online services* often suffer from this Market Risk roller-coaster, being one moment highly valued and the next irrelevant.

Fit, Audience and Evolution

Two risks further consider how the **Fit** and **Audience** change: Regression Risk and Feature Drift Risk. We call this *evolution* in the sense that:

- Our product's features *evolve* with time, and changes made by the development team.
- Our audience changes and evolves as it is exposed to our product and competing products.
- The world as a whole is an evolving system within which our product exists.

10.9 Applying Feature Risk

Consider Feature Risk carefully next time you are grooming the backlog:

- Can you judge which tasks mitigate the most Feature Risk?

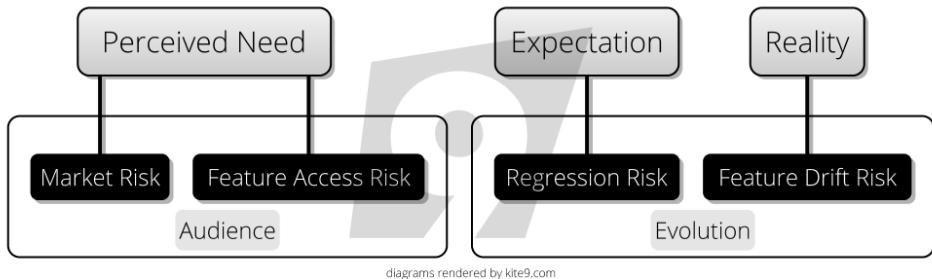


Figure 10.9: Feature Risks Assembled: Audience and Evolution

- Are you delivering features that are valuable to a large audience? How well do you understand your audience? How does the size of the audience for a task impact its importance in the backlog?
- Does the audience *know* that the features exist? How do you communicate feature availability to them?
- How does writing a specification mitigate Fit Risk? For what other reasons are you writing specifications?

In the next chapter, we are going to unpack this third point further. Somewhere between “what the customer wants” and “what you give them” is a *dialog*. In using a software product, users are engaging in a *dialog* with its features. If the features don’t exist, hopefully they will engage in a dialog with the development team to get them added.

These dialogs are prone to risk, and this is the subject of the next chapter, Communication-Risk.

Complexity Risk

Complexity Risk are the risks to your project due to its underlying “complexity”. Over the next few chapters, we’ll break down exactly what we mean by complexity, looking at Dependency Risk and Boundary Risk as two particular sub-types of Complexity Risk. However, in this chapter, we’re going to be specifically focusing on *code you write*: the size of your code-base, the number of modules, the interconnectedness of the modules and how well-factored the code is.

You could think of this chapter, then, as **Codebase Risk**: We’ll look at three separate measures of codebase complexity and talk about Technical Debt, and look at places in which **Codebase Risk** is at it’s greatest.

11.1 Kolmogorov Complexity

The standard Computer-Science definition of complexity, is Kolmogorov Complexity¹. This is:

“... is the length of the shortest computer program (in a pre-determined programming language) that produces the object as output.” - Kolmogorov Complexity, Wikipedia

This is a fairly handy definition for us, as it means that to in writing software to solve a problem, there is a lower bound on the size of the software we write. In practice, this is pretty much impossible to quantify. But that doesn’t really matter: the techniques for *moving in that direction* are all that we are interested in, and this basically amounts to compression.

¹https://en.wikipedia.org/wiki/Kolmogorov_complexity

Let's say we wanted to write a JavaScript program to output this string:

```
abcdabcdabcdabcdabcdabcdabcdabcdabcd
```

We might choose this representation:

```
function out() {  
    return "abcdabcdabcdabcdabcdabcdabcdabcd"  
}  
                                         (7 )  
                                         (45)  
                                         (1 )
```

The numbers in brackets indicate how many symbols each line contains, so in total, this code block contains **53 symbols**, if you count `function`, `out` and `return` as one symbol each.

But, if we write it like this:

```
const ABCD="ABCD";  
  
function out() {  
    return ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+  
          ABCD+ABCD+ABCD;  
}  
                                         (11)  
                                         (7 )  
                                         (16)  
                                         (6 )  
                                         (1 )
```

With this version, we now have **41 symbols** (`ABCD` is a single symbol, because we could have called it anything). And with this version:

```
const ABCD="ABCD";  
  
function out() {  
    return ABCD.repeat(10)  
}  
                                         (11)  
                                         (7 )  
                                         (7 )  
                                         (1 )
```

... we have **26 symbols**.

Abstraction

What's happening here is that we're *exploiting a pattern*: we noticed that `ABCD` occurs several times, so we defined it a single time and then used it over and over, like a stamp. Separating the *definition* of something from the *use* of something as we've done here is called "abstraction". We're going to come

across it over and over again in this part of the book, and not just in terms of computer programs.

By applying techniques such as Abstraction, we can improve in the direction of the Kolmogorov limit. And, by allowing ourselves to say that *symbols* (like `out` and `ABCD`) are worth one complexity point, we've allowed that we can be descriptive in our `function` name and `const`. Naming things is an important part of abstraction, because to use something, you have to be able to refer to it.

Trade-Off

But we could go further down into Code Golf² territory. This javascript program plays FizzBuzz³ up to 100, but is less readable than you might hope:

```
for(i=0;i<100;)document.write(((++i%3?'Fizz':Buzz')+(i%5?'Buzz':i)+"<br>") (62)
```

So there is at some point a trade-off to be made between Complexity Risk and Communication Risk. This is a topic we'll address more in that chapter. But for now, it should be said that Communication Risk is about *misunderstanding*: The more complex a piece of software is, the more difficulty users will have understanding it, and the more difficulty developers will have changing it.

11.2 Connectivity

A second, useful measure of complexity comes from graph theory, and that is the connectivity of a graph:

“... the minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other”

—Connectivity, Wikipedia⁴

To see this in action, have a look at the below graph:

It has 10 vertices, labelled `a` to `j`, and it has 15 edges (or links) connecting the vertices together. If any single edge were removed from this diagram, the 10

²https://en.wikipedia.org/wiki/Code_golf

³https://en.wikipedia.org/wiki/Fizz_buzz

⁴[https://en.wikipedia.org/wiki/Connectivity_\(graph_theory](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)

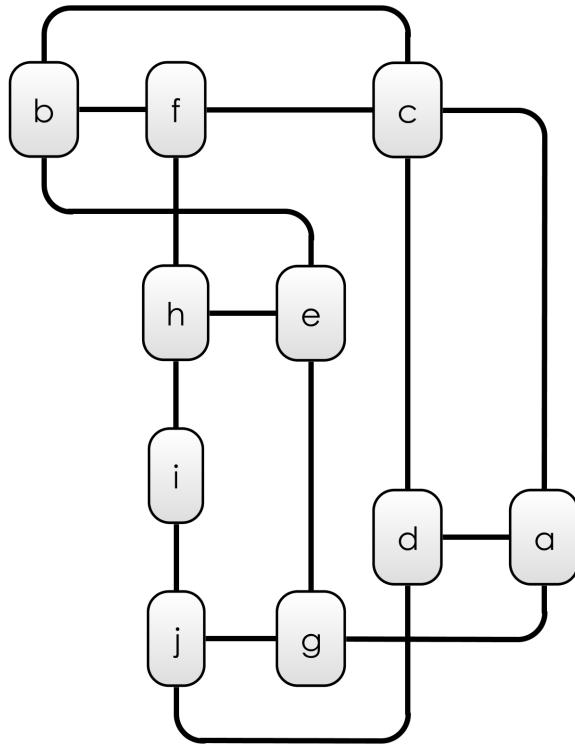


Figure 11.1: Graph 1

vertices would still be linked together. Because of this, we can say that the graph is *2-connected*. That is, to disconnect any single vertex, you'd have to remove *at least* two edges.

As a slight aside, let's consider the **Kolmogorov Complexity** of this graph, by inventing a mini-language to describe graphs. It could look something like this:

```
<item> : [<item>,]* <item>      # Indicates that the item
                                         # before the colon
                                         # has a connection to all
                                         # the items after the colon
```

So our graph could be defined like this:

```

a: b,c,d
b: c,f,e
c: f,d
d: j
e: h,j
f: h
g: j
h: i
i: j

```

(39)

Let's remove some of those extra links:

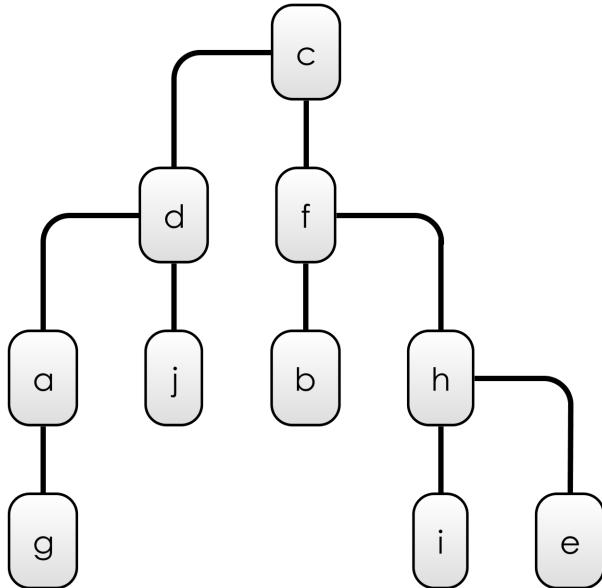


Figure 11.2: Graph 2

In this graph, I've removed 6 of the edges. Now, we're in a situation where if any single edge is removed, the graph becomes *unconnected*. That is, it's broken into distinct chunks. So, it's *1-connected*.

The second graph is clearly simpler than the first. And, we can show this by looking at the **Kolgomorov Complexity** in our little language:

```
a: d,g  
b: f  
c: d,f  
d: j  
f: h  
e: h  
h: i
```

(25)

Connectivity is also **Complexity**. Heavily connected programs/graphs are much harder to work with than less-connected ones. Even *laying out* the first graph sensibly is a harder task than the second (the second is a doddle). But the reason programs with greater connectivity are harder to work with is that changing one module potentially impacts many others.

11.3 Hierarchies and Modularisation

In the second, simplified graph, I've arranged it as a hierarchy, which I can do now that it's only 1-connected. For 10 vertices, we need 9 edges to connect everything up. It's always:

```
edges = vertices - 1
```

Note that I could pick any hierarchy here: I don't have to start at **c** (although it has the nice property that it has two roughly even sub-trees attached to it).

How does this help us? Imagine if **a - j** were modules of a software system, and the edges of the graph showed communications between the different sub-systems. In the first graph, we're in a worse position: who's in charge? What deals with what? Can I isolate a component and change it safely? What happens if one component disappears? But, in the second graph, it's easier to reason about, because of the reduced number of connections and the new hierarchy of organisation.

On the down-side, perhaps our messages have farther to go now: in the original **i** could send a message straight to **j**, but now we have to go all the way via **c**. But this is the basis of Modularisation⁵ and Hierarchy⁶.

As a tool to battle complexity, we don't just see this in software, but everywhere in our lives. Society, business, nature and even our bodies:

⁵https://en.wikipedia.org/wiki/Modular_programming

⁶<https://en.wikipedia.org/wiki/Hierarchy>

- **Organelles** - such as Mitochondria⁷.
- **Cells** - such as blood cells, nerve cells, skin cells in the Human Body⁸.
- **Organs** - like hearts livers, brains etc.
- **Organisms** - like you and me.

The great complexity-reducing mechanism of modularisation is that *you only have to consider your local environment*. Elements of the program that are “far away” in the hierarchy can be relied on not to affect you. This is somewhat akin to the **Principal Of Locality**:

“Spatial locality refers to the use of data elements within relatively close storage locations.”

—Locality Of Reference, *Wikipedia*⁹

11.4 Cyclomatic Complexity

A variation on this graph connectivity metric is our third measure of complexity, Cyclomatic Complexity¹⁰. This is:

$$\text{Cyclomatic Complexity} = \text{edges} - \text{vertices} + 2P,$$

Where **P** is the number of **Connected Components** (i.e. distinct parts of the graph that aren’t connected to one another by any edges).

So, our first graph had a **Cyclomatic Complexity** of 7. ($15 - 10 + 2$), while our second was 1. ($9 - 10 + 2$).

Cyclomatic complexity is all about the number of different routes through the program. The more branches a program has, the greater it’s cyclomatic complexity. Hence, this is a useful metric in Testing and Code Coverage¹¹: the more branches you have, the more tests you’ll need to exercise them all.

11.5 More Abstraction

Although we ended up with our second graph having a **Cyclomatic Complexity** of 1 (the minimum), we can go further through abstraction, because

⁷<https://en.wikipedia.org/wiki/Mitochondrion>

⁸https://en.wikipedia.org/wiki/List_of_distinct_cell_types_in_the_adult_human_body

⁹https://en.wikipedia.org/wiki/Locality_of_reference

¹⁰https://en.wikipedia.org/wiki/Cyclomatic_complexity

¹¹https://en.wikipedia.org/wiki/Code_coverage

this representation isn't minimal from a **Kolmogorov Complexity** point-of-view. For example, we might observe that there are further similarities in the graph that we can "draw out":

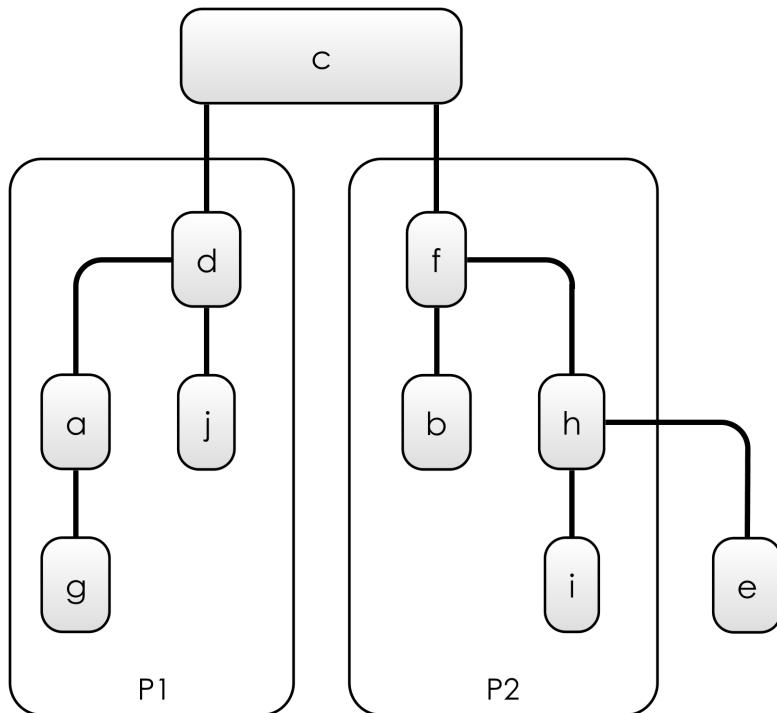


Figure 11.3: Complexity 3

Here, we've spotted that the structure of subgraphs **P1** and **P2** are the same: we can have the same functions there to assemble those. Noticing and exploiting patterns of repetition is one of the fundamental tools we have in the fight against Complexity Risk.

So, we've looked at some measures of software structure complexity, in order that we can say "this is more complex than this". However, we've not really said why complexity entails Risk. So let's address that now by looking at two analogies, Mass and Technical Debt.

11.6 Complexity As Mass

The first way to look at complexity is as **Mass** or **Inertia**: a software project with more complexity has greater **Inertia** or **Mass** than one with less complexity.

Newton's Second Law states:

" $F = ma$, (Force = Mass x Acceleration)"

—Netwon's Laws Of Motion, *Wikipedia*¹²

That is, in order to move your project *somewhere new*, and make it do new things, you need to give it a push, and the more **Mass** it has, the more **Force** you'll need to move (accelerate) it.

Inertia and **Mass** are equivalent concepts in physics:

"mass is the quantitative or numerical measure of a body's inertia, that is of its resistance to being accelerated".

—Inertia, *Wikipedia*¹³

You could stop here and say that the more lines of code a project contains, the higher it's mass. And, that makes sense, because in order to get it to do something new, you're likely to need to change more lines.

But there is actually some underlying sense in which *this is real*, as discussed in this Veritasium¹⁴ video. To paraphrase:

"Most of your mass you owe due to $E=mc^2$, you owe to the fact that your mass is packed with energy, because of the **interactions** between these quarks and gluon fluctuations in the gluon field... what we think of as ordinarily empty space... that turns out to be the thing that gives us most of our mass."

—Your Mass is NOT From the Higgs Boson, *Veritasium*¹⁵

I'm not an expert in physics, *at all*, and so there is every chance that I am pushing this analogy too hard. But, substituting quarks and gluons for pieces

¹²https://en.wikipedia.org/wiki/Newton's_laws_of_motion

¹³https://en.wikipedia.org/wiki/Inertia#Mass_and_inertia

¹⁴<https://www.youtube.com/user/1veritasium>

¹⁵https://www.youtube.com/watch?annotation_id=annotation_3771848421&feature=iv&src_vid=Xo232kyTs00&v=Ztc6QPNuqls

of software we can (in a very handwaving-y way) say that more complex software has more **interactions** going on, and therefore has more mass than simple software.

The reason I am labouring this analogy is to try and make the point that Complexity Risk is really fundamental:

- Feature Risk: like **money**.
- Schedule Risk: like **time**.
- Complexity Risk: like **mass**.

At a basic level, Complexity Risk heavily impacts on Schedule Risk: more complexity means you need more force to get things done, which takes longer.

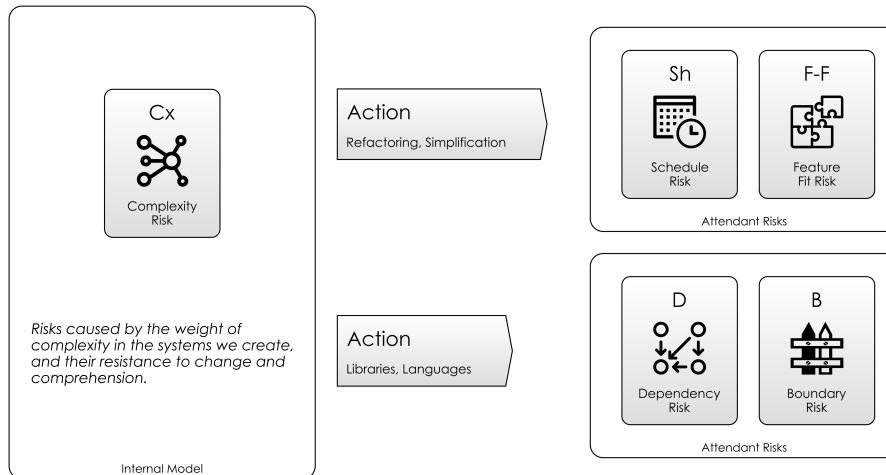


Figure 11.4: Complexity Risk and its mitigations

11.7 Technical Debt

The most common way we talk about unnecessary complexity in software is as Technical Debt:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . The danger occurs when the debt is not repaid. Every

minute spent on not-quite-right code counts as interest on that debt. Entire engineering organisations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.”

Ward Cunningham, 1992¹⁶

Building a perfect first-time solution is a waste, because perfection takes a long time. You’re taking on more attendant Schedule Risk than necessary and Meeting Reality more slowly than you could.

A quick-and-dirty, over-complex implementation mitigates the same Feature Risk and allows you to Meet Reality faster.

But, having mitigated the Feature Risk, you are now carrying more Complexity Risk than you necessarily need, and it’s time to think about how to Refactor¹⁷ the software to reduce this risk again.

11.8 Kitchen Analogy

It’s often hard to make the case for minimising Technical Debt: it often feels that there are more important priorities, especially when technical debt can be “swept under the carpet” and forgotten about until later. (See Discounting The Future.)

One helpful analogy I have found is to imagine your code-base is a kitchen. After preparing a meal (i.e. delivering the first implementation), *you need to tidy up the kitchen*. This is just something everyone does as a matter of *basic sanitation*.

Now of course, you could carry on with the messy kitchen. When tomorrow comes and you need to make another meal, you find yourself needing to wash up saucepans as you go, or working around the mess by using different surfaces to chop on.

It’s not long before someone comes down with food poisoning.

We wouldn’t tolerate this behaviour in a restaurant kitchen, so why put up with it in a software project?

11.9 Feature Creep

In Brooks’ essay “No Silver Bullet - Essence and Accident in Software Engineering”, a distinction is made between:

¹⁶https://en.wikipedia.org/wiki/Technical_debt

¹⁷https://en.wikipedia.org/wiki/Code_refactoring

- **Essence:** *the difficulties inherent in the nature of the software.*
- **Accident:** *those difficulties that attend its production but are not inherent.* - Fred Brooks, *No Silver Bullet*¹⁸

The problem with this definition is that we are accepting features of our software as *essential*.

The Risk-First approach is that if you want to mitigate some Feature Risk then you have to pick up Complexity Risk as a result. But, that's a *choice you get to make*.

Therefore, Feature Creep¹⁹ (or Gold Plating²⁰) is a failure to observe this basic equation: instead of considering this trade off, you're building every feature possible. This has an impact on Complexity Risk, which in turn impacts Communication Risk and also Schedule Risk.

Sometimes, feature-creep happens because either managers feel they need to keep their staff busy, or the staff decide on their own that they need to keep themselves busy. But now, we can see that basically this boils down to bad risk management.

“Perfection is Achieved Not When There Is Nothing More to Add, But When There Is Nothing Left to Take Away”

Antoine de Saint-Exupery²¹

11.10 Dead-End Risk

Dead-End Risk is where you build functionality that you *think* is useful, only to find out later that actually, it was a dead-end, and is superseded by something else.

For example, let's say that the Accounting sub-system needed password protection (so you built this). Then the team realised that you needed a way to *change the password* (so you built that). Then, that you needed to have more than one user of the Accounting system so they would all need passwords (OK, fine).

Finally, the team realises that actually logging-in would be something that all the sub-systems would need, and that it had already been implemented more thoroughly by the Approvals sub-system.

¹⁸https://en.wikipedia.org/wiki/No_Silver_Bullet

¹⁹https://en.wikipedia.org/wiki/Feature_creep

²⁰[https://en.wikipedia.org/wiki/Gold_plating_\(software_engineering\)](https://en.wikipedia.org/wiki/Gold_plating_(software_engineering))

²¹<http://www.quotationspage.com/quote/26979.html>

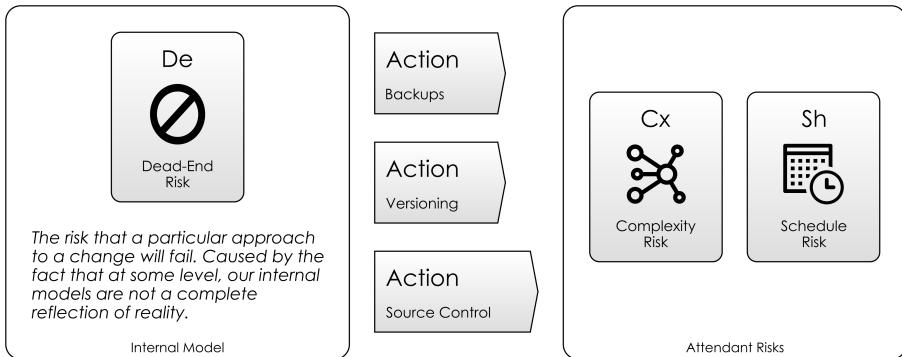


Figure 11.5: Dead-End Risk

At this point, you realise you're in a **Dead End**:

- **Option 1:** You carry on making minor incremental improvements to the accounting password system (carrying the extra Complexity Risk of the duplicated functionality).
- **Option 2:** You rip out the accounting password system, and merge in the Approvals system, surfacing new, hidden Complexity Risk in the process, due to the difficulty in migrating users from the old to new way of working.
- **Option 3:** You start again, trying to take into account both sets of requirements at the same time, again, possibly surfacing new hidden Complexity Risk due to the combined approach.

Sometimes, the path from your starting point to your goal on the Risk Landscape will take you to dead ends: places where the only way towards your destination is to lose something, and do it again another way.

This is because you surface new Hidden Risk along the way. And the source of a lot of this hidden risk will be unexpected Complexity Risk in the solutions you choose. This happens a lot.

Source Control

Version Control Systems²² like Git²³ are a useful mitigation of Dead-End Risk, because it means you can *go back* to the point where you made the

²²https://en.wikipedia.org/wiki/Version_control

²³<https://en.wikipedia.org/wiki/Git>

bad decision and go a different way. Additionally, they provide you with backups against the often inadvertent Dead-End Risk of someone wiping the hard-disk.

The Re-Write

Option 3, Rewriting code or a whole project can seem like a way to mitigate Complexity Risk, but it usually doesn't work out too well. As Joel Spolsky says:

There's a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is the interesting observation: they are probably wrong. The reason that they think the old code is a mess is because of a cardinal, fundamental law of programming:
It's harder to read code than to write it. - Things You Should Never Do, Part 1, *Joel Spolsky*²⁴

The problem that Joel is outlining here is that the developer mistakes hard-to-understand code for unnecessary Complexity Risk. Also, perhaps there is Agency Risk because the developer is doing something that is more useful to him than the project. We're going to return to this problem in again Communication Risk.

11.11 Where Complexity Hides

Complexity isn't spread evenly within a software project. Some problems, some areas, have more than their fair share of issues. We're going to cover a few of these now, but be warned, this is not a complete list by any means:

- Memory Management
- Protocols / Types
- Algorithmic (Space and Time) Complexity
- Concurrency / Mutability
- Networks / Security

Memory Management

Memory Management is another place where Complexity Risk hides:

²⁴<https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>

"Memory leaks are a common error in programming, especially when using languages that have no built in automatic garbage collection, such as C and C++."

—Memory Leak, Wikipedia²⁵

Garbage Collectors²⁶ (as found in Javascript or Java) offer you the deal that they will mitigate the Complexity Risk of you having to manage your own memory, but in return perhaps give you fewer guarantees about the *performance* of your software. Again, there are times when you can't accommodate this Operational Risk, but these are rare and usually only affect a small portion of an entire software-system.

Protocols And Types

Whenever two components of a software system need to interact, they have to establish a protocol for doing so. There are lots of different ways this can work, but the simplest example I can think of is where some component **a** calls some function **b**. e.g:

```
function b(a, b, c) {
    return "whatever" // do something here.
}

function a() {
    var bOut = b("one", "two", "three");
    return "something "+bOut;
}
```

If component **b** then changes in some backwards-incompatible way, say:

```
function b(a, b, c, d /* new parameter */) {
    return "whatever" // do something here.
}
```

Then, we can say that the protocol has changed. This problem is so common, so endemic to computing that we've had compilers that check function arguments since the 1960's²⁷. The point being is that it's totally possible for

²⁵https://en.wikipedia.org/wiki/Memory_leak

²⁶[https://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

²⁷<https://en.wikipedia.org/wiki/Compiler>

the compiler to warn you about when a protocol within the program has changed.

The same is basically true of Data Types²⁸: whenever we change the **Data Type**, we need to correct the usages of that type. Note above, I've given the JavaScript example, but I'm going to switch to TypeScript now:

```
interface BInput {  
    a: string,  
    b: string,  
    c: string,  
    d: string  
}  
  
function b(in: BInput): string {  
    return "whatever" // do something here.  
}
```

Now, of course, there is a tradeoff: we *mitigate* Complexity Risk, because we define the protocols / types *once only* in the program, and ensure that usages all match the specification. But the tradeoff is (as we can see in the TypeScript code) more *finger-typing*, which some people argue counts as Schedule Risk.

Nevertheless, compilers and type-checking are so prevalent in software that clearly, you have to accept that in most cases, the trade-off has been worth it: Even languages like Clojure²⁹ have been retro-fitted with type checkers³⁰.

We're going to head into much more detail on this in the chapter on Protocol Risk.

Space and Time Complexity

So far, we've looked at a couple of definitions of complexity in terms of the codebase itself. However, in Computer Science there is a whole branch of complexity theory devoted to how the software *runs*, namely Big O Complexity³¹.

Once running, an algorithm or data structure will consume space or run-time dependent on its characteristics. As with Garbage Collectors), these

²⁸https://en.wikipedia.org/wiki/Data_type

²⁹<https://clojure.org>

³⁰<https://github.com/clojure/core.typed/wiki/User-Guide>

³¹https://en.wikipedia.org/wiki/Big_O_notation

characteristics can introduce Performance Risk which can easily catch out the unwary. By and large, using off-the-shelf data structures and algorithms helps, but you still need to know their performance characteristics.

The Big O Cheat Sheet³² is a wonderful resource to investigate this further.

Concurrency / Mutability

Although modern languages include plenty of concurrency primitives, (such as the `java.util.concurrent`³³ libraries), concurrency is *still* hard to get right.

Race conditions³⁴ and Deadlocks³⁵ *thrive* in over-complicated concurrency designs: complexity issues are magnified by concurrency concerns, and are also hard to test and debug.

Recently, languages such as Clojure have introduced persistent collections³⁶ to alleviate concurrency issues. The basic premise is that any time you want to *change* the contents of a collection, you get given back a *new collection*. So, any collection instance is immutable once created. The tradeoff is again attendant Performance Risk to mitigate Complexity Risk.

An important lesson here is that choice of language can reduce complexity: and we'll come back to this in Software Dependency Risk.

Networking / Security

There are plenty of Complexity Risk perils in *anything* to do with networked code, chief amongst them being error handling and (again) protocol evolution.

In the case of security considerations, exploits *thrive* on the complexity of your code, and the weaknesses that occur because of it. In particular, Schneier's Law says, never implement your own cryptographic scheme:

“Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis.”

Bruce Schneier, 1998³⁷

³²<http://bigocheatsheet.com>

³³<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html>

³⁴https://en.wikipedia.org/wiki/Race_condition

³⁵<https://en.wikipedia.org/wiki/Deadlock>

³⁶https://en.wikipedia.org/wiki/Persistent_data_structure

³⁷https://en.wikipedia.org/wiki/Bruce_Schneier#Cryptography

Luckily, most good languages include cryptographic libraries that you can include to mitigate these Complexity Risks from your own code-base.

This is a strong argument for the use of libraries. But, when should you use a library and when should you implement yourself? This is again covered in the chapter on Software Dependency Risk.

Big Ball Of Mud

Although Type-Systems help for avoiding Communication Risk in the small, when software systems grow large it becomes hard to communicate their intent and keep their degree of connectivity low.

“A big ball of mud is a software system that lacks a perceivable architecture. Although undesirable from a software engineering point of view, such systems are common in practice due to business pressures, developer turnover and code entropy.”

—Big Ball Of Mud, *Wikipedia*³⁸

costs associated with complexity risk

CHANGE is also more risky why?

https://en.wikipedia.org/wiki/Big_ball_of_mud

³⁸https://en.wikipedia.org/wiki/Big_ball_of_mud

Communication Risk

Communication Risk is the risk of communication between entities *going wrong*, due to loss or misunderstanding. Consider this: if we all had identical knowledge, there would be no need to do any communicating at all, and therefore and also no Communication Risk.

But, people are not all-knowing oracles. We rely on our *senses* to improve our Internal Models of the world. There is Communication Risk here - we might overlook something vital (like an on-coming truck) or mistake something someone says (like "Don't cut the green wire").

Communication Risk isn't just for people; it affects computer systems too.

12.1 A Model Of Communication

In 1948, Claude Shannon proposed this definition of communication:

"The fundamental problem of communication is that of reproducing at one point, either exactly or approximately, a message selected at another point."

—A Mathematical Theory Of Communication, *Claude Shannon*¹ <!– tweet-end –>

And from this same paper, we get the following (slightly adapted) model.

We move from top-left ("I want to send a message to someone") to bottom left, clockwise, where we hope the message has been understood and believed. (I've added this last box to Shannon's original diagram.)

¹https://en.wikipedia.org/wiki/A_Mathematical_Theory_of_Communication

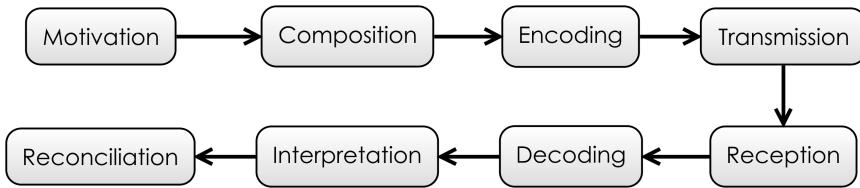


Figure 12.1: Shannon's Communication Model

One of the chief concerns in Shannon's paper is the step between **Transmission** and **Reception**. He creates a theory of information (measured in **bits**), the upper-bounds of information that can be communicated over a channel and ways in which Communication Risk between these processes can be mitigated by clever **Encoding** and **Decoding** steps.

But it's not just transmission. Communication Risk exists at each of these steps. Let's imagine a short exchange where someone, **Alice** is trying to send a message to **Bob**:

- **Alice** might be **motivated** to send a message to tell **Bob** something, only to find out that *he already knew it*, or it wasn't useful information for them.
- In the **composition** stage, **Alice** might mess up the *intent* of the message: instead of "Please buy chips" she might say, "Please buy chops".
- In the **encoding** stage, **Alice** might not speak clearly enough to be understood, and...
- In the **transmission** stage, **Alice** might not say it loudly enough for **Bob** to...
- **receive** the message clearly (maybe there is background noise).
- Having heard **Alice** say something, can **Bob** **decode** what was said into a meaningful sentence?
- Then, assuming that, will they **interpret** correctly which type of chips (or chops) **Alice** was talking about? Does "Please buy chips" convey all the information they need?
- Finally, assuming *everything else*, will **Bob** believe the message? Will they **reconcile** the information into their Internal Model and act on it? Perhaps not, if **Bob** thinks that there are chips at home already.

12.2 Approach To Communication Risk

There is a symmetry about the steps going on in Shannon's diagram, and we're going to exploit this in order to break down Communication Risk into its main types.

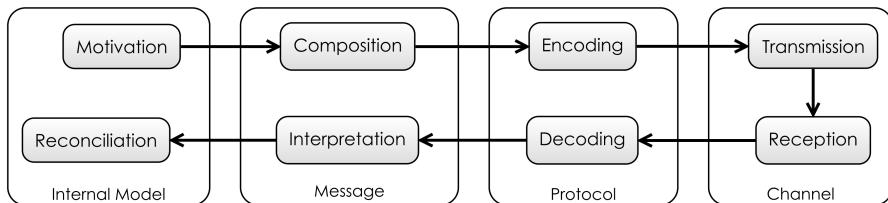


Figure 12.2: Communication Risk 2

To get inside Communication Risk, we need to understand **Communication** itself, whether between *machines, people or products*: we'll look at each in turn. In order to do that, we're going to examine four basic concepts in each of these settings:

- Channels², the medium via which the communication is happening.
- Protocols³ - the systems of rules that allow two or more entities of a communications system to transmit information.
- Messages⁴: The information we want to convey.
- Internal Models: the sources and destinations for the messages. Updating internal models (whether in our heads or machines) is the reason why we're communicating.

And, as we look at these four areas, we'll consider the Attendant Risks of each.

12.3 Channels

There are lots of different types of media for communicating (e.g. TV, Radio, DVD, Talking, Posters, Books, Phones, The Internet, etc.) and they all have different characteristics. When we communicate via a given medium, it's called a *channel*.

²https://en.wikipedia.org/wiki/Communication_channel

³https://en.wikipedia.org/wiki/Communication_protocol

⁴<https://en.wikipedia.org/wiki/Message>

The channel *characteristics* depend on the medium, then. Some obvious ones are cost, utilisation, number of people reached, simplex or duplex (parties can transmit and receive at the same time), persistence (a play vs a book, say), latency (how long messages take to arrive) and bandwidth (the amount of information that can be transmitted in a period of time).

Channel characteristics are important: in a high-bandwidth, low-latency situation, **Alice** and **Bob** can *check* with each other that the meaning was transferred correctly. They can discuss what to buy, they can agree that **Alice** wasn't lying or playing a joke.

The channel characteristics also imply suitability for certain *kinds* of messages. A documentary might be a great way of explaining some economic concept, whereas an opera might not be.

12.4 Channel Risk

Shannon discusses that no channel is perfect: there is always the **risk of noise** corrupting the signal. A key outcome from Shannon's paper is that there is a tradeoff: within the capacity of the channel (the **Bandwidth**), you can either send lots of information with *higher* risk that it is wrong, or less information with *lower* risk of errors. And, rather like the Kolgomorov complexity result, the more *randomness* in the signal, the less compressible it is, and therefore the more *bits* it will take to transmit.



- Risks due to the inadequacy of the physical channel used to communicate our messages. e.g. noise, loss, interception, corruption.

Figure 12.3: Communication Channel Risk

But channel risk goes wider than just this mathematical example: messages might be delayed or delivered in the wrong order, or not be acknowledged when they do arrive. Sometimes, a channel is just an inappropriate way of communicating. When you work in a different time-zone to someone else on your team, there is *automatic* Channel Risk, because instantaneous communication is only available for a few hours' a day.

When channels are **poor-quality**, less communication occurs. People will try to communicate just the most important information. But, it's often impossible to know a-priori what constitutes "important". This is why Extreme Programming recommends the practice of Pair Programming⁵ and siting all the developers together: although you don't know whether useful communication will happen, you are mitigating Channel Risk by ensuring high-quality communication channels are in place.

At other times, channels can contain so much information that we can't hope to receive all the messages. In these cases, we don't even observe the whole channel, just parts of it. For example, you might have a few YouTube channels that you subscribe to, but hundreds of hours of video are being posted on YouTube every second, so there is no way you can keep up with all of it.

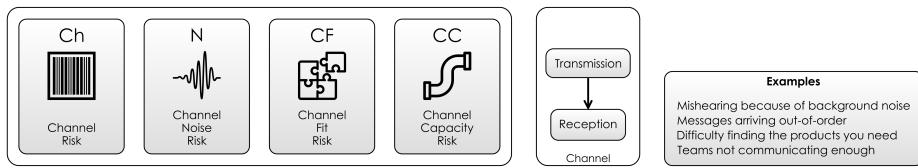


Figure 12.4: Communication Channels

Marketing Communications

When we are talking about a product or a brand, mitigating Channel Risk is the domain of Marketing Communications⁶. How do you ensure that the information about your (useful) project makes it to the right people? How do you address the right channels?

This works both ways. Let's look at some of the **Channel Risks** from the point of view of a hypothetical software tool, **D**, which would really useful in my software:

- The concept that there is such a thing as **D** which solves my problem isn't something I'd even considered.
- I'd like to use something like **D**, but how do I find it?
- There are multiple implementations of **D**, which is the best one for the task?
- I know **D**, but I can't figure out how to solve my problem in it.

⁵https://en.wikipedia.org/wiki/Pair_programming

⁶https://en.wikipedia.org/wiki/Marketing_communications

- I've chosen **D**, I now need to persuade my team that **D** is the correct solution...
- ... and then they also need to understand **D** to do their job too.

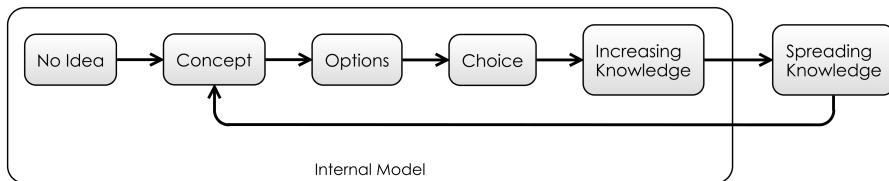


Figure 12.5: Communication Marketing

Internal Models don't magically get populated with the information they need: they fill up gradually, as shown in this diagram. Popular products and ideas *spread*, by word-of-mouth or other means. Part of the job of being a good technologist is to keep track of new **Ideas**, **Concepts** and **Options**, so as to use them as Dependencies when needed.

12.5 Protocols

In this chapter, I want to examine the concept of Communication Protocols and how they relate to Abstraction.

So, to do this, let's look in a bit of detail at how web pages are loaded. When considering this, we need to broaden our terminology. Although so far we've talked about **Senders** and **Receivers**, we now need to talk from the point of view of who-depends-on-who. If you're *depended on*, then you're a "Server", whereas if you require communication with something else, you're a "Client". Thus, clients depend on servers in order to load pages.

This is going to involve (at least) six separate protocols, the top-most one being the HTTP Protocol⁷. As far as the HTTP Protocol is concerned, a *client* makes an **HTTP Request** at a specific URL and the **HTTP Response** is returned in a predictable format that the browser can understand.

Let's have a quick look at how that works with a `curl` command, which allows me to load a web page from the command line. We're going to try and load Google's preferences page, and see what happens. If I type:

⁷https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

```
> curl -v http://google.com/preferences
```

1. DNS - Domain Name System

Then, the first thing that happens is this:

```
* Rebuilt URL to: http://google.com/
* Trying 216.58.204.78...
```

At this point, curl has used DNS⁸ to *resolve* the address “google.com” to an IP address. This is some Abstraction: instead of using the machine’s IP Address⁹ on the network, 216.58.204.78, I can use a human-readable address, google.com. The address google.com doesn’t necessarily resolve to that same address each time: *They have multiple IP addresses for google.com*. But, for the rest of the curl request, I’m now set to just use this one.

2. IP - Internet Protocol

But this hints at what is beneath the abstraction: although I’m loading a web-page, the communication to the Google server happens by IP Protocol¹⁰ - it’s a bunch of discrete “packets” (streams of binary digits). You can think of a packet as being like a real-world parcel or letter.

Each packet consists of two things:

- An address, which tells the network components (such as routers and gateways) where to send the packet, much like you’d write the address on the outside of a parcel.
- The *payload*, the stream of bytes for processing at the destination. Like the contents of the parcel.

But, even this concept of “packets” is an Abstraction. Although all the components of the network interoperate with this protocol, we might be using Wired Ethernet, or WiFi, 4G or *something else*.

⁸https://en.wikipedia.org/wiki/Domain_Name_System

⁹https://en.wikipedia.org/wiki/IP_address

¹⁰https://en.wikipedia.org/wiki/Internet_Protocol

3. 802.11 - WiFi Protocol

I ran this at home, using WiFi, which uses IEEE 802.11 Protocol¹¹, which allows my laptop to communicate with the router wirelessly, again using an agreed, standard protocol. But even *this* isn't the bottom, because this is actually probably specifying something like MIMO-OFDM¹², giving specifications about frequencies of microwave radiation, antennas, multiplexing, error-correction codes and so on. And WiFi is just the first hop: after the WiFi receiver, there will be protocols for delivering the packets via the telephony system.

4. TCP - Transmission Control Protocol

Anyway, the next thing that happens is this:

```
* TCP_NODELAY set
* Connected to google.com (216.58.204.78) port 80 (#0)
```

The second obvious Abstraction going on here is that curl now believes it has a TCP¹³ connection. The TCP connection abstraction gives us the surety that the packets get delivered in the right order, and retried if they go missing. Effectively it *guarantees* these things, or that it will have a connection failure if it can't keep it's guarantee.

But, this is a fiction - TCP is built on the IP protocol, packets of data on the network. So there are lots of packets floating around which say "this connection is still alive" and "I'm message 5 in the sequence" and so on in order to maintain this fiction. But that means that the HTTP protocol can forget about this complexity and work with the fiction of a connection.

5. HTTP - Hypertext Transfer Protocol

Next, we see this:

```
> GET /preferences HTTP/1.1      (1)
> Host: google.com              (2)
> User-Agent: curl/7.54.0       (3)
> Accept: */*                   (4)
>                                (5)
```

¹¹https://en.wikipedia.org/wiki/IEEE_802.11

¹²<https://en.wikipedia.org/wiki/MIMO-OFDM>

¹³https://en.wikipedia.org/wiki/Transmission_Control_Protocol

This is now the HTTP protocol proper, and these 5 lines are sending information *over the connection* to the Google server.

- (1) says what version of HTTP we are using, and the path we're loading (/preferences in this case).
- (2) to (4) are *headers*. They are name-value pairs, separated with a colon. The HTTP protocol specifies a bunch of these names, and later versions of the protocol might introduce newer ones.
- (5) is an empty line, which indicates that we're done with the headers, please give us the response. And it does:

```
< HTTP/1.1 301 Moved Permanently
< Location: http://www.google.com/preferences
< Content-Type: text/html; charset=UTF-8
< Date: Sun, 08 Apr 2018 10:24:34 GMT
< Expires: Tue, 08 May 2018 10:24:34 GMT
< Cache-Control: public, max-age=2592000
< Server: gws
< Content-Length: 230
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
<
<HTML><HEAD><meta http-equiv="content-type"
content="text/html; charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
</BODY></HTML>
* Connection #0 to host google.com left intact
```

There's a lot going on here, but we can break it down really easily into 3 chunks:

1. The first line is the HTTP Status Code¹⁴. 301 is a code meaning that the page has moved.
2. The next 9 lines are HTTP headers again (name-value pairs). The Location: directive tells us where the page has moved to. Instead of trying `http://google.com/preferences`, we should have used:

¹⁴https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

<http://www.google.com/preferences>.

3. The lines starting <HTML> are now some HTML to display on the screen to tell the user that the page has moved.

6. HTML - Hypertext Mark-Up Language

Although HTML¹⁵ is a language, a language is also a protocol. (After all, language is what we use to encode our ideas for transmission as speech.) In the example we gave, this was a very simple page telling the client that it's looking in the wrong place. In most browsers, you don't get to see this: the browser will understand the meaning of the 301 error and redirect you to the location.

Let's look at all the protocols we saw here:

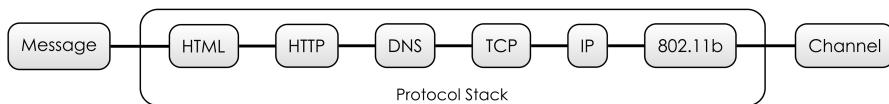


Figure 12.6: Protocol Stack

Each protocol "passes on" to the next one in the chain. On the left, we have the representation most suitable for the *messages*: HTTP is designed for browsers to use to ask for and receive web pages. As we move right, we are converting the message more and more into a form suitable for the Channel: in this case, microwave transmission.

By having a stack of protocols, we are able to apply Separation Of Concerns¹⁶, each protocol handling just a few concerns:

- HTML Abstraction: A language for describing the contents of a web-page.
- HTTP Abstraction: Name-Value pairs, agreed on by both curl and Google, URLs and error codes.
- DNS Abstraction: Names of servers to IP Addresses.
- TCP Abstraction: The concept of a "connection" with guarantees about ordering and delivery.
- IP Abstraction: "Packets" with addresses and payloads.
- WiFi Abstraction: "Networks", 802.11 flavours.

¹⁵<https://en.wikipedia.org/wiki/HTML>

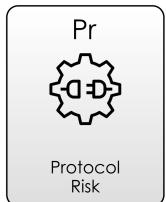
¹⁶https://en.wikipedia.org/wiki/Separation_of_concerns

- Transmitters, Antennas, error correction codes, etc.

HTTP “stands on the shoulders of giants”. Not only does it get to use pre-existing protocols like TCP and DNS to make it’s life easier, it got 802.11 “for free” when this came along and plugged into the existing IP protocol. This is the key value of abstraction: you get to piggy-back on *existing* patterns, and use them yourself.

The protocol mediates between the message and the channel. Where this goes wrong, we have Protocol Risk. This is a really common issue for IT systems, but also sometimes for human communication too.

12.6 Protocol Risk



- Risks due to the failure of encoding or decoding messages between two parties in communication.

Figure 12.7: Protocol Risk

Generally, any time where you have different parts of a system communicating with each other, and one part can change incompatibly with another you have Protocol Risk.

Locally, (within our own project), where we have control, we can mitigate this risk using compile-time checking (as discussed already in Complexity Risk), which essentially forces all clients and servers to agree on protocol. But, the wider the group that you are communicating with, the less control you have and the more chance there is of Protocol Risk.

Let’s look at some types of Protocol Risk:

Protocol Incompatibility Risk

The people you find it *easiest* to communicate with are your friends and family, those closest to you. That’s because you’re all familiar with the same protocols. Someone from a foreign country, speaking a different language

and having a different culture, will essentially have a completely incompatible protocol for spoken communication to you.

Within software, there are also competing, incompatible protocols for the same things, which is maddening when your protocol isn't supported. Although the world seems to be standardizing, there used to be *hundreds* of different image formats. Photographs often use TIFF¹⁷, RAW¹⁸ or JPEG¹⁹, whilst we also have SVG²⁰ for vector graphics, GIF²¹ for images and animations and PNG²² for other bitmap graphics.

Protocol Versioning Risk

Even when systems are talking the same protocol, there can be problems. When we have multiple, different systems owned by different parties, on their own upgrade cycles, we have **Protocol Versioning Risk**: the risk that either client or server could start talking in a version of the protocol that the other side hasn't learnt yet. There are various mitigating strategies for this. We'll look at two now: **Backwards Compatibility** and **Forwards Compatibility**.

Protocol Complexity

tbd. abstraction - virtue between two vices. Postel's law.

Backward Compatibility

Backwards Compatibility mitigates Protocol Versioning Risk. Quite simply, this means, supporting the old format until it falls out of use. If a server is pushing for a change in protocol it either must ensure that it is Backwards Compatible with the clients it is communicating with, or make sure they are upgraded concurrently. When building web services²³, for example, it's common practice to version all APIs so that you can manage the migration. Something like this:

- Server publishes /api/v1/something.
- Clients use /api/v1/something.
- Server publishes /api/v2/something.
- Clients start using /api/v2/something.

¹⁷<https://en.wikipedia.org/wiki/TIFF>

¹⁸https://en.wikipedia.org/wiki/Raw_image_format

¹⁹<https://en.wikipedia.org/wiki/JPEG>

²⁰https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

²¹<https://en.wikipedia.org/wiki/GIF>

²²https://en.wikipedia.org/wiki/Portable_Network_Graphics

²³https://en.wikipedia.org/wiki/Web_service

- Clients (eventually) stop using /api/v2/something.
- Server retires /api/v2/something API.

Forward Compatibility

HTML and HTTP provide “graceful failure” to mitigate Protocol Risk: while it’s expected that all clients can parse the syntax of HTML and HTTP, it’s not necessary for them to be able to handle all of the tags, attributes and rules they see. The specification for both these standards is that if you don’t understand something, ignore it. Designing with this in mind means that old clients can always at least cope with new features, but it’s not always possible.

JavaScript *can’t* support this: because the meaning of the next instruction will often depend on the result of the previous one.

Does human language support this? To some extent! New words are added to our languages all the time. When we come across a new word, we can either ignore it, guess the meaning, ask or look it up. In this way, human language has **Forward Compatibility** features built in.

Protocol Implementation Risk

A second aspect of Protocol Risk exists in heterogeneous computing environments, where protocols have been independently implemented based on standards. For example, there are now so many different browsers, all supporting different levels of HTTP, HTML and JavaScript that it becomes impossible to test comprehensively over all the different versions. To mitigate as much Protocol Risk as possible, generally we run tests in a subset of browsers, and use a lowest-common-denominator approach to choosing protocol and language features.

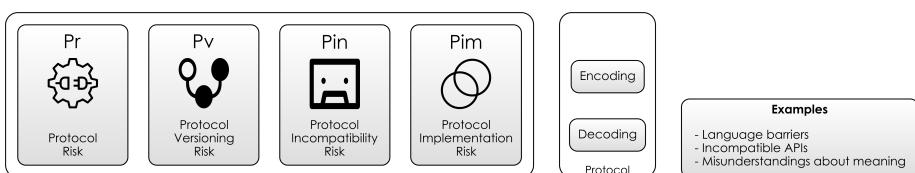


Figure 12.8: Communication Protocols Risks



- Risks caused by the difficulty of composing and interpreting messages in the communication process.

Figure 12.9: Message Risk

12.7 Messages

Although Shannon's Communication Theory is about transmitting **Messages**, messages are really encoded **Ideas** and **Concepts**, from an **Internal Model**.

Internal Model Assumption Risk

When we construct messages in a conversation, we have to make judgements about what the other person already knows. When talking to children, it's often hard work because they *assume* that you have knowledge of everything they do. This is called Theory Of Mind²⁴: the appreciation that your knowledge is different to other people's, and adjusting your messages accordingly.

When teaching, this is called The Curse Of Knowledge²⁵: teachers have difficulty understanding students' problems *because they already understand the subject*. For example, if I want to tell you about a new JDBC Driver²⁶, this pre-assumes that you know what JDBC is: the message has a dependency on prior knowledge.

Message Dependency Risk

A second, related problem is actually Dependency Risk, which is covered more thoroughly in the next chapter. Often, messages assume that you have followed everything up to that point already, otherwise again, your Internal Model will not be rich enough to understand the new messages.

This happens when messages get missed, or delivered out of order. In the past, TV shows were only aired once a week at a particular time. So writers

²⁴https://en.wikipedia.org/wiki/Theory_of_mind

²⁵https://en.wikipedia.org/wiki/Curse_of_knowledge

²⁶https://en.wikipedia.org/wiki/JDBC_driver

were constrained plot-wise by not knowing whether their audience would have seen the previous week's episode. Therefore, often the state of the show would "reset" week-to-week, allowing you to watch it in *any* order.

The same **Message Dependency Risk** exists for computer software: if there is replication going on between instances of an application, and one of the instances misses some messages, you end up with a "Split Brain²⁷" scenario, where later messages can't be processed because they refer to an application state that doesn't exist. For example, a message saying:

```
Update user 53's surname to 'Jones'
```

only makes sense if the application has previously had the message

```
Create user 53 with surname 'Smith'
```

Misinterpretation Risk

People don't rely on rigorous implementations of abstractions like computers do; we make do with fuzzy definitions of concepts and ideas. We rely on Abstraction to move between the name of a thing and the *idea of a thing*.

While machines only process *information*, people's brains run on concepts and ideas. For people, abstraction is critical: nothing exists unless we have a name for it. Our world is just atoms, but we don't think like this. *The name is the thing*.

"The famous pipe. How people reproached me for it! And yet, could you stuff my pipe? No, it's just a representation, is it not? So if I had written on my picture "This is a pipe", I'd have been lying!"

—Rene Magritte, of *The Treachery of Images*²⁸

This brings about Misinterpretation Risk: names are not *precise*, and concepts mean different things to different people. We can't be sure that people have the same meaning for concepts that we have.

²⁷[https://en.wikipedia.org/wiki/Split-brain_\(computing\)](https://en.wikipedia.org/wiki/Split-brain_(computing))

²⁸https://en.wikipedia.org/wiki/The_Treachery_of_Images

Invisibility Risk

Another cost of Abstraction is Invisibility Risk. While abstraction is a massively powerful technique, (as we saw above in the chapter on Protocols, it allows things like the Internet to happen) it lets the function of a thing hide behind the layers of abstraction and become invisible.

Invisibility Risk In Software

As soon as you create a function, you are doing abstraction. You are saying: "I now have this operation. The details, I won't mention again, but from now on, it's called *f*" And suddenly, "*f*" hides. It is working invisibly. Things go on in *f* that people don't necessarily need to understand. There may be some documentation, or tacit knowledge around what *f* is, and what it does, but it's not necessarily right. Referring to *f* is a much simpler job than understanding *f*.

We try to mitigate this via (for the most part) documentation, but this is a terrible deal: because we can't understand the original, (un-abstracted) implementation, we now need to write some simpler documentation, which *explains* the abstraction, in terms of further abstractions, and this is where things start to get murky.

Invisibility Risk is mainly Hidden Risk. (Mostly, *you don't know what you don't know.*) But you can carelessly *hide things from yourself* with software:

- Adding a thread to an application that doesn't report whether it's worked, failed, or is running out of control and consuming all the cycles of the CPU.
- Redundancy can increase reliability, but only if you know when servers fail, and fix them quickly. Otherwise, you only see problems when the last server fails.
- When building a web-service, can you assume that it's working for the users in the way you want it to?

When you build a software service, or even implement a thread, ask yourself: "How will I know next week that this is working properly?" If the answer involves manual work and investigation, then your implementation has just cost you in Invisibility Risk.

Invisibility Risk In Conversation

Invisibility Risk is risk due to information not sent. But because humans don't need a complete understanding of a concept to use it, we can cope with

some Invisibility Risk in communication, and this saves us time when we're talking. It would be *painful* to have conversations if, say, the other person needed to understand everything about how cars worked in order to discuss cars.

For people, Abstraction is a tool that we can use to refer to other concepts, without necessarily knowing how the concepts work. This divorcing of "what" from "how" is the essence of abstraction and is what makes language useful.

The debt of Invisibility Risk comes due when you realise that *not* being given the details *prevents* you from reasoning about it effectively. Let's think about this in the context of a project status meeting, for example:

- Can you be sure that the status update contains all the details you need to know?
- Is the person giving the update wrong or lying?
- Do you know enough about the details of what's being discussed in order to make informed decisions about how the project is going?

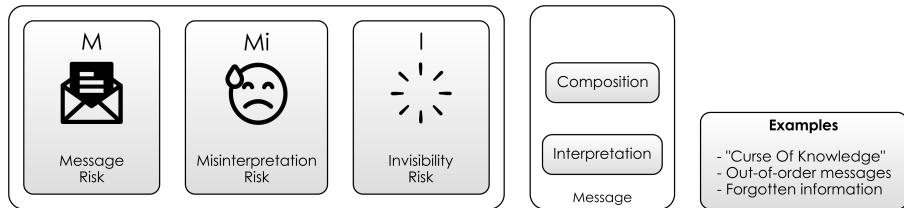


Figure 12.10: Message Risk

12.8 Internal Models

So finally, we are coming to the root of the problem: communication is about transferring ideas and concepts from one Internal Model to another.

The communication process so far has been fraught with risks, but we have a few more to come.

Trust & Belief Risk

Although protocols can sometimes handle security features of communication (such as Authentication²⁹ and preventing man-in-the-middle attacks³⁰), trust goes further than this, it is the flip-side of Agency Risk, which we will look at later: can you be sure that the other party in the communication is acting in your best interests?

Even if the receiver trusts the communicator, they may not believe the message. Let's look at some reasons for that:

- Weltanschauung (World View)³¹: The ethics, values and beliefs in the receiver's Internal Model may be incompatible to those from the sender.
- Relativism³² is the concept that there are no universal truths. Every truth is from a frame of reference. For example, what constitutes *offensive language* is dependent on the listener.
- Psycholinguistics³³ is the study of humans acquire languages. There are different languages and dialects, (and *industry dialects*), and we all understand language in different ways, take different meanings and apply different contexts to the messages.

From the point-of-view of Marketing Communications choosing the right message is part of the battle. You are trying to communicate your idea in such a way as to mitigate Trust & Belief Risk.

Learning-Curve Risk

If the messages we are receiving force us to update our Internal Model too much, we can suffer from the problem of "too steep a Learning Curve³⁴" or "Information Overload³⁵", where the messages force us to adapt our Internal Model too quickly for our brains to keep up.

Commonly, the easiest option is just to ignore the information channel completely in these cases.

Reading Code

It's often been said that code is *harder to read than to write*:

²⁹<https://en.wikipedia.org/wiki/Authentication>

³⁰https://en.wikipedia.org/wiki/Man-in-the-middle_attack

³¹https://en.wikipedia.org/wiki/World_view

³²<https://en.wikipedia.org/wiki/Relativism>

³³<https://en.wikipedia.org/wiki/Psycholinguistics>

³⁴https://en.wikipedia.org/wiki/Learning_curve

³⁵https://en.wikipedia.org/wiki/Information_overload

"If you ask a software developer what they spend their time doing, they'll tell you that they spend most of their time writing code. However, if you actually observe what software developers spend their time doing, you'll find that they spend most of their time trying to understand code." -

When Understanding Means Rewriting, *Coding Horror*³⁶

By now it should be clear that it's going to be *both* quite hard to read and write: the protocol of code is actually designed for the purpose of machines communicating, not primarily for people to understand. Making code human readable is a secondary concern to making it machine readable.

But now we should be able to see the reasons it's harder to read than write too:

- When reading code, you are having to shift your Internal Model to wherever the code is, accepting decisions that you might not agree with and accepting counter-intuitive logical leaps. i.e. Learning Curve Risk. (*cf. Principle of Least Surprise*³⁷)
- There is no Feedback Loop between your Internal Model and the Reality of the code, opening you up to Misinterpretation Risk. When you write code, your compiler and tests give you this.
- While reading code *takes less time* than writing it, this also means the Learning Curve is steeper.

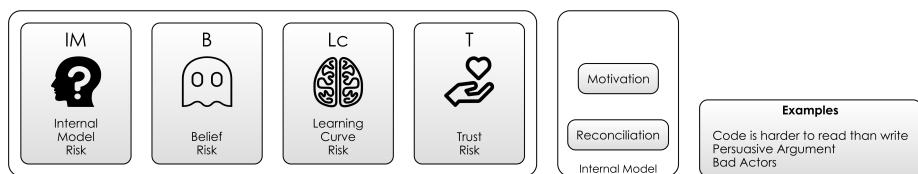


Figure 12.11: Internal Model Risks

³⁶<https://blog.codinghorror.com/when-understanding-means-rewriting/>

³⁷https://en.wikipedia.org/wiki/Principle_of_least_astonishment

12.9 Communication Risk Wrap Up

So, here's a summary of where we've arrived with our model of communication risk:

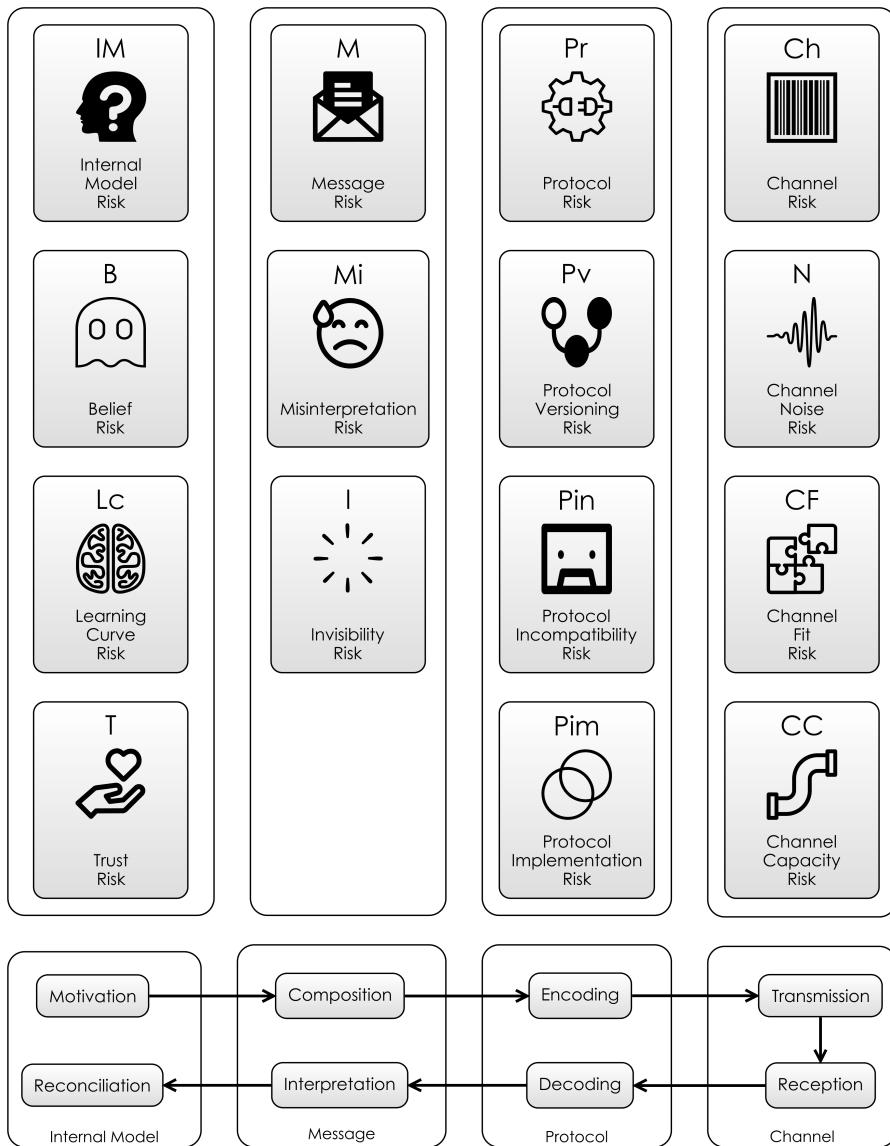


Figure 12.12: Communication 2

tbd, So next it's time to look at Complexity Risk.

this seems complex tbd.

Part III

Application

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.

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.

14.1 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. "Attack at dawn" allows disparate army units to avoid the Coordination Risk inherent in "attack on my signal". This is a *good reason for estimating*, because by using events you are mitigating Coordination 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.

14.2 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, Wikipedia¹.

In their research “Anchoring and Adjustment in Software Estimation”, Aranda and Easterbrook² asked developers split into three groups (A, B and Control) to give individual estimates on how long a piece of software would take to build. They were each given the same specification. However:

- Group A was given the hint: “I admit I have no experience with software, but I guess it will take about two months to finish”.
- Group B were given the same hint, except with 20 months.

How long would members of each group estimate the work to take? The results were startling. On average:

- Group A estimated 5.1 months.
- The Control Group estimated 7.8 months.
- Group B estimated 15.4 months.

The anchor mattered more than experience, how formal the estimation method, or *anything else*. *We can't estimate time at all.*

14.3 Is Risk To Blame?

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).

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

²<http://www.cs.toronto.edu/%7Esme/papers/2005/ESEC-FSE-05-Aranda.pdf>

- 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.

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

14.4 Old Saw No. 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, Techopedia³

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 existing risk, and generating less Attendant Risk.
- It stands to reason then, that someone taking action *faster* will leave us with less Schedule Risk.
- However, if they are *more expensive*, they may leave us with greater Funding Risk afterwards.

³<https://www.techopedia.com/definition/31673/10x-developer>

- 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.

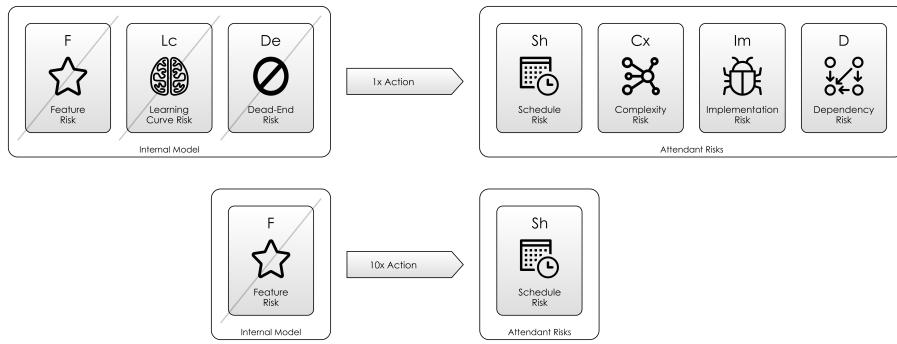


Figure 14.1: 1x Task vs 10X Task

Does the “10X Developer” even exist? 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.

The easiest way to be the “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.

14.5 Old Saw No. 2: Quality, Speed, Cost: Pick Any Two

"The Project Management Triangle (called also the Triple Constraint, Iron Triangle and Project Triangle) is a model of the constraints of project management. While its origins are unclear, it has been used since at least the 1950s. It contends that:

- 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*⁴

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).
- Or, by assuming a lottery win, the project's budget is fine. (Although I would have *huge* Funding Risk because *what are the chances of winning the lottery?*).
- Brooks' Law contradicts this by saying you can't trade budget for deadlines:

⁴https://en.wikipedia.org/wiki/Project_management_triangle

“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, Wikipedia⁵

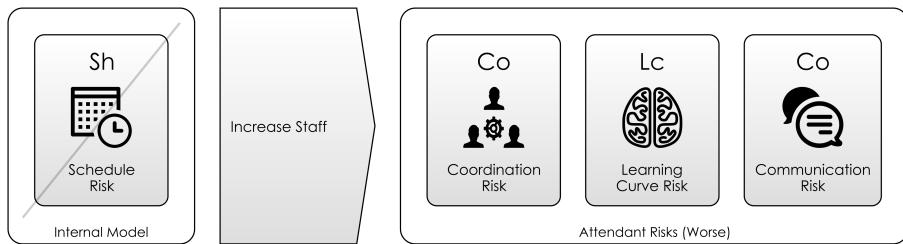


Figure 14.2: Brooks’ Law, Risk-First Style

Focusing on the three variables of the iron triangle 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 its 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.

Old Saw No. 3: Parkinson’s Law

We’ve already looked at Parkinson’s Law in the chapter on Agency Risk, but let’s recap:

“Parkinson’s law is the adage that ‘work expands so as to fill the time available for its completion.’” Parkinson’s Law, Wikipedia⁶

⁵https://en.wikipedia.org/wiki/Brooks'_law

⁶https://en.wikipedia.org/wiki/Parkinson's_law

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 exchange free time to reduce 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 McConnell⁷. 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?" (1)

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:

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

⁷<http://amzn.eu/d/eTWKOsK>

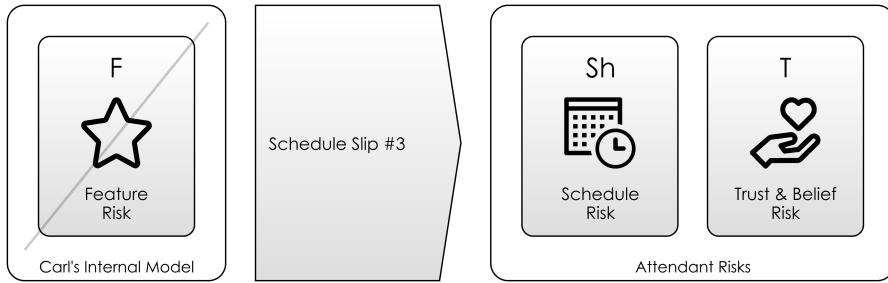


Figure 14.3: Carl's Schedule Slip increases Trust and Belief Risks

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), the schedule has slipped again, and Bill has threatened Carl's job. Why did 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:

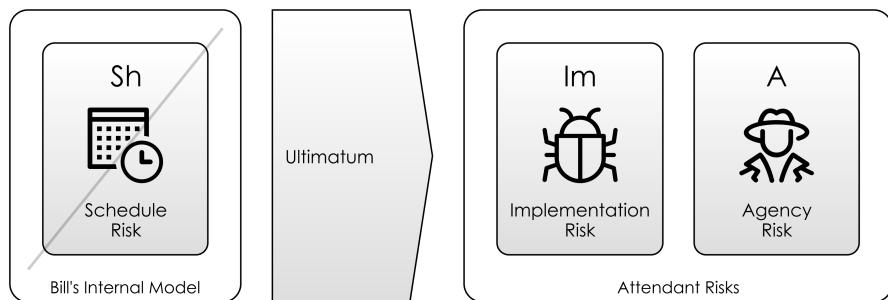


Figure 14.4: Bill's Ultimatum

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, 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.

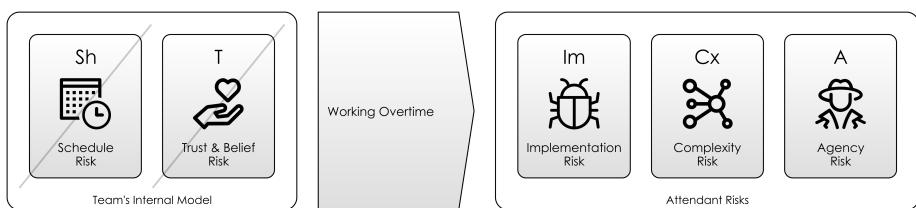


Figure 14.5: Team Response

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?

14.6 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-Hours⁸ or Story Points⁹:

"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

⁸<https://en.wikipedia.org/wiki/Man-hour>

⁹<https://www.atlassian.com/agile/project-management/estimation>

counter-intuitive, but that abstraction is actually helpful because it pushes the team to make tougher decisions around the difficulty of work.”

—Story Points, Atlassian¹⁰

Second, the strategy of picking the two-day action with the greatest Pay-Off is *often good*. (After all, this is just Gradient Descent, and that’s a perfectly good way for training Machine Learning¹¹ 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 ends up being the same.

14.7 Risk-First Estimating

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

- **From the “10X Developer” Saw:** the difference made by experience implies that a lot of the effort on a project comes from Learning Curve Risk and Dead End Risk.
- **From “Quality, Speed, Cost”:** 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.
- **From the Parkinson’s Law:** 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.

¹⁰<https://www.atlassian.com/agile/project-management/estimation>

¹¹https://en.wikipedia.org/wiki/Machine_learning

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:

- 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:

- "Adding this new database is problematic because it's going to massively increase our Dependency Risk."
- "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 Feature Implementation Risk. How about we try and get agreement on a specification?"

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

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.

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.



Figure 14.6: Journey via the Central Line

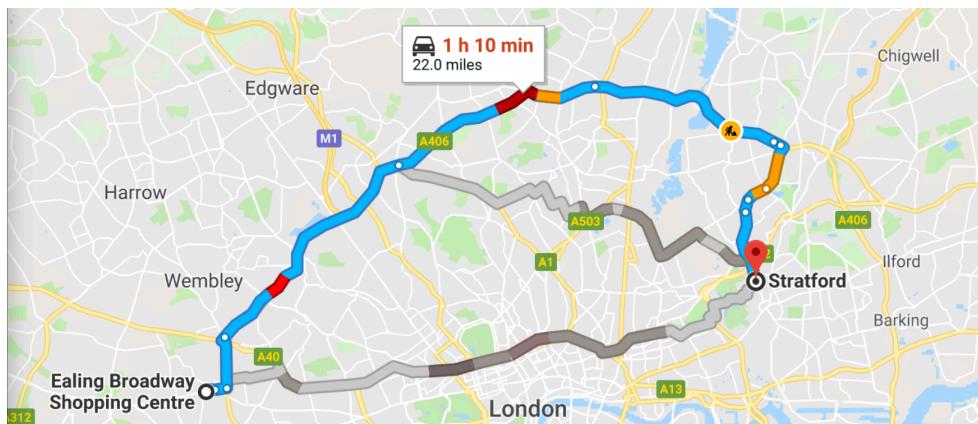


Figure 14.7: Journey by Car

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).

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*.

Because the modern Risk Landscape is so complex:

- There can be orders of magnitude difference in *time*, with very little difference in destination.

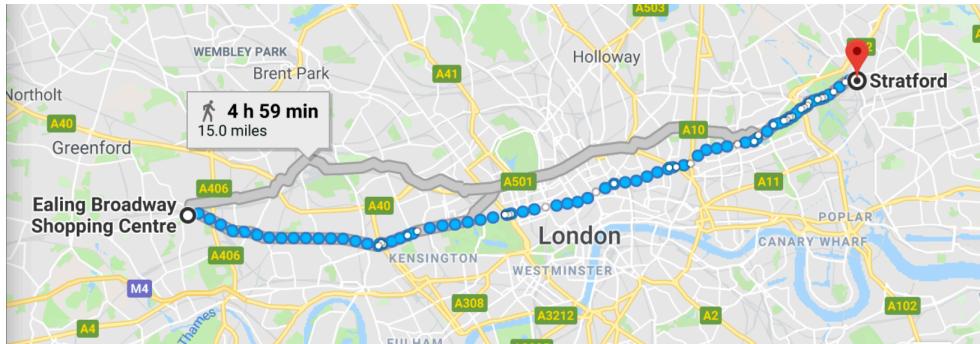


Figure 14.8: Journey on Foot

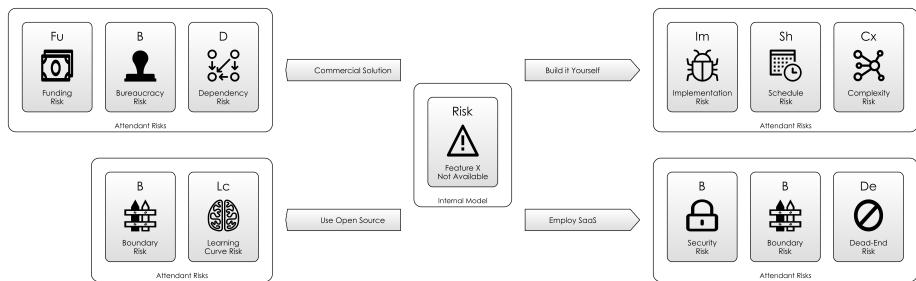


Figure 14.9: Possible Moves On The Risk Landscape

- 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.
 - Test the market with these and meet reality early.
- 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."

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:

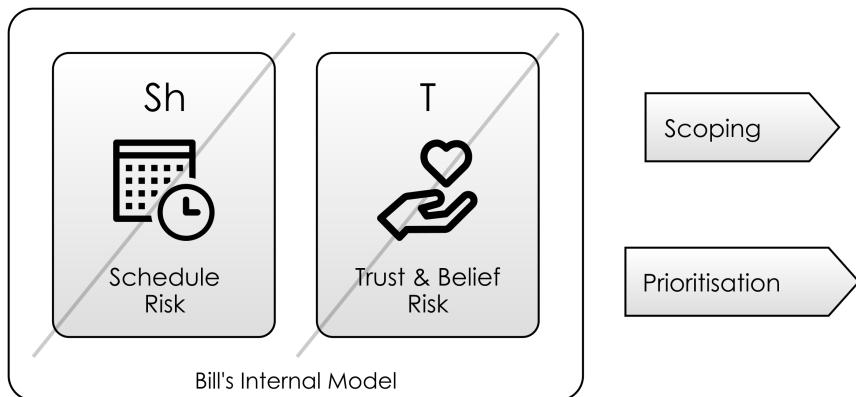


Figure 14.10: Identifying The Action

"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.