Risk-First

Software Development De-Risked

Volume 1: The Menagerie

Rob Moffat

Risk First: The Menagerie

By Rob Moffat

ISBN: 9781717491855

Credits

tbd

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

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

Thanks to:

Books In The Series

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

Online

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

Published By

Kite9 Ltd.
14 Manor Close

Colchester
CO6 4AR

Contents

Preface

Welcome to Risk-First!

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

Why

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

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

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

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

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

This book considers that perspective: that building software is all about *managing risk*, and that these methodologies are acknowledgements of this fact, and they differ because they have *different ideas* about which are the most important *risks to manage*.

What This Is

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

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

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

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

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

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

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

[2] https://tvtropes.org

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

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

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

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

Patterns For Practitioners

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

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

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

Towards a "Periodic Table"

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

What This is Not

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

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

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

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

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

Who

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

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

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

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

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

For Developers

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

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

For Project Managers and Team Leads

All too often, Project Managers don't have a full grasp of the technical details of their projects. And this is perfectly normal, as the specialization belongs below them. However, projects fail because risks materialize, and risks materialize because the devil is in those details.

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

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

How

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

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

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

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

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

Where

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

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

A Note on References

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

On to The Executive Summary

[7] https://github.com

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

[9] https://wikipedia.org

Executive Summary

1. There are Lots of Ways of Running Software Projects

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

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

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

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

2. We can Look at Projects in Terms of Risks

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

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

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

For example:

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

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

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

3. We Can Break Down Risks on a Project Methodically

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

For example:

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

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

With this in place, we can:

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

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

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

For example:

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

Different tools are appropriate for mitigating different types of risks.

5. Different Methodologies for Different Risk Profiles

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

For example:

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

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

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

6. Driving Development With a Risk-First Perspective

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

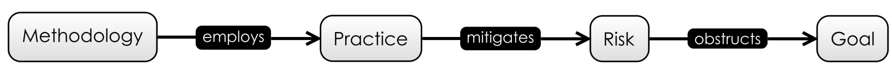


Figure 1: Methdologies, Risks, Practices

How do we take this further?

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

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

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

For example:

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

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

Still interested? Then dive into reading the introduction.

Part I

Introduction

CHAPTER 1

A Simple Scenario

First up, I'm going to introduce a simple model for thinking about risk.

1.1 A Simple Scenario

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.

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.

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.

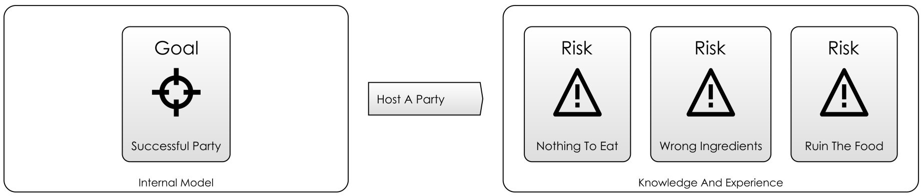


Figure 1.1: Goal In Mind, with the risks you know about

When we go about preparing this wonderful evening, we can with these risks and try to mitigate them: 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 mitigating Attendant Risks depends on how great we think they are: for example, if you know it'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*).

Hidden Risks

There are also hidden Attendant Risks that you might not know about: if you're poaching eggs for dinner, you might know that fresh eggs poach best. These are the "Unknown Unknowns" of Rumsfeld's model[1].

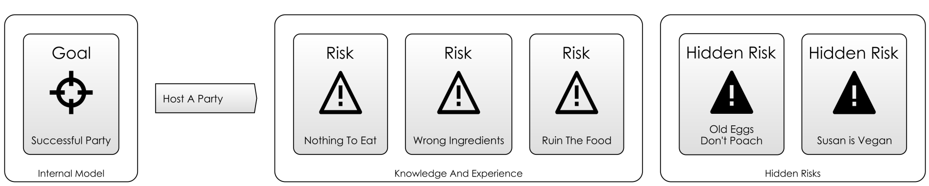


Figure 1.2: Goal In Mind, the risks you know about and the ones you don't

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

Different people will evaluate the risks differently. (That is, worry about them more or less.) They'll also *know* about different risks. They might have cooked the recipe before, or organised lots more dinner parties than you.

How we evaluate the risks, and which ones we know about depends on our **knowledge** and **experience**, then. And that varies from person to person (or team to team). Lets call this our Internal Model, and it's something we build on and improve with experience (of organising dinner parties, amongst everything else).

Model Meets Reality

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

Our model is forced into contact with reality, and the model changes. We may be forced to take further, mitigating actions to deal with these new risks, as shown in this diagram.

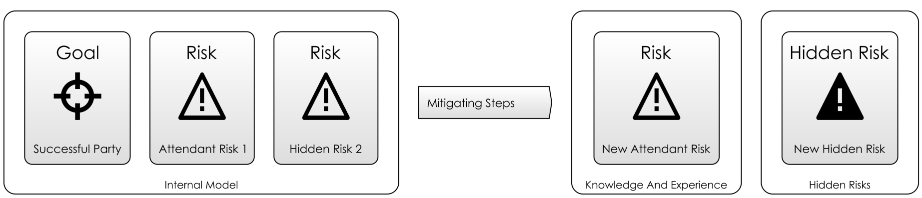


Figure 1.3: How taking action affects Reality, and also changes your internal model

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

1.2 On To Software

In this website, we're going to look at the risks in the software process and how these are mitigated by the various methodologies you can choose from.

Let's examine the scenario of a new software project, and expand on the simple model being outlined above: instead of a single person, we are likely to have a team, and our model will not just exist in our heads, but in the code we write.

On to Development Process

CHAPTER 2

Development Process

In the previous chapter we looked at a simple model for risks on any given activity.

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.

. . . All being well, the code is released to production.

Now, it might be waterfall, it might be agile, we're not going to commit to specifics at this stage. It's probably not perfect, but let's just assume that *it works for this project* and everyone is reasonably happy with it.

I'm not saying this is the *right* process, or even a *good* process: you could add code review, a pilot, integration testing, whatever. We're just doing some analysis of *what process gives us*.

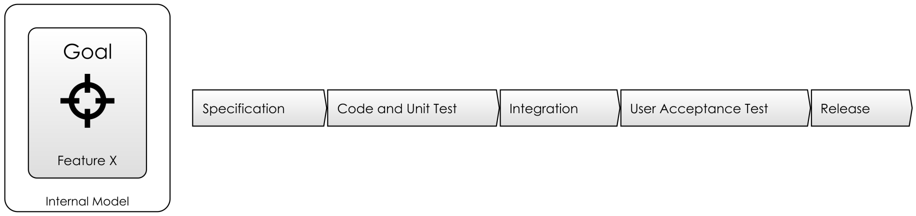


Figure 2.1: A Simple Development Process

What's happening here? Why these steps?

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.

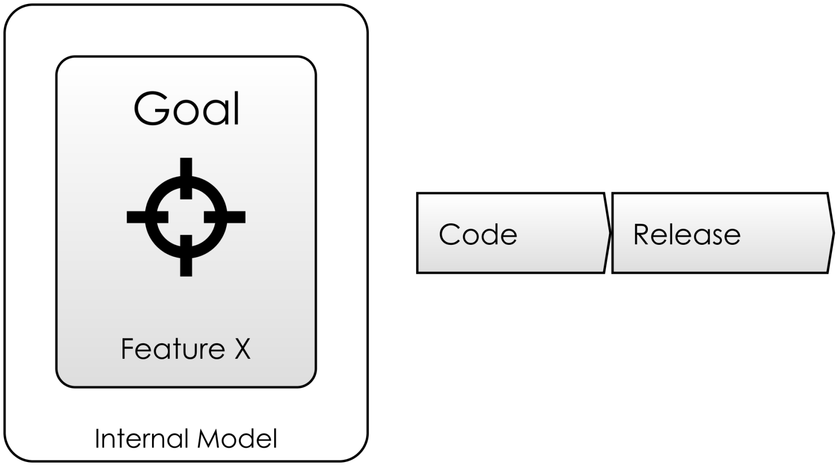


Figure 2.2: A Dangerous Development Process

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 ready to materialize. 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.

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:

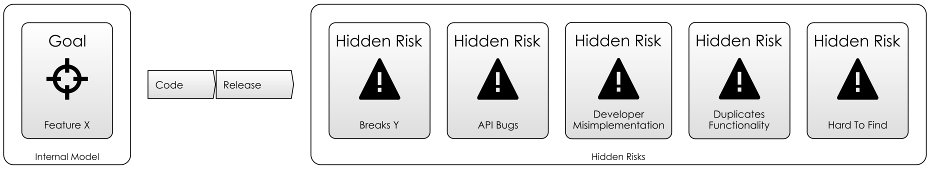


Figure 2.3: Development Process - Hidden Risks

Specification

The first stage of the journey for the feature is that it meets the Business Analyst (BA). The *purpose* of the BA is to examine new goals for the project and try to integrate them with *reality as they understands it*. A good BA might take a feature request and vet it against the internal logic of the project, 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 Risk and make it clear to everyone on the project.

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 BAs 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.

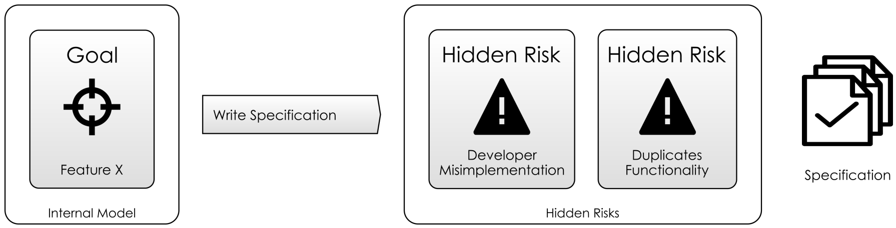


Figure 2.4: BA Specification: exposing hidden risks as soon as possible

Code And Unit Test

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

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

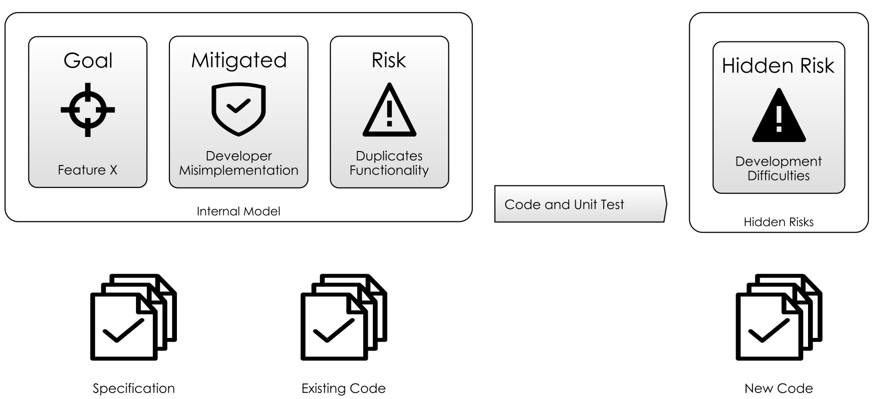


Figure 2.5: Coding Process: exposing more hidden risks as you code

A couple of things about this diagram:

- In boxes, we are showing Risks, which exist within Internal Models, whereas:

- Beneath them, we are showing actual *physical artifacts* which exist in the real world.
- *Actions* might *meet* reality, but they are *changing reality* too, by producing these artifacts.

Integration

Integration is where we run *all* the tests on the project, and compile *all* the code in a clean environment: the "reality" of the development environment can vary from one developer's machine to another.

So, this stage is about the developer's committed code meeting a new reality: the clean build.

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

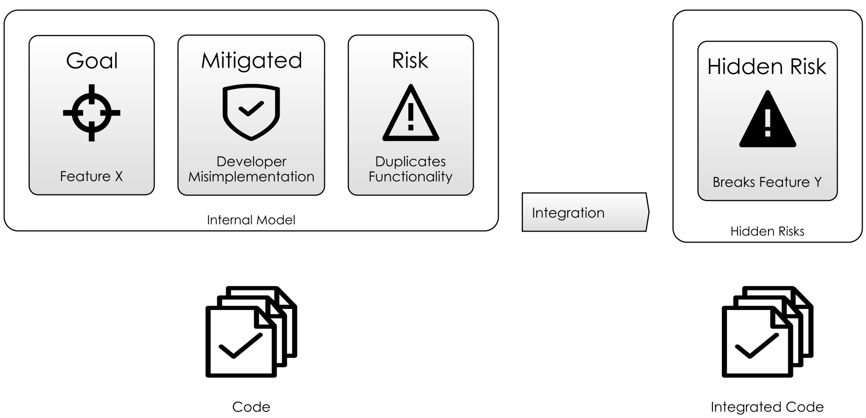


Figure 2.6: Integration testing exposes hidden risks before you get to production

User Acceptance Test

Is where our 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 the user's difficulty in finding the feature.

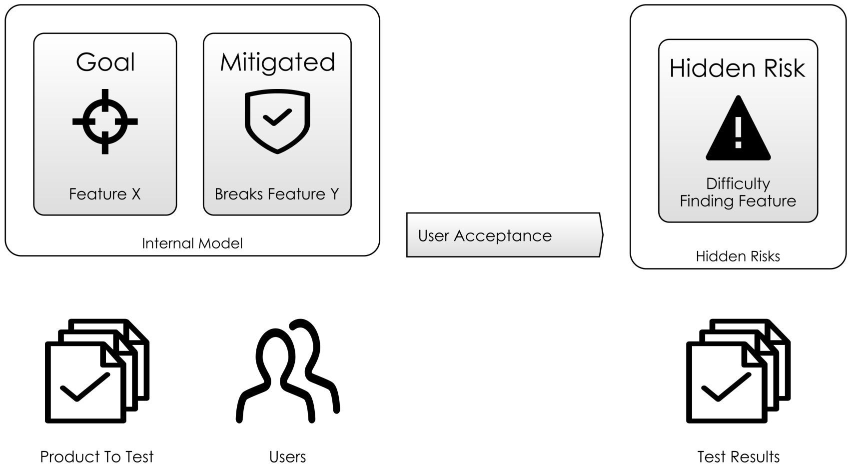


Figure 2.7: UAT - putting tame users in front of your software is better than real ones, where the risk is higher

2.4 Observations

A couple of things:

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. Part of the purpose of this site is to help with this and try and provide a taxonomy for different types of risks.

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? It's pretty hard to draw a line and say exactly.

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

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

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

CHAPTER 3

Meeting Reality

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

3.1 Revisiting the Model

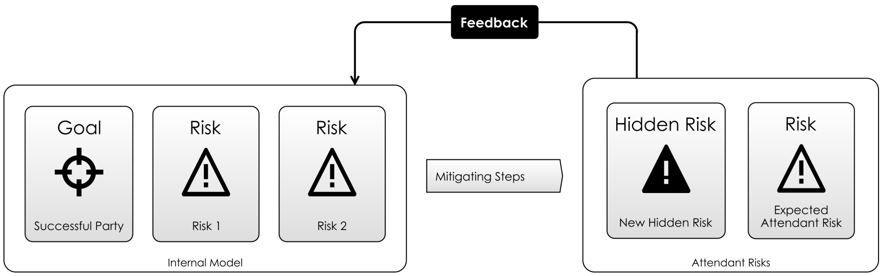
In A Simple Scenario, we looked at a basic model for how **Reality** and our Internal Model interacted with each other: we take action based on out Internal Model, hoping to **change Reality** with some positive outcome.

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

It should be no surprise to see that there is a *recursive* nature about this:

1. 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 (otherwise, what would be the point of doing them?)
2. The actions we take towards achieving a Goal In Mind each have their *own* Goal In Mind. And because of this, when we take action, we have to consider and evaluate the Hidden Risks exposed by that action. That is, there are many ways to achieving a goal, and these different ways expose different Hidden Risks.

So, let's see how this kind of recursion looks on our model. Note that here, I am showing *just one possible action*, in reality, you'll have choices.



.

Hopefully, if you've read along so far, this model shouldn't be too hard to understand. But, how is it helpful?

3.2 "Navigating the Risk Landscape"

So, 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 accrues the *least risk* to get it done: each action you take in trying to achieve the overall Goal In Mind will have it's Attendant Risks, and it's the experience you bring to bear on these that will help you navigate through them smoothly.

Ideally, when you take an action, you are trading off a big risk for a smaller one. Take Unit Testing for example. Clearly, writing Unit Tests adds to the amount of development work, so on it's own, it adds Schedule Risk. However, if you write *just enough* of the right Unit Tests, you should be short-cutting the time spent finding issues in the User Acceptance Testing (UAT) stage, so you're hopefully trading off a larger Schedule Risk from UAT and adding a smaller risk to **Development**.

You can think of taking actions 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 solve a higher-level Attendant Risk will leave you with a worse Attendant Risks. Almost certainly, this will have been a Hidden Risk when you embarked on the action, otherwise you'd not have chosen it.

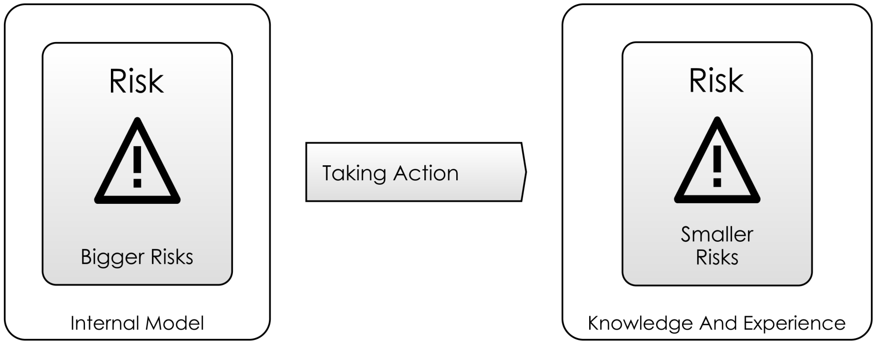


Figure 3.1: Navigating The Risk Landscape

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

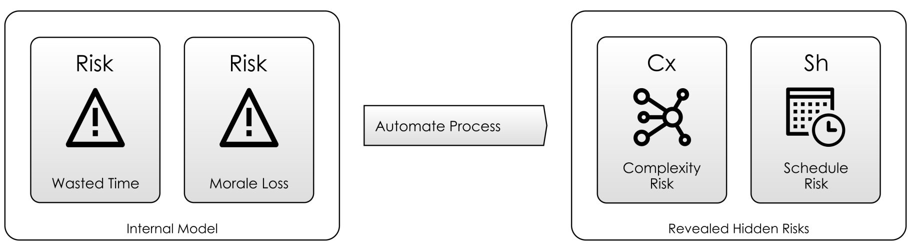


Figure 3.2: Hidden Risks of Automation

Another Quick Example: MongoDB

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

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

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.

The data storage Attendant Risk was mitigated easily with MongoDB. However, the new Attendant Risk of licensing bureacracy eventually proved too great, and MongoDB had to be abandoned after much investment of time.

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

3.3 The Cost Of Meeting Reality

Meeting reality 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 - sometimes that is expensive. But we can meet it in "limited ways".

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 looked at the use of UAT in order to Meet Reality earlier. By performing the UAT, we meet reality sooner. The *cost* of this is that we also expend precious time on that process which adds risk to the schedule.

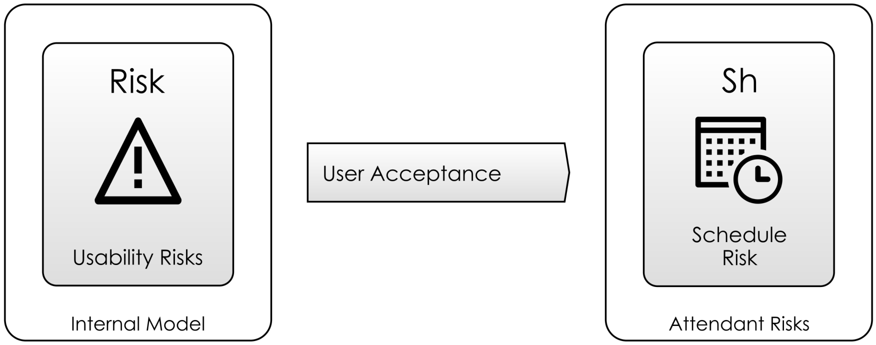


Figure 3.3: Testing flushes out hidden risk, but increases Schedule Risk

YAGNI

As a flavour of what's to come, let's look at YAGNI[2], an acronym for You Aren't Gonna Need It. Martin Fowler says:

> YAGNI originally is an acronym that stands for "You Aren't Gonna Need It". It is a mantra from ExtremeProgramming 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".
>
> This principle was first discussed and fleshed out on Ward's Wiki[3]

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

But, there is always the opposite opinion: You *Are* Gonna Need It[4]. As a simple example, we often add log statements in our code as we write it, though following YAGNI strictly says we shouldn't.

Which is right?

Now, we can say: do the work *if it mitigates your Attendant Risks*.

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

[3] http://wiki.c2.com/?YouArentGonnaNeedIt

[4] http://wiki.c2.com/?YouAreGonnaNeedIt

- 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, you're reducing risk.

Do The Simplest Thing That Could Possibly Work

Another mantra from Kent Beck (originator of the Extreme Programming[5] methodology, is "Do The Simplest Thing That Could Possibly Work", which is closely related to YAGNI and is about looking for solutions which are simple. Our risk-centric view of this strategy would be:

- Every action you take on a project has it's own Attendant Risks.
- The bigger or more complex the action, the more Attendant Risk it'll have.
- The reason you're taking action *at all* is because you're trying to reduce risk elsewhere on the project
- Therefore, the biggest payoff is whatever action *works* to remove that risk, whilst simultaneously picking up the least amount of new Attendant Risk.

So, "Do The Simplest Thing That Could Possibly Work" is really a helpful guideline for Navigating the Risk Landscape.

3.4 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.

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

CHAPTER 4

All Risk Management

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

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

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

4.1 RAID Log

Many project managers will be familiar with the RAID Log. It's simply four columns on a spreadsheet:

- Risks
- Actions
- Issues
- Decisions

Let's try and put the following Attendant 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.

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

- 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?

4.2 Let's Go Again

This is a completely made-up example, deliberately chosen to be hard to categorize. 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 Risk:

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.
- If you're getting a business sign-off for something, this is mitigating a Too Many Cooks-style *stakeholder risk*.
- If you're reading the specification, that's mitigating the type of "Developer Misimplementation Risk" we saw in the last chapter.

Every Action Carries 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?

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.

An Issue is Just A Type of Risk

- Because issues need to be solved. . .
- And solving 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, so stop agonising over which column to put them in.

Every Decision is a Risk.

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

4.3 Goals Are Risks Too

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

So we have:

- Risks, which exist within Internal Models.
- Actions, which are operations we perform when we Meet Reality.
- Artifacts, which are the results of Meeting Reality.

Goals live inside our Internal Model too. 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:

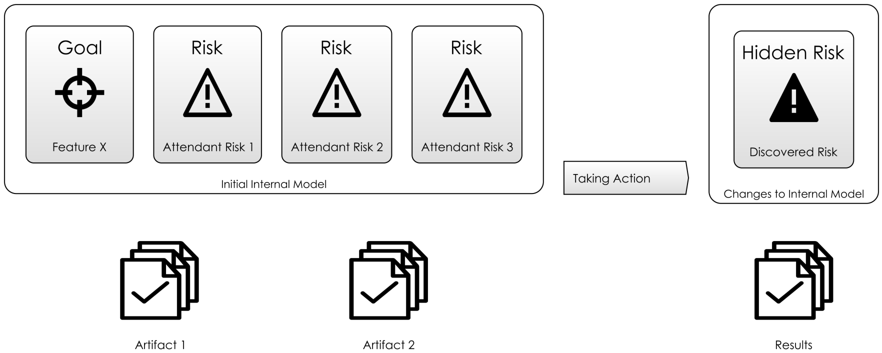


Figure 4.1: Risk-First Diagram Language

Goal	Restated As A Risk
Build a Wall	Mitigate the risk of something getting in / out
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 the starting point, a movement, and the destination. From a redundancy perspective, any one of these can be determines 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.

4.4 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 and give up.

4.5 What To Do?

It makes it much easier to tackle the RAID log if there's only one list: all you do is pick either the *most important* risk on the list, or the *most urgent*, or the risk you can make *the biggest impact on*, and deal with it.

In the next chapter, A Software Project Scenario we'll give an example of this in action.

CHAPTER 5

Software Project Scenario

Where do the risks of the project lie?

How do we decide what *needs to be done today* on a software project?

Let's look again at the simple risk framework from the introduction and try to apply it at the level of the *entire project*.

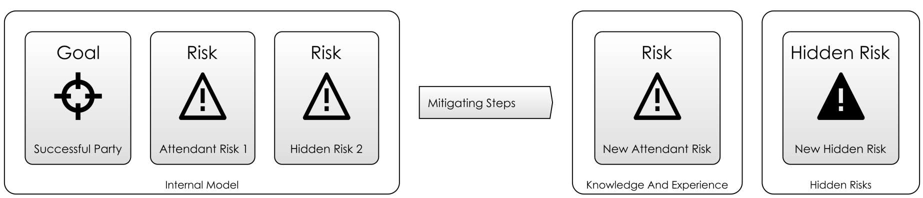


Figure 5.1: Taking action changes reality, but it changes your perception of the attendant risks too

5.1 Today's Action

How should we decide how to spend our time today?

What actions should we take? (In Scrum[1] terminology, what is our *Sprint Goal*?).

Let's say that we are working in a Startup, on a project building an online service. As we discussed previously, we could call this a Goal, but we could also restate it as mitigating Risk, which gives more clarity about our intentions. For the startup, launching the Online Service is about mitigating (Funding-Risk), in order to keep the startup alive.

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

5.2 Attendant Risks

What are the Attendant Risks that come with that goal? Here are some to get us started:

1. The users can't access the system
2. The data gets lost, stolen.
3. The data is wrong or corrupted
4. There are bugs that prevent the functionality working
5. The functionality isn't there that the user needs (Feature Risk).
6. Our Internal Model of the market is poor, and we could be building the wrong thing.

I'm sure you can think of some more.

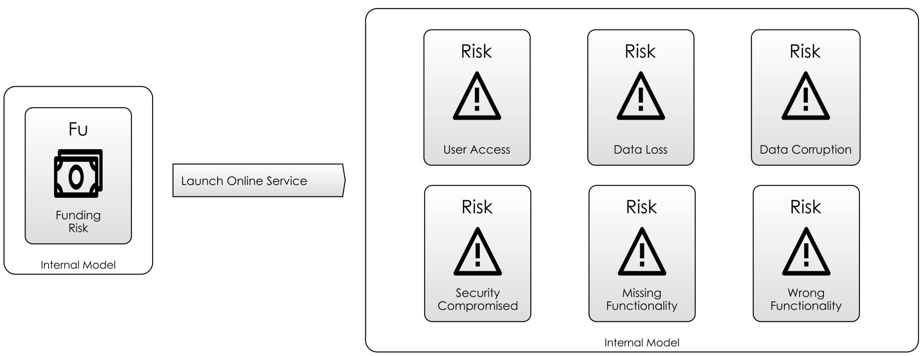


Figure 5.2: Our Goal, With Attendant Risks

5.3 Evaluating The Risks

Next, we can look at each of these risks and consider the threat they represent. The same Attendant Risks will be evaluated differently depending on the *nature of the project* and the mitigations you already have in place. For example:

- If they **can't access it**, does that mean that they're stuck unable to get on the train? Or they can't listen to music?
- If the **data is lost**, does this mean that no one can get on the plane? Or that the patients have to have their CAT scans done again? Or that people's private information is scattered around the Internet?

- If the **data is wrong**, does that mean that the wrong people get sent their parcels? Do they receive the wrong orders? Do they end up going to the wrong courses?
- If the **security is compromised** and all the data exposed, does that mean confidential details are shared on the Internet, or that they just need to create a new password?
- If there is **missing functionality**, will they not buy the system? Will they use a competitor's product? Will they waste time doing things a harder or less optimal way?
- If our **Internal Model is wrong**, then is there a chance we are building something for a non-existent market? Or annoying our customers? Or leaving an opportunity for competitors?

A Single Attendant Risk: Getting Hacked

Let's consider a single risk: that the website gets hacked, and sensitive data is stolen. How we evaluate this risk is going to depend on a number of factors:

- How many users we have
- The importance of the data
- How much revenue will be lost
- Risk of litigation
- etc.

Ashley Maddison

In 2016, Ashley Maddison, a website specialising in aiding extra-marital affairs, suffered a data breach[2], revealing the personal details of all of their clients on the Internet. The sensitive nature of their business contrasts sharply with the reported sloppiness of their security arrangements.

Can our risk model explain what happened here?

- It's possible that for the developers in question, security concerns were a Hidden Risk: they were not aware of the dangers of being hacked.
- It's possible that *at the time of setting up the security arrangements*, stronger security was considered, but the evaluation of the risk was low: Perhaps, the risk of not shipping quickly was deemed greater. And so they ignored this concern.

[2] https://en.wikipedia.org/wiki/Ashley_Madison_data_breach

- It's also possible that *for Ashley Maddison* the perceived impact of any data-breach was considered low. The impact is on the *customers*, rather than the *company*.
- But, as the number of users of the sites increased, the risk increased too, but there was (apparently) no re-evaluation of the risk otherwise they would have addressed it. This was a costly *failure to update the Internal Model*.

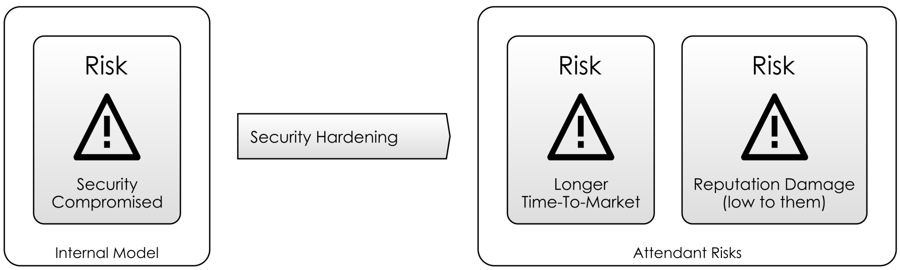


Figure 5.3: Attendant Risks of Improved Security

This story highlights a couple of important lessons. First, when exposing a service on the Internet, it's now a good idea to *look for trouble*: you should go out and try and improve your Internal Model. Thankfully, this is what sites like OWASP[3] are for: they *tell you about the Attendant Risks* and further, try to provide some evaluation of them to guide your actions. Second, you shouldn't expect an organisation with a morally-questionable business model to have your best interests at heart.

5.4 Actions Are Dependent On Risk Evaluation

So, *evaluating risks against one another other* allows us to determine what actions we should pursue on any given day, by answering the question:

Which actions give us the biggest benefit in terms of mitigating Attendant Risk?

For example, it's worth considering that if we're just starting this project, risks 1-4 in our diagram above are *negligible*. It makes complete sense from a risk-evaluation perspective that we're only going to spend time building functionality or improving our understanding of the market.

[3] https://www.owasp.org/index.php/Top_10-2017_Top_10

5.5 Tacit and Explicit Modelling

As we saw in the example of the Dinner Party, creating an internal model is something *we just do*: we have this functionality in our brains already. When we scale this up to a whole project team, we can expect the individuals on the project to continue to do this, but we might also want to consider *explicitly* creating a risk register for the whole project.

In the next chapter, we're going to take a quick aside into looking at some Risk Theory around this.

CHAPTER 6

Risk Theory

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

6.1 Risk Registers

In the previous chapter Software Project Scenario we saw how you try to look across the Attendant Risks of the project, in order to decide what to do next.

A Risk Register[1] can help with this. From Wikipedia:

> A typical risk register contains:
>
> - A risk category to group similar risks
> - The risk breakdown structure identification number
> - A brief description or name of the risk to make the risk easy to discuss
> - The impact (or consequence) if event actually occurs rated on an integer scale
> - The probability or likelihood of its occurrence rated on an integer scale
> - The Risk Score (or Risk Rating) is the multiplication of Probability and Impact and is often used to rank the risks.
> - Common mitigation steps (e.g. within IT projects) are Identify, Analyse, Plan Response, Monitor and Control.

This is Wikipedia's example:

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

Category	Name	RBS ID	Probability	Impact	Mitigation	Contingency	Risk Score after Mitigation	Action By	Action When
Guests	The guests find the party boring	1.1.	low	medium	Invite crazy friends, provide sufficient liquor	Bring out the karaoke	2		within 2hrs
Guests	Drunken brawl	1.2.	medium	low	Don't invite crazy friends, don't provide too much liquor	Call 911	x		Now
Nature	Rain	2.1.	low	high	Have the party indoors	Move the party indoors	0		10mins
Nature	Fire	2.2.	highest	highest	Start the party with instructions on what to do in the event of fire	Implement the appropriate response plan	1	Everyone	As per plan

Figure 6.1: Wikipedia Risk Register

Some points about this description:

This is a Bells-and-Whistles Description

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" to Wikipedia's Risk Register description. Most of the time, it's going to be in your head, or in discussion with the team, rather than written down.

Most of the value of the **Risk-First** approach is *in conversation*. Later, we'll have an example to show how this can work out.

Probability And Impact

Sometimes, it's better to skip these, 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.

6.2 Graphical Analysis

The Wikipedia page also includes this wonderful diagram showing you risks of a poorly run barbecue party:

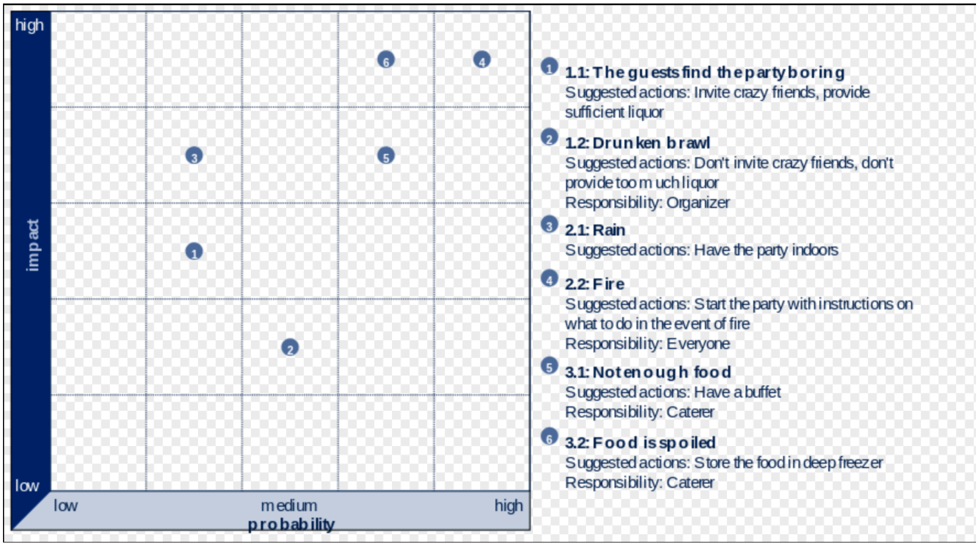


Figure 6.2: Wikipedia Risk Register

This type of graphic is *helpful* in deciding what to do next, although personally I prefer to graph the overall **Risk Score** against the **Cost of Mitigation**: easily mitigated, but expensive risks can therefore be dealt with first (hopefully).

6.3 Unknown Unknowns

In Wikipedia's example, this fictitious BBQ has high fire risk, so one should begin mitigating there.

But, does this feel right? 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.

6.4 Risk And Uncertainty

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

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

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

This is not how **Risk-First** sees it:

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

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

Finally, Crisis Management is *still just Risk Management*: the crisis (Earthquake, whatever) has *happened*. You can't manage it because it's in the past.

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

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

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

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

All you can do is Risk Manage the future (minimize further casualties and human suffering, for example).

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

6.6 Invariances #1: Panic Invariance

You would expect then, that any methods for managing software delivery should be *invariant* to the level of crisis in the project. If, for example, a project proceeds using Scrum) 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 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*.

I call this **Panic Invariance**: the methodology shouldn't need to change given the amount of pressure or importance on the table.

6.7 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, and this was an early win for Einstein's Relativity.

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, the bigger risks increase too fast for it to work.

[6] `https://en.wikipedia.org/wiki/Sunk_costs`

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 that I am exploring with Risk First is trying to place methodologies and practices within a framework to say *when* they are applicable.

6.8 Value vs Speed

Value

"Upside Risk" isn't a commonly used term: industry tends to prefer "value", as in "Is this a value-add project?". There is plenty of theory surrounding **Value**, such as Porter's Value Chain[7] and Net Present Value[8]. This is all fine so long as we remember:

- **The 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.

Risk-First is a particular *view* on reality. It's not the only one. However, I am going to try and make the case that it's an underutilized one that has much to offer us.

Speed

For example, in Rapid Development[9] by Steve McConnell we have the following diagram:

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

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

[9] http://a.co/d/ddWGTB2

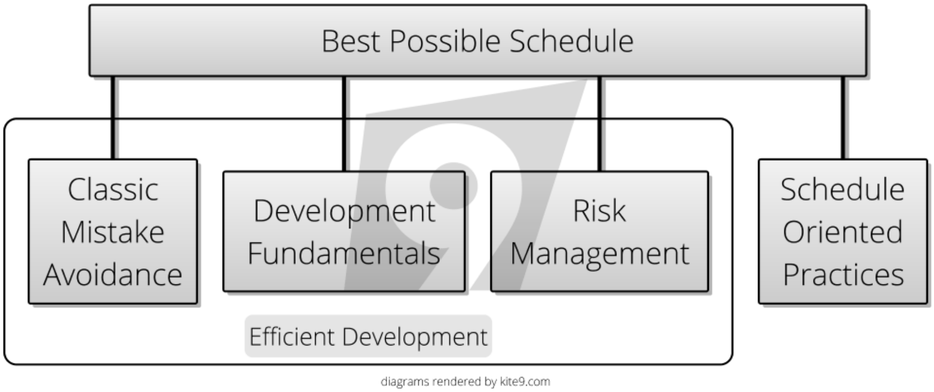


Figure 6.3: Pillars, From Rapid Development By Steve McConnell

And, this is *fine*, McConnell is structuring the process from the perspective of *delivering as quickly as possible*. However, here, I want to turn this on it's head. Exploring 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:

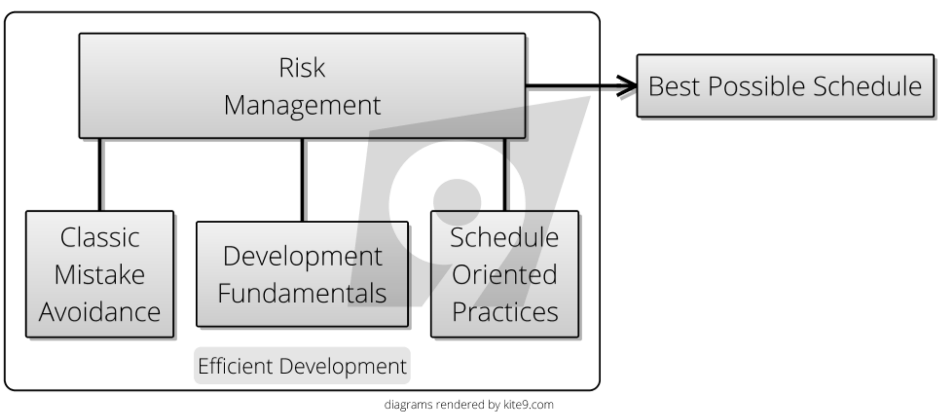


Figure 6.4: Pillars, re-arranged

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 unaddressed Operational Risk, and irreparably damages trust in the product.

Eisenhower's Box

If we can view software delivery from the point of view of *value*, then why can't we apply the same tools to Risk too? In order to do this, let's review "Eisenhower's Box" model. This considers two variables:

- How valuable the work is (Importance)
- How soon it is needed (Urgency)

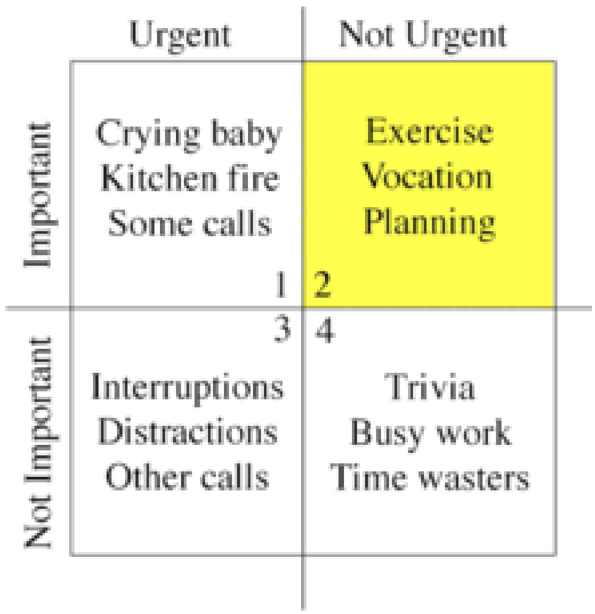


***Figure 6.5:** A basic "Eisenhower box" to help evaluate urgency and importance. Items may be placed at more precise points within each quadrant. - Time Management,* **Wikipedia**[10]

Here, we're considering a synthesis of both *time* and *value*. But Net Present Value allows us to discount value in the future, which offers us a way to reconcile these two variables. In the diagram, you can see two future payments, one of č100 due in one year, one of č150 due in 10 years. By discounting at a given rate (here at 6% per year) we can compare their worth *now*.

6.9 Net Present Risk

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

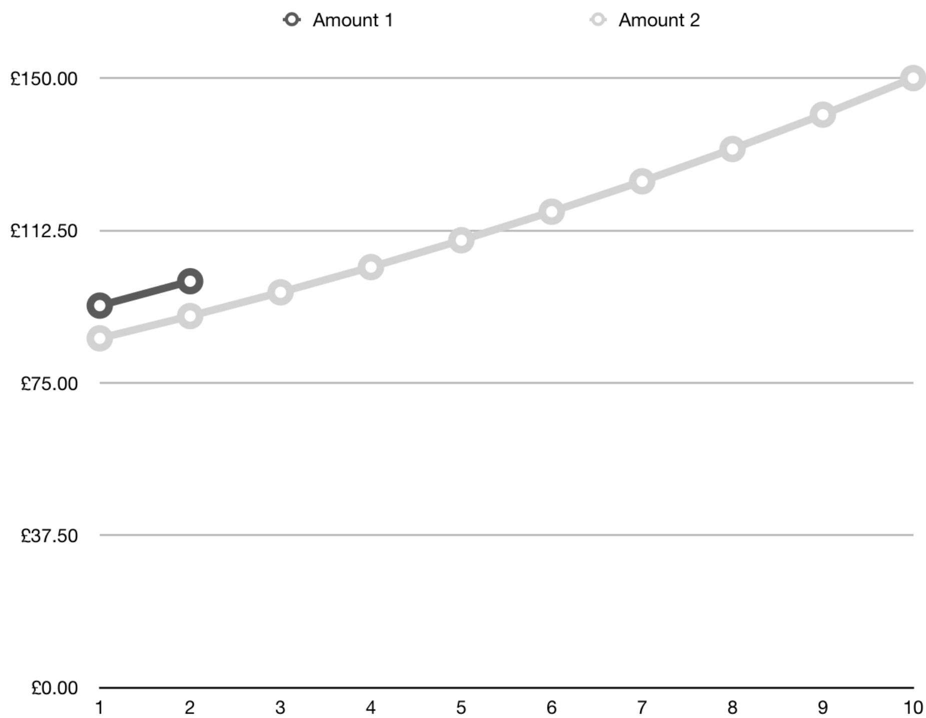


Figure 6.6: Net Present Value Discounting

> Net Present Risk is the *Importance* 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 3 risks:

- Risk **A**, which will cost you č50 in 5 year's time.
- Risk **B**, which will cost you č70 in 8 year's time.
- Risk **C**, which will cost you č120 in 18 year'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 year. A graph of the discounted risks looks like this:

On this basis, the biggest risk is **B**, at about č45. If we *reduce* the discount factor to 3%, we get a different result:

Now, risk **C** is bigger.

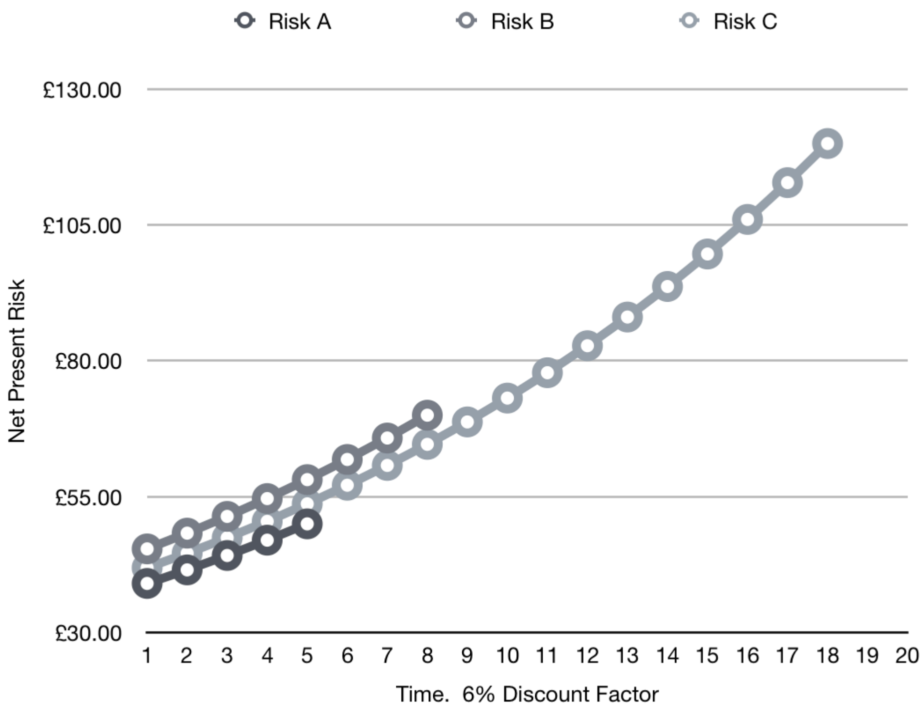


Figure 6.7: Net Present Risk, 6% Discount Rate

Because this is *Net* Present Risk, we can also use it to make decisions about whether or not to mitigate risks. Let's consider the cost of mitigating each risk *now*:

- Risk **A** costs č20 to mitigate
- Risk **B** costs č50 to mitigate
- Risk **C** costs č100 to mitigate

Which is the best deal?

Well, under the 6% regime, only Risk **A** is worth mitigating, because you spend č20 today to get rid of č40 of risk (today). The NPR is positive at around č20, whereas for **B** and **C** mitigations it's under water.

But, under a 3% regime, risk **A** and **B** are *both* worth mitigating.

Discounting the Future To Zero

I have worked in teams sometimes where the blinkers go down, and the only thing that matters is *now*. They may apply a rate of 60% per-day, which

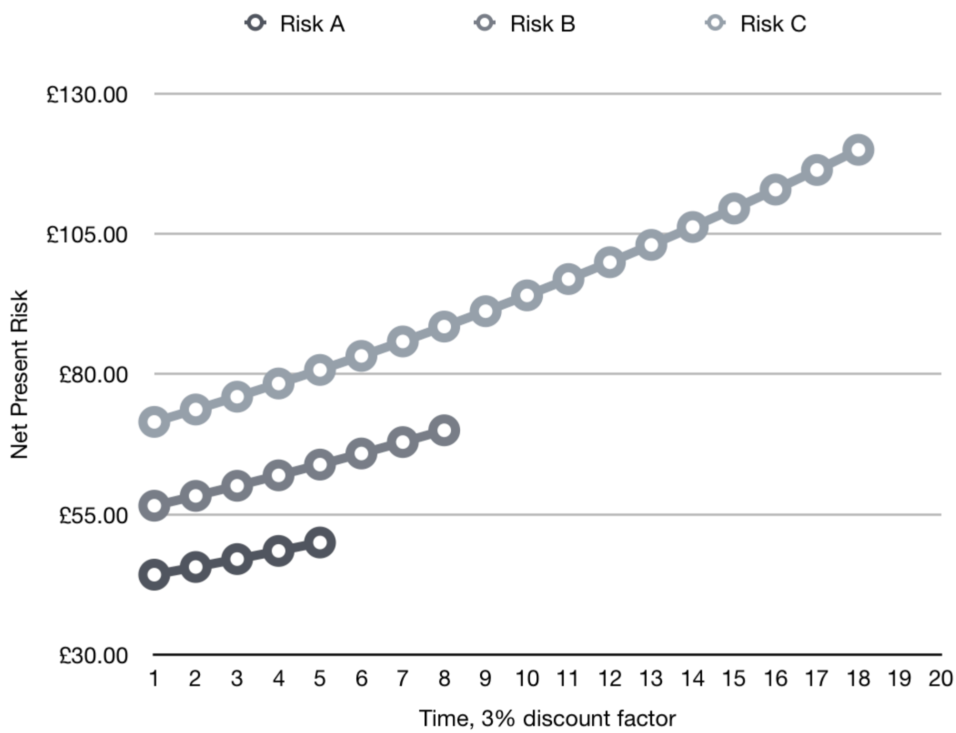


Figure 6.8: Net Present Risk, 3% Discount Rate

means that anything with a horizon over a week is irrelevant. Regimes of such hyper-inflation[11] 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: č80 cost, happening *tomorrow*
- Risk B: č500 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 with the sack on a daily basis, and therefore feel they have to react. Publically-listed companies also often 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*:

[11] https://en.wikipedia.org/wiki/Hyperinflation

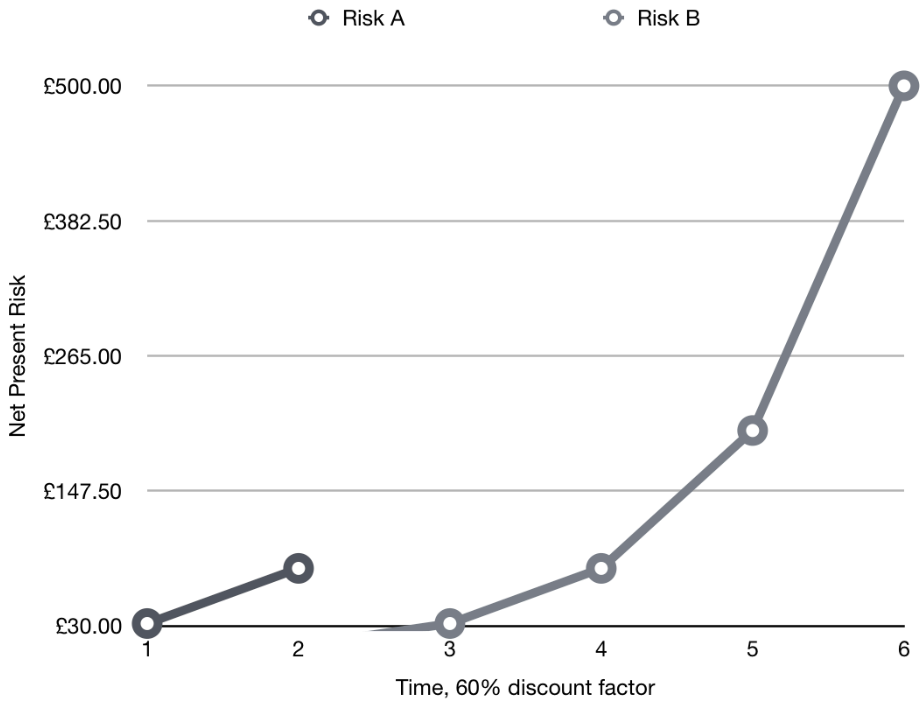


Figure 6.9: Net Present Risk, 60% Discount Rate

> "Here is Edward Bear coming downstairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way... if only he could stop bumping for a moment and think of it!"
>
> —A. A. Milne, *Winne-the-Pooh*[12]

6.10 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. In fact, my view is that you should *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 evidence for an approach like this. We *can* look

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

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 first of all misused and then gamed.

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? I would argue 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?"

6.11 Pay-Off

Because everything we're dealing with is Risk, we can't know in advance how well any action we take will pay off. Taking Action is a lot like placing a bet. But, if we can *weigh separate risks in our hands* we should be able to intuit what the Pay-Off of a given Action will be.

The fruits of this gambling are revealed when we meet reality, and we can see whether our bets really paid off.

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

CHAPTER 7

Cadence

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

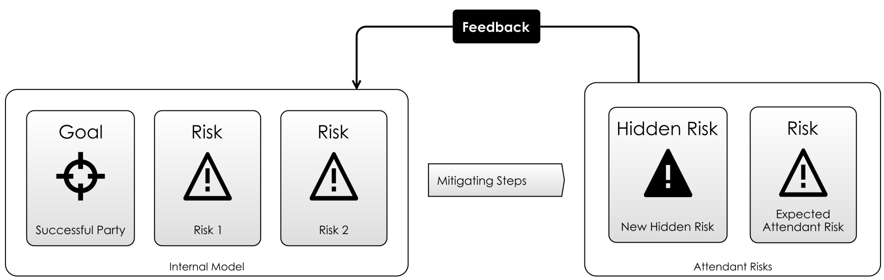


Figure 7.1: Meeting Reality: reality is changed and so is your internal model.

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

How *fast* should we go round this loop? Is there a right answer? 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] (CD) 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

One thing that often astounds me is how developers can ignore the fast feedback loops at the bottom of the list, because the ones nearer the top *will do*. In the worst cases, changing two lines of code, running the build script, deploying and then manually testing out a feature. And then repeating.

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

The Testing Pyramid[2] hints at this truth:

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

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

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

Production

You could take this chapter to mean that Continuous Delivery (CD) is always and everywhere a good idea. I *guess* 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 the fastest feedback loop possible, *which actually does give you feed back.*

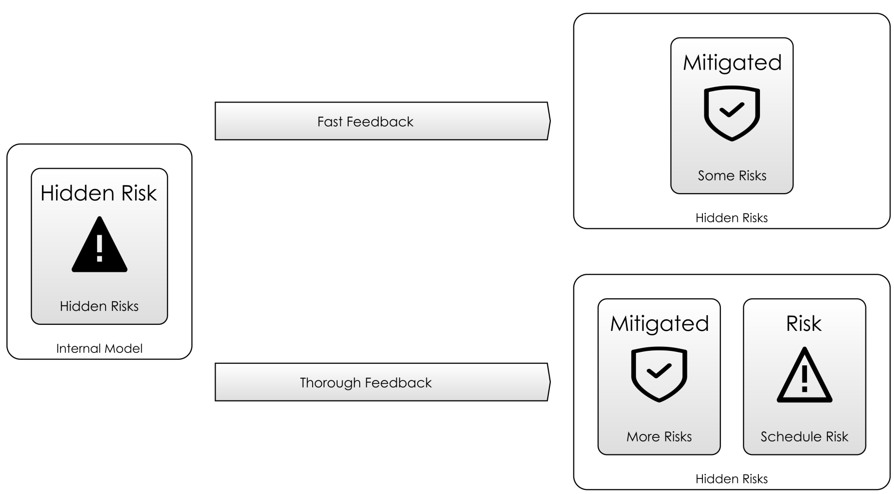


Figure 7.2: Different actions have different feedback loops

7.1 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.

- Some Risk Theory was an aside, looking at some terminology and the useful concept of a Risk Register.

- Finally, above, 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.

Let's have a look at an example of how this might work.

CHAPTER 8

De Risking

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

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

Some simple examples of this might be:

- Safety-nets and ropes de-risk climbing. But, the activity of climbing itself is otherwise much unchanged.
- Backups and Source-Control de-risk the development process by reducing the impact of computer failure. Our process is changed *slightly* by this imposition, but we're not massively inconvenienced.
- tbd.
- tbd.

Let's look at some common strategies for De-Risking. This is probably not an exhaustive set, but it's a good starting point.

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

8.1 Mitigate

Mitigating the risk is taking steps towards minimising either it's likelihood or impact (as we discussed in the Risk Theory 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.)

8.2 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.

8.3 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/

8.4 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 usually mitigating more risk than they cause. Another example is using particular software dependencies: I might build a mobile application which requires a Facebook account to log in. This might give rise to the risk that some people 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. As we've discussed previously, what we decide to mitigate on a daily basis will depend on the evaluation of where we can best mitigate Attendant Risk.

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. One useful test I have found is whether "This would be a nice problem to have". For example:

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

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

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

Learned Helplessness

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

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

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

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

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

8.6 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 are **exploiting** the risk of my house burning down by selling me insurance. This is a common pattern: wherever there is risk, there is likely to be a way to profit from it.

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

8.7 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:

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

> "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*[6]

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

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

- Concentrate on Risks, not Goals (goals are risks in disguise, anyway)
- Every action you take is De-Risking *something* (or should be)
- tbd.

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

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

CHAPTER 9

A Conversation

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

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

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

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

9.1 A Risk Conversation

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

Bob: Tell me.

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

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

Eve: Right, and it *did* flush out Hidden Risk: some people using Windows 7, downloading Excel spreadsheets on the site, couldn't download them: for some reason, that combination didn't support anything greater than TLS version 1.0. So, we had to back it out.

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

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

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

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

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

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

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

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

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

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

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

Eve: Yes. How about for now, we go with the isolated server, and review next week? If it's working out, then great, we continue with it. Otherwise, if we're not making progress next week, then it'll be because our isolation solution is meeting more risk than we originally thought. We can try the settings change in that case.

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

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

9.2 Aftermath

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

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

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

Part II

Risk

CHAPTER 10

Risk Landscape

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.

10.1 The Financial Crisis

In the Financial Services[1] industry, whole *departments* exist to calculate things like:

- Market Risk[2]: the risk that the amount some asset you hold/borrow/have loaned is going to change in value.
- Credit Risk[3]: the risk that someone who owes you a payment at a specific point in time might not pay it back.
- Liquidity Risk[4]: 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 č100

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

[2] https://en.wikipedia.org/wiki/Market_risk

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

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

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 Easing[5].
- All of which had massive impacts on the markets in ways that none of the Market Risk models foresaw.

All the Risks were correlated[6]. That is, they were affected by the *same underlying events*, or *each other*.

10.2 The Risk Landscape Again

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

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 Landscape[7]. 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 Descent[8] 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

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

[6] https://www.investopedia.com/terms/c/correlation.asp

[7] https://en.wikipedia.org/wiki/Fitness_landscape

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

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*.

10.3 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 Theory[9], or Concurrency Theory[10]. 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.

10.4 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.

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

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

10.5 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.

CHAPTER 11

Feature Risk

Feature Risk is the category of software risk to do with features that have to be in your software. It is the risk that you face by *not having features that your clients need*.

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!

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.

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 it's variations.

11.1 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*. Now, as usual, you could call this an issue, but we're calling it a Risk because it's not clear exactly *how many* people are affected, or how badly.

- This might manifest itself as complete *absence* of something you need, e.g "Where is the word count?"
- It could be that the implementation isn't complete enough, e.g "why can't I add really long numbers in this calculator?"

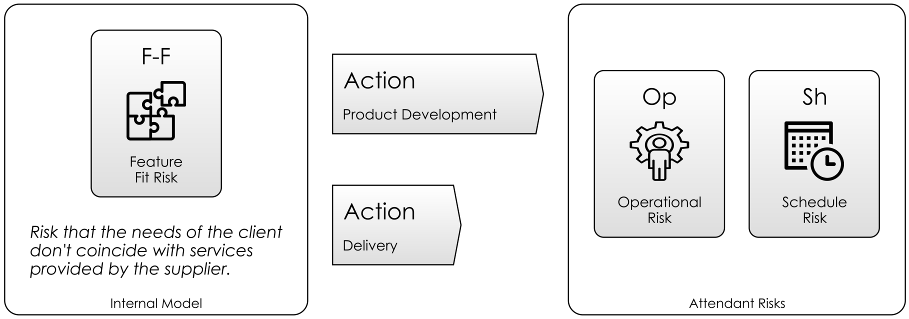


Figure 11.1: Feature Risk

11.2 Implementation Risk

Feature Risk also includes things that don't work as expected: That is to say, bugs[1]. 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*.

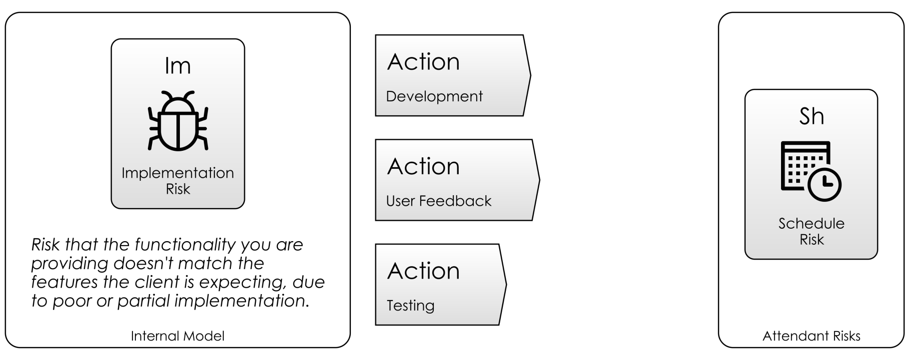


Figure 11.2: Implementation Risk

(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 or Documentation or simply Poor User Interface and we'll look at that more in Communication Risk.)

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

11.3 Regression Risk

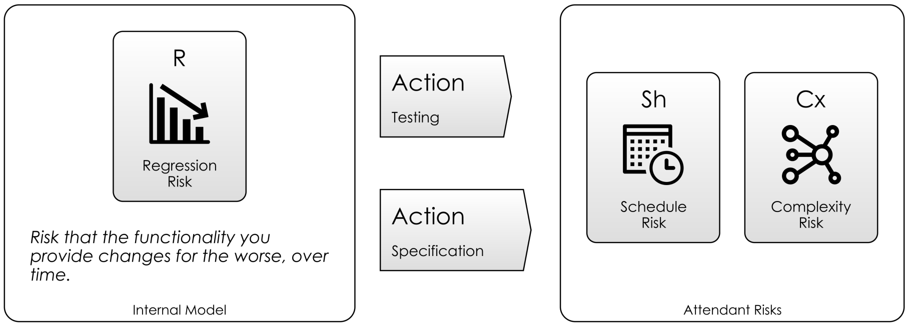


Figure 11.3: Regression Risk

Regression Risk is basically 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.

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

11.4 Conceptual Integrity Risk

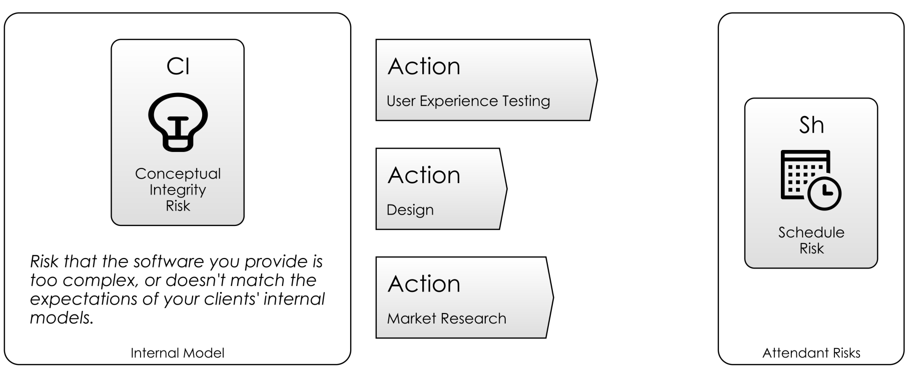


Figure 11.4: Conceptual Integrity Risk

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 Phones[2] are a real-life example: although it *seemed* like the market wanted more and more features added to their phones, Apple's iPhone[3] 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.

Which leads to Greenspun's 10th Rule:

> "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."
>
> —Greenspun's 10th Rule, *Wikipedia*[4]

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 differences you can find. The aim of a product is to capture some needs at a *general* level: you can't hope to "please all of the people all of the time".

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

11.5 Feature Access Risk

Sometimes, features can work for some people and not others: this could be down to Accessibility[5] issues, language barriers or localisation.

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

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

[4] https://en.wikipedia.org/wiki/Greenspun's_tenth_rule

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

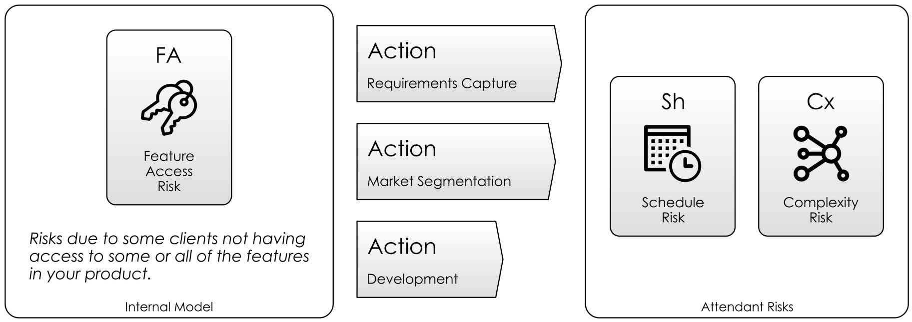


Figure 11.5: Feature Access Risk

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 Segmentation[6]: trying to work out *who* your product appeals to, and tailoring it to that particular market, but for technologists, increasing Feature Access means increasing complexity: you have to deliver the software on more platforms, localised in more languages, with different configurations of features at different price-points. Mitigating Feature Access Risk therefore means increased effort and complexity (which we'll come to later).

Market Risk

Feature Access Risk is related, of course, to Market Risk, which I introduced on the Risk Landscape page as being the value that the market places on a particular asset. Since the product you are building is your asset, it makes sense that you'll face Market Risk on it:

> "Market risk is the risk of losses in positions arising from movements in market prices."
>
> —Market Risk, *Wikipedia*[7]

I face market risk when I own (i.e. have a *position* in) some Apple[8] stock. Apple's's stock price will decline if a competitor brings out an amazing

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

[7] https://en.wikipedia.org/wiki/Market_risk

[8] http://apple.com

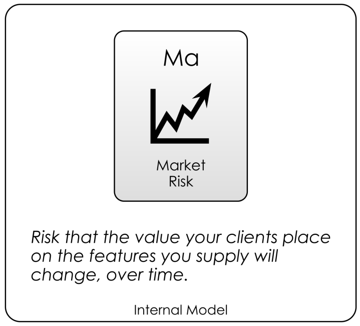


Figure 11.6: Market Risk

product, or if fashions change and people don't want their products any more.

In the same way, *you* have Market Risk on the product or service you are building: the *market* decides what it is prepared to pay for this, and it tends to be outside your control.

11.6 Feature Drift Risk

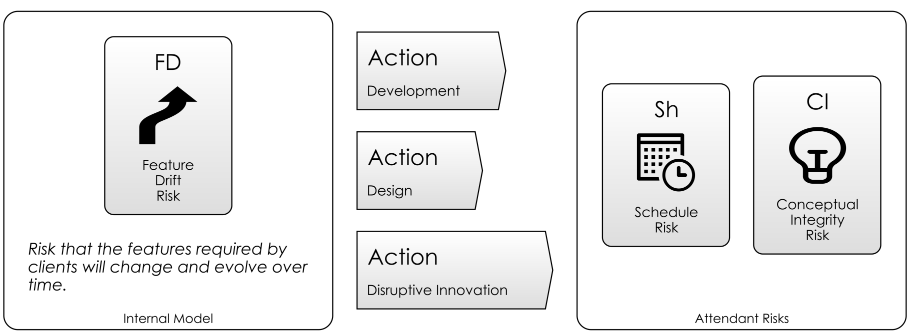


Figure 11.7: Feature Drift Risk

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. querty[9] 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.

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 shows[10]. 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?

11.7 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:

[9] `https://en.wikipedia.org/wiki/QWERTY`

[10] `https://designers.hubspot.com/blog/the-history-of-web-design-infographic`

- 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.

11.8 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*[11].

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

[11] https://en.wikipedia.org/wiki/Survival_of_the_fittest

- Expectation: What the user *expects*
- Reality: What they actually *get*.

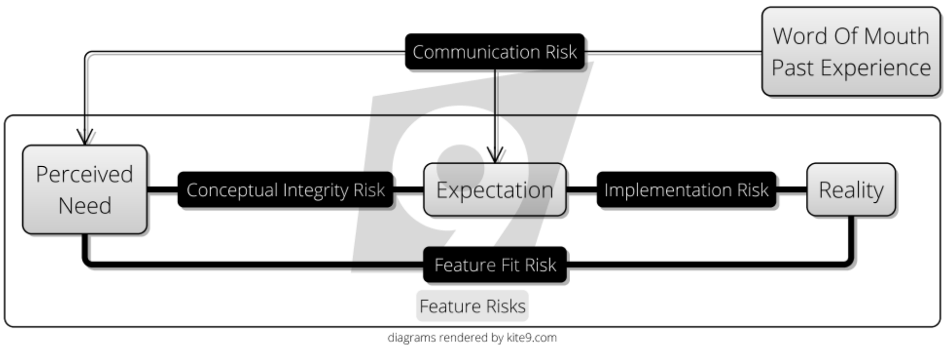


Figure 11.8: Feature Risks Assembled: Fit Risks

For further reading, you can check out The Service Quality Model[12], whcih 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.

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 it's 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.

[12] `http://en.wikipedia.org/SERVQUAL`

- The world as a whole is an evolving system within which our product exists.

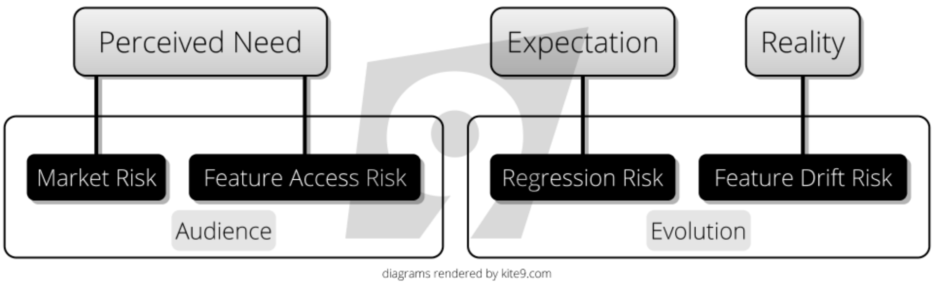


Figure 11.9: Feature Risks Assembled: Audience and Evolution

11.9 Applying Feature Risk

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

- Can you judge which tasks mitigate the most Feature Risk?
- 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 it's 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.

CHAPTER 12

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.

12.1 Kolmogorov Complexity

The standard Computer-Science definition of complexity, is Kolmogorov Complexity[1]. 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.

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

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

```
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

We might choose this representation:

```
function out() {                                        (7 )
    return "abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd" (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";                                      (11)

function out() {                                        (7 )
    return ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+          (16)
        ABCD+ABCD+ABCD;                                 (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";                                      (11)

function out() {                                        (7 )
    return ABCD.repeat(10)                              (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 Golf[2] territory. This javascript program plays FizzBuzz[3] up to 100, but is less readable than you might hope:

```
for(i=0;i<100;)document.write(((++i%3?'':'Fizz')+
(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.

12.2 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, *Wikipedia*[4])

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

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

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

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

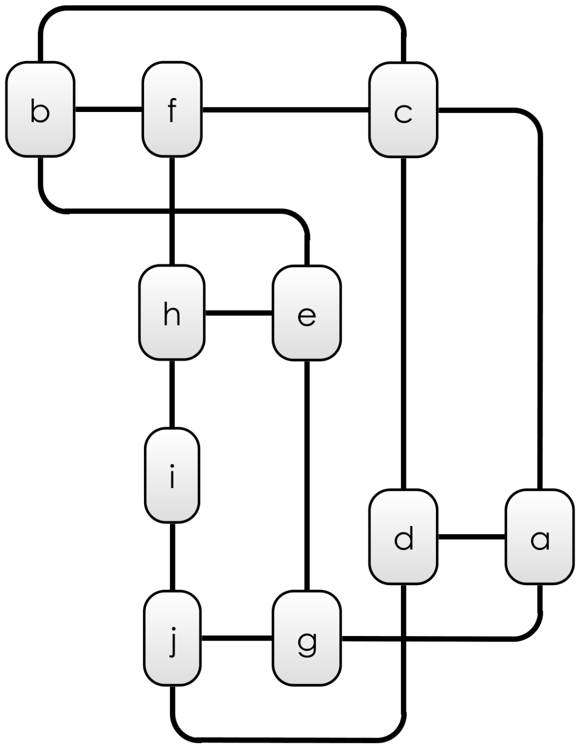


Figure 12.1: Graph 1

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:

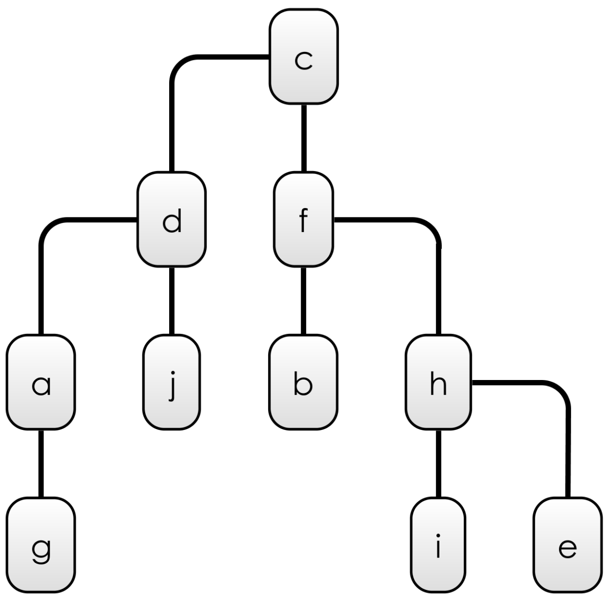


Figure 12.2: Graph 2

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.

12.3 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 heirarchy 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 Modularisation[5] and Hierarchy[6].

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:

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

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

- **Organelles** - such as Mitochondria[7].
- **Cells** - such as blood cells, nerve cells, skin cells in the Human Body[8].
- **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]

12.4 Cyclomatic Complexity

A variation on this graph connectivity metric is our third measure of complexity, Cyclomatic Complexity[10]. This is:

```
Cyclomatic Complexity = edges  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 Coverage[11]: the more branches you have, the more tests you'll need to exercise them all.

12.5 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

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

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

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

[11] https://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":

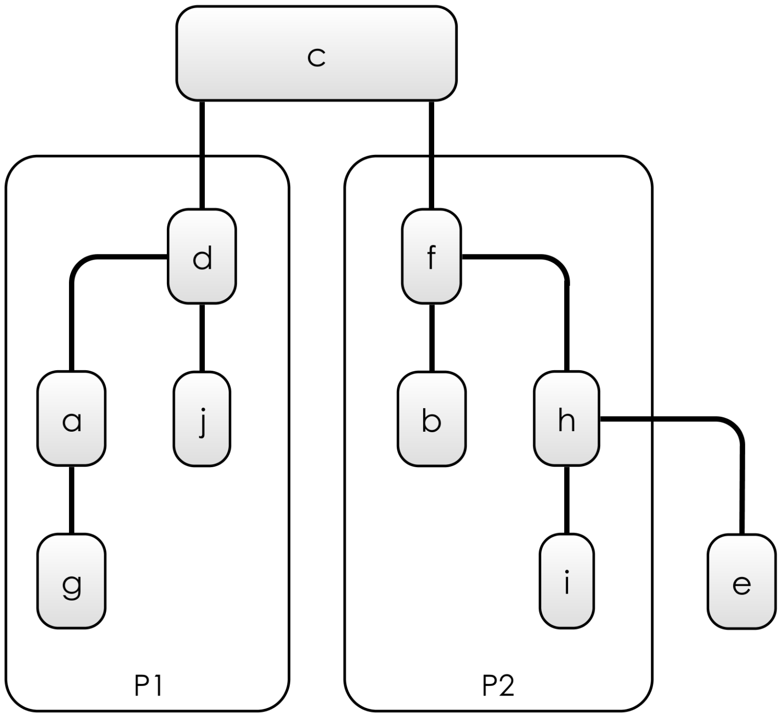


Figure 12.3: Complexity 3

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.

12.6 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 = *m***a**, (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 Veritasium[14] video. To paraphrase:

> "Most of your mass you owe due to E=mcš, 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

[12] https://en.wikipedia.org/wiki/Newtons_laws_of_motion

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

[14] https://www.youtube.com/user/1veritasium

[15] https://www.youtube.com/watch?annotation_id=annotation_3771848421&feature=iv&src_vid=Xo232kyTsO0&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.

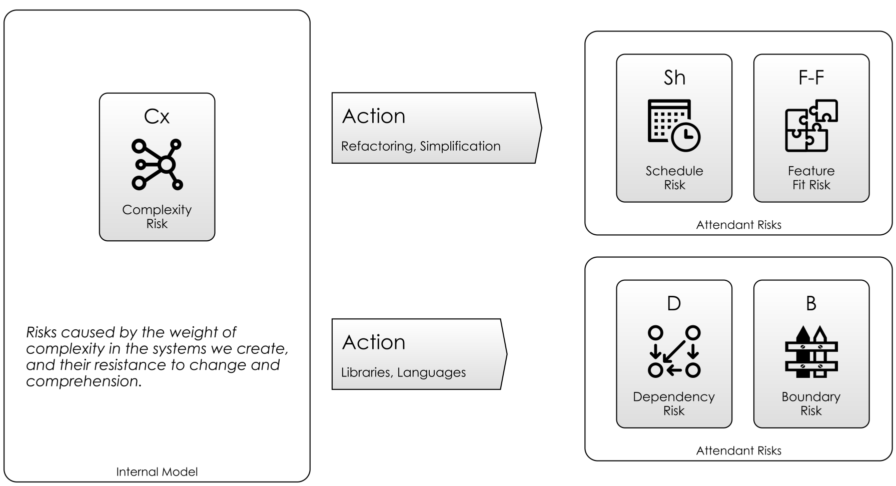


Figure 12.4: Complexity Risk and it's mitigations

12.7 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, 1992[16]

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 Refactor[17] the software to reduce this risk again.

12.8 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?

12.9 Feature Creep

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

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

[17] https://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 Creep[19] (or Gold Plating[20]) 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-Exupery[21]

12.10 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.

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

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

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

[21] http://www.quotationspage.com/quote/26979.html

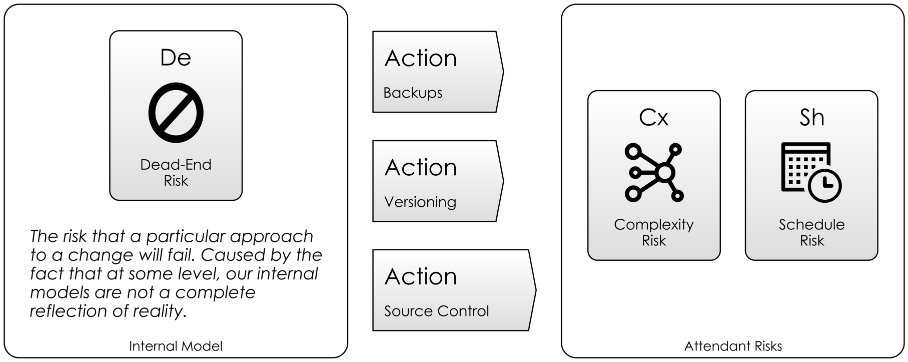


Figure 12.5: Dead-End Risk

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 Systems[22] like Git[23] are a useful mitigation of Dead-End Risk, because it means you can *go back* to the point where you made the

[22]https://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.

12.11 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, *Wikipedia*[25]

Garbage Collectors[26] (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's[27]. The point being is that it's totally possible for

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

[26] 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 Types[28]: 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 Clojure[29] have been retro-fitted with type checkers[30].

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 Complexity[31].

Once running, an algorithm or data structure will consume space or runtime dependent on it's characteristics. As with Garbage Collectors), these

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

[29] https://clojure.org

[30] https://github.com/clojure/core.typed/wiki/User-Guide

[31] https://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 Sheet[32] 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 conditions[34] and Deadlocks[35] *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 collections[36] 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, 1998[37]

[32] http://bigocheatsheet.com

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

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

[35] https://en.wikipedia.org/wiki/Deadlock

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

[37] https://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

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

CHAPTER 13

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.

13.1 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.)

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

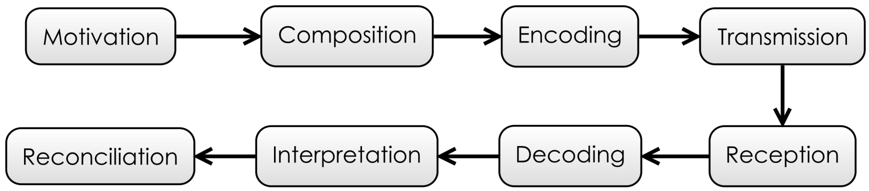


Figure 13.1: Shannon's Communication Model

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.

13.2 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 it's main types.

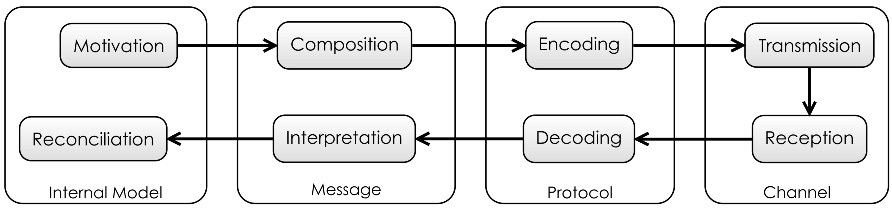


Figure 13.2: Communication Risk 2

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:

- Channels[2], the medium via which the communication is happening.
- Protocols[3] - the systems of rules that allow two or more entities of a communications system to transmit information.
- Messages[4]: 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.

13.3 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*.

[2] `https://en.wikipedia.org/wiki/Communication_channel`

[3] `https://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.

13.4 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.

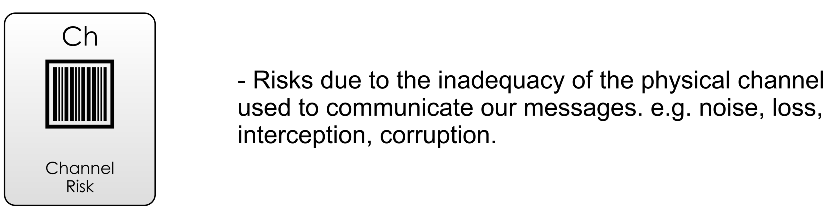


Figure 13.3: Communication Channel Risk

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 Programming[5] 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.

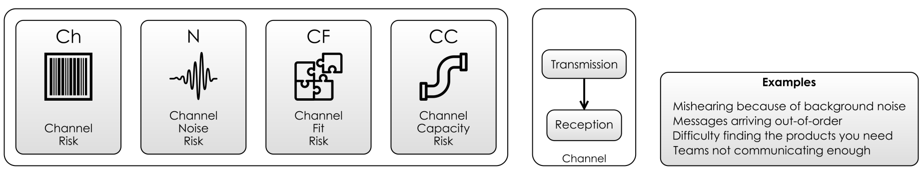


Figure 13.4: Communication Channels

Marketing Communications

When we are talking about a product or a brand, mitigating Channel Risk is the domain of Marketing Communications[6]. 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 looks 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.

[5] https://en.wikipedia.org/wiki/Pair_programming
[6] https://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.

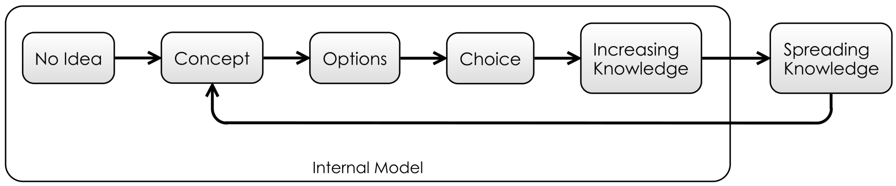


Figure 13.5: Communication Marketing

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.

13.5 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 Protocol[7]. 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:

[7] https://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 DNS[8] to *resolve* the address "google.com" to an IP address. This is some Abstraction: instead of using the machine's IP Address[9] 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 Protocol[10] - 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*.

[8] https://en.wikipedia.org/wiki/Domain_Name_System
[9] https://en.wikipedia.org/wiki/IP_address
[10] https://en.wikipedia.org/wiki/Internet_Protocol

3. 802.11 - WiFi Protocol

I ran this at home, using WiFi, which uses IEEE 802.11 Protocol[11], 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-OFDM[12], 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 TCP[13] 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)
```

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

[12] https://en.wikipedia.org/wiki/MIMO-OFDM

[13] https://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 Code[14]. `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:

[14] https://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 HTML[15] 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:

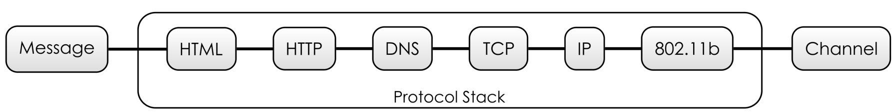


Figure 13.6: Protocol Stack

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 Concerns[16], 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

[16] https://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.

13.6 Protocol Risk

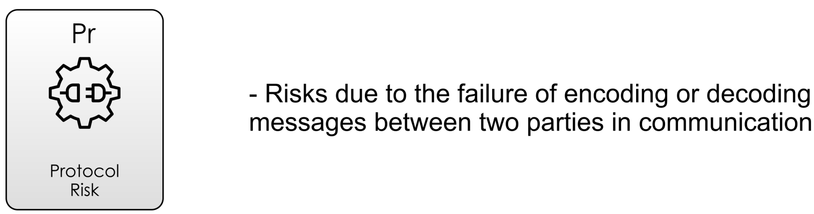


Figure 13.7: Protocol Risk

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 TIFF[17], RAW[18] or JPEG[19], whilst we also have SVG[20] for vector graphics, GIF[21] for images and animations and PNG[22] 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 services[23], 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

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

[19] https://en.wikipedia.org/wiki/JPEG

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

[21] https://en.wikipedia.org/wiki/GIF

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

[23] https://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 its 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.

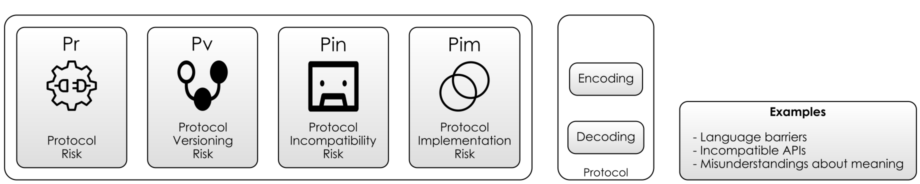


Figure 13.8: Communication Protocols Risks

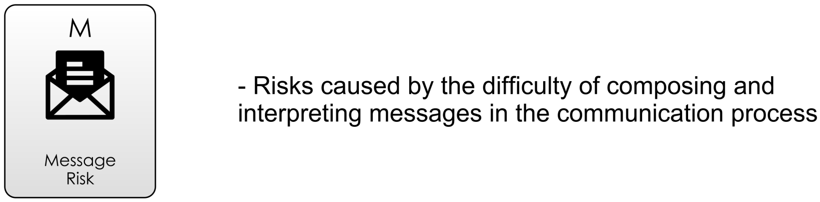


Figure 13.9: Message Risk

13.7 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 Mind[24]: the appreciation that your knowledge is different to other people's, and adjusting you messages accordingly.

When teaching, this is called The Curse Of Knowledge[25]: teachers have difficulty understanding students' problems *because they already understand the subject*. For example, if I want to tell you about a new JDBC Driver[26], 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

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

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

[26] https://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 Brain[27]" 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)

[28] https://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?

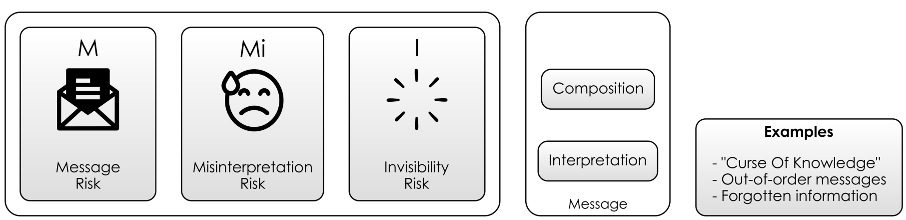


Figure 13.10: Message Risk

13.8 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 Authentication[29] and preventing man-in-the-middle attacks[30]), 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.
- Relativism[32] 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.
- Psycholinguistics[33] is the study of humans aquire 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 Curve[34]" or "Information Overload[35]", 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

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

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

[32] https://en.wikipedia.org/wiki/Relativism

[33] https://en.wikipedia.org/wiki/Psycholinguistics

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

[35] https://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.

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*[36]*)*
- 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.

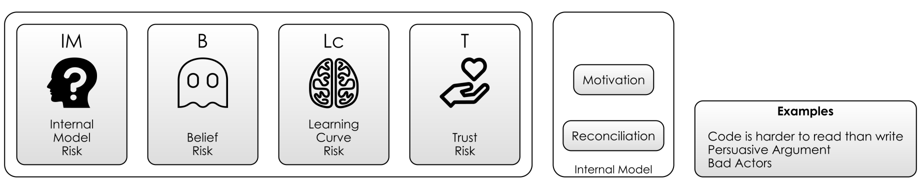


Figure 13.11: Internal Model Risks

13.9 Communication Risk Wrap Up

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

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

this seems complex tbd.

[36] `https://en.wikipedia.org/wiki/Principle_of_least_astonishment`

Figure 13.12: Communication 2

CHAPTER 14

Dependency Risk

Dependency Risk is the risk you take on whenever you have a dependency on something (or someone) else. One simple example could be that the software service you write might depend on a server to run on. If the server goes down, the service goes down too. In turn, the server depends on electricity from a supplier, as well as a network connection from a provider. If either of these dependencies aren't met, the service is out of commission.

Dependencies can be on *events*, *people*, *teams*, *processes*, *software*, *services*, *money*: pretty much *any resource*. Dependencies add risk to any project because the reliability of the project itself is now a function involving the reliability of the dependency.

In order to avoid repetition, and also to break down this large topic, we're going to look at this over 6 chapters:

- In this first chapter will look at dependencies *in general*, and apply our existing risk categorizations to understand Dependency Risk.
- Next, we'll look at Scarcity Risk, because time and money are scarce resources in every project.
- We'll cover Deadline Risk, and discuss the purpose of Events and Deadlines, and how they enable us to co-ordinate around dependency use.
- Then, we'll move on to look specifically at Software Dependency Risk, covering using libraries, software services and building on top of the work of others.
- After, we'll take a look at Process Risk, which is still Dependency Risk, but we'll be considering more organisational factors and how bureaucracy comes into the picture.

- Next, we'll take a closer look at Boundary Risk and Dead-End Risk. These are the risks you face in choosing the wrong things to depend on.
- Finally, we'll wrap up this analysis with a look at some of the specific problems around working with other people or businesses in Agency Risk.

14.1 Why Have Dependencies?

Luckily for us, the things we depend on in life are, for the most part, abundant: water to drink, air to breathe, light, heat and most of the time, food for energy.

This isn't even lucky though: life has adapted to build dependencies on things that it can *rely* on.

Although life exists at the bottom of the ocean around hydrothermal vents[1], it is a very different kind of life to us, and has a different set of dependencies given it's circumstances.

This tells us a lot about Dependency Risk right here:

- On the one hand, depending on something else is very often helpful, and quite often essential. (For example, all animals that *move* seem to depend on oxygen).
- However, as soon as you have dependencies, you need to take into account of their *reliability*. (Living near a river or stream gives you access to fresh water, for example).
- Successful organisms *adapt* to the dependencies available to them (like the thermal vent creatures).
- There is likely to be *competition* for a dependency when it is scarce (think of droughts and famine).

So, dependencies are a trade-off. They give with one hand and take with the other. Our modern lives are full of dependency (just think of the chains of dependency needed for putting a packet of biscuits on a supermarket shelf, for example), but we accept this extra complexity because it makes life *easier*.

tbd. diagram, mitigating feature risk with a dependency?

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

14.2 Simple Made Easy

In Rich Hickey's talk, Simple Made Easy[2] he discusses the difference between *simple* software systems and *easy* (to use) ones, heavily stressing the virtues of simple over easy. It's an incredible talk and well worth watching.

But. Living systems are not simple. Not anymore. They evolved in the direction of increasing complexity because life was *easier* that way. In the "simpler" direction, life is first *harder* and then *impossible*, and then an evolutionary dead-end.

Depending on things makes *your job easier*. It's just division of labour[3] and dependency hierarchies, as we saw in Hierarchies and Modularisation.

Our economic system and our software systems exhibit the same tendency-towards-complexity. For example, the television in my house now is *vastly more complicated* than the one in my home when I was a child. But, it contains much more functionality and consumes much less power and space.

14.3 Reliability Risk

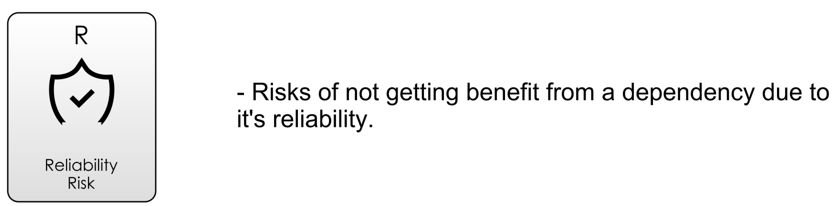


Figure 14.1: Reliability Risk

Deadline Risk is really a kind of reliability issue: if you can understand which parties are unreliable, you have a much better handle on your Deadline Risk.

Luckily, there is quite a lot of existing science around reliability. For example:

- If a component **A** depends on component **B**, unless there is some extra redundancy around **B**, then **A** *can't* be more reliable than **B**.
- Is **A** or **B** a Single Point Of Failure[4] in a system?

[2] https://www.infoq.com/presentations/Simple-Made-Easy

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

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

- Are there bugs in **B** that are going to prevent it working correctly in all circumstances?

This kind of stuff is encapsulated in the science of Reliability Engineering[5]. For example, Failure mode and effects analysis (FEMA)[6]:

> "...was one of the first highly structured, systematic techniques for failure analysis. It was developed by reliability engineers in the late 1950s to study problems that might arise from malfunctions of military systems."
>
> —FEMA, *Wikipedia*[7]

This was applied on NASA missions, and then more recently in the 1970's to car design following the Ford Pinto exploding car[8] affair.

14.4 Communication Risk

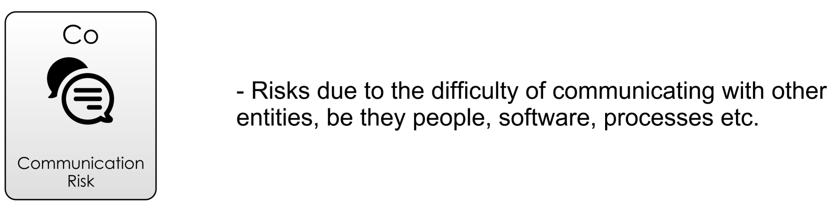


Figure 14.2: Communication Risk

We've already looked at communication risk in a lot of depth, and we're going to go deeper still in Software Dependency Risk, but let's look at some general issues around communicating dependencies. In the Communication Risk chapter we looked at Marketing Communications and talked about the levels of awareness that you could have with dependencies. i.e.

- The concept that there is such a thing as **D** which solves my problem isn't something I'd even considered.

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

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

[7] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis

[8] https://en.wikipedia.org/wiki/Ford_Pinto#Design_flaws_and_ensuing_lawsuits

- 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.

Let's apply this to our Bus scenario:

- Am I aware that there is public transport in my area?
- How do I find out about the different options?
- How do I choose between buses, taxis, cars etc.
- How do I understand the timetable, and apply it to my problem?

Silo Mentality

Finding out about bus schedules is easy. But in a large company, Communication Risk and especially Invisibility Risk are huge problems. This tends to get called "Silo Mentality[9]", that is, ignoring what else is going on in other divisions of the company or "not invented here"[10] syndrome:

> "In management the term silo mentality often refers to information silos in organisations. Silo mentality is caused by divergent goals of different organisational units."
>
> —Silo Mentality, *Wikipedia*[11]

Ironically, *more communication* might not be the answer - if channels are provided to discover functionality in other teams you can still run into Trust Risk (why should I believe in the quality of this dependency?) Or Channel Risk in terms of too low a signal-to-noise ratio, or desperate self-promotion.

Silo Mentality is exacerbated by the problems you would face in *budgeting* if suddenly all the divisions in an organisation started providing dependencies for each other. This starts to require a change to organisational structure towards being a set of individual businesses marketing services to one another, rather than a division-based one. We'll look more closely at these kind of organisational issues in the Coordination Risk chapter.

[9]https://en.wikipedia.org/wiki/Information_silo#Silo_mentality
[10]https://en.wikipedia.org/wiki/Not_invented_here
[11]https://en.wikipedia.org/wiki/Information_silo#Silo_mentality

14.5 Complexity Risk

Dependencies are usually a mitigation for Complexity Risk, and we'll investigate that in much more detail in Software Dependency Risk. The reason for this is that a dependency gives you an abstraction: you no longer need to know *how* to do something, (that's the job of the dependency), you just need to interact with the dependency properly to get the job done. Buses are *perfect* for people who can't drive, after all.

But this means that all of the issues of abstractions that we covered in Communication Risk apply:

- There is Invisibility Risk because you probably don't have a full view of what the dependency is doing. Nowadays, bus stops have a digital "arrivals" board which gives you details of when the bus will arrive, and shops publish their opening hours online. But, abstraction always means the loss of some detail.
- There is Misinterpretation Risk, because often the dependency might mistake your instructions. This is endemic in software, where it's nearly impossible to describe exactly what you want up-front.

14.6 Fit Risk

Sometimes, the bus will take you to lots of in-between places you *didn't* want to go. This is Fit Risk and we saw this already in the chapter on Feature Risk. There, we considered two problems:

- The feature (or now, dependency) doesn't provide all the functionality you need. This was Fit Risk. An example might be the supermarket not stocking everything you wanted.
- The feature / dependency provides far too much, and you have to accept more complexity than you need. This was Conceptual Integrity Risk. An example of this might be the supermarket being *too big*, and you spend a lot longer navigating it than you wanted to.

14.7 Dead-End Risk and Boundary Risk

When you choose something to depend on, you can't be certain that it's going to work out in your favour. Sometimes, the path from your starting point to your goal on the Risk Landscape will take you to dead ends: places where

Figure 14.3: Feature Fit: A Two-Dimensional Problem (at least)

the only way towards your destination is to lose something, and do it again another way. This is Dead End Risk, which we looked at before.

Boundary Risk is another feature of the Risk Landscape: when you make a decision to use one dependency over another, you are picking a path on the risk landscape that *precludes* other choices. After all, there's not really much cost in a Dead End if you've not had to follow a path to get to it.

We're also going to look at Boundary Risk in more detail later, but I want to introduce it here. Here are some examples:

- If I choose to program some software in Java, I will find it hard to integrate libraries from other languages. The dependencies available to Java software are different to those in Javascript, or C#. Having gone down a Java route, there are *higher risks* associated with choosing incompatible technologies. Yes, I can pick dependencies that use C# (still), but I am now facing greater complexity risk than if I'd just chosen to do everything in C# in the first place.
- If I choose one database over another, I am *limited to the features of that database*. This is not the same as a dead-end: I can probably build what I want to build, but the solution will be "bounded" by the dependency choices I make. One of the reasons we have standards like Java Database Connectivity (JDBC)[12] is to mitigate Dead End Risk around databases, so that we can move to a different database later.

[12]https://en.wikipedia.org/wiki/Java_Database_Connectivity

- If I choose to buy a bus ticket, I've made a decision not to travel by train, even though later on it might turn out that the train was a better option. Buying the bus ticket is Boundary Risk: I may be able to get a refund, but having chosen the dependency I've set down a path on the risk landscape.

14.8 Managing Dependency Risk

Arguably, managing Dependency Risk is *what Project Managers do*. Their job is to meet the Goal by organising the available dependencies into some kind of useful order.

There are *some* tools for managing dependency risk: Gantt Charts[13] for example, arrange work according to the capacity of the resources (i.e. dependencies) available, but also the *dependencies between the tasks*. If task **B** requires the outputs of task **A**, then clearly task **A** comes first and task **B** starts after it finishes. We'll look at this more in Process Risk.

We'll look in more detail at project management in the *practices* part, later. But now let's get into the specifics with Scarcity Risk.

[13] https://en.wikipedia.org/wiki/Gantt_chart

CHAPTER 15

Scarcity Risk

Any resource that you depend on can suffer from *scarcity*,

Schedule Risk is the term for risks you face because of *lack of time*.

You could also call this "Chronological Risk" or just "Time Risk" if you wanted to.

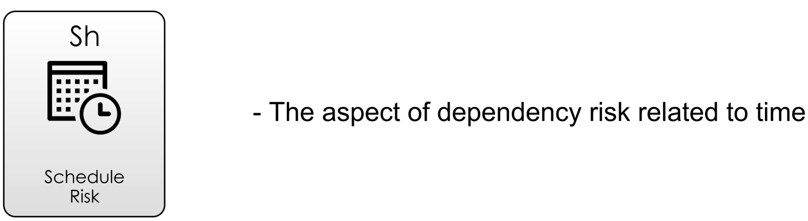


Figure 15.1: Schedule Risk

Schedule Risk is very pervasive, and really underlies *everything* we do. People *want* things, but they *want them at a certain time*. We need to eat and drink every day, for example. We might value having a great meal, but not if we have to wait three weeks for it.

And let's go completely philosophical for a second: Were you to attain immortality, you'd probably not feel the need to buy *anything*. You'd clearly have no *needs*, and anything you wanted, you could create yourself within your infinite time-budget. Rocks don't need money, after all.

Let's look at some specific kinds of Schedule Risk.

15.1 Scarcity Risk

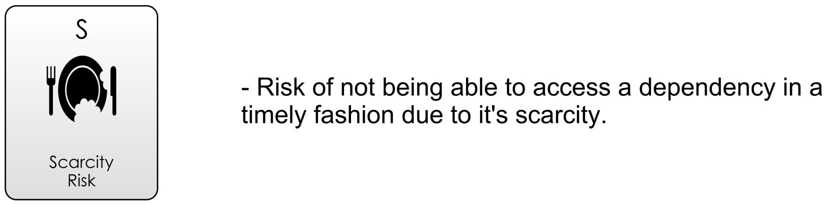


Figure 15.2: Scarcity Risk

Let's get back to the bus (which, hopefully, is still working). What if, when it arrives, it's already full of passengers? Let's term this, Scarcity Risk - the chance that a dependency is over-subscribed and you can't use it the way you want. This is clearly an issue for nearly every kind of dependency: buses, supermarkets, concerts, teams, services and people.

You could also call this *availability risk* or *capacity risk* of the resource. Here are a selection of mitigations:

- **Buffers**: Smoothing out peaks and troughs in utilisation.
- **Reservation Systems**: giving clients information *ahead* of the dependency usage about whether the resource will be available to them.
- **Graceful degradation**: Ensuring *some* service in the event of over-subscription. It would be no use allowing people to cram onto the bus until it can't move.
- **Demand Management**: Having different prices during busy periods helps to reduce demand. Having "first class" seats means that higher-paying clients can get service even when the train is full. Uber[1] adjust prices in real-time by so-called Surge Pricing[2]. This is basically turning Scarcity Risk into a Market Risk problem.
- **Queues**: Again, these provide a "fair" way of dealing with scarcity by exposing some mechanism for prioritising use of the resource. Buses operate a first-come-first-served system, whereas emergency departments in hospitals triage according to need.
- **Pools**: Reserving parts of a resource for particular customers.
- **Horizontal Scaling**: allowing a scarce resource to flexibly scale according to how much demand there is. (For example, putting on extra

[1] https://www.uber.com

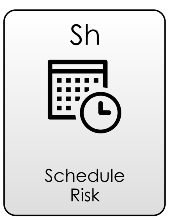
[2] https://www.uber.com/en-GB/drive/partner-app/how-surge-works/

buses when the trains are on strike, or opening extra check-outs at the supermarket.)

Much like Reliability Risk, there is science for it:

- Queue Theory[3] is all about building mathematical models of buffers, queues, pools and so forth.
- Logistics[4] is the practical organisation of the flows of materials and goods around things like Supply Chains[5].
- And Project Management[6] is in large part about ensuring the right resources are available at the right times. We'll be taking a closer look at that in Risk-First Part 3 chapters on Prioritisation and the Project Management Body Of Knowledge[7].

15.2 Schedule Risk



- The aspect of dependency risk related to time.

Figure 15.3: Schedule Risk

By agreeing a *time* and *place* for something to happen, you're introducing Deadline Risk. Miss the deadline, and you miss the bus, or the start of the meeting or get fined for not filling your tax return on time.

As discussed above, *schedules* (such as bus timetables) exist so that *two or more parties can coordinate*, and Deadline Risk is on *all* of the parties. While there's a risk I am late, there's also a risk the bus is late. I might miss the start of a concert, or the band might keep everyone waiting.

[3] https://en.wikipedia.org/wiki/Queueing_theory
[4] https://en.wikipedia.org/wiki/Logistics
[5] https://en.wikipedia.org/wiki/Supply_chain
[6] https://en.wikipedia.org/wiki/Project_management
[7] https://en.wikipedia.org/wiki/Project_Management_Body_of_Knowledge

Each party can mitigate Deadline Risk with *slack*. That is, ensuring that the exact time of the event isn't critical to your plans:

- Don't build into your plans a *need* to start shopping at 9am.
- Arrive at the bus-stop *early*.

The amount of slack you build into the schedule is likely dependent on the level of risk you face: I tend to arrive a few minutes early for a bus, because the risk is *low* (there'll be another bus along soon), however I try to arrive over an hour early for a flight, because I can't simply get on the next flight straight away, and I've already paid for it, so the risk is *high*.

Deadline Risk becomes very hard to manage when you have to coordinate actions with lots of tightly-constrained events. So what else can give? We can reduce the number of *parties* involved in the event, which reduces risk, or, we can make sure all the parties are in the same *place* to begin with.

15.3 Opportunity Risk

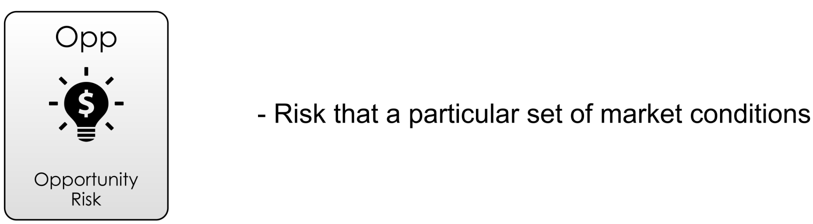


Figure 15.4: Opportunity Risk

Opportunity Risk is really the concern that whatever we do, we have to do it *in time*. If we wait too long, we'll miss the Window Of Opportunity[8] for our product or service.

Any product idea is necessarily of it's time: the Goal In Mind will be based on observations from a particular Internal Model, reflecting a view on reality at a specific *point in time*.

How long will that remain true for? This is your *opportunity*: it exists apart from any deadlines you set yourself, or funding options. It's purely, "how long will this idea be worth doing?"

[8] `https://en.wikipedia.org/wiki/Window_of_opportunity`

With any luck, decisions around *funding* your project will be tied into this, but it's not always the case. It's very easy to under-shoot or overshoot the market completely and miss the window of opportunity.

The iPad

For example, let's look at the iPad[9], which was introduced in 2010 and was hugely successful.

This was not the first tablet computer. Apple had already tried to introduce the Newton[10] in 1989, and Microsoft had released the Tablet PC[11] in 1999. But somehow, they both missed the Window Of Opportunity. Possibly, the window existed because Apple had changed changed the market with their release of the iPhone, which left people open to the idea of a tablet being "just a bigger iPhone".

But maybe now, the iPad's window is closing? We have more *wearable computers* like the Apple Watch[12], and voice-controlled devices like Alexa[13] or Siri[14]. Peak iPad was in 2014, according to this graph[15].

So, it seems Apple timed the iPad to hit the peak of the Window of Opportunity.

But, even if you time the Window Of Opportunity correctly, you might still have the rug pulled from under your feet due to a different kind of Schedule Risk, such as. . .

15.4 Funding Risk

On a lot of software projects, you are "handed down" deadlines from above, and told to deliver by a certain date or face the consequences. But sometimes you're given a budget instead, which really just adds another layer of abstraction to the Schedule Risk: That is, do I have enough funds to cover the team for as long as I need them?

This grants you some leeway as now you have two variables to play with: the *size* of the team, and *how long* you can run it for. The larger the team, the shorter the time you can afford to pay for it.

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

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

[11] https://en.wikipedia.org/wiki/Microsoft_Tablet_PC

[12] https://en.wikipedia.org/wiki/Apple_Watch

[13] https://en.wikipedia.org/wiki/Amazon_Alexa

[14] https://en.wikipedia.org/wiki/Siri

[15] https://www.statista.com/statistics/269915/global-apple-ipad-sales-since-q3-2010/

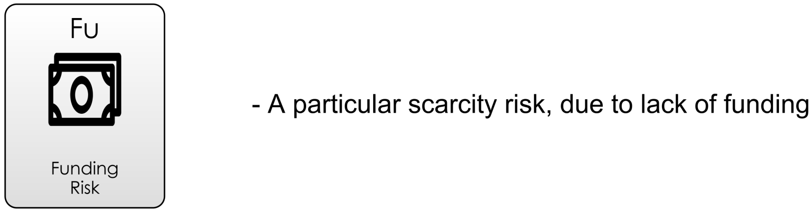


Figure 15.5: Funding Risk

In startup circles, this "amount of time you can afford it" is called the "Runway"[16]: you have to get the product to "take-off" before the runway ends. So you could term this component as "Runway Risk".

Startups often spend a lot of time courting investors in order to get funding and mitigate this type of Schedule Risk. But, this activity usually comes at the expense of Opportunity Risk and Feature Risk, as usually the same people are trying to raise funds as build the project itself.

15.5 Staff Risk

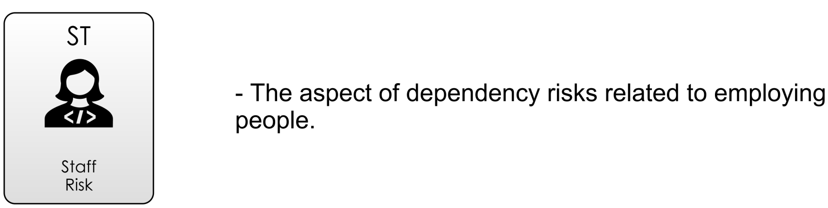


Figure 15.6: Staff Risk

If a startup has a "Runway", then the chances are that the founders and staff do too, as this article explores[17]. It identifies the following risks:

- Company Cash: The **Runway** of the startup itself

[16] https://en.wiktionary.org/wiki/runway

[17] https://www.entrepreneur.com/article/223135

- Founder Cash: The **Runway** for a founder, before they run out of money and can't afford their rent.
- Team Cash: The **Runway** for team members, who may not have the same appetite for risk as the founders do.

You need to consider how long your staff are going to be around, especially if you have Key Man Risk[18] on some of them. People like to have new challenges, or move on to live in new places, or simply get bored. The longer your project goes on for, the more Staff Risk you will have to endure, and you can't rely on getting the best staff for failing projects.

In the chapter on Coordination-Risk we'll look in more detail at the non-temporal components of Staff Risk.

15.6 Red-Queen Risk

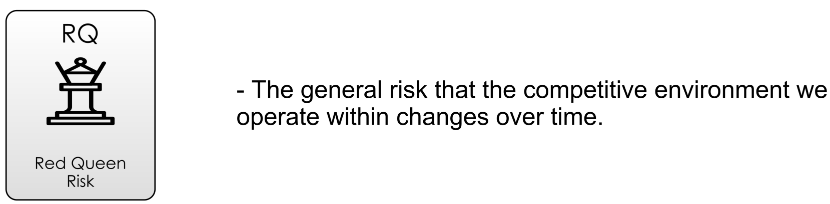


Figure 15.7: Red Queen Risk

A more specific formulation of Schedule Risk is Red Queen Risk, which is that whatever you build at the start of the project will go slowly more-and-more out of date as the project goes on.

This is named after the Red Queen quote from Alice in Wonderland:

> "My dear, here we must run as fast as we can, just to stay in place. And if you wish to go anywhere you must run twice as fast as that." - Lewis Carroll, *Alice in Wonderland*[19]

The problem with software projects is that tools and techniques change *really fast*. In 2011, 3DRealms released Duke Nukem Forever after 15 years in development[20], to negative reviews:

[18] https://en.wikipedia.org/wiki/Key_person_insurance#Key_person_definition

[19] https://www.goodreads.com/quotes/458856-my-dear-here-we-must-run-as-fast-as-we

[20] https://en.wikipedia.org/wiki/Duke_Nukem_Forever

> ... most of the criticism directed towards the game's long loading times, clunky controls, offensive humor, and overall aging and dated design.

Now, they didn't *deliberately* take 15 years to build this game (lots of things went wrong). But, the longer it took, the more their existing design and code-base were a liability rather than an asset.

Personally, I have suffered the pain on project teams where we've had to cope with legacy code and databases because the cost of changing them was too high. And any team who is stuck using Visual Basic 6.0[21] is here. It's possible to ignore Red Queen Risk for a time, but this is just another form of Technical Debt which eventually comes due.

15.7 Schedule Risk and Feature Risk

In the chapter on Feature Risk we looked at Market Risk, the idea that the value of your product is itself at risk from the morés of the market, share prices being the obvious example of that effect. In Finance, we measure this using *money*, and we can put together probability models based on how much money you might make or lose.

With Schedule Risk, the underlying measure is *time*:

- "If I implement feature X, I'm picking up something like 5 days of Schedule Risk."
- "If John goes travelling that's going to hit us with lots of Schedule Risk while we train up Anne."

... and so on. Clearly, in the same way as you don't know exactly how much money you might lose or gain on the stock-exchange, you can't put precise numbers on Schedule Risk either.

Schedule Risk, then, is *fundamental* to every dependency. But now it's time to get into the *specifics*, and look at Software Dependencies.

[21] https://en.wikipedia.org/wiki/Visual_Basic

CHAPTER 16

Deadline Risk

Events as mitigation for coordination risk. diagram.

16.1 Deadline Risk

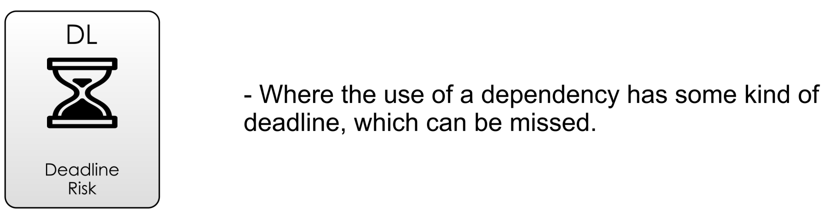


Figure 16.1: Deadline Risk

Often when running a software project, you're given a team of people and told to get something delivered by a certain date. i.e. you have an artificially-imposed Deadline on delivery.

What happens if you miss the deadline? It could be:

- The funding on the project runs out, and it gets cancelled.
- You have to go back to a budgeting committee, and get more money.
- The team gets replaced, because of lack of faith.

.. or something else.

Deadlines can be set by an authority in order to *sharpen focus* and reduce Coordination Risk. This is how we arrive at tools like SMART Objectives[1] and KPI's (Key Performance Indicators)[2]. Time scales change the way we evaluate goals, and the solutions we choose.

In JFK's quote:

> First, I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the Earth.

The 9-year timespan came from an authority figure (the president) and helped a huge team of people coordinate their efforts and arrive at a solution that would work within a given time-frame.

Compare with this quote:

> "I love deadlines. I love the whooshing noise they make as they go by." - Douglas Adams[3]

As a successful author, Douglas Adams *didn't really care* about the deadlines his publisher's gave him. The Deadline Risk was minimal for him, because the publisher wouldn't be able to give his project to someone else to complete.

Sometimes, deadlines are set in order to *coordinate work between teams*. The classic example being in a battle, to coordinate attacks. When our deadlines are for this purpose, we're heading towards Coordination Risk territory.

Student Syndrome

Student Syndrome[4] is, according to Wikipedia:

> Student syndrome refers to planned procrastination, when, for example, a student will only start to apply themselves to an assignment at the last possible moment before its deadline.

Arguably, there is good psychological, evolutionary and risk-based reasoning behind procrastination: the further in the future the Deadline Risk is, the

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

[2] https://en.wikipedia.org/wiki/Performance_indicator

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

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

more we discount it. If we're only ever mitigating our *biggest risks*, then deadlines in the future don't matter so much, do they? And, putting efforts into mitigating future risks that *might not arise* is wasted effort.

Or at least, that's the argument. If you're Discounting the Future To Zero then you'll be pulling all-nighters in order to deliver any assignment.

So, the problem with Student Syndrome is that the *very mitigation* for Schedule Risk (allowing more time) is an Attendant Risk that *causes* Schedule Risk: you'll work towards the new, generous deadline more slowly, and you'll end up revealing Hidden Risk *later* than you would have with the original, pressing deadline . . . and you end up being late because of them.

We'll look at mitigations for this in Part 3's chapter on Prioritisation.

16.2 Event Dependencies

Let's start with dependencies on *events*.

We rely on events occuring all the time in our lives, and so this is a good place to start in our analysis of Dependency Risk generally. And, as we will see, all the risks that apply to events pretty much apply to all the other kinds of dependencies we'll look at.

Arguably, the event dependencies are the simplest to express, too: usually, a *time* and a *place*. For example: - "I can't start shopping until the supermarket opens at 9am", or - "I must catch my bus to work at 7:30am".

In the first example, you can't *start* something until a particular event happens. In the latter example, you must *be ready* for an event at a particular time.

Events Mitigate Risk. . .

Having an event occur in a fixed time and place is mitigating risk:

- By taking the bus, we are mitigating our own Schedule Risk: we're (hopefully) reducing the amount of time we're going to spend on the activity of getting to work.
- Events are a mitigation for Coordination Risk: A bus needn't necessarily *have* a fixed timetable: it could wait for each passenger until they turned up, and then go. (A bit like ride-sharing works). This would be a total disaster from a Coordination Risk perspective, as one person could cause everyone else to be really really late. Having a fixed time for doing something mitigates Coordination Risk by turning it into Schedule

Risk. Agreeing a date for a product launch, for example, allows lots of teams to coordinate their activities.
- It's not entirely necessary to even take the bus: you could walk, or go by another form of transport. But, effectively, this just swaps one dependency for another: if you walk, this might well take longer and use more energy, so you're just picking up Schedule Risk in another way. If you drive, you have a dependency on your car instead. So, there is often an *opportunity cost* with dependencies. Using the bus might be a cheap way to travel. You're therefore imposing less Dependency Risk on a different scarce resource - your money.

But, Events Lead To Attendant Risk

By *deciding to use the bus* we've Taken Action.

Figure 16.2: Action Diagram showing risks mitigated by having an **event**

However, as we saw in A Simple Scenario, this means we pick up Attendant Risks.

So, we're going to look at Dependency Risk for our toy events (bus, supermarket) from 7 different perspectives, many of which we've already touched on in the other chapters.

- Scarcity Risk
- Schedule Risk
- Reliability Risk
- Communication Risk
- Complexity Risk
- Feature Fit Risk
- Dead-End Risk and Boundary Risk

(Although you might be able to think of a few more.)

Let's look at each of these in turn.

CHAPTER 17

Software Dependency Risk

In this chapter, we're going to look specifically at *Software* dependencies, although many of the concerns we'll raise here apply equally to all the other types of dependency we outlined in Dependency Risk.

17.1 Kolmogorov Complexity: Cheating

In the earlier chapter on Complexity Risk we tackled Kolmogorov Complexity, and the idea that your codebase had some kind of minimal level of complexity based on the output it was trying to create. This is a neat idea, but in a way, we cheated. Let's look at how.

We were trying to figure out the shortest (Javascript) program to generate this output:

```
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

And we came up with this:

```
const ABCD="ABCD";                          (11 symbols)

function out() {                            (7 symbols)
    return ABCD.repeat(10)                  (7 symbols)
}                                           (1 symbol)
```

Which had **26** symbols in it.

Now, here's the cheat: The `repeat()` function was built into Javascript in 2015 in ECMAScript 6.0[1]. If we'd had to program it ourselves, we might have

[1] http://www.ecma-international.org/ecma-262/6.0/

added this:

```
function repeat(s,n) {                          (10 symbols)
    var a=[];                                   (7 symbols)
    while(a.length<n){                          (9 symbols)
        a.push(s)                               (6 symbols)
    }                                           (1 symbol)
    return a.join('');                          (10 symbols)
}                                               (1 symbol)
```

. . . which would be an extra **44** symbols (in total **70**), and push us completely over the original string encoding of **53** symbols. So, *encoding language is important*.

Conversely, if ECMAScript 6.0 had introduced a function called `abcdRepeater(n)` we'd have been able to do this:

```
function out() {                                (7 symbols)
    return abcdRepeater(10)                     (6 symbols)
}                                               (1 symbol)
```

.. and re-encode to **14** symbols. Now, clearly there are some problems with all this:

1. Clearly, *language matters*: the Kolmogorov complexity is dependent on the language, and the features the language has built in.
2. The exact Kolmogorov complexity is uncomputable anyway (it's the *theoretical* minimum program length). It's just a fairly abstract idea, so we shouldn't get too hung up on this. There is no function to be able to say, "what's the Kolmogorov complexity of string X"
3. What is this new library function we've created? Is `abcdRepeater` going to be part of *every* Javascript? If so, then we've shifted Codebase Risk away from ourselves, but we've pushed Communication Risk and Dependency Risk onto every *other* user of Javascript. (Why these? Because `abcdRepeater` will be clogging up the documentation and other people will rely on it to function correctly.)
4. Are there equivalent functions for every single other string? If so, then compilation is no longer a tractable problem: is `return abcdRepeater(10)` correct code? Well, now we have a massive library of different `XXXRepeater` functions to compile against to see if it is. . . So, what we *lose* in Kolmogorov Complexity we gain in Runtime Complexity.

5. Language design, then, is about *ergonomics*. After you have passed the relatively low bar of providing Turing Completeness[2], the key is to provide *useful* features that enable problems to be solved, without over-burdening the user with features they *don't* need. And in fact, all software is about this.

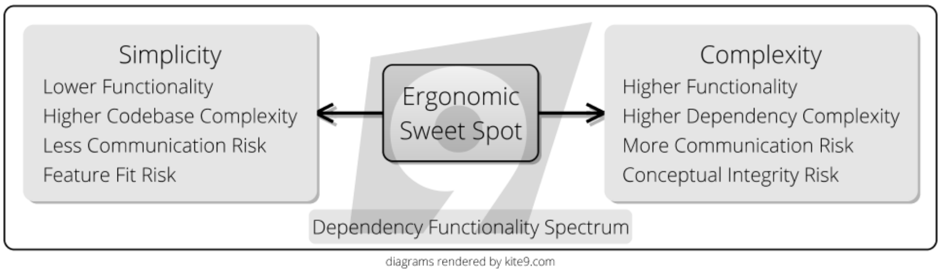


Figure 17.1: Software Dependency Ergonomics: finding the sweet spot between too many features and too few

17.2 Ergonomics Examined

Have a look at some physical tools, like a hammer, or spanner. To look at them, they are probably *simple* objects, obvious, strong and dependable. Their entire behaviour is encapsulated in their form. Now, if you have a drill or sander to hand, look at the design of this too. If it's well-designed, then from the outside it is simple, perhaps with only one or two controls. Inside, it is complex and contains a motor, perhaps a transformer, and is maybe made of a hundred different components.

But outside, the form is simple, and designed for humans to use. This is *ergonomics*[3]:

> "Human factors and ergonomics (commonly referred to as Human Factors), is the application of psychological and physiological principles to the (engineering and) design of products, processes, and systems. The goal of human factors is to reduce human error, increase productivity, and enhance safety and comfort with a specific focus on the interaction between the human and the thing of interest."
>
> —Human Factors and Ergonomics, *Wikipedia*[4]

[2] `https://en.wikipedia.org/wiki/Turing_completeness`

[3] `https://en.wikipedia.org/wiki/Human_factors_and_ergonomics`

[4] `https://en.wikipedia.org/wiki/Human_factors_and_ergonomics`

Interfaces

The interface of a tool is the part we touch and interact with. By striving for simplicity, the interface reduces Communication Risk.

The interface of a system expands when you ask it to do a wide variety of things. An easy-to-use drill does one thing well: it turns drill-bits at useful levels of torque for drilling holes and sinking screws. But if you wanted it to also operate as a lathe, a sander or a strimmer (all basically mechanical things going round) you would have to sacrifice the ergonomic simplicity for a more complex interface, probably including adapters, extensions, handles and so on.

So, we now have split complexity into two: - The inner complexity of the tool (how it works internally, it's own Kolmogorov Complexity). - The complexity of the instructions that we need to write to make the tool work (the interface Kolmogorov Complexity).

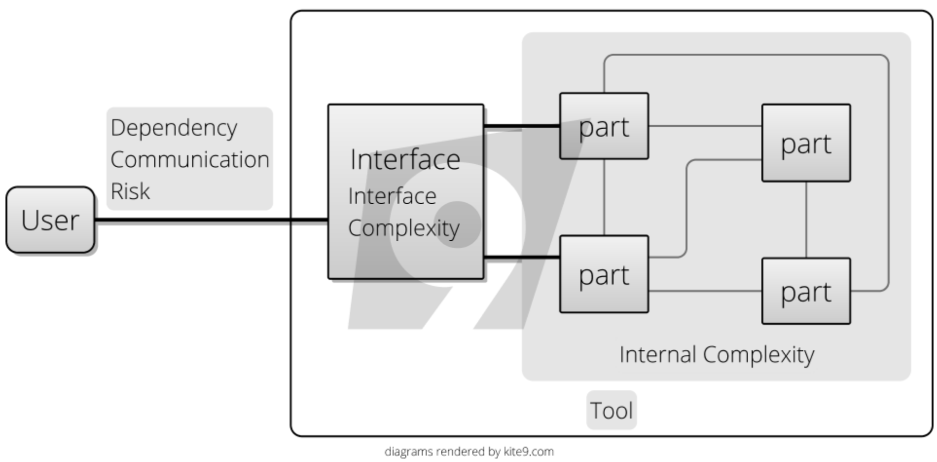


Figure 17.2: Types of Complexity For a Software Dependency

Software Tools

In the same way as with a hand-tool, the bulk of the complexity of a software tool is hidden behind it's interface. But, the more complex the *purpose* of the tool, the more complex the interface will be.

Software is not constrained by *physical* ergonomics in the same way as a tool is. But ideally, it should have conceptual ergonomics: ideally, complexity is hidden away from the user behind the Application Programming Interface (API)[5]. This is the familiar concept of Abstraction we've already looked at.

That is, the tool should be as simple to use and understand as possible. This is the Principal Of Least Astonishment:

- **The abstractions should map easily to how the user expects the tool to work.** For example, I *expect* the trigger on a drill to start the drill turning.
- **The abstractions should leverage existing idioms and knowledge.** In a new car, I *expect* to know what the symbols on the dashboard mean, because I've driven other cars.
- **The abstractions provide me with only the functions I need.** Because everything else is confusing and gets in the way.

The way to win, then, is to allow a language to be extensible as-needed with features written by third parties. By supplying mechanisms for extension a language can provide insurances against the Boundary Risk of adopting it.

17.3 Types Of Software Dependencies

There are lots of ways you can depend on software. Here though, we're going to focus on just three main types: 1. **Code Your Own**: write some code ourselves to meet the dependency. 2. **Software Libraries**: importing code from the Internet, and using it in our project. Often, libraries are Open Source (this is what we'll consider here). 3. **Software as a Service**: calling a service on the Internet, (probably via `http`) This is often known as SaaS, or Software as a Service[6].

All 3 approaches involve a different risk-profile. Let's look at each in turn, from the perspective of which risks get mitigated, and which risks are accentuated.

1. Code Your Own

Initially, writing our own code was the only game in town: when I started programming, you had a user guide, BASIC and that was pretty much it. Tool support was very thin-on-the-ground. Programs and libraries could be distributed as code snippets *in magazines* which could be transcribed and run, and added to your program. This spirit lives on somewhat in StackOverflow

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

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

and JSFiddle, where you are expected to "adopt" others' code into your own project.

One of the hidden risks of embarking on a code-your-own approach is that the features you need are *not* apparent from the outset. What might appear to be a trivial implementation of some piece of functionality can often turn into it's own industry as more and more hidden Feature Risk is uncovered.

For example, as we discussed in our earlier treatment of Dead-End Risk, building log-in screens *seemed like a good idea*. However, this gets out-of-hand fast when you need: - A password reset screen - To email the reset links to the user - An email verification screen - A lost account screen - Reminders to complete the sign up process - . . . and so on.

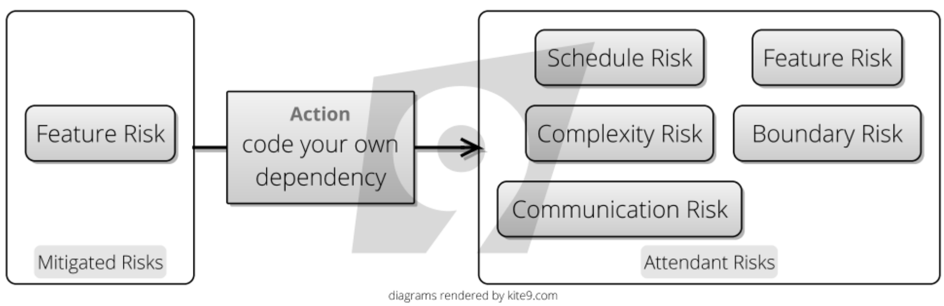


Figure 17.3: Code-Your-Own mitigates immediate feature risk, but at the expense of schedule risk, complexity risk and communication risk. There is also a hidden risk of features you don't yet know you need.

Unwritten Software

Sometimes, you will pick up a dependency on *unwritten software*. This commonly happens when work is divided amongst team members, or teams.

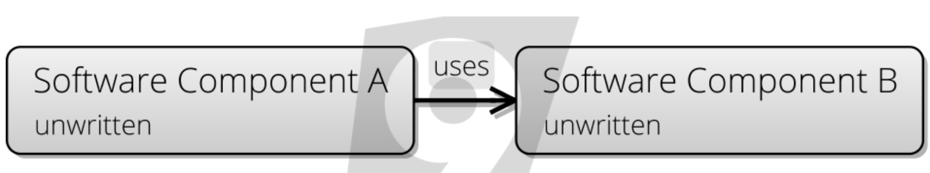


Figure 17.4: Sometimes, a module you're writing will depend on unwritten code

If a component **A** of our project *depends* on **B** for some kind of processing, you might not be able to complete **A** before writing **B**. This makes *scheduling* the project harder, and if component **A** is a risky part of the project, then the chances are you'll want to mitigate risk there first.

But it also hugely increases Communication Risk because now you're being asked to communicate with a dependency that doesn't really exist yet, *let alone* have any documentation.

There are a couple of ways to do this:

- **Standards**: If component **B** is a database, a queue, mail gateway or something else with a standard interface, then you're in luck. Write **A** to those standards, and find a cheap, simple implementation to test with. This gives you time to sort out exactly what implementation of **B** you're going for. This is not a great long-term solution, because obviously, you're not using the *real* dependency- you might get surprised when the behaviour of the real component is subtly different. But it can reduce Schedule Risk in the short-term.
- **Coding To Interfaces**: If standards aren't an option, but the surface area of **B** that **A** uses is quite small and obvious, you can write a small interface for it, and work behind that, using a Mock[7] for **B** while you're waiting for finished component. Write the interface to cover only what **A** *needs*, rather than everything that **B** *does* in order to minimise the risk of Leaky Abstractions[8].

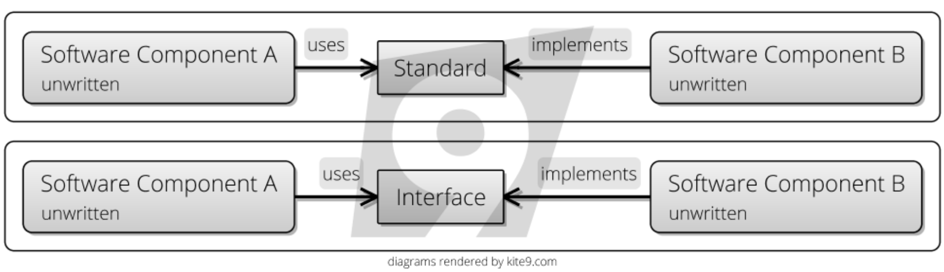


Figure 17.5: Coding to a standard on an interface breaks the dependency on unwritten software

Conway's Law

If the dependency is being written by another person, another team or in another country, communication risks pile up. When this happens, you

[7] https://en.wikipedia.org/wiki/Mock_object

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

will want to minimise *as much as possible* the interface complexity, since the more complex the interface, the worse the Communication Risk will be. The tendency then is to make the interfaces between teams or people *as simple as possible*, modularising along these organisational boundaries.

In essence, this is Conway's Law[9]:

> organisations which design systems ... are constrained to produce designs which are copies of the communication structures of these organisations.

2. Software Libraries

By choosing a particular software library, we are making a move on the Risk Landscape in the hope of moving to place with more favourable risks. Typically, using library code offers a Schedule Risk and Complexity Risk Silver Bullet. But, in return we expect to pick up: - Communication Risk: because we now have to learn how to communicate with this new dependency. - Boundary Risk - because now are limited to using the functionality provided by this dependency. We have chosen it over alternatives and changing to something else would be more work and therefore costly.

But, it's quite possible that we could wind up in a worse place than we started out, by using a library that's out-of-date, riddled with bugs or badly supported. i.e. Full of new, hidden Feature Risk.

It's *really easy* to make bad decisions about which tools to use because the tools don't (generally) advertise their deficiencies. After all, they don't generally know how *you* will want to use them.

Software Libraries - Hidden Risks

Currently, choosing software dependencies looks like a "bounded rationality"-type process:

> "Bounded rationality is the idea that when individuals make decisions, their rationality is limited by the tractability of the decision problem, the cognitive limitations of their minds, and the time available to make the decision."
>
> —Bounded Rationality, *Wikipedia*[10]

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

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

Unfortunately, we know that most decisions *don't* really get made this way. We have things like Confirmation Bias[11] (looking for evidence to support a decision you've already made) and Cognitive Inertia[12] (ignoring evidence that would require you to change your mind) to contend with.

But, leaving that aside, let's try to build a model of what this decision making process *should* involve. Luckily, other authors have already considered the problem of choosing good software libraries, so let's start there.

In the table below, I am summarising three different sources (linked at the end of the chapter), which give descriptions of which factors to look for when choosing open-source libraries.

Some take-aways:

- Feature Risk is a big concern. How can you be sure that the project will do what you want it to do ahead of schedule? Will it contain bugs or missing features? By looking at factors like *release frequency* and *size of the community* you get a good feel for this which is difficult to fake.
- Boundary Risk is also very important. You are going to have to *live* with your choices for the duration of the project, so it's worth spending the effort to either ensure that you're not going to regret the decision, or that you can change direction later.
- Third is Communication Risk: how well does the project deal with it's users? If a project is "famous", then it has communicated its usefulness to a wide, appreciative audience. Avoiding Communication Risk is also a good reason to pick *tools you are already familiar with*.

Complexity Risk?

One thing that none of the sources consider (at least from the outset) is the Complexity Risk of using a solution: - Does it drag in lots of extra dependencies that seem unnecessary for the job in hand? If so, you could end up in Dependency Hell[13], with multiple, conflicting versions of libraries in the project. - Do you already have a dependency providing this functionality? So many times, I've worked on projects that import a *new* dependency when some existing (perhaps transitive) dependency has *already brought in the functionality*. For example, there are plenty of libraries for JSON[14] marshaling, but if I'm also using a web framework the chances are it already has a dependency on one already. - Does it contain lots of functionality that isn't relevant

[11] https://en.wikipedia.org/wiki/Confirmation_bias
[12] https://en.wikipedia.org/wiki/Cognitive_inertia
[13] https://en.wikipedia.org/wiki/Dependency_hell
[14] https://en.wikipedia.org/wiki/JSON

	Co Coordination Risk	B Boundary Risk	F Feature Risk	Co Communication Risk	**Sources**
Is the project "famous"?		☒	☒	☒	[sd2] [sd3]
Is there evidence of a large user community on user forums or e-mail list archives?		☒	☒	☒	[sd1]
Who is developing and maintaining the project? (Track Record)		☒	☒	☒	[sd3]
What are the mechanisms for supporting the software (community support, direct email, dedicated support team), and how long will the support be available? The more support, the better			☒	☒	[sd3]
Is the API to your liking?				☒	[sd2]
Are there examples of using the software successfully in the manner you want to use it?			☒		[sd1]
Are all the features you need, and think you will need, included?			☒		[sd1]
How mature is the project?			☒		[sd2]
In respect to the software licence, do you have the right to use the software in its intended production environment, or the right to distribute it along with your software?			☒		[sd1]
What is its deprecation or versioning policy? Does it have one? If not then it may be more unstable and features may disappear without warning between versioning, especially if releases are frequent.		☒	R Regression Risk	☒	[sd1]
What does the codebase look like?			Im Implementation Risk		[sd1]
How frequent are its releases?			☒		[sd1] [sd2] [sd3]
How well documented is the project?				☒	[sd2]
Does the software have evidence of a sustainable future (e.g. is there a roadmap)?				☒	[sd2]
Does the software support open standards? If it does, it will be easier to replace the software should it come to the end of its lifetime		☒			[sd1]
Does the version you intend to use come from a forked open-source project, or is it from the original source project? If so, which source is more appropriate?		☒			[sd1]
Are there any alternatives to the software?		☒			[sd1]
Has your community converged on using a particular software package?	☒				[sd1]
Totals	**1**	**9**	**15**	**8**	

Figure 17.6: Software Dependencies

to the task you want it to accomplish? e.g. Using Java when a shell script would do (on a non-Java project)

To give an extreme example of this, I once worked on an application which used Hazlecast[15] to cache log-in session tokens for a 3rd party data-source. But, the app is only used once every month, and session IDs can be obtained in milliseconds. So... why cache them? Although Hazlecast is an excellent choice for in-memory caching across multiple JVMs, it is a complex piece of software (after all, it does lots of stuff). By doing this, you have introduced extra dependency risk, cache invalidation risks, networking risks, synchronisation risks and so on, for actually no benefit at all... Unless, it's about CV Building.

Sometimes, the amount of complexity *goes up* when you use a dependency for *good reason*. For example, in Java, you can use Java Database Connectivity (JDBC) to interface with various types of database. Spring Framework[16] (a popular Java library) provides a thing called a `JDBCTemplate`. This actually makes your code *more* complex, and can prove very difficult to debug. However, it prevents some security issues, handles resource disposal and makes database access more efficient. None of those are essential to interfacing with the database, but not using them is Technical Debt that can bite you later on.

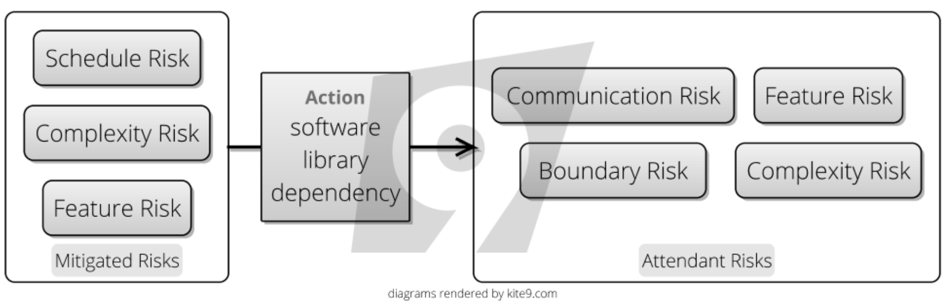


Figure 17.7: Software Libraries Risk Tradeoff

3. Software as a Service

Businesses opt for Software as a Service (SaaS) because: - It vastly reduces the Complexity Risk they face in their organisations. e.g. managing the software or making changes to it. - Payment is usually based on *usage*, mitigating Schedule Risk. e.g. Instead of having to pay for in-house software administrators, they can leave this function to the experts. - Potentially, you

[15] `https://en.wikipedia.org/wiki/Hazelcast`

[16] `https://en.wikipedia.org/wiki/Spring_Framework`

out-source the Operational Risk to a third party. e.g. ensuring availability, making sure data is secure and so on.

SaaS is now a very convenient way to provide *commercial* software. Popular examples of SaaS might be SalesForce[17], or GMail[18]. Both of which follow the commonly-used Freemium[19] model, where the basic service is provided free, but upgrading to a paid account gives extra benefits.

By providing the software on their own servers, the commercial organisation has a defence against *piracy*, as well as being able to control the Complexity Risk of the their environment (e.g. not having to support *every* version of the software that's ever been released).

Let's again recap the risks raised in some of the available literature:

Some take-aways:

- Clearly, Operational Risk is now a big concern. By depending on a third-party organisation you are tying yourself to its success or failure in a much bigger way than just by using a piece of open-source software. What happens to data security, both in the data centre and over the Internet?
- With Feature Risk you now have to contend with the fact that the software will be upgraded *outside your control*, and you may have limited control over which features get added or changed.
- Boundary Risk is a also a different proposition: you are tied to the software provider by *a contract*. If the service changes in the future, or isn't to your liking, you can't simply fork the code (like you could with an open source project).

17.4 A Matrix of Options

We've looked at just 3 different ways of providing a software dependency: SaaS, Libraries and code-your-own.

But these are not the only ways to do it, and there's clearly no one *right* way. Although here we have looked just at "Commercial SaaS" and "Free Open Source", in reality, these are just points in a two-dimensional space involving *Pricing* and *Hosting*.

Let's expand this view slightly and look at where different pieces of software sit on these axes:

[17] https://en.wikipedia.org/wiki/Salesforce.com

[18] https://en.wikipedia.org/wiki/Gmail

[19] https://en.wikipedia.org/wiki/Freemium

	Op Operational Risk	B Boundary Risk	F Feature Risk	Co Communication Risk	Sh Schedule Risk	**Sources**
How does the support process hold up in your trial runs?	☒					[sd4]
What's the backup plan? (It's vital that you understand how your data are protected, and what redundancies are available should your SaaS provider have an outage.)	☒	☒				[sd4] [sd5]
What happens to your data if you sever ties with the vendor?		☒				[sd4]
Are your current and future user environments supported? (e.g. Browser Compatibility)		☒	☒			[sd4]
Can you test in parallel? (i.e. run existing and new SaaS system together)			☒			[sd4]
How does functionality compare to maturity?			☒			[sd4]
What's the pricing model? (What might cause a price increase?)		☒			☒	[sd4]
What migration and training assistance options are available?				☒		[sd4]
What integration options are available? (Are there APIs you can use to get at your data?)				☒		[sd5]
Security (What standards and controls are in place?)	☒					[sd5]
Service Level Agreements (SLAs) (What are the guarantees? What happens when the service levels are not met?)	☒					[sd5]
Global Reach. (Is the service usable everywhere you need it?)	☒					[sd5]
Totals	**5**	**4**	**3**	**2**	**1**	

Figure 17.8: Software As A Service Dependencies

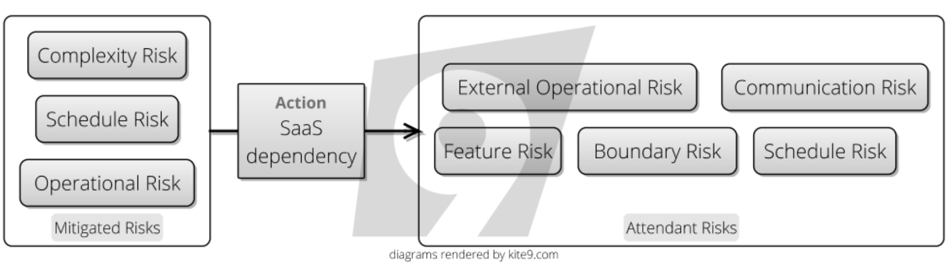


Figure 17.9: Risk Tradeoff From Using _Software as a Service (SaaS)

Pricing	On Premises 3rd Party	In Cloud / Browser 3rd Party	Risk Profile
Free	**OSS Libraries** • Tools • Java • Firefox • Linux • Programming Languages	**Freemium** • Splunk • Spotify • GitHub	• Low Boundary Risk Drives Adoption • Value In Network Effect
Advertising Supported	**Commercial Software** • Phone Apps • e.g. Angry Birds	**Commercial SaaS** • Google Search • GMail • Twitter	• Low Boundary Risk • High Availability Of Substitutes
Monthly / Metered Subscription	**Commercial Software** • Oracle Database • Windows • Office	**Commercial SaaS** • Office 365 • Amazon Web Services • SalesForce	Easy arguments for reduced: • Complexity Risk • Communication Risk • Coordination Risk Higher Boundary Risk

Figure 17.10: Software Dependencies, Pricing, Delivery Matrix Risk Profiles

- Where there is value in the Network Effect[20], it's often a sign that the software will be free, or open source: programming languages and Linux are the obvious examples of this. Bugs are easier to find when there are lots of eyes looking, and learning the skill to use the software has less Boundary Risk if you know you'll be able to use it at any point in the future.
- At the other end of the spectrum, clients will happily pay for software if it clearly **reduces complexity**. Take Amazon Web Services (AWS)[21]. The essential trade here is that you substitute the complexity of hosting and maintaining various pieces of software, in exchange for monthly payments (Funding Risk for you). Since the AWS *interfaces* are specific to Amazon, there is significant Boundary Risk in choosing this option.
- In the middle there are lots of **substitute options** and therefore high competition. Because of this, prices are pushed towards zero, and and therefore often advertising is used to monetarise the product. Angry Birds[22] is a classic example: initially, it had demo and paid versions, however Rovio[23] discovered there was much more money to be made through advertising than from the paid-for app[24].

Managing Risks

So far, we've considered only how the different approaches to Software Dependencies change the landscape of risks we face to mitigate some Feature Risk or other.

But with Software Dependencies we can construct dependency networks to give us all kinds of features and mitigate all kinds of risk. That is, *the features we are looking for are to mitigate some kind of risk*.

For example, I might start using WhatsApp[25] for example, because I want to be able to send my friends photos and text messages. However, it's likely that those same features are going to allow us to mitigate Communication Risk and Coordination Risk when we're next trying to meet up.

Let's look at some:

[20] https://en.wikipedia.org/wiki/Network_effect

[21] https://en.wikipedia.org/wiki/Amazon_Web_Services

[22] https://en.wikipedia.org/wiki/Angry_Birds

[23] https://en.wikipedia.org/wiki/Rovio_Entertainment

[24] https://www.deconstructoroffun.com/blog/2017/6/11/how-angry-birds-2-multiplied-quadrupled-revenue-in-a-year

[25] https://en.wikipedia.org/wiki/WhatsApp

Risk	Examples of Software Mitigating That Risk
Coordination Risk	Calendar tools, Bug Tracking, Distributed Databases
Map-And-Territory-Risk	The Internet, generally. Excel, Google, "Big Data", Reporting tools
Schedule-Risk	Planning Software, Project Management Software
Communication-Risk	Email, Chat tools, CRM tools like SalesForce, Forums, Twitter, Protocols
Process-Risk	Reporting tools, online forms, process tracking tools
Agency-Risk	Auditing tools, transaction logs, Time-Sheet software, HR Software
Operational-Risk	Support tools like ZenDesk, Grafana, InfluxDB, Geneos
Feature-Risk	Every piece of software you use!

17.5 Back To Ergonomics

What's clear from this analysis is that software dependencies don't *conquer* any risk - the moves they make on the Risk Landscape are *subtle*. Whether or not you end up in a more favourable position risk-wise is going to depend heavily on the quality of the execution and the skill of the implementor.

In particular, *choosing* dependencies can be extremely difficult. As we discussed above, the usefulness of any tool depends on its fit for purpose, it's *ergonomics within a given context*. It's all too easy to pick a good tool for the wrong job:

> "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."
>
> —Abraham Maslow, *Toward a Psychology of Being*[26] <!– tweet-end –>

With software dependencies, we often have to live with the decisions we make for a long time. In my experience, given the Boundary Risks associated with getting this wrong, not enough time is spent really thinking about this in advance.

[26] https://en.wiktionary.org/wiki/if_all_you_have_is_a_hammer,_everything_looks_like_a_nail

Let's take a closer look at this problem in the next chapter, Boundary Risk.

Sources
sd1 - Defending your code against dependency problems[27]
sd2 - How to choose an open source library[28]
sd3 - Open Source - To use or not to use[29]
sd4 - SaaS Checklist - Nine Factors to Consider[30]
sd5 - How to Evaluate SaaS Vendors[31]

[27] https://www.software.ac.uk/resources/guides/defending-your-code-against-dependency-problems

[28] https://stackoverflow.com/questions/2960371/how-to-choose-an-open-source-library

[29] https://www.forbes.com/sites/forbestechcouncil/2017/07/20/open-source-to-use-or-not-to-use-and-how-to-choose/2/#39e67e445a8c

[30] https://www.zdnet.com/article/saas-checklist-nine-factors-to-consider-when-selecting-a-vendor/

[31] http://sandhill.com/article/how-to-evaluate-saas-vendors-five-key-considerations/

CHAPTER 18

Process Risk

Process Risk, as we will see, is the risk you take on whenever you embark on completing a *process*.

> "**Process:** A process is a set of activities that interact to achieve a result."
>
> —Process, *Wikipedia*[1]

In the software development world (and the business world generally) processes commonly involve *forms*: If you're filling out a form (whether on paper or on a computer) then you're involved in a process of some sort, whether an "Account Registration" process, "Loan Application" process or "Consumer Satisfaction Survey" process. But sometimes, they involve events occurring: a build process[2] might start after you commit some code, for example.

18.1 The Purpose Of Process

Process exists to mitigate other kinds of risk, and for this reason, we'll be looking at them again in Part 3: Practices, where we'll look at how you can design processes to mitigate risks on your own project.

Until we get there, let's look at some examples of how process can mitigate other risks:

- Coordination Risk: You can often use process to help people coordinate. For example, a Production Line[3] is a process where work being done

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

[2] https://en.wikipedia.org/wiki/Software_build

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

by one person is pushed to the next person when it's done. A meeting booking process ensures that people will all attend a meeting together at the same place and time, and that a room is available for it.

- Dependency Risk: You can use processes to make dependencies explicit and mitigate dependency risk. For example, a process for hiring a car will make sure there is a car available at the time you need it. Alternatively, if we're processing a loan application, we might need evidence of income or bank statements. We can push this Dependency Risk onto the person asking for the loan, by making it part of the process and not accepting the application until this has been provided.
- Complexity Risk: Working *within a process* can reduce the amount of Complexity you have to deal with. We accept that processes are going to slow us down, but we appreciate the reduction in risk this brings. (They allow us to trade Complexity for Schedule risk). For example, setting up a server might be complex, but filling in a form to do the job might simplify things. Clearly, the complexity hasn't gone away, but it's hidden within the process. Process therefore can provide Abstraction. mcdonalds. tbd
- Operational Risk: Operational Risk encompasses the risk of people *not doing their job properly*. But, by having a process, (and asking, did this person follow the process?) you can draw a distinction between a process failure and a personnel failure. For example, making a loan to a money launderer *could* be a failure of the loan agent. But, if they followed the *process*, it's a failure of the Process itself.

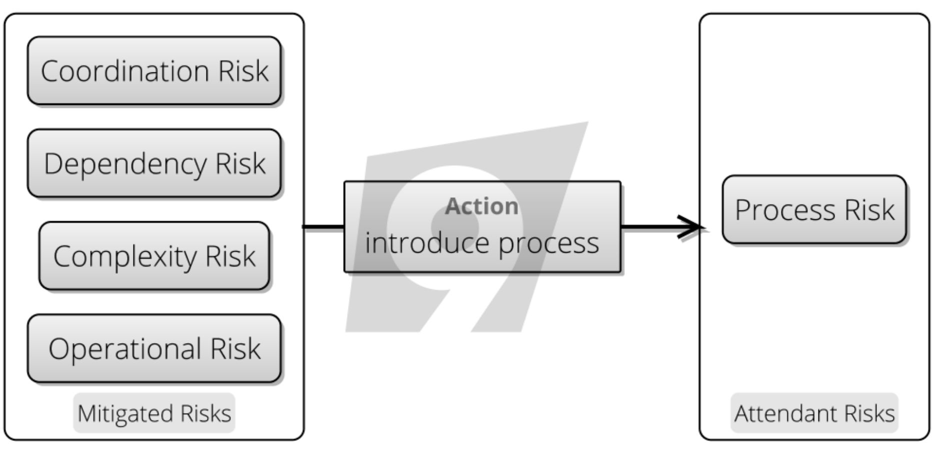


Figure 18.1: Introducing process can mitigate many risks for a team, but there are attendant process risks created.

These are all examples of Risk Mitigation for the *owners* of the process. The *consumers* of the process end up picking up most of the Process Risks. Let's see how this comes about.

18.2 Evolution Of Business Process

Before we get to examining what constitutes Process Risks, let's consider how processes *form*. Specifically, we're going to look at Business Process[4]:

> "**Business Process** or **Business Method** is a collection of related, structured activities or tasks that in a specific sequence produces a service or product (serves a particular business goal) for a particular customer or customers."
>
> —Business Process, *Wikipedia*[5]

Business Processes often arise in response to an unmet need within an organisation. And, as we said above, they are usually there to mitigate other risks. Let's look at an example life-cycle of how that can happen.

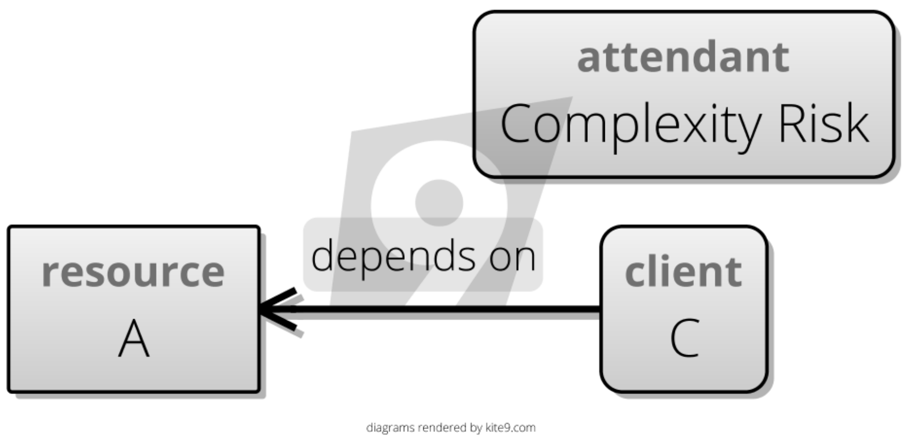


Figure 18.2: Step 0: Clients `C` need `A` to do their jobs

0. Let's say, there exists a group of people inside a company `C`, which need a certain something `A` in order to get their jobs done. It might be a producing a resource, or dealing with some source of complexity, or whatever.

[4] `https://en.wikipedia.org/wiki/Business_process`
[5] `https://en.wikipedia.org/wiki/Business_process`

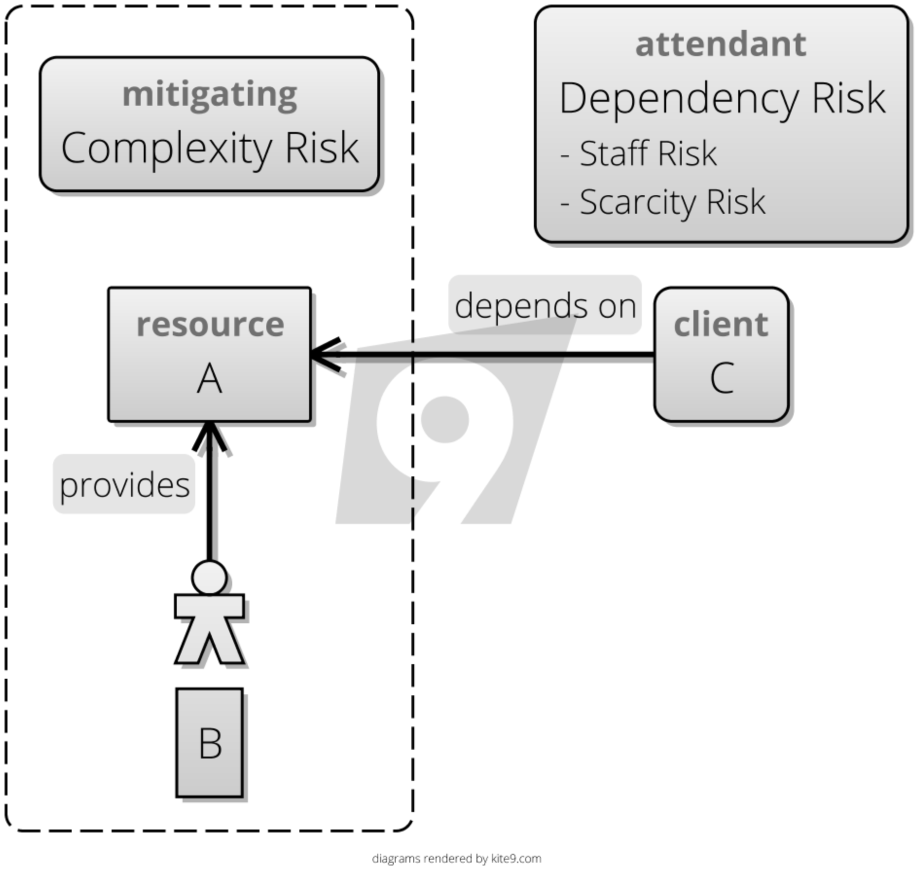


Figure 18.3: Step 1: Person B doing A for company C

1. Person `B` in a company starts producing `A` *as a service to others*. This is really useful! It makes the the lives of clients in `C` much easier as they have an easier path to `A` than before. `B` gets busy keeping `C` happy. No one cares. But then, `B` goes on holiday. `A` doesn't get done, and people now care: the Dependency Risk is suddenly apparent.

2. Either, `B` co-opts other people to help, gets given a team (`T`), or someone else forms a team `T` containing `B` to get the job done "properly".

- `T` is responsible for doing `A`, but it needs to supply the company with `A` reliably and responsibly, otherwise there will be trouble, so they try and please all of their clients as far as possible.

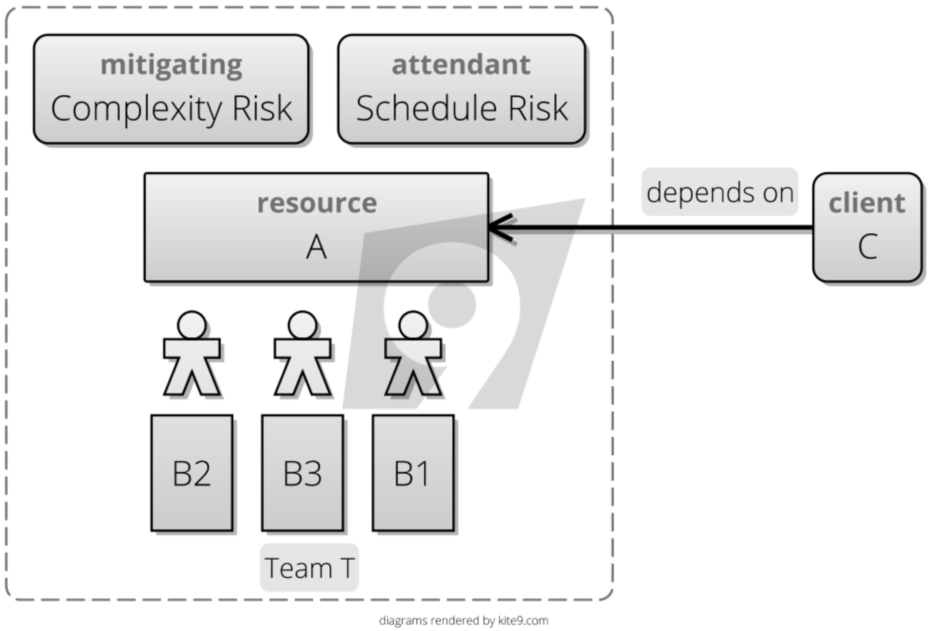


Figure 18.4: Step 2: Team T is created to do A for Company C

- This is a good deal for their clients within `C`, but because there is a lot of variation in what the clients ask for, `T` end up absorbing a lot of Complexity Risk and are overworked.
- This is attendant Schedule Risk: They either need to streamline what they are doing, or get a larger budget, because all this extra Complexity impacts their ability to reliably deliver `A`.

3. `T` organises bureaucratically, so that there is a controlled process (`P`) by which `A` can be accessed. Like a cell, they have arranged a protective barrier around themselves, the strength of which depends on the power conferred to them by control of `A`.

- `P` probably involves filling in a form (or following some other Protocol).
- `T` can now deal with requests on a first-come-first-served basis and deal with them all in the same way: Complexity Risks are now the problem of the form-filler in `C`.
- `T` has mitigated Schedule Risk issues by drawing a line around the amount of Complexity Risk they are willing to take on.
- `C` now has Process Risk: will their requirements for `A` be met by `T`? They have to submit to the process to find out…

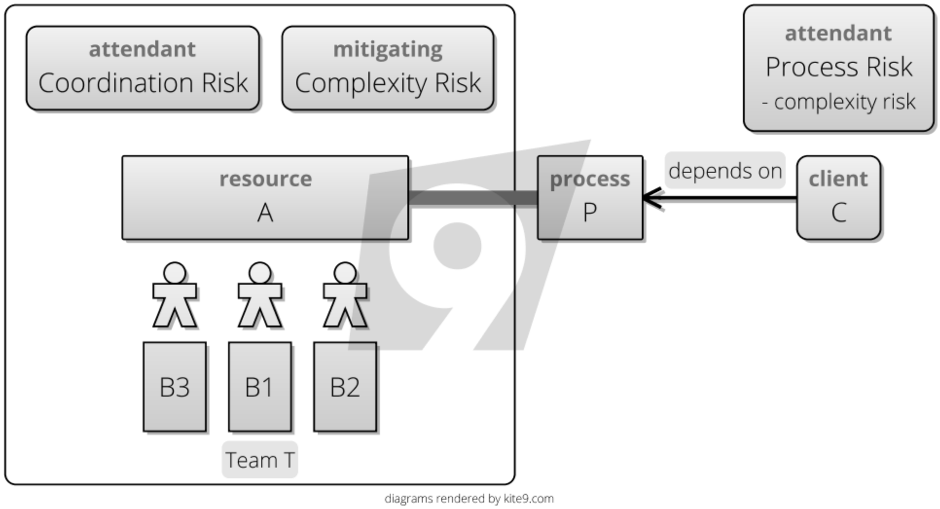


Figure 18.5: Team T protects itself from complexity with a process, P

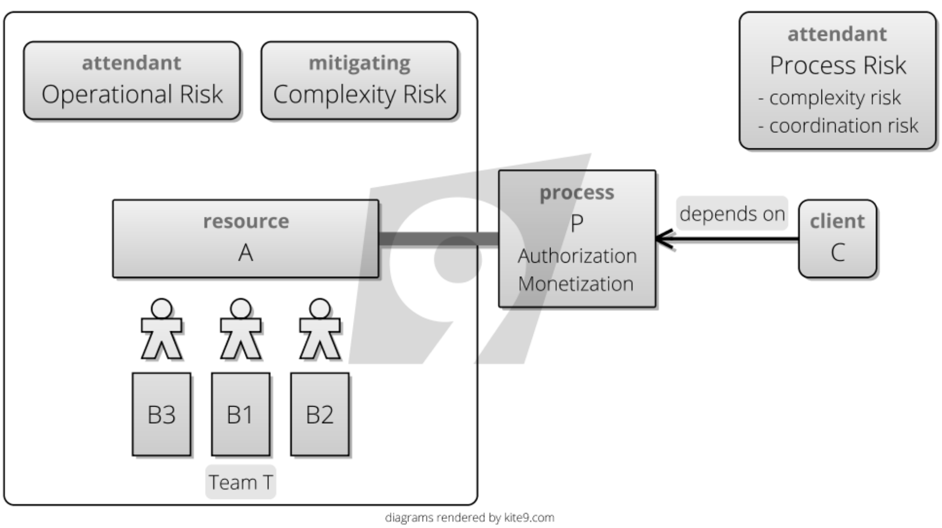


Figure 18.6: Team T protects itself from Coordination issues with sign-offs or other barriers

4. But it's hard to make sure the right clients get access to `A` at the right times, and it's necessary to synchronise access across company `C`. (A Coordination Risk issue.)

- `T` reacts and sets up sign-off, authorisation or monetary barriers around `A`, moving the Coordination Risk issue out of their team.
- But, for `C`, this *again* increases the Process Risk involved in using `A`.

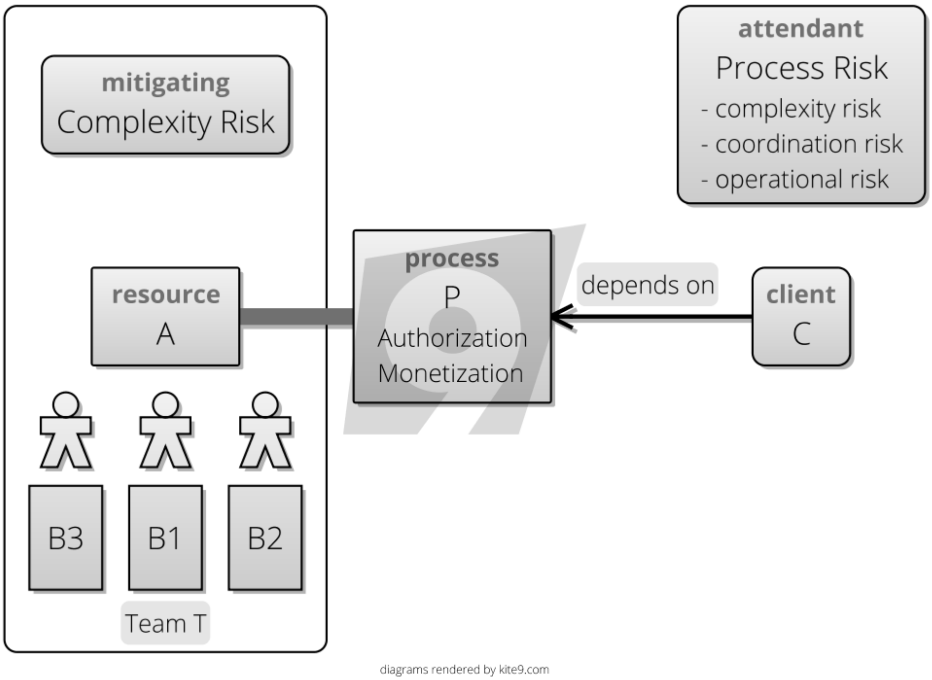


Figure 18.7: Team T increases bureaucratic load, and pushes Process Risk onto C

5. But, there are abuses of `A`: people either misuse it, or use it too much. (These are Operational Risks).

- `T` reacts by *increasing* the amount of *process* to use `A`, mitigating Operational Risk within their team, but. . .
- This corresponds to greater Process Risk for clients in company `C`.

6. Person `D`, who has experience working with team `T` acts as a middleman for customers requiring some variant of `A` for a subset of `C`. They are able to help navigate the bureaucratic process (handle with Process Risk). The cycle potentially starts again: will `D` end up becoming a new team, with a new process?

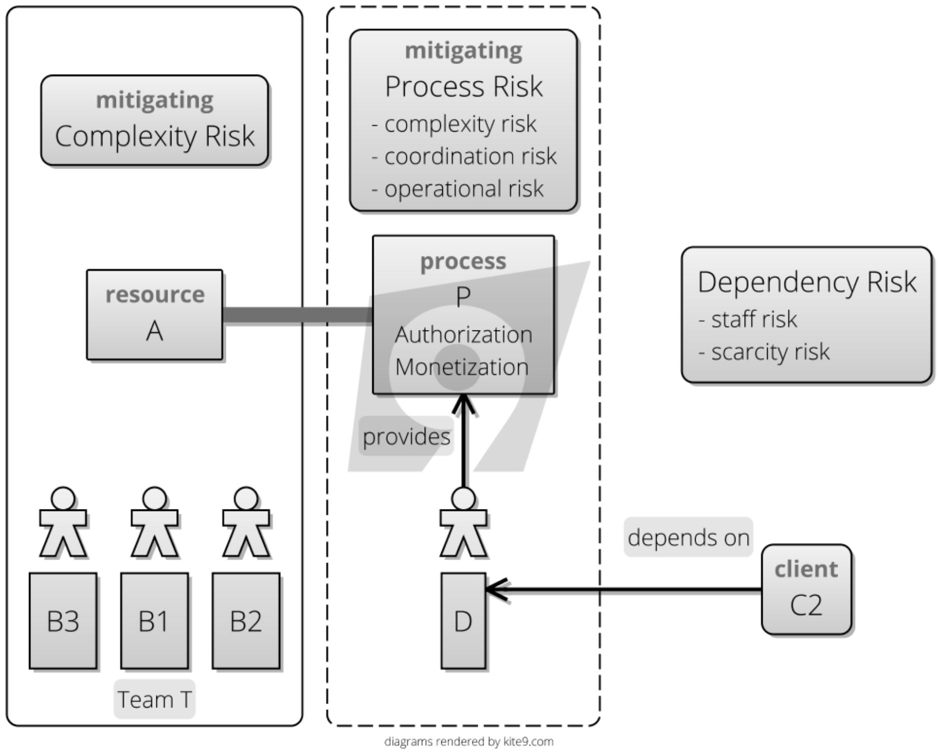


Figure 18.8: Person D acts as a middleman for customers needing some variant of **`A`**

In this example, you can see how the organisation evolves to mitigate risk around the use (and misuse) of `A`: First, Complexity Risk, then Coordination Risk, then Dependency Risk and finally, the Process Risk was created to mitigate everything else. This is an example of *Process following Strategy*:

> In this conception, you can see how the structure of an organisation (the teams and processes within it, the hierarchy of control) will 'evolve' from the resources of the organisation and the strategy it pursues. Processes evolve to meet the needs of the organisation." - Minzberg, *Strategy Safari*[6]

18.3 An Example - Release Process

For many years I have worked in the Finance Industry, and it's given me time to observe how, across an entire industry, process can evolve, both in

[6] `http://www.mintzberg.org/books/strategy-safari`

response to regulatory pressure but also because of organisational maturity, and mitigating risks:

1. Initially, I could release software by logging onto the production accounts with a password that everyone knew, and deploy software or change data in the database.
2. The first issue with this is bad actors: How could you know that the numbers weren't being altered in the databases? Production auditing came in so that at least you could tell *who was changing what*, in order to point the blame later.
3. But, there was still plenty of scope for deliberate or accidental damage. I personally managed to wipe production data on one occasion by mistaking it for a development environment. Eventually, passwords were taken out of the hands of developers and you needed approval to "break glass" to get onto production.
4. Change Requests were introduced. This is another approval process which asks you to describe what you want to change in production, and why you want to change it. In most places, this was quite an onerous process, so the unintended consequence was that release cadence was reduced.
5. The change request software is generally awful, making the job of raising change requests tedious and time-consuming. Therefore, developers would *automate* the processes for release, sometimes including the process to write the change request. This allowed them to improve release cadence, at the expense of owning more code.
6. Auditors didn't like the fact that this automation existed, because effectively, that meant that developers could get access to production with the press of a button, effectively taking you back to step 1. So auditing of Change Requests had to happen.

. . . and so on.

18.4 Process Risks

Process Risk, then, is a type of Dependency Risk, where you are relying on a process. In a way, it's no different from any other kind of Dependency Risk. But Process Risk manifests itself in fairly predictable ways:

- Reliability Risk: If *people* are part of the process, there's the chance that they forget to follow the process itself, and miss steps or forget

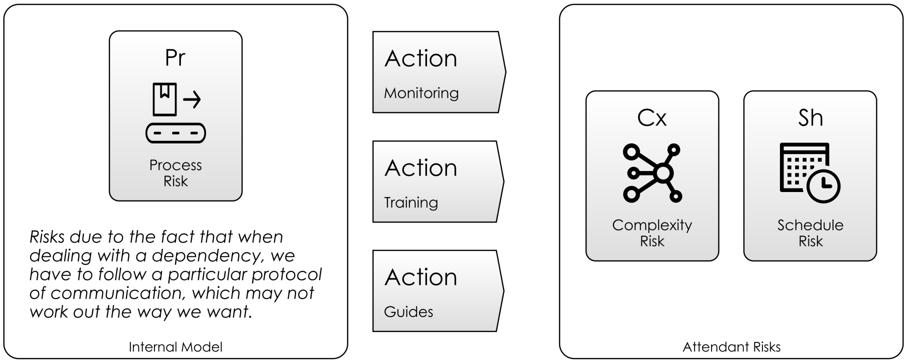


Figure 18.9: Process Risk

your request completely. The reliability is related to the amount of Complexity Risk the process is covering.

- Invisibility Risk: Usually, processes (like other dependencies) trade Complexity Risk for visibility: it's often not possible to see how far along a process is to completion. Sometimes, you can do this to an extent. For example, when I send a package for delivery, I can see roughly how far it's got on the tracking website. But, this is still less-than-complete information, and is a representation of reality.
- Fit Risk: You have to be careful to match the process to the outcome you want. Sometimes, it's easy to waste time on the wrong process.
- Dead-End Risk: Even if you have the right process, initiating a process has no guarantee that your efforts won't be wasted and you'll be back where you started from. The chances of this happening increase as you get further from the standard use-case for the process, and the sunk cost increases with the length of time the process takes to report back.
- Agency Risk: Due to Parkinson's Law, see below.
- Operational Risk: Where processes fail, this is often called Operational Risk, which we'll address further in it's own chapter.
- Credit Risk: Where you pay for something to be done, but then end up without the outcome you want. Let's look at that in more detail.

Processes And Invisibility Risk

Processes tend to work well for the common cases, because *practice makes perfect*. but they are really tested when unusual situations occur. Expanding processes to deal with edge-cases incurs Complexity Risk, so often it's better to try and have clear boundaries of what is "in" and "out" of the process' domain.

Sometimes, processes are *not* used commonly. How can we rely on them anyway? Usually, the answer is to build in extra feedback loops anyway:

- Testing that backups work, even when no backup is needed.
- Running through a disaster recovery scenario at the weekend.
- Increasing the release cadence, so that we practice the release process more.

The feedback loops allow us to perform Retrospectives and Reviews to improve our processes.

Bureaucracy Risk

Where we've talked about process evolution above, the actors involved have been acting in good faith: they are working to mitigate risk in the organisation. The Process Risk that accretes along the way is an *unintended consequence*: There is no guarantee that the process that arises will be humane and intuitive. Many organisational processes end up being baroque or Kafka-esque, forcing unintuitive behaviour on their users. This is partly because process design is *hard*, and it's difficult to anticipate all the various ways a process will be used ahead-of-time. So, some of Bureaucracy Risk is due to Complexity.

But Parkinson's Law[7] takes this one step further: the human actors shaping the organisation will abuse their positions of power in order to further their own careers (this is Agency Risk, which we will come to in a future chapter):

> "Parkinson's law is the adage that"work expands so as to fill the time available for its completion". It is sometimes applied to the growth of bureaucracy in an organisation… He explains this growth by two forces: (1) 'An official wants to multiply subordinates, not rivals' and (2) 'Officials make work for each other.' "
>
> —Parkinson's Law, *Wikipedia*[8]

This implies that there is a tendency for organisations to end up with *needless levels of Bureaucratic Risk*.

[7] https://en.wikipedia.org/wiki/Parkinsons_law

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

Credit Risk

Where the process you depend on is being run by a third-party organisation, (or that party depends on you) you are looking at Credit Risk (also known as Counterparty Risk):

> "A credit risk is the risk of default on a debt that may arise from a borrower failing to make required payments... For example... A business or consumer does not pay a trade invoice when due [or] A business does not pay an employee's earned wages when due"
>
> —Credit Risk, *Wikipedia*[9]

Money is *changing hands* between you and the supplier of the process, and often, the money doesn't transfer *at the same time* as the process is performed. Let's look at an example: Instead of hosting my website on a server in my office, I could choose to host my software project with an online provider. I am trading Complexity Risk for Credit Risk, because now, I have to care that the supplier is solvent.

There's a couple of ways this could go wrong: They may *take my payment*, but then turn off my account. Or, they could go bankrupt, and leave me with the costs of moving to another provider (this is also Dead-End Risk).

Mechanisms like insurance[10], contracts[11] and guarantees[12] help mitigate this risk at the cost of complexity and expense.

Sign-Offs

Often, Processes will include sign-off steps. The Sign-Off is an interesting mechanism: - By signing off on something for the business, people are usually in some part staking their reputation on something being right. - Therefore, you would expect that sign-off involves a lot of Agency Risk: people don't want to expose themselves in career-limiting ways. - Therefore, the bigger the risk they are being asked to swallow, the more cumbersome and protracted the sign off process.

Often, Sign Offs boil down to a balance of risk for the signer: on the one hand, *personal, career risk* from signing off, on the other, the risk of upsetting

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

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

[11] https://en.wikipedia.org/wiki/Contract

[12] https://en.wikipedia.org/wiki/Guarantee

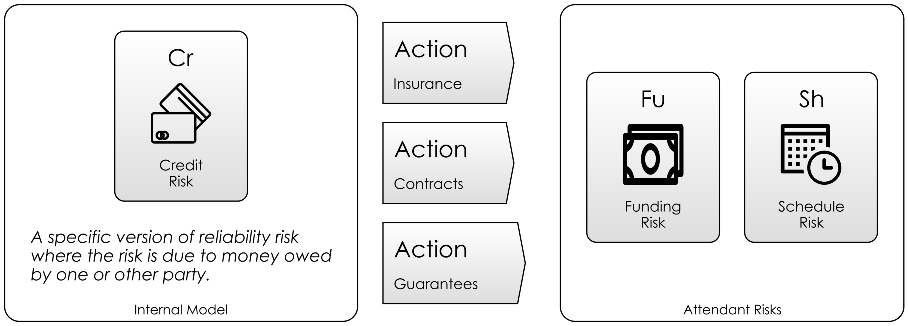


Figure 18.10: Credit Risk

the rest of the staff waiting for the sign-off, and the Dead End Risk of all the effort gone into getting the sign off if they don't.

This is a nasty situation, but there are a couple of ways to de-risk this: - break Sign Offs down into bite-size chunks of risk that are acceptable to those doing the sign-off. - Agree far-in-advance the sign-off criteria. As discussed in Risk Theory, people have a habit of heavily discounting future risk, and it's much easier to get agreement on the *criteria* than it is to get the sign-off.

Software Processes

tbd

tbd. processes as a response to legal environment.

CHAPTER 19

Boundary Risk

In the previous few chapters on Dependency Risk we've touched on Boundary Risk several times, but now it's time to tackle it head-on and discuss this important type of risk.

In terms of the Risk Landscape, Boundary Risk is exactly as it says: a *boundary*, *wall* or other kind of obstacle in your way to making a move you want to make. This changes the nature of the Risk Landscape, and introduces a maze-like component to it. It also means that we have to make *decisions* about which way to go, knowing that our future paths are constrained by the decisions we make.

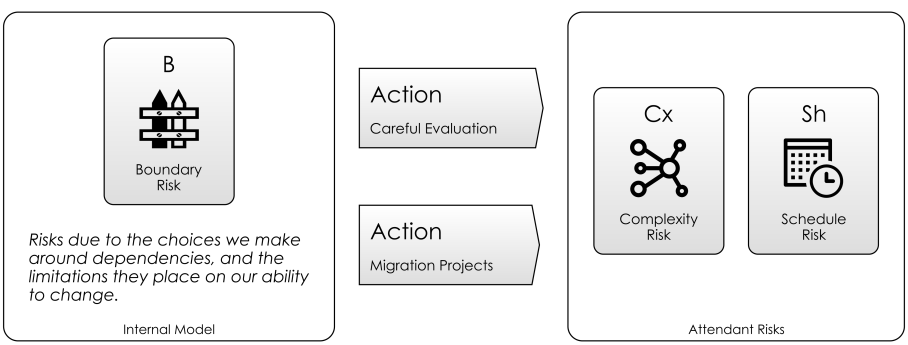


Figure 19.1: Boundary Risk

And, as we discussed in Complexity Risk, there is always the chance we end up at a Dead End, and we've done work that we need to throw away. In this case, we'll have to head back and make a different decision.

19.1 Emergence Through Choice

Boundary Risk is an emergent risk, which exists at the intersection of Complexity Risk, Dependency Risk and Communication Risk. Because of that, it's going to take a bit of time to pick it apart and understand it, so we're going to build up to this in stages.

Let's start with an obvious example: Musical Instruments. Let's say you want to learn to play some music. There are a *multitude* of options available to you, and you might choose an *uncommon* instrument like a Balalaika[1] or a Theremin[2], or you might choose a *common* one like a piano or guitar. In any case, once you start learning this instrument, you have picked up the three risks above:

- Dependency Risk You have a *physical* Dependency on it in order to play music, so get to the music shop and buy one.
- Communication Risk: You have to *communicate* with the instrument in order to get it to make the sounds you want. And you have Learning Curve Risk in order to be able to do that.
- Complexity Risk: As *a music playing system*, you now have an extra component (the instrument), with all the attendant complexity of looking after that instrument, tuning it, and so on.

Those risks are true for *any* instrument you choose. However, if you choose the *uncommon* instrument like the Balalaika, you have *worse* Boundary Risk, because the *ecosystem* for the balalaika is smaller. It might be hard to find a tutor, or a band needing a balalaika. You're unlikely to find one in a friend's house (compared to the piano, say).

Even choosing the Piano has Boundary Risk. By spending your time learning to play the piano, you're mitigating Communication Risk issues, but *mostly*, your skills won't be transferable to playing the guitar. Your decision to choose one instrument over another cements the Boundary Risk: you're following a path on the Risk Landscape and changing to a different path is *expensive*.

Also, it stands to reason that making *any* choice is better than making *no* choice, because you can't try and learn *all* the instruments. Doing that, you'd make no meaningful progress on any of them.

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

[2] `https://en.wikipedia.org/wiki/Theremin`

19.2 Boundary Risk For Software Dependencies

Let's look at a software example now.

As discussed in Software Dependency Risk, if we are going to use a software tool as a dependency, we have to accept the complexity of it's Interface, and learn the protocol of that interface. If you want to work with it, you have to use it's protocol, it won't come to you.

Let's take a look at a hypothetical system structure, in the accompanying diagram. In this design, we have are transforming data from the `input` to the `output`. But how should we do it?

- We could go via `a`, using the Protocols of `a`, and having a dependency on `a`.
- We could go via `b`, using the Protocols of `b`, and having a dependency on `b`.
- We could choose the middle route, and avoid the dependency, but potentially pick up lots more Complexity Risk and Schedule Risk.

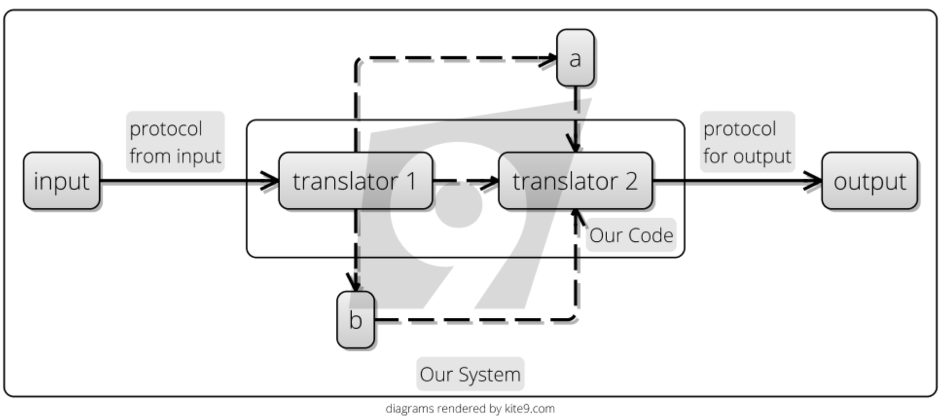


*Figure 19.2: Our System receives data from the **`input`**, translates it and sends it to the **`output`**. But which dependency should we use for the translation, if any?*

This is a basic **Translation** job from `input` to `output`. Since we are talking about **Translation**, we are clearly talking about Communication Risk again: our task in **Integrating** all of these components is *to get them to talk to each other*.

From a Cyclomatic Complexity point of view, this is a very simple structure, with low Complexity. But the choice of approach presents us with Boundary

Risk, because we don't know that we'll be able to make them *talk to each other* properly:

- Maybe `a` outputs dates in a strange calendar format that we won't understand.
- Maybe `b` works on some streaming API basis, that is incompatible with the input protocol.
- Maybe `a` runs on Windows, whereas our code runs on Linux.

. . . and so on.

19.3 Boundary Risk Pinned Down

Wherever we integrate dependencies with complex Protocols, we potentially have Boundary Risk. The more complex the systems being integrated, the higher the risk. When we choose software tools or libraries to help us build our systems, we are trading Complexity Risk for Boundary Risk. It is:

- The *sunk cost* of the Learning Curve we've overcome to integrate the dependency, when it fails to live up to expectations.
- The likelihood of, and costs of changing in the future.
- The rarity of alternatives (or, conversely, the risk of Lock In).

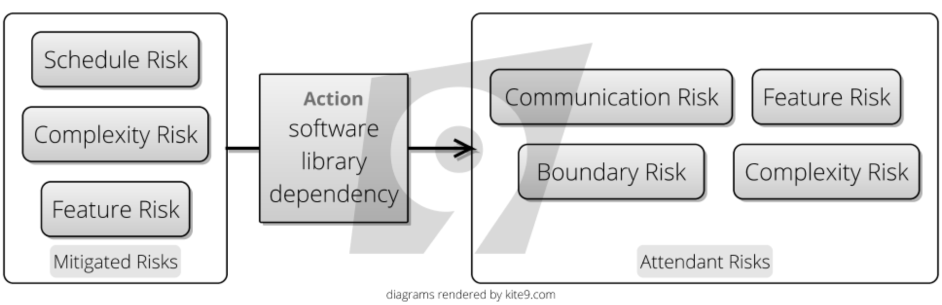


Figure 19.3: The tradeoff for using a library

As we saw in Software Dependency Risk, Boundary Risk is a big factor in choosing libraries and services. However, it can apply to any kind of dependency:

- If you're depending on a Process or Organisation, they might change their products or quality, making the effort you put into the relationship worthless.

- If you're depending on Staff, they might leave, meaning your efforts on training them don't pay back as well as you hoped.
- If you're depending on an Event occurring at a particular time, you might have a lot of work to reorganise your life if it changes time or place.

19.4 Avoiding Boundary Risk Now...

Because of Boundary Risk's relationship to Learning Curve Risk, we can avoid accreting it by choose the *simplest* and *fewest* dependencies for any job. Let's look at some examples:

- `mkdirp` is an npm[3] module defining a single function. This function takes a single string parameter and recursively creating directories. Because the protocol is so simple, there is almost no Boundary Risk.
- Using a database with a JDBC driver comes with *some* Boundary Risk: but the boundary is specified by a standard. Although the standard doesn't cover every aspect of the behaviour of the database, it does minimise risk, because if you are familiar with one JDBC driver, you'll be familiar with them all, and swapping one for another is relatively easy.
- Using a framework like Spring[4], Redux[5] or Angular[6] comes with higher Boundary Risk: you are expected to yield to the framework's way of behaving throughout your application. You cannot separate the concern easily, and swapping out the framework for another is likely to leave you with a whole new set of assumptions and interfaces to deal with.

19.5 ... And In The Future

Unless your project *ends*, you can never be completely sure that Boundary Risk *isn't* going to stop you making a move you want. For example:

- `mkdirp` might not work on a new device's Operating System, forcing you to swap it out.

[3] https://www.npmjs.com

[4] https://spring.io

[5] https://redux.js.org

[6] https://angularjs.org

- You might discover that the database you chose satisfied all the features you needed at the start of the project, but came up short when the requirements changed later on.
- The front-end framework you chose might go out-of-fashion, and it might be hard to find developers interested in working on the project because of it.

This third point is perhaps the most interesting aspect of Boundary Risk: how can we ensure that the decisions we make now are future-proof? In order to investigate this further, let's look at three things: Plugins, Ecosystems and Evolution (again).

19.6 Plugins, Ecosystems and Evolution

On the face of it, WordPress[7] and Drupal[8] *should* be very similar:

- They are both Content Management Systems[9]
- They both use a LAMP (Linux, Apache, MySql, PHP) Stack[10]
- They were both started around the same time (2001 for Drupal, 2003 for WordPress)
- They are both Open-Source, and have a wide variety of Plugins[11]. That is, ways for other programmers to extend the functionality in new directions.

In practice, they are very different. This could be put down to different *design goals*: it seems that WordPress was focused much more on usability, and an easy learning curve, whereas Drupal supported plugins for building things with complex data formats. It could also be down to the *design decisions*: although they both support Plugins[12], they do it in very different ways.

(Side note: I wasn't short of go-to examples for this. I could have picked on Team City[13] and Jenkins[14] here (Continuous Integration[15] tools), or Maven[16]

[7] https://en.wikipedia.org/wiki/WordPress

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

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

[10] https://en.wikipedia.org/wiki/LAMP_(software_bundle)

[11] https://en.wikipedia.org/wiki/Plug-in_(computing)

[12] https://en.wikipedia.org/wiki/Plug-in_%28computing%29

[13] https://en.wikipedia.org/wiki/TeamCity

[14] https://en.wikipedia.org/wiki/Jenkins_(software)

[15] https://en.wikipedia.org/wiki/Continuous_integration

[16] https://en.wikipedia.org/wiki/Apache_Maven

and Gradle[17] (build tools). All of these support plugins), and the *choice* of plugins is dependent on which I've chosen, despite the fact that the platforms are solving pretty much the same problems.)

Ecosystems and Systems

The quality, and choice of plugins for a given platform, along with factors such as community and online documentation is often called its ecosystem[18]:

> "as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them"
>
> —Software Ecosystem, *Wikipedia*[19]

You can think of the ecosystem as being like the footprint of a town or a city, consisting of the buildings, transport network and the people that live there. Within the city, and because of the transport network and the amenities available, it's easy to make rapid, useful moves on the Risk Landscape. In a software ecosystem it's the same: the ecosystem has gathered together to provide a way to mitigate various different Feature Risks in a common way.

tbd: talk about complexity within the boundary. (increased convenience?)

Ecosystem size is one key determinant of Boundary Risk: a *large* ecosystem has a large boundary circumference. Boundary Risk is lower because your moves on the Risk Landscape are unlikely to collide with it. The boundary *got large* because other developers before you hit the boundary and did the work building the software equivalents of bridges and roads and pushing it back so that the boundary didn't get in their way.

In a small ecosystem, you are much more likely to come into contact with the edges of the boundary. *You* will have to be the developer that pushes back the frontier and builds the roads for the others. This is hard work.

Evolution

In the real world, there is a tendency for *big cities to get bigger*. The more people that live there, the more services they provide, and therefore, the more immigrants they attract. And, it's the same in the software world. In both cases, this is due to the Network Effect:

[17] https://en.wikipedia.org/wiki/Gradle

[18] https://en.wikipedia.org/wiki/Software_ecosystem

[19] https://en.wikipedia.org/wiki/Software_ecosystem

> "A network effect (also called network externality or demand-side economies of scale) is the positive effect described in economics and business that an additional user of a good or service has on the value of that product to others. When a network effect is present, the value of a product or service increases according to the number of others using it."
>
> —Network Effect, *Wikipedia*[20]

You can see the same effect in the adoption rates of WordPress and Drupal, shown in the chart below. Note: this is over *all sites on the internet*, so Drupal accounts for hundreds of thousands of sites. In 2018, WordPress is approximately 32% of all web-sites. For Drupal it's 2%.

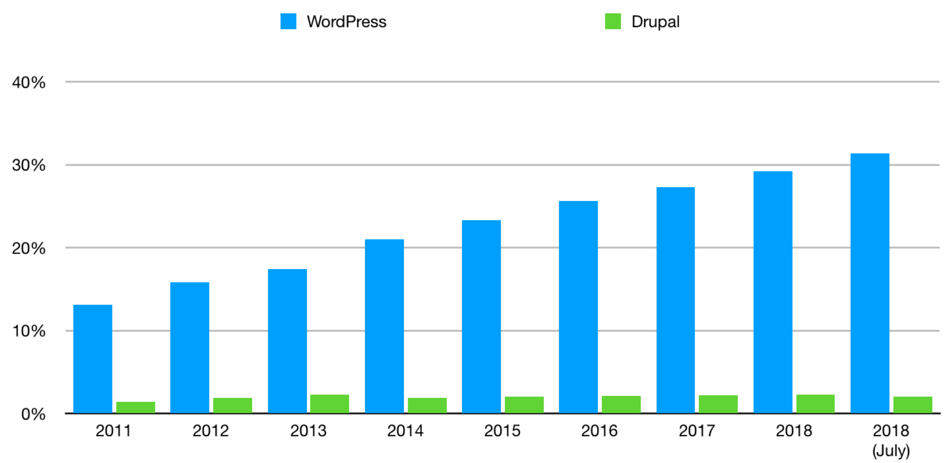


Figure 19.4: WordPress vs Drupal adoption over 8 years, according to w3techs.com[21]

Did WordPress gain this march because it was better than Drupal? That's arguable. That it's this way round could be *entirely accidental*, and a result of Network Effect.

And maybe, they aren't comparable: Given the same problems, the people in each ecosystem have approached them and solved them in different ways. And, this has impacted the 'shape' of the abstractions, and the protocols you use in each. Complexity *emerges*, and the ecosystem gets more complex and opinionated, much like the way in which the network of a city will evolve over time in an unpredictable way.

[20] `https://en.wikipedia.org/wiki/Network_effect`

But, by now, if they *are* to be compared side-by-side, WordPress *should be better* due to the sheer number of people in this ecosystem who are. . .

- Creating web sites.
- Using those sites.
- Submitting bug requests.
- Fixing bugs.
- Writing documentation.
- Building plugins.
- Creating features.
- Improving the core platform.

But, there are two further factors to consider. . .

1. The Peter Principle

When a tool or platform is popular, it is under pressure to increase in complexity. This is because people are attracted to something useful, and want to extend it to new purposes. This is known as *The Peter Principle*:

> "The Peter principle is a concept in management developed by Laurence J. Peter, which observes that people in a hierarchy tend to rise to their 'level of incompetence'."
>
> —The Peter Principle, *Wikipedia*[22]

Although designed for *people*, it can just as easily be applied to any other dependency you can think of. Let's look at Java[23] as an example of this.

Java is a very popular platform. Let's look at how the number of public classes (a good proxy for the boundary) has increased with each release:

Why does this happen?

- More and more people are using Java for more and more things. It's popularity begets more popularity.
- Human needs are *fractal* in complexity. You can always find ways to make a dependency *better* (For some meaning of better).
- There is Feature Drift Risk: our requirements evolve with time. Android Apps[24] weren't even a thing when Java 3 came out, for example, yet they are all written in Java now, and Java has had to keep up.

[22] https://en.wikipedia.org/wiki/Peter_principle

[23] https://en.wikipedia.org/wiki/Java_(software_platform)

[24] https://en.wikipedia.org/wiki/Android_software_development

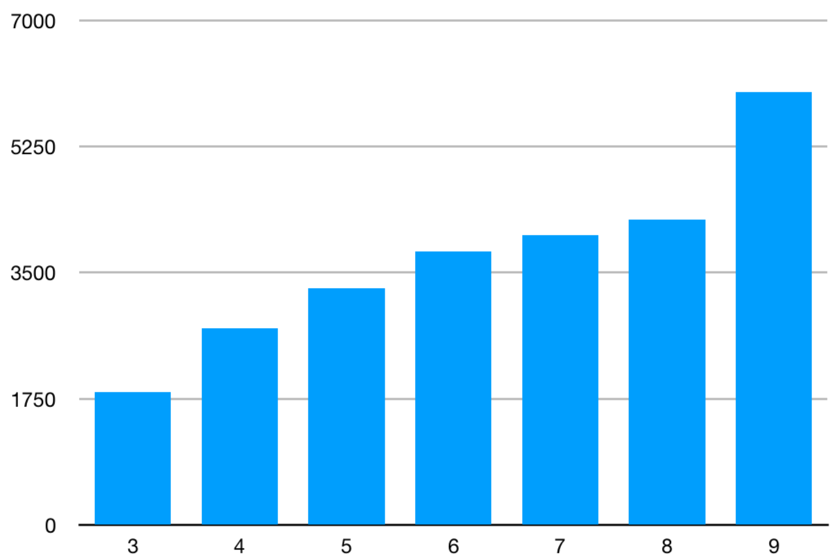


Figure 19.5: Java Public Classes By Version (3-9)

2. Backward Compatibility

As we saw in Software Dependency Risk, The art of good design is to afford the greatest increase in functionality with the smallest increase in complexity possible, and this usually means Refactoring[25]. But, this is at odds with Backward Compatibility.

Each new version has a greater functional scope than the one before (pushing back Boundary Risk), making the platform more attractive to build solutions in. But this increases the Complexity Risk as there is more functionality to deal with.

Focus vs Over-Reach

You can see in the diagram the Peter Principle at play: as more responsibility is given to a dependency, the more complex it gets, and the greater the learning curve to work with it. Large ecosystems like Java react to Learning Curve Risk by having copious amounts of literature to read or buy to help, but it is still off-putting.

[25] `https://en.wikipedia.org/wiki/Refactoring`

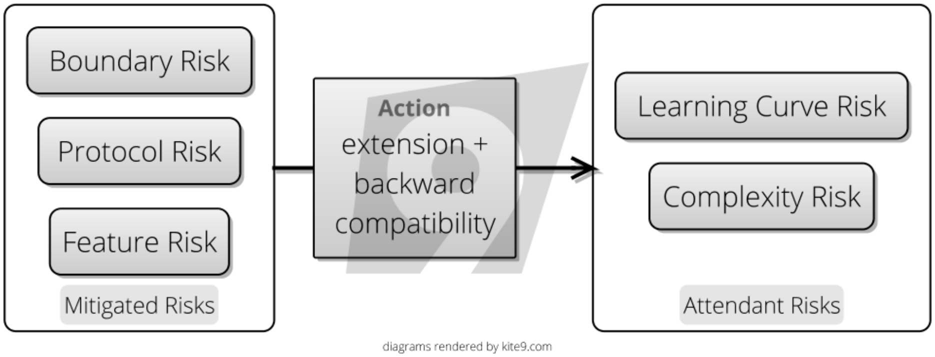


Figure 19.6: The Peter Principle: Backward Compatibility + Extension leads to complexity and learning curve risk

Because Complexity is Mass, large ecosystems can't respond quickly to Feature Drift. This means that when the world changes, *new* systems will come along to plug the gaps.

This implies a trade-off: - Sometimes it's better to accept the Boundary Risk innate in a smaller system than try to work within the bigger, more complex system.

example:

In the late 80's and 90's there was a massive push towards *building functionality in the database*. Relational Database Management Systems (RDBMSs)[26] were all-in-one solutions, expensive platforms that you purchased and built *everything* inside. However, this dream didn't last:

why? (need some research here).

This tbd

tbd. diagram here.

19.7 Beating Boundary Risk With Standards

Sometimes, technology comes along that allows us to cross boundaries, like a *bridge* or a *road*. This has the effect of making it easy to to go from one self-contained ecosystem to another. Going back to WordPress, a simple example might be the Analytics Dashboard[27] which provides Google Analytics[28] functionality inside WordPress.

[26] https://en.wikipedia.org/wiki/Relational_database

[27] https://en-gb.wordpress.org/plugins/google-analytics-dashboard-for-wp/

[28] https://en.wikipedia.org/wiki/Google_Marketing_Platform

I find, a lot of code I write is of this nature: trying to write the *glue code* to join together two different *ecosystems*.

Standards allow us to achieve the same thing, in one of two ways:

- **Mode 1: Abstract over the ecosystems.** Provide a *standard* protocol (a *lingua franca*) which can be converted down into the protocol of any of a number of competing ecosystems.
- **Mode 2: Force adoption.** All of the ecosystems start using the standard for fear of being left out in the cold. Sometimes, a standards body is involved, but other times a "de facto" standard emerges that everyone adopts.

Let's look at some examples:

- ASCII[29]: fixed the different-character-sets boundary risk by being a standard that others could adopt. Before everyone agreed on ASCII, copying data from one computer system to another was a massive pain, and would involve some kind of translation. Unicode[30] continues this work. (**Mode 1**)

- C[31]: The C programming language provided a way to get the same programs compiled against different CPU instruction sets, therefore providing some *portability* to code. The problem was, each different operating system would still have it's own libraries, and so to support multiple operating systems, you'd have to write code against multiple different libraries. (**Mode 2**)

- Java[32] took what C did and went one step further, providing interoperability at the library level. Java code could run anywhere where Java was installed. (**Mode 2**)

- Internet Protocol: As we saw in Communication Risk, the Internet Protocol (IP) is the *lingua franca* of the modern Internet. However, at one period of time, there were many competing standards. and IP was the ecosystem that "won", and was subsequently standardized by the IETF[33]. (**Mode 1**)

[29] `https://en.wikipedia.org/wiki/ASCII`

[30] `https://en.wikipedia.org/wiki/Unicode`

[31] `https://en.wikipedia.org/wiki/C_(programming_language)`

[32] `https://en.wikipedia.org/wiki/Java_(programming_language)`

[33] `https://en.wikipedia.org/wiki/Internet_Engineering_Task_Force`

19.8 Complex Boundaries

As shown in the above diagram, mitigating Boundary Risk involves taking on complexity. The more Protocol Complexity there is to bridge the two ecosystems, the more Complex the bridge will necessarily be.

From examining the [Protocol Risk][br1] at each end of the bridge you are creating, you can get a rough idea of how complex the endeavour will be:

- If it's low-risk at both ends, you're probably going to be able to knock it out easily. Like translating a date, or converting one file format to another.
- Where one of the protocols is *evolving*, you're definitely going to need to keep releasing new versions. The functionality of a `Calculator` app on my phone remains the same, but new versions have to be released as the phone APIs change, screens change resolution and so on.
- tbd

Where boundaries

tbd Trying to create a complex, fractal surface. User requirements are fractal in nature.

Object-Relational Mapping

For example, Object Relational Mapping (ORM)[34] has long been a problem in software. This is Boundary-Crossing software trying to bridge the gap between Relational Databases and Object-Oriented Languages[35] like Java). Building a *general* library that does this and is useful tbd said:

> 'Object/Relational Mapping is the Vietnam of Computer Science'
> - Ted Neward[36]

This is a particularly difficult problem because the two ecosystems are so *rich* and *complex* in the functionality they expose. But what are the alternatives?

- Either back to building functionality within the database again, using stored procedures

[34] https://en.wikipedia.org/wiki/Object-relational_mapping

[35] https://en.wikipedia.org/wiki/Object-oriented_programming

[36] http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

Protocol Risk From A	Protocol Risk From B	Resulting Bridge Complexity	Example
Low	Low	Simple	Changing from one date format to another.
High	Low	Moderate	Status Dashboard, tbd
High	High	Complex	Object-Relational Mapping (ORM) Tools, (see below)
High + Evolving	Low	Moderate, Versioned	Simple Phone App, e.g. note-taker or calculator
Evolving	High	Complex	Modern browser (see below)
Evolving	Evolving	Very Complex	Google Search, Scala (see below)

- Building Object Databases[37]. It's interesting that neither of these really worked out.
- Custom-building the bridge between the systems, one database call at-a-time in your own software.

This is tbd hobson's choice, there is strong debate about whether ORM is a worse trade of mitigated Boundary Risk for attendant Complexity Risk or not, and clearly will depend on your circumstances.

Scala

Mapping between complex boundaries is especially difficult if the Boundaries are evolving and changing as you go. This means in ecosystems that are changing rapidly, you are unlikely to be able to create lasting bridges between them. Given that Java) is an old, large and complex ecosystem, you would imagine that it would have a slow-enough rate of change that abstracting technologies can be built on top of it safely.

Indeed, we see that happening with Clojure[38] and Kotlin[39], two successful languages built on top of the Java Virtual Machine (JVM)[40] and offering compatibility with it.

Scala[41] is arguably the first mainstream language that tried to do the same thing: it is trying to build a Functional Programming[42] paradigm on top of the JVM, which traditionally has an Object Oriented paradigm.

The problem faced by Scala is that Java didn't stay still: as soon as they demonstrated some really useful features in Scala (i.e. stream-based processing), Java moved to include this new functionality too. If they hadn't, the developer community would have slowly drifted away and used Scala instead.

So, in a sense, Scala is a *success story*: they were able to force change to Java. But, once Java had changed, Scala was in the difficult position of having two sets of competing features in the platform: the existing Scala streams, and the new Java streams.

Clojure can interoperability with Java because on one side, the boundary is simple: lisp is a simple language which lends itself to re-implementation

[37] https://en.wikipedia.org/wiki/Object_database

[38] https://en.wikipedia.org/wiki/Clojure

[39] https://en.wikipedia.org/wiki/Kotlin_(programming_language)

[40] https://en.wikipedia.org/wiki/Java_virtual_machine

[41] https://en.wikipedia.org/wiki/Scala_(programming_language)

[42] https://en.wikipedia.org/wiki/Functional_programming

within other platforms. Therefore, the complexity of the bridge is *simple*: all that needs to be provided is a way to call methods from Java to Clojure.

Scala and Java have a complex relationship because Scala creates it's own complex boundary: it is syntactically and functionally a broad language with lots of features. And so is Java. Mapping from one to the other is therefore

for interoperability here. Why is one so different from the other?

Browsers

Web browsers are another surprisingly complex boundary. They have to understand the following protocols:

- HTTP for loading resources (as we already reviewed in Communication Risk
- HTML Pages, for describing the content of web pages.
- Various image formats
- JavaScript[43] for web-page *interactivity*
- CSS[44] for web-page styling, animation and so on.
- ... and several others.

Handling any one of these protocols alone is a massive endeavour, so browsers are built on top of Software Libraries which handle each concern, for example, Networking Libraries, Parsers[45] and so on.

One way of looking at the browser is that it is a *function*, where those elements listed above are the *inputs* to the function, and the output is *what is displayed on the screen*, as shown in the image below.

tbd. browser as a function

There are three specific problems that make this a really complex boundary:

1. All of the standards above are *evolving and improving*. And, although HTML5[46] (say) is a reasonably well-specified standard, in reality, web pages tend not to adhere exactly to the letter of it. People make mistakes in the HTML they write, and it's up to the browser to try and figure out what they *meant* to write, rather than what they did write. This makes the *input* to the function extremely complex.

[43] https://en.wikipedia.org/wiki/JavaScript

[44] https://en.wikipedia.org/wiki/Cascading_Style_Sheets

[45] https://en.wikipedia.org/wiki/Parsing#Computer_languages

[46] https://en.wikipedia.org/wiki/HTML5

2. Similarly, the *output* of the function is not well defined either, and relies a lot on people's *subjective aesthetic judgement*. For example, if you insert a `<table>` into an HTML page, the specification doesn't say anything about exactly how big the table should be, the size of it's borders, the spacing of the content and so on. At least, initially, *none* of this was covered by the HTML Specification. The CSS specification is over time clearing this up, but it's not *exactly nailed down*, which means...

3. That because there are various different browsers (Chrome[47], Safari[48], Internet Explorer[49], Microsoft Edge[50], Firefox[51] etc.) and each browser has multiple different versions, released over a period of many years, you cannot, as a web-page developer know, *a priori* what your web-page will look like to a user.

As developers trying to build software to be delivered over the Internet, this is therefore a source of common Boundary Risk. If you were trying to build software to work in *all browsers* and *all versions*, this problem would be nearly insurmountable. So, in order to tackle this risk, we do the following:

- We pick a small (but commonly used) subset of browsers, and use features from the specifications that we know commonly work in that subset.
- We test across the subset. Again, testing is *harder than it should be*, because of problem 2 above, that the expected output is not exactly defined. This generally means you have to get humans to apply their *subjective aesthetic judgement*, rather than getting machines to do it.
- There is considerable pressure on browser developers to ensure consistency of behaviour across the implementations. If all the browsers work the same, then we don't face the Boundary Risk of having to choose just one to make our software work in. However, it's not always been like this...

19.9 Vendor Lock-In

In the late 1990s, faced with the emergence of the nascent World Wide Web, and the Netscape Navigator[52] browser, Microsoft[53] adoped a strategy known

[47] https://en.wikipedia.org/wiki/Google_Chrome
[48] https://en.wikipedia.org/wiki/Safari_(web_browser)
[49] https://en.wikipedia.org/wiki/Internet_Explorer
[50] https://en.wikipedia.org/wiki/Microsoft_Edge
[51] https://en.wikipedia.org/wiki/Firefox
[52] https://en.wikipedia.org/wiki/Netscape_Navigator
[53] https://en.wikipedia.org/wiki/Microsoft

as Embrace and Extend[54]. The idea of this was to subvert the HTML standard to their own ends by *embracing* the standard and creating their own browser Internet Explorer and then *extending* it with as much functionality as possible, which would then *not work* in Netscape Navigator. They then embarked on a campaign to try and get everyone to "upgrade" to Internet Explorer. In this way, they hoped to "own" the Internet, or at least, the software of the browser, which they saw as analogous to being the "operating system" of the Internet, and therefore a threat to their own operating system, Windows[55].

There are two questions we need to ask about this, from the point-of-view of understanding Boundary Risk:

1. Why was this a successful strategy?
2. Why did they stop doing this?

Let's look at the first question then. Yes, it was a successful strategy. In the 1990s, browser functionality was rudimentary. Developers were *desperate* for more features, and for more control over what appeared on their web-pages. And, Internet Explorer was a free download (or, bundled with Windows). By shunning other browsers and coding just for IE, developers pushed Boundary Risk to the consumers of the web pages and in return mitigated Feature Fit Risk: they were able to get more of the functionality they wanted in the browser.

It's worth pointing out, *this was not a new strategy*:

- Processor Chip manufacturers had done something similar in the tbds: by providing features (instructions) on their processors that other vendors didn't have, they made their processors more attractive to system integrators. However, since the instructions were different on different chips, this created Boundary Risk for the integrators. Intel and Microsoft were able to use this fact to build a big ecosystem around Windows running on Intel chips (so called, WinTel).
- We have two main *mobile* ecosystems: Apple's iOS[56] and Google's Android[57], which are both *very* different and complex ecosystems with large, complex boundaries. They are both innovating as fast as possible to keep users happy with their features. Tools like Xamarin[58] exist which allow you to build

[54]https://en.wikipedia.org/wiki/Embrace_and_extend
[55]https://en.wikipedia.org/wiki/Microsoft_Windows
[56]https://en.wikipedia.org/wiki/IOS
[57]https://en.wikipedia.org/wiki/Android_(operating_system)
[58]https://en.wikipedia.org/wiki/Xamarin

- Currently, Amazon Web Services (AWS) are competing with Microsoft Azure[59] and Google Cloud Platform[60] over building tools for Platform as a Service (PaaS)[61] (running software in the cloud). They are both racing to build new functionality, but at the same time it's hard to move from one vendor to another as there is no standardisation on the tools.
- As we saw above, Database vendors tried to do the same thing with features in the database. Oracle particularly makes money over differentiating itself from competitors by providing features that other vendors don't have. Tom tbd provides a compelling argument for using these features thus:

  tbd.

The next question, is why did Microsoft *stop* pursuing this strategy? It seems that the answer is because they were made to. tbd.

19.10 Everyday Boundary Risks

Boundary Risk occurs all the time. Let's look at some ways:

- **Configuration**: When software has to be deployed onto a server, there has to be configuration (usually on the command line, or via configuration property files) in order to bridge the boundary between the *environment it's running in* and the *software being run*. Often, this is setting up file locations, security keys and passwords, and telling it where to find other files and services.
- **Integration Testing**: Building a unit test is easy. You are generally testing some code you have written, aided with a testing framework. Your code and the framework are both written in the same language, which means low boundary risk. But, to *integration test* you need to step outside this boundary and so it becomes much harder. This is true whether you are integrating with other systems (providing or supplying them with data) or parts of your own system (say testing the client-side and server parts together).
- **User Interface Testing**: If you are supplying a user-interface, then the interface with the user is already a complex, under-specified risky protocol. Although tools exist to automate UI testing (such as Selenium[62],

[59] https://en.wikipedia.org/wiki/Microsoft_Azure
[60] https://en.wikipedia.org/wiki/Google_Cloud_Platform
[61] https://en.wikipedia.org/wiki/Platform_as_a_service
[62] https://en.wikipedia.org/wiki/Selenium_(software)

these rarely satisfactorily mitigate this protocol risk: can you be sure that the screen hasn't got strange glitches, that the mouse moves correctly, that the proportions on the screen are correct on all browsers?

- **Jobs**: When you pick a new technology to learn and add to your CV, it's worth keeping in mind how useful this will be to you in the future. It's career-limiting to be stuck in a dying ecosystem and need to retrain.
- **Teams**: if you're given license to build a new product within an existing team, are you creating Boundary Risk by using tools that the team aren't familiar with?
- **Organisations**: Getting teams or departments to work with each other often involves breaking down Boundary Risk. Often the departments use different tool-sets or processes, and have different goals making the translation harder. tbd

Boundary Risk and Change

You can't always be sure that a dependency now will always have the same guarantees in the future:

- **Ownership changes** Microsoft buys GitHub[63]. What will happen to the ecosystem around GitHub now?
- **Licensing changes**. (e.g. Oracle[64] buys **Tangosol** who make Coherence[65] for example). Having done this, they increase the licensing costs of Tangosol to huge levels, milking the Cash Cow[66] of the installed user-base, but ensuring no-one else is likely to use it.
- **Better alternatives become available**: As a real example of this, I began a project in 2016 using Apache Solr[67]. However, in 2018, I would probably use ElasticSearch[68]. In the past, I've built web-sites using Drupal and then later converted them to use WordPress.

19.11 Patterns In Boundary Risk

In Feature Risk, we saw that the features people need change over time. Let's get more specific about this:

[63] https://en.wikipedia.org/wiki/GitHub

[64] http://oracle.com

[65] https://en.wikipedia.org/wiki/Oracle_Coherence

[66] https://en.wikipedia.org/wiki/Cash_cow

[67] https://en.wikipedia.org/wiki/Apache_Solr

[68] https://en.wikipedia.org/wiki/Elasticsearch

- Human need is Fractal[69]. This means that over time, software products have evolved to more closely map to human needs. Software that would have delighted us ten years ago lacks the sophistication we expect today.
- Software and hardware are both is improving with time, due to evolution and the ability to support greater and greater levels of complexity.
- Abstractions build too. As we saw in Process Risk, we *encapsulate* earlier abstractions in order to build later ones.

If all this is true, the only thing we can expect in the future is that the lifespan of any ecosystem will follow an arc through creation, adoption, growth, use and finally either be abstracted over or abandoned.

tbd diagram.

Although our discipline is a young one, we should probably expect to see "Software Archaeology" in the same way as we see it for biological organisms. Already we can see the dead-ends in the software evolutionary tree: COBOL and BASIC languages, CASE systems. Languages like FORTH live on in PostScript, SQL is still embedded in everything

Boundary risk is *inside* and *outside*

[69] `https://en.wikipedia.org/wiki/Fractal`

CHAPTER 20

Agency Risk

Coordinating a team is difficult enough when everyone on the team has a single Goal. But, people have their own goals, too. Sometimes, the goals harmlessly co-exist with the team's goal, but other times they don't.

This is Agency Risk. This term comes from finance and refers to the situation where you (the "principal") entrust your money to someone (the "agent") in order to invest it, but they don't necessarily have your best interests at heart. They may instead elect to invest the money in ways that help them, or outright steal it.

> "This dilemma exists in circumstances where agents are motivated to act in their own best interests, which are contrary to those of their principals, and is an example of moral hazard."
>
> —Principal-Agent Problem, *Wikipedia*[1]

The less visibility you have of the agent's activities, the bigger the risk. However, the whole *point* of giving the money to the agent was that you would have to spend less time and effort managing it.

Agency Risk clearly includes the behaviour of Bad Actors[2]. But, this is a very strict definition of Agency Risk. In software development, we're not lending each other money, but we are being paid by the project sponsor, so they are assuming Agency Risk by employing us.

As we saw in the previous chapter on Process Risk, Agency Risk doesn't just apply to people: it can apply to *running software* or *whole teams*.

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

[2] https://en.wiktionary.org/wiki/bad_actor

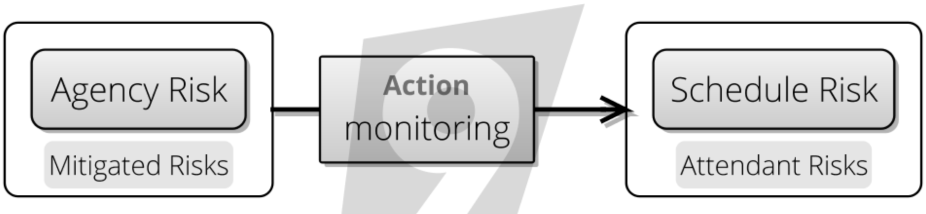


Figure 20.1: Mitigating Agency Risk Through Monitoring

Let's look at some examples of borderline Agency Risk situations, in order to sketch out where the domain of this risk lies.

20.1 Personal Lives

We can't (shouldn't) expect people on a project to sacrifice their personal lives for the success of the project, right? Except that "Crunch Time"[3] is exactly how some software companies work:

> "Game development... requires long working hours and dedication from their employees. Some video game developers (such as Electronic Arts) have been accused of the excessive invocation of"crunch time"."Crunch time" is the point at which the team is thought to be failing to achieve milestones needed to launch a game on schedule. "
>
> —Crunch Time, *Wikipedia*[4]

People taking time off, going to funerals, looking after sick relatives and so on are all Agency Risk, but they should be *accepted* on the project. They are a necessary Attendant Risk of having *staff* rather than *slaves*.

20.2 The Hero

> "The one who stays later than the others is a hero."
>
> —Hero Culture, *Ward's Wiki*[5] <!– tweet-end –>

Conversely, Heroes put in more hours and try to rescue projects single-handedly, often cutting corners like team communication and process in order to get there.

[3] `https://en.wikipedia.org/wiki/Video_game_developer#%22Crunch_time%22`

[4] `https://en.wikipedia.org/wiki/Video_game_developer#"Crunch_time"`

[5] `http://wiki.c2.com/?HeroCulture`

Sometimes, projects don't get done without heroes. But other times, the hero has an alternative agenda than just getting the project done:

- A need for control, and for their own vision.
- A preference to work alone.
- A desire for recognition and acclaim from colleagues.
- For the job security of being a Key Man[6].

A team *can* make use of heroism, but it's a double-edged sword. The hero can becomes a bottleneck to work getting done, and because want to solve all the problems themselves, they under-communicate.

20.3 Consultancies

When you work with an external consultancy, there is *always* more Agency Risk than with a direct employee. This is because as well as your goals and the employee's goals, there is also the consultancy's goals.

This is a good argument for not using consultancies, but sometimes the technical expertise they bring can outweigh this risk.

Also, try to look for *hungry* consultancies: if you being a happy client is valuable to them, they will work at a discount (either working cheaper, harder or longer or more carefully) as a result.

20.4 CV Building

This is when someone decides that the project needs a dose of "Some Technology X", but in actual fact, this is either completely unhelpful to the project (incurring large amounts of Complexity Risk), or merely less useful than something else.

It's very easy to spot CV building: look for choices of technology that are incongruently complex compared to the problem they solve, and then challenge by suggesting a simpler alternative.

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

20.5 Career Risk

20.6 Devil Makes Work

Heroes can be useful, but *underused* project members are a nightmare. The problem is, people who are not fully occupied begin to worry that actually, the team would be better off without them, and then wonder if their jobs are at risk.

The solution to this is "busy-work": finding tasks that, at first sight, look useful, and then delivering them in an over-elaborate way (Gold Plating)) that'll keep them occupied. This will leave you with more Complexity Risk than you had in the first place.

Even if they don't worry about their jobs, doing this is a way to stave off *boredom*.

20.7 Pet Projects

> A project, activity or goal pursued as a personal favourite, rather than because it is generally accepted as necessary or important. - Pet Project, *Wiktionary*[7]

Sometimes, budget-holders have projects they value more than others without reference to the value placed on them by the business. Perhaps the project has a goal that aligns closely with the budget holder's passions, or its related to work they were previously responsible for.

Working on a pet project usually means you get lots of attention (and more than enough budget), but due to Map and Territory Risk, it can fall apart very quickly under scrutiny.

20.8 Morale Risk

> Morale, also known as Esprit de Corps is the capacity of a group's members to retain belief in an institution or goal, particularly in the face of opposition or hardship - Morale, *Wikipedia*[8]

Sometimes, the morale of the team or individuals within it dips, leading to lack of motivation. Morale Risk is a kind of Agency Risk because it really

[7] https://en.wiktionary.org/wiki/pet_project

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

means that a team member or the whole team isn't committed to the Goal, may decide their efforts are best spent elsewhere. Morale Risk might be caused by:

- External factors: Perhaps the employees' dog has died, or they're simply tired of the industry, or are not feeling challenged.
- If the team don't believe a goal is achievable, they won't commit their full effort to it. This might be due to to a difference in the evaluation of the risks on the project between the team members and the leader.
- If the goal isn't considered sufficiently worthy, or the team isn't sufficiently valued.
- In military science, a second meaning of morale is how well supplied and equipped a unit is. This would also seem like a useful reference point for IT projects. If teams are under-staffed or under-equipped, this will impact on motivation too.

20.9 Hubris & Ego

It seems strange that humans are over-confident. You would have thought that evolution would drive out this trait but apparently it's not so:

> "Now, new computer simulations show that a false sense of optimism, whether when deciding to go to war or investing in a new stock, can often improve your chances of winning."
> —Evolution of Narcissism, *National Geographic*[9]

In any case, humans have lots of self-destructive tendencies that *haven't* been evolved away, and we get by.

Development is a craft, and ideally, we'd like developers to take pride in their work. Too little pride means lack of care, but too much pride is *hubris*, and the belief that you are better than you really are. Who does hubris benefit? Certainly not the team, and not the goal, because hubris blinds the team to hidden risks that they really should have seen.

Although over-confidence might be a useful trait when bargaining with other humans, the thesis of everything so far is that Meeting Reality will punish your over-confidence again and again.

Perhaps it's a little unfair to draw out one human characteristic for attention. After all, we are riddled with biases. There is probably an interesting article

[9] https://news.nationalgeographic.com/news/2011/09/110914-optimism-narcissism-overconfidence-hubris-evolution-science-nature/

to be written about the effects of different biases on the software development and project management processes. (This task is left as an exercise for the reader.)

20.10 Software Processes And Teams

Agency Risk doesn't just refer to people - it refers to anything which has agency over it's actions.

> "Agency is the capacity of an actor to act in a given environment... Agency may either be classified as unconscious, involuntary behaviour, or purposeful, goal directed activity (intentional action)."
>
> —Agency, *Wikipedia*[10])

There is significant Agency Risk in running software *at all*. Since computer systems follow rules we set for them, we shouldn't be surprised when those rules have exceptions that lead to disaster. For example:

- A process continually writing log files until the disks fill up, crashing the system.
- Bugs causing data to get corrupted, causing financial loss.
- Malware infecting a system, and sending your passwords and data to undesirables.

Agency Risk also covers *whole teams* too. It's perfectly possible that a team within an organisation develops Goals that don't align with those of the overall organisation. For example:

- A team introduces excessive Bureaucracy in order to avoid work it doesn't like.
- A team gets obsessed with a particular technology, or their own internal process improvement, at the expense of delivering business value.
- A marginalised team forces their services on other teams in the name of "consistency". (This can happen a lot with "Architecture", "Branding" and "Testing" teams, sometimes for the better, sometimes for the worse.)

[10] `https://en.wikipedia.org/wiki/Agency_(philosophy`

20.11 Security

Security

Intersecting both the internal and external environments are security concerns.

Interestingly, security is handled in very similar ways at all sorts of levels:

- **Walls**: defences *around* the complex system, to protect it's parts from the external environment.
- **Doors**: ways to get *in* and *out* of the complex system, possibly with *locks*.
- **Guards**: to make sure only the right things go in and out. (i.e. to try and keep out *Bad Actors*).
- **Police**: to defend from *within* the system, against Agency Risk and *invaders*.
- **Subterfuge**: Hiding, camouflage, disguises, pretending to be something else. tbd

These work various levels in our own bodies: our *cells* have *cell walls* around them, and *cell membranes* that act as the guards to allow things in and out. Our *bodies* have *skin* to keep the world out, and we have *mouths*, *eyes*, *pores* and so on to allow things in and out. We have an *immune system* to act as the police.

Our societies work in similar ways: in medieval times, a city would have walls, guards and doors to keep out intruders. Nowadays, we have customs control, borders and passports.

We're waking up to the realisation that our software systems need to work the same way: we have Firewalls to protect our organisations, we lock down *ports* on servers to ensure there are the minimum number of doors to guard and we *police* the servers ourselves with monitoring tools and anti-virus software.

```
- Security Risk
  - Hacking
  - Denial Of Service
  - Security, Trust and Complexity
  - oWASp
```

tbd, How much do compilers do for you? Now, they prevent many kinds of security error. Libraries too.

Security risk thrives on complexity. As does Agency Risk generally.

20.12 It's About Goals

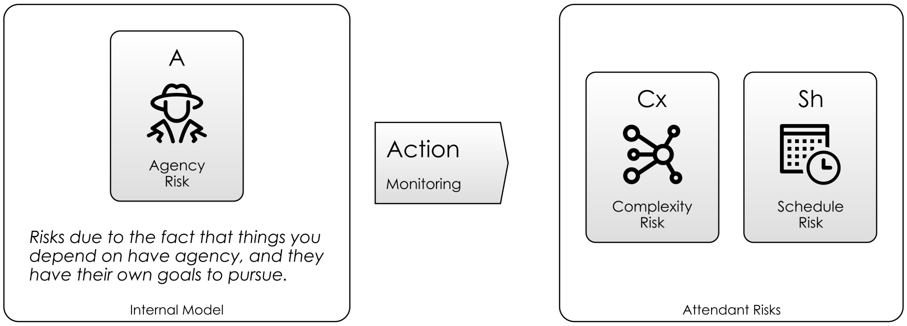


Figure 20.2: Agency Risk

We've looked here at some illustrative examples of Agency Risk. But as we stated at the beginning, Agency Risk at any level comes down to differences of Goals between the different agents, whether they are *people*, *teams* or *software*.

So, having looked at agents *individually*, it's time to look more closely at Goals, and the Attendant Risks when aligning them amongst multiple agents.

On to Coordination Risk. . .

CHAPTER 21

Coordination Risk

Coordination Risk is the risk that, a group of people (or processes), maybe with a similar Goal In Mind they can fail to coordinate on a way to meet this goal and end up making things worse. Coordination Risk is embodied in the phrase "Too Many Cooks Spoil The Broth": more people, opinions or agents often make results worse.

As in Agency Risk, we are going to use the term *agent*, which refers to anything with agency[1] in a system to decide it's own fate. That is, an agent has an Internal Model, and can take actions based on it. Here, we're going to work on the assumption that the agents *are* working towards a common Goal, even though in reality it's not always the case, as we saw in the chapter on Agency Risk.

In this chapter, we'll first build up A Model Of Coordination Risk and what exactly coordination means and why we do it. Then, we'll look at some classic Problems of Coordination. Then, we're going to consider agency at several different levels (because of Scale Invariance) . We'll look at: - Team Decision Making, - Living Organisms, - Larger Organisations and the staff within them, - and Software Processes.

. . . and we'll consider how Coordination Risk is a problem at each scale.

But for now, let's crack on and examine where Coordination Risk comes from.

21.1 A Model Of Coordination Risk

Earlier, in Dependency Risk, we looked at various resources (time, money, people, events etc) and showed how we could Depend On Them, taking on

[1] https://github.com/risk-first/website/wiki/Agency-Risk#software-processes-and-teams

risk. Here, however, we're looking at the situation where there is *competition for those dependencies*, that is, Scarcity Risk: other parties want to use them in a different way.

Competition

The basic problem of Coordination Risk, then, is *competition*. Sometimes, competition is desireable (such as in sports and in markets), but sometimes competition is a waste and cooperation would be more efficient. Without coordination, we would deliberately or accidentally compete for the same Dependencies, which is wasteful.

Why is this wasteful?

One argument could come from Diminishing Returns[2], which says that the earlier units of a resource (say, chocolate bars) give you more benefit than later ones.

We can see this in the graph below. Let's say A and B compete over a resource, of which there are 5 units available. For every extra A takes, B loses one. The X axis shows A's consumption of the resource, so the biggest benefit to A is in the consumption of the first unit.

As you can see, by *sharing*, it's possible that the *total benefit* is greater than it can be for either individual. But sharing requires coordination. Further, the more competitors involved, the *worse* a winner-take-all outcome is for total benefit.

Just two things are needed for competition to occur:

- Individual agents, trying to achieve Goals.
- Scarce Resources, which the agents want to use as Dependencies.

Coordination via Communication

The only way that the agents can move away from competition towards coordination is via Communication, and this is where their coordination problems begin.

You might think, therefore, that this is just another type of Communication Risk problem, and that's often a part of it, but even with synchronized Internal Models, coordination risk can occur. Imagine the example of people all trying to madly leave a burning building. They all have the same information

[2] https://en.wikipedia.org/wiki/Diminishing_returns

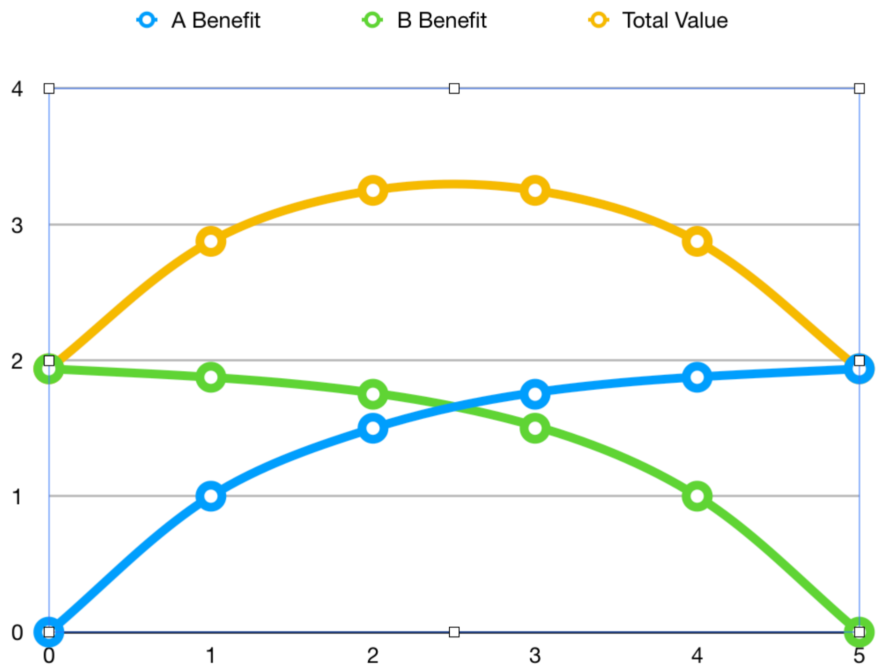


Figure 21.1: Sharing Resources. 5 units are available, and the X axis shows A's consumption of the resource. B gets whatever remains. Total benefit is maximised somewhere in the middle

(the building is on fire). If they coordinate, and leave in an orderly fashion, they might all get out. If they don't, and there's a scramble for the door, more people might die.

But commonly, Coordination Risk occurs where people have different ideas about how to achieve a goal, and they have different ideas because they have different evaluations of the Attendant Risk. As we saw in the chapter on Communication Risk, we can only hope to synchronize Internal Models if there are high-bandwidth Channels available for communication.

21.2 Problems Of Coordination

Let's unpack this idea, and review some classic problems of coordination, none of which can be addressed without good communication:

1. **Merging Data**. If you are familiar with the source code control system, Git, you will know that this is a *distributed* version control system. That

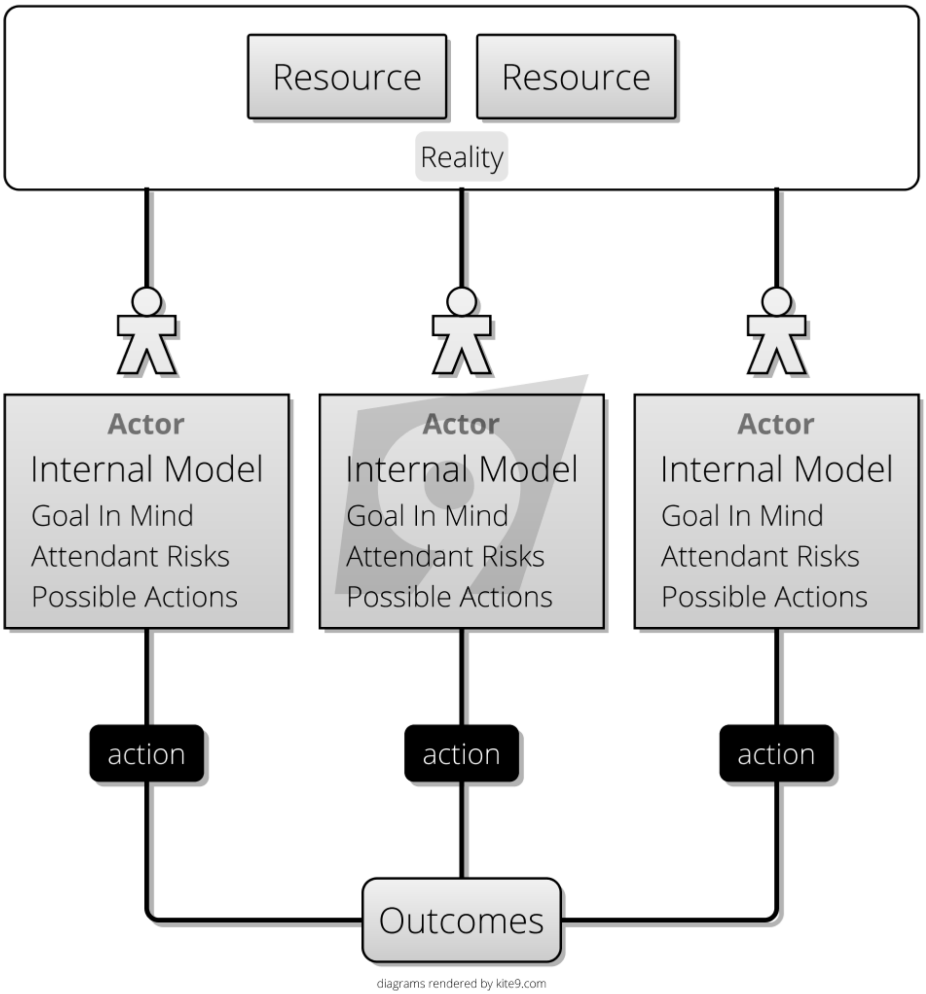


Figure 21.2: A model of competition: scarce resources, and individual agents competing for them.

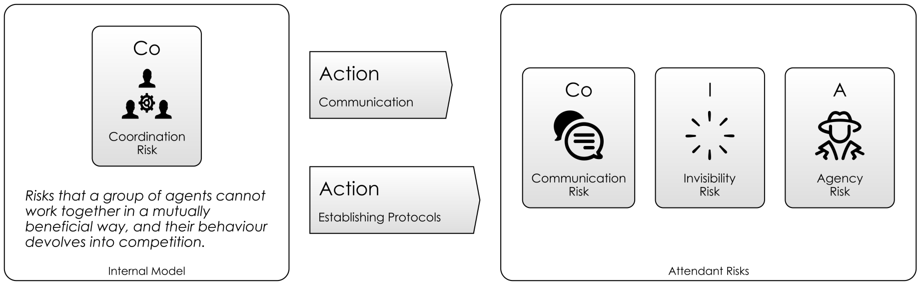


Figure 21.3: Coordination Risk - Mitigated by Communication

means that two or more people can propose changes to the same files without knowing about each other. This means that at some later time, Git then has to merge (or reconcile) these changes together. Git is very good at doing this automatically, but sometimes, different people can independently change the same lines of code and these will have to be merged manually. In this case, a human arbitrator "resolves" the difference, either by combining the two changes or picking a winner.

2. **Consensus**. Making group decisions (as in elections) is often decided by votes. But having a vote is a coordination issue, and requires that everyone has been told the rules:

- Where will the vote be held?
- How long do you provide for the vote?
- What do you do about absentees?
- What if people change their minds in the light of new information?
- How do you ensure everyone has enough information to make a good decision?

3. **Factions**. Sometimes, it's hard to coordinate large groups at the same time, and "factions" can occur. That the world isn't a single big country is probably partly a testament to this: countries are frequently separated by geographic features that prevent the easy flow of communication (and force). We can also see this in distributed systems, with the "split brain") problem. This is where a network of processes becomes disconnected (usually due to a network failure between data centers), and you end up with two, smaller networks with different knowledge. We'll address in more depth later.

4. Resource Allocation[3]: Ensuring that the right people are doing the right work, or the right resources are given to the right people is a coordination issue. On a grand scale, we have Logistics, and Economic Systems[4]. On a small scale, the office's *room booking system* solves the coordination issue of who gets a meeting room using a first-come-first-served booking algorithm.

5. Deadlock: Deadlock refers to a situation where, in an environment where multiple parallel processes are running, the processing stops and no-one can make progress because the resources each process needs are being reserved by another process. This is a specific issue in Resource Allocation, but it's one we're familiar with in the computer science industry. Compare with Gridlock[5], where traffic can't move because other traffic is occupying the space it wants to move to already.

6. Race Conditions: A race condition is where we can't be sure of the result of a calculation, because it is dependent on the ordering of events within a system. For example, two separate threads writing the same memory at the same time (one ignoring and over-writing the work of the other) is a race.

7. **Contention**: Where there is Scarcity Risk for a Dependency, we might want to make sure that everyone gets fair use of it, by taking turns, booking, queueing and so on. As we saw in Scarcity Risk, sometimes, this is handled for us by the Dependency itself. However if it isn't, it's the *users* of the dependency who'll need to coordinate to use the resource fairly, again, by communicating with each other.

21.3 Team Decision Making

Within a team, Coordination Risk is at it's core about resolving Internal Model conflicts in order that everyone can agree on a Goal In Mind and cooperate on getting it done. Therefore, Coordination Risk is worse on projects with more members, and worse in organizations with more staff.

If you are engaged in a solo project, do you suffer from Coordination Risk at all? Maybe: sometimes, you can feel "conflicted" about the best way to solve a problem. And weirdly, usually *not thinking about it* helps. Sleeping too. (Rich Hickey calls this "Hammock Driven Development[6]"). This is probably

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

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

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

[6]https://www.youtube.com/watch?v=f84n5oFoZBc

because, unbeknownst to you, your subconscious is furiously communicating internally, trying to resolve these conflicts itself, and will let you know when it's come to a resolution.

Vroom and Yetton[7] introduced a model of group decision making which delineated five different styles of decision making within a team. These are summarised in the table below (**AI, AII, CI, CII, GII**). To this, I have added a sixth (**UI**), which is the *uncoordinated* option, where everyone competes. In the accompanying diagrams I have adopted the following convention: - Thin lines with arrow-heads show a flow of *information*, either one-way or two-way. - Thick lines show a flow of *opinion*. - Boxes with corners are *decision makers*, whereas curved corners don't have a part in the decision.

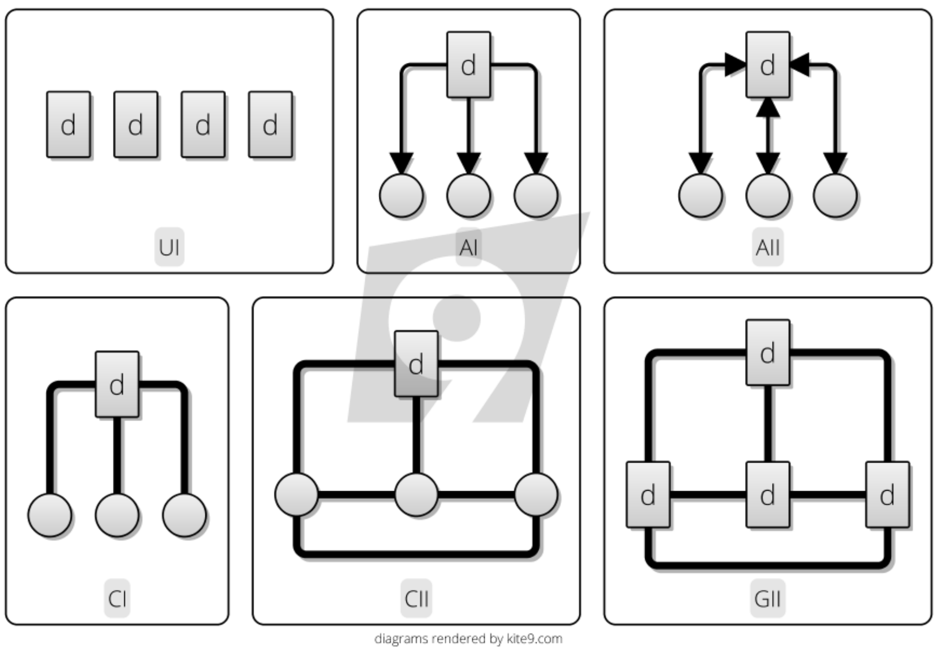


Figure 21.4: Vroom And Yetton Decision Making Styles. "d" indicates authority in making a decision. Thin lines with arrow-heads show information flow, whilst thick lines show* opinions *being passed around.

At the top, you have the *least* consultative styles, and at the bottom, the *most*. At the top, decisions are made with just the leader's Internal Model but moving down, the Internal Models of the rest of the team are increasingly brought into play.

[7]https://en.wikipedia.org/wiki/Vroom-Yetton_decision_model

Type	People Involved In Decision	Opinions	Channels Of Communication	Coordination Risk	Description
UI	1	1	0	Competition	*No Coordination*
AI	1	1	**s** (One message to each **subordinate**)	Maximum Coordination Risk	Autocratic, top-down
AII	1	1	2 x **s** (Messages from/to each **subordinate**)		Autocratic, with information flow up.
CI	1	1 + **s**	> 2 x **s**		Individual Consultations
CII	1	1 + **s**	> **s**2		Group Consultation
GII	1 + **s**	1 + **s**	> **s**2	Maximum Communication Risk, Schedule Risk	Group Consultation with voting

The decisions at the top are faster, but don't do much for mitigating **Coordination Risk**. The ones below take longer, (incurring Schedule Risk) but mitigate more **Coordination Risk**. Group decision-making inevitably involves everyone *learning*, and improving their Internal Models.

The trick is to be able to tell which approach is suitable at which time. Everyone is expected to make decisions *within their realm of expertise*: you can't have developers continually calling meetings to discuss whether they should be using an Abstract Factory[8] or a Factory Method[9], this would waste time. The critical question is therefore, "what's the biggest risk?" - Is the Coordination Risk greater? Are we going to suffer Dead End Risk if the decision is made wrongly? What if people don't agree with it? Poor leadership has an impact on Morale too. - Is the Schedule Risk greater? If you have a 1-hour meeting with eight people to decide a decision, that's *one man day* gone right there: group decision making is *expensive*.

Hopefully, this model shows how *organisation* can reduce Coordination Risk. But, to make this work, we need more *communication*, and this has attendant complexity and time costs. So, we can draw this diagram of our move on the Risk Landscape:

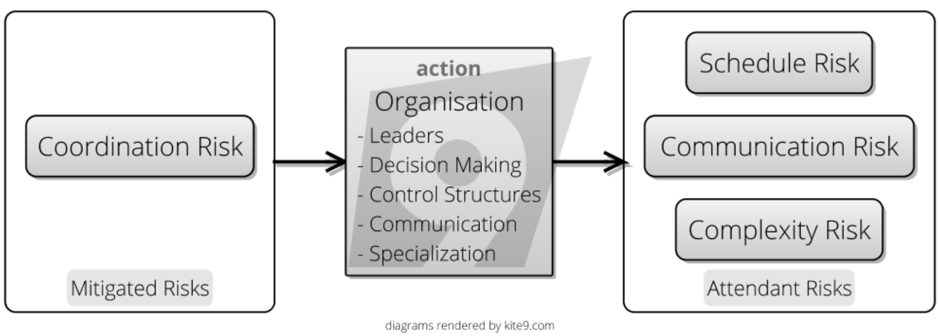


Figure 21.5: Coordination Risk traded for Complexity Risk, Schedule Risk and Communication Risk

Staff As Agents

Staff in a team have a dual nature: they are **Agents** and **Resources** at the same time. The team depends on staff for their resource of *labour*, but they're also part of the decision making process of the team, because they have agency over their own actions.

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

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

Part of Coordination Risk is about trying to mitigate differences in Internal Models. So it's worth considering how varied people's models can be: - Different skill levels - Different experiences - Expertise in different areas - Preferences - Personalities

The job of harmonzing this on a project would seem to fall to the team leader, but actually people are self-organising to some extent. This process is called Team Development[10]:

> "The forming–storming–norming–performing model of group development was first proposed by Bruce Tuckman in 1965, who said that these phases are all necessary and inevitable in order for the team to grow, face up to challenges, tackle problems, find solutions, plan work, and deliver results."
>
> —Tuckman's Stages Of Group Development, *Wikipedia*[11]

Specifically, this describes a process whereby a new group will form and then be required to work together. In the process, they will have many *disputes*. Ideally, the group will resolve these disputes internally and emerge as a *Team*, with a common Goal In Mind.

They can be encouraged with orthogonal practices such as team-building exercises[12] (generally, submitting everyone to extreme experiences in order to bond them together). With enough communication bandwidth and detente, a motivated team will self-organise code reviews, information exchange and improve their practices.

As decribed above, the job of Coordination is Resource Allocation, and so the skills of staff can potentially be looked at as resources to allocate. This means handling Coordination Risk issues like:

- People leaving, taking their Internal Models and expertise with them Key Man Risk.
- People requiring external training, to understand new tools and techniques Learning-Curve Risk.
- People being protective about their knowledge in order to protect their jobs Agency Risk.
- Where there are mixed ability levels, senior developers not helping juniors as it "slows them down".
- People not getting on and not helping each other.

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

[11] https://en.wikipedia.org/wiki/Tuckmans_stages_of_group_development

[12] https://en.wikipedia.org/wiki/Team_building

> "As a rough rule, three programmers organised into a team can do only twice the work of a single programmer of the same ability - because of time spent on coordination problems."
>
> Gerald Wienberg, The Psychology of Computer Programming[13]

21.4 In Living Organisms

Vroom and Yetton's organisational style isn't relevant to just teams of people. We can see it in the natural world too. Although *the majority* of cellular life on earth (by weight) is single celled organisms[14], the existence of *humans* (to pick a single example) demonstrates that sometimes it's better to try to mitigate Coordination Risk and work as a team, accepting the Complexity Risk and Communication Risk this entails. As soon as cells start working together, they either need to pass *resources* between them, or *control* and *feedback*.

For example, in the human body, we have various systems[15]:

- The Respiratory System[16] which is responsible for ensuring that Red Blood Cells[17] are replenished with Oxygen, as well as disposing of Carbon Dioxide.
- The Digestive System[18] which is responsible for extracting nutrition from food and putting them in our Blood Plasma[19].
- The Circulatory System[20] which is responsible for moving blood cells to all the rest of the body.
- The Nervous System[21] which is responsible for collecting information from all the parts of the body, dealing with it in the Brain[22] and issuing commands.
- The Motor System[23] which contains muscles and bones, and allows us to move about.

. . . and many others. Each of these systems contains organs, which contain tissues, which contain cells of different types. (Even cells are complex systems

[13] https://en.wikipedia.org/wiki/Gerald_Weinberg
[14] http://www.stephenjaygould.org/library/gould_bacteria.html
[15] https://en.wikipedia.org/wiki/List_of_systems_of_the_human_body
[16] https://en.wikipedia.org/wiki/Respiratory_system
[17] https://en.wikipedia.org/wiki/Red_blood_cell
[18] https://en.wikipedia.org/wiki/Human_digestive_system
[19] https://en.wikipedia.org/wiki/Blood_plasma
[20] https://en.wikipedia.org/wiki/Circulatory_system
[21] https://en.wikipedia.org/wiki/Nervous_system
[22] https://en.wikipedia.org/wiki/Brain
[23] https://en.wikipedia.org/wiki/Motor_system

containing multiple different, communicating sub-systems.) There is huge Complexity Risk here: the entire organism fails if one of these systems fail (they are Single Points Of Failure, although we can get by despite the failure of one lung or one leg say).

Some argue[24] that the human nervous system is the most complex known artifact in the universe: there is huge attendant Communication Risk to running the human body. But, given the success of humanity as a species, you must conclude that these steps on the evolutionary Risk Landscape have benefitted us in our ecological niche.

The key observation from looking at biology is this: most of the cells in the human body *don't get a vote*. Muscles in the motor system have an **AI** or **AII** relationship with the brain - they do what they are told, but there are often nerves to report pain back. The only place where **CII** or **GII** *could* occur is in our brains, when we try to make a decision and weigh up the pros and cons.

This means that there is a deal: *most* of the cells in our body accede control of their destiny to "the system". Living within the system of the human body is a better option than going it alone. Occasionally, due to mutation, we can end up with Cancer[25], which is where one cell genetically "forgets" its purpose in the whole system and goes back to selfish individual self-replication (**UI**). We have White Blood Cells[26] in the body to shut down this kind of behaviour and try to kill the rogue cells. In the same way, society has a police force to stop undesireable behaviour amongst its citizens.

21.5 Large Organisations

Working in a large organisation often feels like being a cell in a larger organism. Just as cells live and die, but the organism goes on, in the same way, workers come and go from a large company but the organisation goes on. By working in an organisation, we give up self-control and competition and accept **AI** and **AII** power structures above us, but we trust that there is symbiotic value creation on both sides of the employment deal.

Less consultative decision making styles are more appropriate then when we don't have the luxury of high-bandwidth channels for discussion, or when the number of parties rises above a room-full of people. As you can see from the table above, for **CII** and **GII** decision-making styles, the amount of communication increases non-linearly with the number of participants,

[24] https://www.quora.com/What-is-the-most-complex-object-in-the-universe

[25] https://en.wikipedia.org/wiki/Cancer

[26] https://en.wikipedia.org/wiki/White_blood_cell

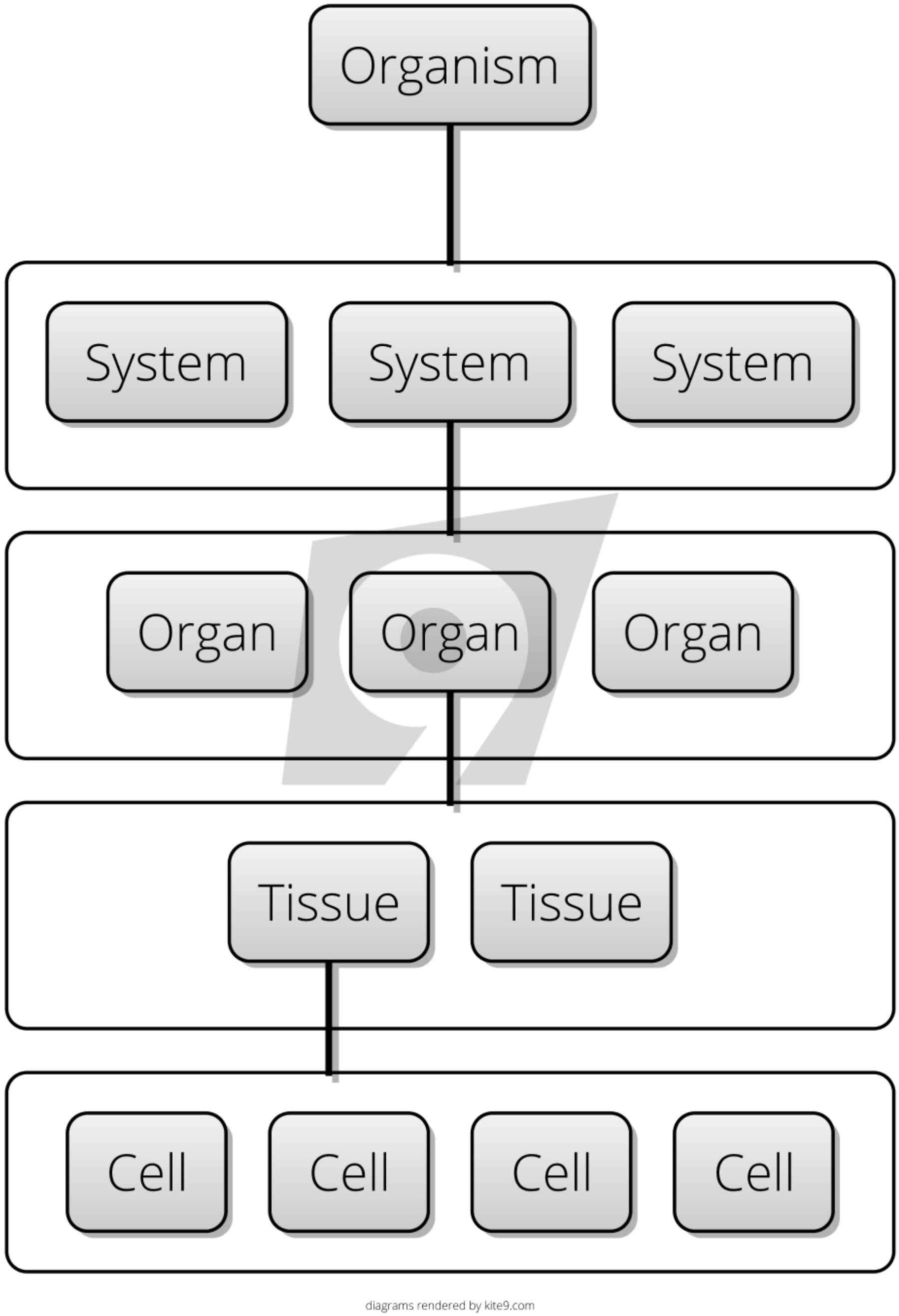


Figure 21.6: Hierarchy of Function in the Human Body

so we need something simpler. As we saw in the Complexity Risk chapter, hierarchies are an excellent way of economizing on number of different communication channels, and we use these frequently when there are lots of parties to coordinate.

In large organisations, teams are created and leaders chosen for those teams precisely to mitigate Communication Risk. We're all familiar with this: control of the team is ceded to the leader, who takes on the role of 'handing down' direction from above, but also 'reporting up' issues that cannot be resolved within the team. In Vroom and Yetton's model, this is moving from a **GII** or **CII** to an **AI** or **AII** style of leadership.

As shown in the diagram above, we end up with a hierarchy of groups, each having it's own decision-making style. The team leader at the bottom level is a *decision maker* within his team, but moving up, doesn't have decision making power in the next team up.. and so on.

Sometimes, parts of an organisation are encouraged *not* to coordinate, but to compete. In the diagram above, we have an M-Form[27] organisation, composed of *competing divisions*.

Clearly, this is just a *model*, it's not set in stone and decision making styles usually change from day-to-day and decision to decision. The same is not true in our software - *rules are rules*.

21.6 In Software Processes

It should be pretty clear that we are applying the Scale Invariance rule to Coordination Risk: all of the problems we've described as affecting teams, also affect software, although the scale and terrain are different. Software processes have limited *agency* - in most cases they follow fixed rules set down by the programmers, rather than self-organising like people can (so far).

As before, in order to face Coordination Risk in software, we need multiple agents all working together. Coordination Risks (such as race conditions or deadlock) only really occurs where *more than one thing is happening at a time*. This means we are considering *at least* multi-threaded software and anything above that (multiple CPUs, servers, data-centres and so on).

CAP Theorem

The CAP Theorem[28] has a lot to say about Coordination Risk. Imagine talking to a distributed database, where your request (*read* or *write*) can be handled

[27] https://en.wikipedia.org/wiki/Multi-divisional_form

[28] https://en.wikipedia.org/wiki/CAP_theorem

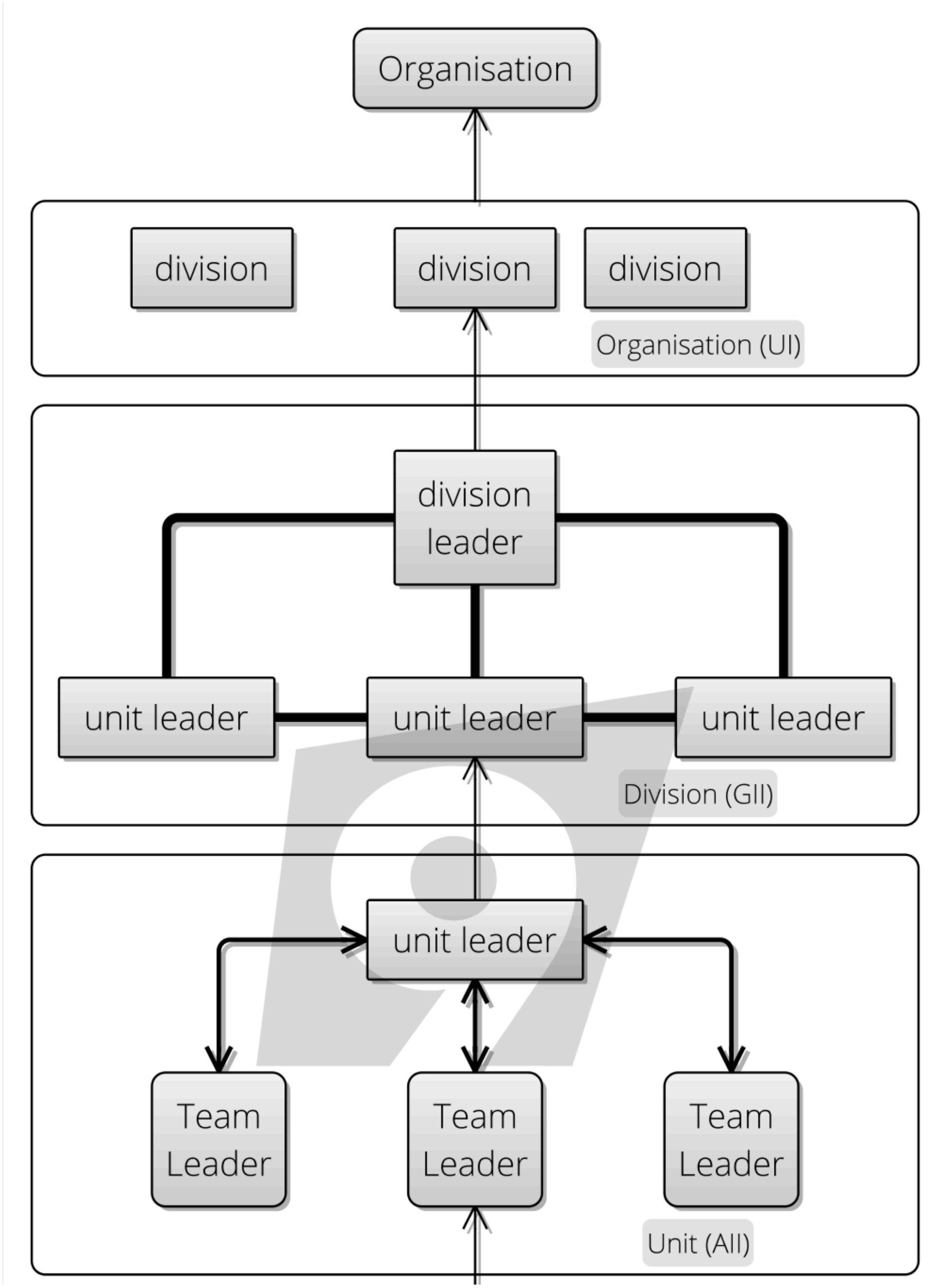


Figure 21.7: Hierarchy of Function in an Organisation

by one of many agents.

In the diagram below, we have just two agents `1` and `2`, in order to keep things simple. `User A` *writes something* to the database, then `User B` *reads it back* afterwards.

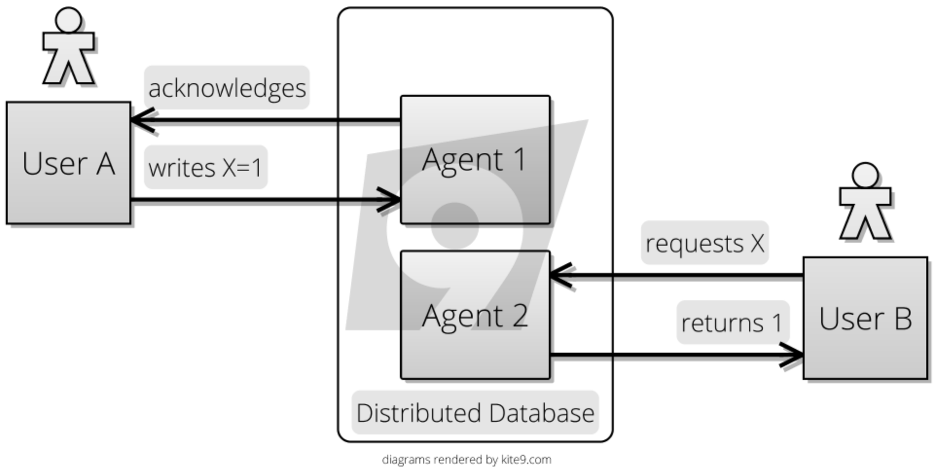


Figure 21.8: User A and User B are both using a distributed database, managed by Agents 1 and 2, whom each have their own Internal Model

According to the CAP Theorem, there are three properties we could desire in such a system:

- **Consistency**: Every read receives the most recent value from the last write.
- **Availability**: Every request receives a response.
- **Partition tolerance**: The system can operate despite the isolation (lack of communication with) some of it's agents.

The CAP Theorem states that this is a Trilemma[29]. That is, you can only have two out of the three properties.

There are plenty of resources on the internet that discuss this in depth, but let's just illustrate with some diagrams to show how this plays out. In our diagram example, we'll say that *any* agent can receive the read or write. So this might be a **GII** decision making system, because all the agents are going to need to coordinate to figure out what the right value is to return for a read,

[29] `https://en.wikipedia.org/wiki/Trilemma`

and what the last value written was. In these, the last write (setting X to 1) was sent to Agent 1 which then becomes *isolated*, and can't be communicated with, due to network failure. What will User B get back?

With an AP System

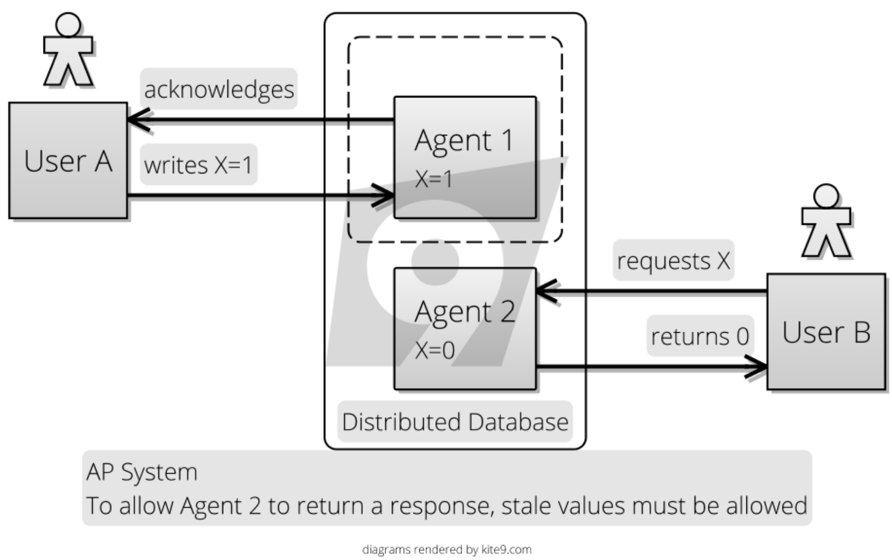
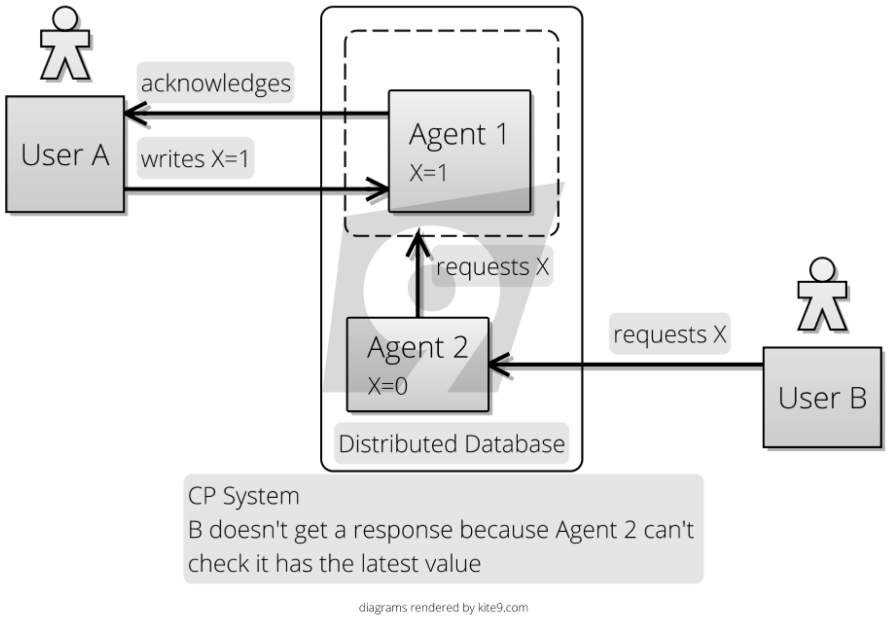


Figure 21.9: In an AP system, the User B will get back a **stale value** *for X*

With `AP`, you can see that `User B` is getting back a *stale value*. `AP` scenarios lead to Race Conditions: `Agent 1`s availability determines what value `User B` gets back.

With an CP System



.

Where Agent 2 is left waiting for Agent 1 to re-appear, we are *blocked*. So CP systems lead to Deadlock scenarios.

With an CA System

Finally, if we have a CA system, we are essentially saying that *only one agent is doing the work*. (You can't partition a single agent, after all). But this leads to Resource Allocation and **Contention** around use of the scarce resource of `Agent 2`'s attention. (Both Coordination Risk issues we met earlier.)

Some Real-Life Examples

This sets an upper bound on Coordination Risk: we *can't* get rid of it completely in a software system, -or- a system on any other scale. Fundamentally, coordination problems are inescapable at some level. The best we can do is mitigate it by agreeing on protocols and doing lots of communication.

Let's look at some real-life examples of how this manifests in software.

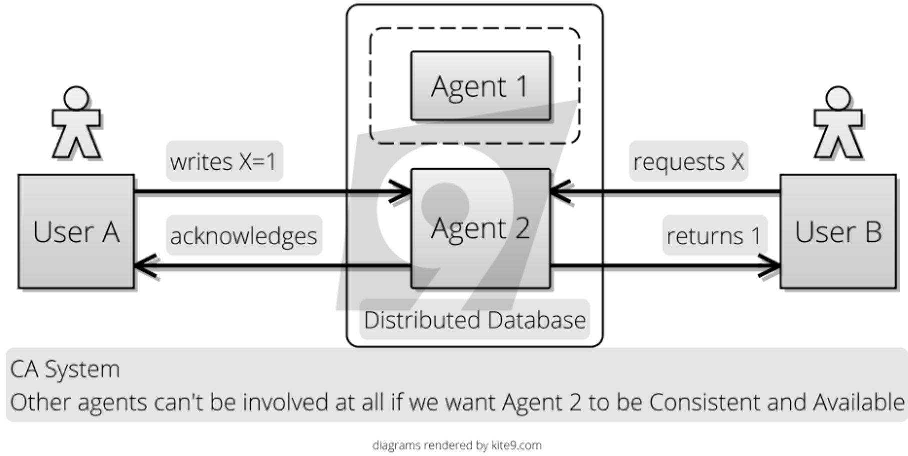


Figure 21.10: In an CA system, we can't have partition tolerance, so in order to be consistent a single Agent has to do all the work

ZooKeeper

First, ZooKeeper[30] is an Open-Source datastore, which is used a lot for coordinating a distributed systems, and storing things like configuration information across them. If the configuration of a distributed system gets changed, it's important that *all of the agents in the system know about it*, otherwise... disaster.

This *seems* trivial, but it quickly gets out-of-hand: what happens if only some of the agents receive the new information? What happens if a datacentre gets disconnected while the update is happening? There are lots of edge-cases.

ZooKeeper handles this by communicating inter-agent with it's own protocol. It elects a **master agent** (via voting), turning it into an **AI**-style team. If the master is lost for some reason, a new leader is elected. *Writes* are then coordinated via the **master agent** who makes sure that a *majority of agents* have received and stored the configuration change before telling the user that the transaction is complete. Therefore, ZooKeeper is a `CP` system.

Git

Second, git is a (mainly) write-only ledger of source changes. However, as we already discussed above, where different agents make incompatible changes, someone has to decide how to resolve the conflicts so that we have a single source of truth.

[30] `https://zookeeper.apache.org`

The Coordination Risk just *doesn't go away*.

Since multiple users can make all the changes they like locally, and merge them later, Git is an `AP` system: individual users may have *wildly* different ideas about what the source looks like until the merge is complete.

Bitcoin

Finally, Bitcoin (BTC)[31] is a write-only distributed ledger[32], where agents *compete* to mine BTC, but also at the same time record transactions on the ledger. BTC is also `AP`, in a similar way to Git. But new changes can only be appended if you have the latest version of the ledger. If you append to an out-of-date ledger, your work will be lost.

Because it's based on outright competition, if someone beats you to completing a mining task, then your work is wasted. So, there is *huge* Coordination Risk.

For this reason, BTC agents coordinate into mining consortia[33], so they can avoid working on the same tasks at the same time. But this in itself is a problem, because the whole *point* of BTC is that it's competitive, and no one entity has control. So, mining pools tend to stop growing before they reach 50% of the BTC network's processing power. Taking control would be politically disastrous[34] and confidence in the currency (such as there is) would likely be lost.

21.7 Communication Is For Coordination

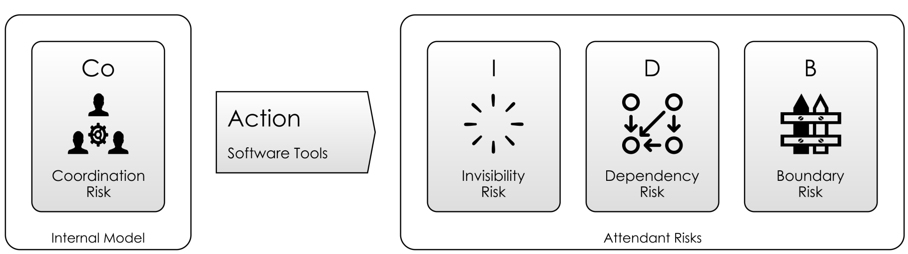


Figure 21.11: Coordination Risk - Mitigated by Software Tools

[31] `https://en.wikipedia.org/wiki/Bitcoin`

[32] `https://en.wikipedia.org/wiki/Distributed_ledger`

[33] `https://en.bitcoin.it/wiki/Comparison_of_mining_pools`

[34] `https://www.reddit.com/r/Bitcoin/comments/5fe9vz/in_the_last_24hrs_three_mining_pools_have_control/`

So, now we have a fundamental limit on how much Coordination Risk we can mitigate. And, just as there are plenty of ways to mitigate Coordination Risk within teams of people, organisations or living organisms, so it's the case in software.

Earlier in this chapter, we questioned whether Coordination Risk was just another type of Communication Risk. However, it should be clear after looking at the examples of competition, cellular life and Vroom and Yetton's Model that this is exactly *backwards*:

- Most single-celled life has no need for communication: it simply competes for the available resources. If it lacks anything it needs, it dies.
- There are *no* lines of communication on the **UI** decision-type. It's only when we want to avoid competition, by sharing resources and working towards common goals that we need to communicate.
- Therefore, the whole point of communication *is for coordination*.

In the next chapter, Map And Territory Risk, we're going to look at some new ways in which systems can fail, despite their attempts to coordinate.

CHAPTER 22

Map And Territory Risk

As we discussed in the chapter on Abstraction, our understanding of the world is entirely informed by the names we give things and the abstractions we create.

(In the same way, **Risk-First** is about *identifying patterns* within software development and calling them out.)

Our Internal Models are a model of the world based on these patterns, and their relationships.

So there is a translation going on here: observations about the arrangement of *atoms* in the world get turned into patterns of *information* (measured in bits and bytes).

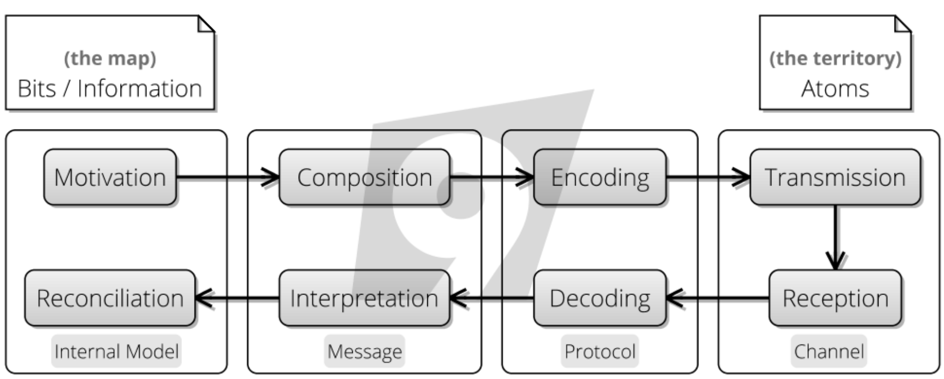


Figure 22.1: Maps and Territories, and Communication happening between them

Map And Territory Risk is the risk we face because we base our behaviour on our Internal Models rather than reality itself. It comes from the expression "Confusing the Map for the Territory", attributed to Alfred Korzybski:

> "Polish-American scientist and philosopher Alfred Korzybski remarked that"the map is not the territory" and that "the word is not the thing", encapsulating his view that an abstraction derived from something, or a reaction to it, is not the thing itself. Korzybski held that many people *do* confuse maps with territories, that is, confuse models of reality with reality itself."
>
> —Map-Territory Relation, *Wikipedia*[1]

In this chapter, we're going to make a case for analysing Map and Territory Risk along the same axes we introduced for Feature Risk, that is **Fitness**, **Audience** and **Evolution**. After that, we are going to widen the scope by looking at Map and Territory Risk within the context of **machines**, **people**, **hierarchies** and **markets**.

tbd - diagram of how our actions are based on the map, not the territory.

22.1 Fitness

In the picture shown here, from the Telegraph newspaper, the driver *trusted* the SatNav to such an extent that he didn't pay attention to the road-signs around him, and ended up getting stuck.

This wasn't borne of stupidity, but experience: SatNavs are pretty reliable. *So many times* the SatNav had been right, that the driver stopped *questioning its fallibility*.

So, there are two Map and Territory Risks here:

- The Internal Model of the *SatNav* contained information that was wrong: the track had been marked up as a road, rather than a path.
- The Internal Model of the *driver* was wrong: his abstraction of "the SatNav is always right" turned out to be only *mostly* accurate.

22.2 Internal Models as Dependencies, Features

What are the risks at play here? We've already looked in detail at the Dependency Risks involved in relying on something like a SatNav, in the Software

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

Figure 22.2: Sat Nav Blunder Sends Asda Van Crashing Narrow Footpath - Telegraph Newspaper

Dependency Risk chapter. But here, we are really looking at the *Internal Models themselves* as a source of Dependency Risk too.

We could argue that the SatNav and the Driver's Internal Model had bugs in them. That is, they both suffer the Feature Implementation Risk we saw in the Feature Risk chapter. If a SatNav has too much of this, you'd end up not trusting it, and getting a new one. With your *personal* Internal Model, you can't buy a new one, but you may learn to *trust certain abstractions less*, as this driver did.

In the Feature Risk chapter, we broke down Feature Risk on three axes: **Fitness**, **Evolution** and **Audience**.

Lets do this again and see how each type of Feature Risk can manifest in the Internal Model:

As with Features in a product, Information in an internal model has at least these three dimensions:

- **Fitness**: as discussed above with the SatNav example, this is how closely the information matches reality, and how *useful that is to us*

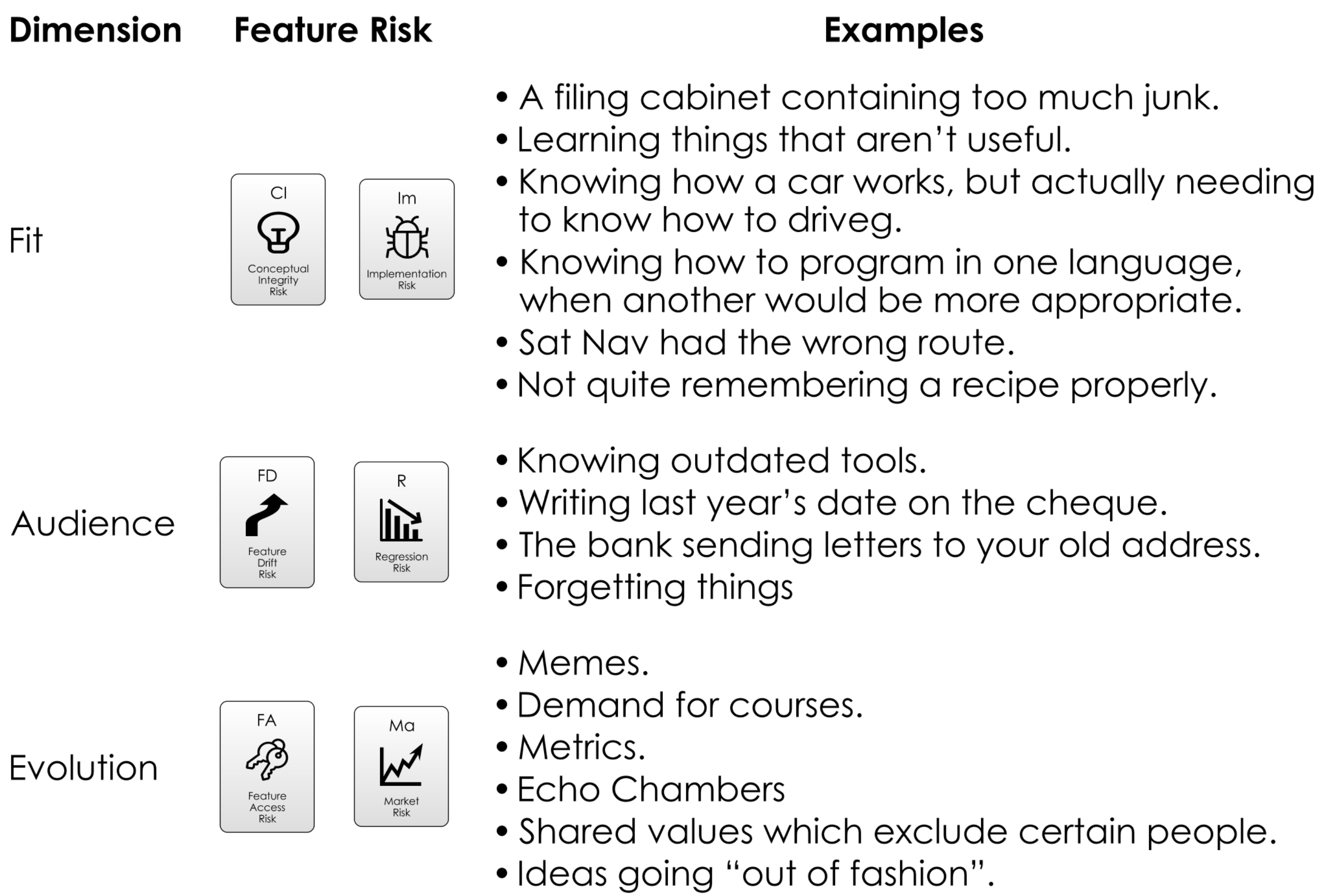


Figure 22.3: Feature Risk, as manifested in the Internal Model

(models that contain too much detail are as bad as models with too little).

- **Audience**: is all about how a piece of information is *shared* between many Internal Models, and it's this we are going to address further now.
- **Evolution**: is all about how Internal Models change when they meet reality, and we'll cover that last.

22.3 Audience

We already know a lot about Internal Models and audience, as these have been the subject of previous chapters:

- We know from looking at Communication Risk that communication allows us to *share* information between Internal Models.
- We know from Coordination Risk the difficulties inherent in aligning Internal Models so that they cooperate.
- Job markets show us that there is demand for people with certain *skills*. This demonstrates to us that Market Risk is as applicable to Internal Models containing certain information as it is to products containing Features. This was the focus of the Ecosystem discussion in Boundary Risk.

. . . And, we're all familiar with *memes*:

> "A meme acts as a unit for carrying cultural ideas, symbols, or practices, that can be transmitted from one mind to another through writing, speech, gestures, rituals, or other imitable phenomena with a mimicked theme."
>
> —Meme, *Wikipedia*[2]

Therefore, we should be able to track the rise-and-fall of *ideas* much as we can track stock prices. And in effect, this is what Google Trends[3] does. In the chart below, we can see the relative popularity of two search terms over time. This is probably as good an indicator as any of the audience for an abstraction at any point in time.

[2] https://en.wikipedia.org/wiki/Meme

[3] https://trends.google.com

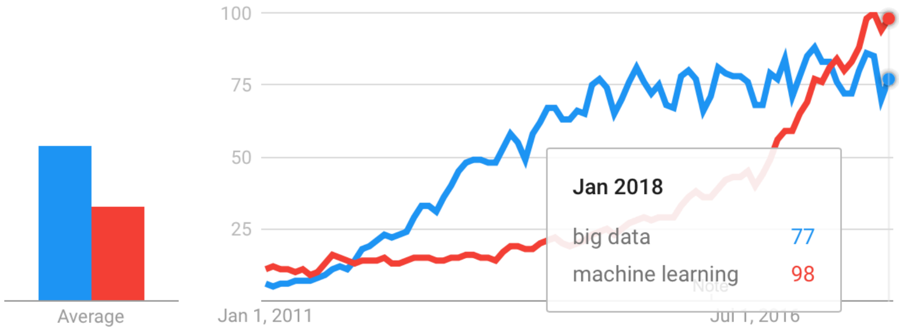


Figure 22.4: Relative popularity of "Machine Learning" and "Big Data" as search terms on Google Trends, 2011-2018

Example: Hype Cycles

Most ideas (and most products) have a slow, hard climb to wide-scale adoption. But some ideas seem to disperse much more rapidly and are picked up quickly because they are exciting and promising, having greater "memetic potential" within society. One way this evolution manifests itself in the world is though the Hype Cycle[4]:

> "The hype cycle is a branded graphical presentation developed and used by the American research, advisory and information technology firm Gartner, for representing the maturity, adoption and social application of specific technologies. The hype cycle provides a graphical and conceptual presentation of the maturity of emerging technologies through five phases."
>
> —Hype Cycle, *Wikipedia*[5]

The five phases (and the "Hype" itself) are shown in the chart below, with the thick black line being "Hype":

Also in this diagram we are showing where the hype originates:

- The **saturation** of the idea within the audience (a dotted line).

[4] https://en.wikipedia.org/wiki/Hype_cycle
[5] https://en.wikipedia.org/wiki/Hype_cycle

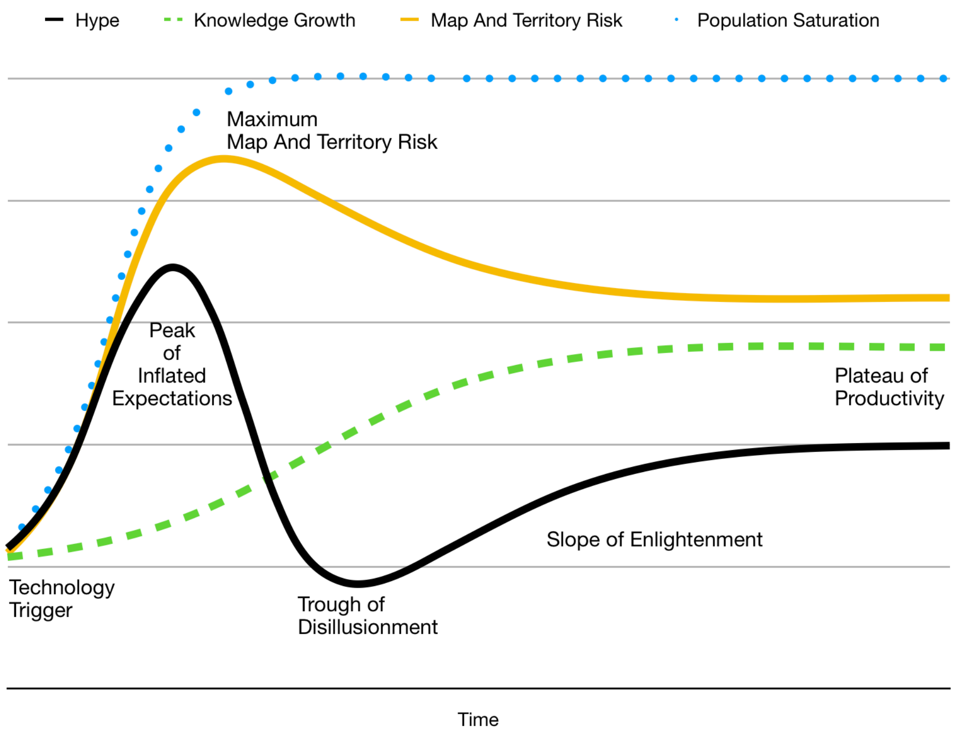


Figure 22.5: Hype Cycle, along with Map & Territory Risk

- The **amount known** about the idea by the audience (a Learning Curve, if you will, a dashed line).

Both of these are modelled with Cumulative Distribution[6] curves. From these two things, we can figure out where our maximum Map and Territory Risk lies: it's the point where awareness of an idea is farthest from the understanding of it. This acts as a "brake" on the **hype** around the idea, corresponding to the "Trough of Disillusionment".

Where the **saturation** and **knowledge** grow together, there is no spike in Map and Territory Risk and we don't see the corresponding "Trough of Disillusionment" at all, as shown in this chart:

22.4 Evolution

The chapter on Communication Risk introduced the following model for ideas:

[6] https://en.wikipedia.org/wiki/Cumulative_distribution_function#Use_in_statistical_analysis

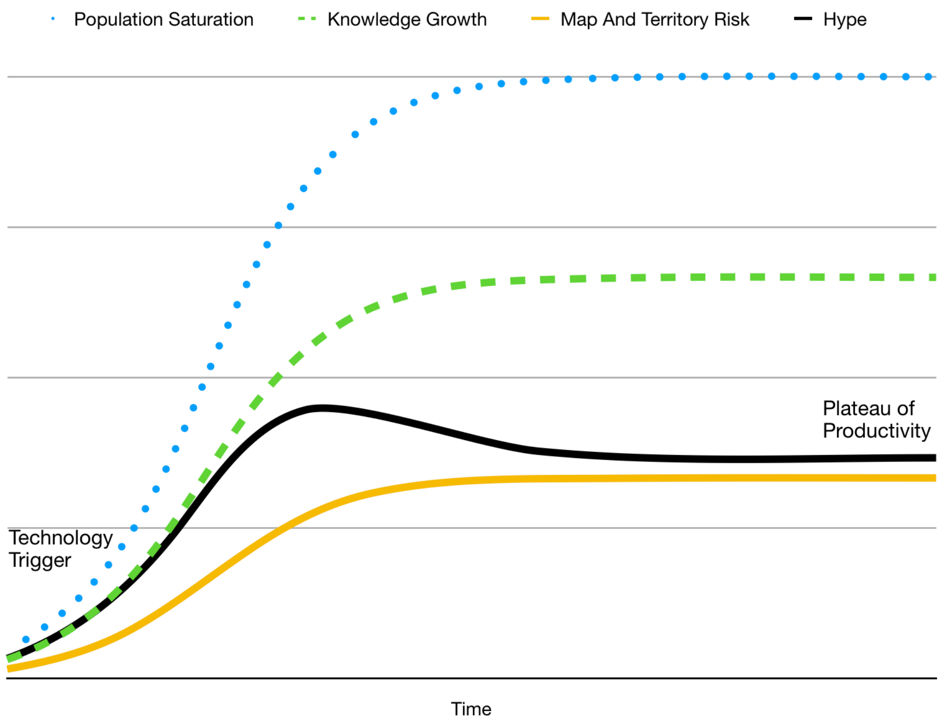


Figure 22.6: Hype Cycle 2: Slower growth of Map and Territory Risk means no "Trough of Disillusionment"

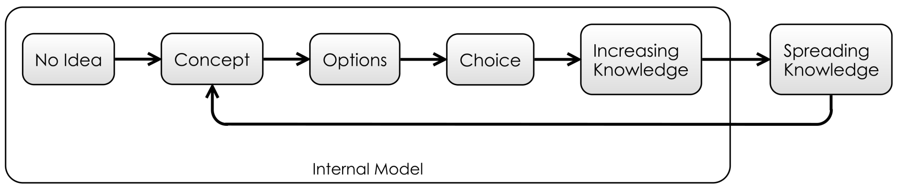


Figure 22.7: Spread of information between Internal Models

But what happens next? As we saw in Boundary Risk, the **Peter Principle** applies, people will use dependencies up to the point when they start breaking down.

Example: Metrics

Let's dive into a specific example now: someone finds a useful new metric that helps in evaluating performance.

It might be:

- **SLOC (Source Lines Of Code)**: i.e. the number of lines of code each developer writes per day/week whatever.
- **Function Points**: the number of function points a person on the team completes, each sprint.
- **Code Coverage**: the number of lines of code exercised by unit tests.
- **Response Time**: the time it takes to respond to an emergency call, say, or to go from a feature request to production.
- **Release cadence**: number of releases a team performs, per month, say.

With some skill, they may be able to *correlate* this metric against some other more abstract measure of success. For example:

> quality is correlated with more releases

> user-satisfaction is correlated with SLOC

> revenue is correlated with response time

Because the *thing on the right* is easier to measure than *the thing on the left*, it becomes used as a proxy (or, Map) for the thing they are really interested in (the Territory). At this point, it's *easy* to communicate this idea with the rest of the team, and *the market value of the idea is high*: it is a useful representation of reality, which is shown to be accurate at a particular point in time.

But *correlation* doesn't imply *causation*. The *cause* might be different:

- quality and number of releases might both be down to the simplicity of the product.
- user satisfaction and SLOC might both be down to the calibre of the developers.

- response time and revenue might both be down to clever team planning.

Metrics are seductive because they simplify reality and are easily communicated. But they *inherently* contain Map and Territory Risk: By relying *only* on the metrics, you're not really *seeing* the reality.

The devil is in the detail.

Reality Evolves

In the case of metrics, this is where they start being used for more than just indicators, but as measures of performance or targets:

- If a team is *told* to do lots of releases, they will perform lots of releases *at the expense of something else*.
- If team members are promoted according to SLOC, they will make sure their code takes up as many lines as possible.
- In the UK, ambulances were asked to wait before admitting patients to Emergency wards, in order that hospitals could meet their targets[7].

Some of this seems obvious: *Of course* SLOC is a terrible measure performance! We're not that stupid anymore. The problem is, it's not so much the *choice* of metric, but the fact that *all* metrics merely approximate reality with a few numbers. The map is *always* simpler than the territory, therefore there can be no perfect metrics.

In the same way that markets evolve to demand more features, our behaviour evolves to incorporate new ideas. The more popular an idea is, the more people will modify their behaviour as a result of it, and the more the world will change. Will the idea still be useful as the world adapts? Although the Hype Cycle model doesn't cover it, ideas and products all eventually have their day and decline in usefulness.

Bad Ideas

There are plenty of ideas which *seem a really good idea at the time* but then end up being terrible. It's only as we *learn about the products* and realize the hidden Map and Territory Risk that we stop using them. While SLOC is a minor offender, CFCs[8] or Leaded Petrol[9] are more significant examples.

[7] https://en.wikipedia.org/wiki/NHS_targets

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

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

The following Hype Cycle chart shows an initially promising idea that turns out to be terrible, and there is a "Period of Inoculation" where the population realise their mistake. There is "negative hype" as they work to phase out the offending idea:

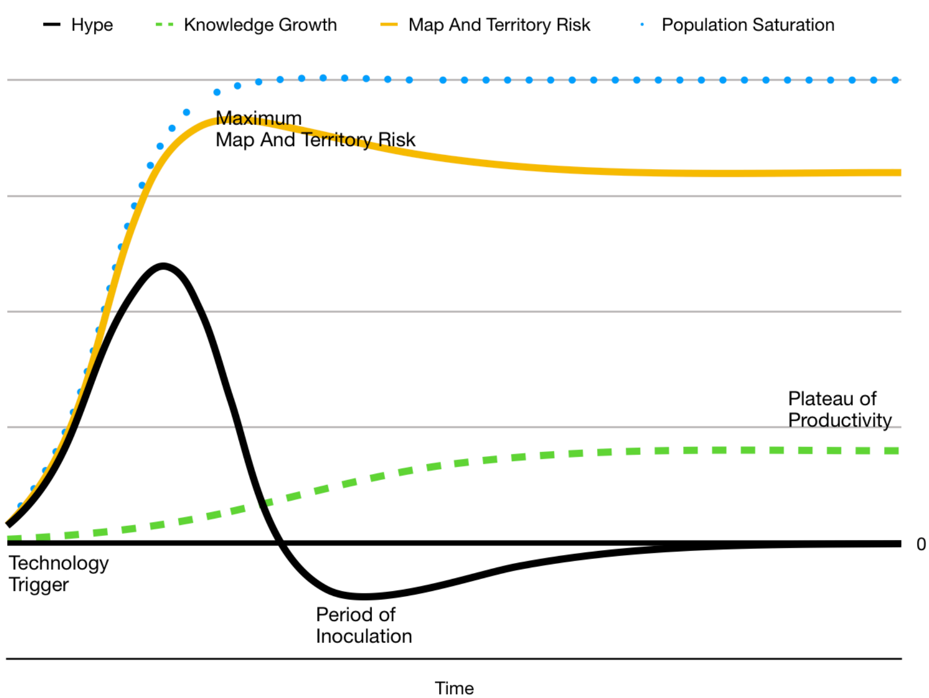


Figure 22.8: Hype Cycle For Something that turns out to be a **bad** *idea*

22.5 Humans and Machines

In the example of the SatNav, we saw how the *quality* of Map and Territory Risk is different for *people* and *machines*. Whereas people *should* be expected show skepticism to new (unlikely) information, our databases accept it unquestioningly. *Forgetting* is an everyday, usually benign part of our human Internal Model, but for software systems it is a production crisis involving 3am calls and backups.

For Humans, Map and Territory Risk is exacerbated by cognitive biases[10]:

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

> "Cognitive biases are systematic patterns of deviation from norm or rationality in judgement, and are often studied in psychology and behavioural economics."
>
> —Cognitive Bias, *Wikipedia*[11]

There are *lots* of cognitive biases. But let's just look at a couple that are relevant to Map and Territory Risk:

- **Availability Heuristic**: People overestimate the importance of knowledge they have been exposed to.
- **The Ostrich Effect**: Which is where dangerous information is ignored or avoided because of the emotions it will evoke.
- **Bandwagon Effect**: People like to believe things that other people believe. Could this be a factor in the existence of the Hype Cycle?

22.6 Hierarchical Organisations

Map And Territory Risk "trickles down" through an organisation. The higher levels have an out-sized ability to pervert the incentives at lower levels because once an organisation begins to pursue a "bullshit objective", the whole company can align to this.

The Huffington Post[12] paints a brilliant picture of how Volkswagen managed to get caught faking their emissions tests. As they point out:

> "The leadership culture of VW probably amplified the problem by disconnecting itself from the values and trajectory of society, by entrenching in what another executive in the auto industry once called a "bullshit-castle"... No engineer wakes up in the morning and thinks: OK, today I want to build devices that deceive our customers and destroy our planet. Yet it happened. Why? Because of hubris at the top."
>
> —Otto Scharmer, *Huffington Post*[13].

This article identifies the following process:

- **De-sensing**: VW Executives ignored *The Territory* society around them (such as the green movement), ensuring their maps were out of date.

[11] https://en.wikipedia.org/wiki/List_of_cognitive_biases

[12] https://www.huffingtonpost.com/otto-scharmer/the-fish-rots-from-the-he_b_8208652.html

[13] https://www.huffingtonpost.com/otto-scharmer/the-fish-rots-from-the-he_b_8208652.html

The top-down culture made it hard for reality to propagate back up the hierarchy.
- **Hubris/Absencing**: They pursued their own metrics of *volume* and *cost*, rather than seeking out others (a la the Availability Heuristic Bias). That is, focusing on their own *Map*, which is *easier* than checking the *Territory*. (See Hubris in the Agency Risk chapter).
- **Deception**: Backed into a corner, engineers had no choice but to find "creative" ways to meet the metrics.
- **Destruction**: Eventually, the truth comes out, to the detriment of the company, the environment and the shareholders. As the article's title summarizes "A fish rots from the head down".

Personal Example

A similar (but less catastrophic) personal story from a bank I worked at, where the objectives end up being mis-aligned *within the company*:

1. My team had been tasked with building automated "smoke tests" of an application. But this was bullshit: We only needed to build these *at all* because the application was so complex. The reason it was so complex was. . .
2. The application was being designed within a "Framework" constructed by the department. However, the framework was only being used by this one application. Building a "reusable" framework which is only used by a single application is bullshit. But, we had to do this because. . .
3. The organisational structure was created along a "matrix", with "business function" on one axis and "functional area" on another. Although we were only building the application for a single business function, it was expected to cater with all the requirements from the an entire "functional area". This was bullshit too, because. . .
4. The matrix structure was largely the legacy of a recent merger with another department. As Conway's Law[14] predicts, our software therefore had to reflect this structure. But this was bullshit because. . .
5. The matrix structure didn't represent reality in any useful way. It was designed to pacify the budget committee at the higher level, and try to demonstrate attributes such as *control* and *governance*. But this was bullshit too, because. . .
6. The budget that was given to our department was really based on how much fear the budget holders currently had of the market regulators. But this was bullshit too, because. . .

[14] https://en.wikipedia.org/wiki/Conway%27s_law

7. At a higher level, the executives had realised that our division wasn't one of the banks strategic strengths, and was working to close it all down anyway.

When faced with so many misaligned objectives, it seemed completely hopeless to concentrate on the task at hand. But then, a colleague was able to nihilistically add one final layer to this onion by saying:

8. "It's all about chasing money, which is bullshit, because life is bullshit."

Picking Fights

It feels like there's no way back from that.

All of life might well be a big Map and Territory illusion. But let's analyse just a bit:

- At each layer, the objectives changed. But, they impacted on the objectives of the layer below.
- Therefore, it seems like the more layers you have, the less likely it is that your objectives become inconsistent between the lower and higher levels.
- On a new project, it seems like a good idea to model this stuff: does the objective of the work you're about to undertake "align" with the objectives at a higher level?

Trying to spot Map and Territory Risk ahead-of-time in this manner seems like a useful way of trying to avoid Vanity Projects, and, if you are good at it, allows you to see which Goals in the organisation are fragile and likely to change. However, usually, if you are working in a team, you have limited agency to decide which projects you feel are valuable.

This comes down to a personal decision: do you want to spend time working on projects that you know are going in the bin? Some developers have the attitude that, so long as they get paid, it doesn't matter. But others are in it for the satisfaction of the work itself, so this ends up being a personal call.

(This theme will be developed further in Staging and Classifying.)

22.7 Markets

So far, we've considered what happens to individuals, teams and organisations when told to optimise around a particular objective. In Coordination Risk we looked at how communication was critical for coordination to happen. And, as we've already discussed, Abstraction is a key part of communication.

The languages we adopt or create are *sets of useful abstractions* that allow us to communicate. But what happens when this goes wrong?

Inadequate Equilibria[15] by Eleizer Yudkovsky, looks at how perverse incentive mechanisms break not just departments, but entire societal systems. He highlights one example involving *academics* and *grantmakers* in academia:

- It's not very apparent which scientists are better than which other scientists.
- One proxy is what they've published (scientific papers) and where they've published (journals).
- Universities want to attract research grants, and the best way to do this is to have the best scientists.
- Because "best" isn't measurable, they use the proxy.
- Therefore, immense power rests in the hands of the journals, since they can control the money-proxy.
- Therefore, journals are able to charge large amounts of money to universities for subscriptions.

> "Now consider the system of scientific journals... Some journals are prestigious. So university hiring committees pay the most attention to publications in that journal. So people with the best, most interesting-looking publications try to send them to that journal. So if a university hiring committee paid an equal amount of attention to publications in lower-prestige journals, they'd end up granting tenure to less prestigious people. Thus, the whole system is a stable equilibrium that nobody can unilaterally defy except at cost to themselves."
>
> —Inadequate Equilibria, *Eleizer Yudkovsky*[16]

As the book points out, while everyone *persists* in using an inadequate abstraction, the system is broken. However, Coordination would be required

[15] https://equilibriabook.com
[16] https://equilibriabook.com/molochs-toolbox/

for everyone to *stop* doing it this way, which is hard work. (At least within a hierarchy, Maps can get fixed.)

This is a *small example* from a much larger, closely argued book, and it's worth taking the time to read a couple of the chapters on this interesting topic.

As usual, this chapter forms a grab-bag of examples in a complex topic. But it's time to move on as there is one last stop we have to make on the Risk Landscape, and that is to look at Operational Risk.

(NB: The Hype Cycle model is available in **Numbers** form here[17].)

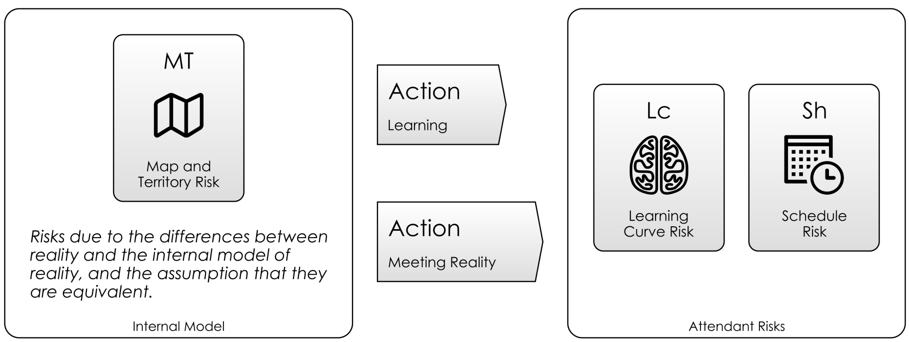


Figure 22.9: Map And Territory Risk

(talk about how operational risk is an extension of this). tbd

[17] `https://github.com/risk-first/website/blob/master/RiskMatrix.numbers`

CHAPTER 23

Operational Risk

In this chapter on Operational Risks, we're going to start considering the realities of running software systems in the real world.

Here, we're going to set the scene by looking at what constitutes an Operational Risk, and then look at the related disciplines of Operations Management and [Operational Risk Management]. Following this background, we'll apply the Risk-First model and dive into the various mitigations for Operational Risk.

23.1 Operational Risks

It's tempting to take a very narrow view of the dependencies of a system, but Operational Risks are often caused by dependencies we don't consider - the *context* within which the system is operating. Here are some examples:

- Staff Dependencies (Staff Risk):
  - Freak weather conditions affecting ability of staff to get to work, interrupting the development and support teams.
  - Reputational damage caused when staff are rude to the customers.
- Infrastructure Dependencies (Reliability Risk):
  - A data-centre going off-line, causing your customers to lose access.
  - A power cut causing backups to fail.
  - Not having enough desks for everyone to sit at.
- Process Dependencies (Process Risk):
  - Regulatory change, which means you have to adapt your business model.

  - Insufficient controls which means you don't notice when some transactions are failing, leaving you out-of-pocket.
  - Data loss because of bugs introduced during an untested release.

- Software Dependencies (Software Dependency Risk):
  - Hackers breaking into the system and bringing your service down.

- Agency Dependencies (Agency Risk):
  - Suppliers deciding to stop supplying you with something you need.
  - Workers going on strike.
  - Employees trying to steal from the company (bad actors).

.. basically, a long laundry-list of everything that can go wrong due to operating in "The Real World".

So, Operational Risk Management[1] is the purview of dealing with all the types of issues listed above:

> "Operational Risk Management is the oversight of Operational Risk, including the risk of loss resulting from inadequate or failed internal processes and systems; human factors; or external events."
>
> —Operational Risk Management, *Wikipedia*[2]

23.2 Operations Management

If we are designing a software system to "live" in the real world, we have to be mindful of the environment we're working in, and adapt our software and processes accordingly. This view of the "wider" system is the discipline of Operations Management. The below diagram (from "Operations Management" by *Slack et al.*[3]) breaks down some of the concerns of this discipline.

In this diagram, a **Transform Process** (the **Operation** itself) is embedded in the **Environment**, which supplies it with three key dependencies:

- The **Resources** it needs, whether *transformed* resources (like electricity or information, say).

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

[2] https://en.wikipedia.org/wiki/Operational_risk_management

[3] http://amzn.eu/d/b6ZjuMu

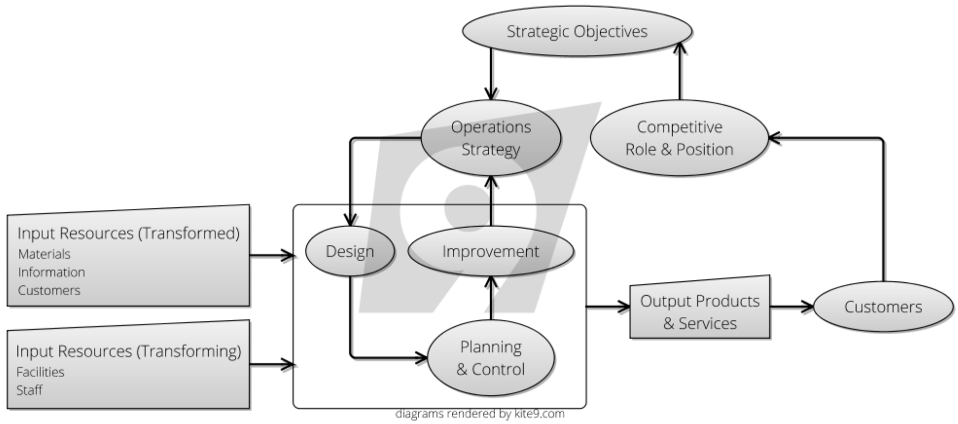


Figure 23.1: A General Model of Operations Management

- It's **Customers**, which supply it with money in return for goods and services, and
- An **Operational Strategy** to follow.

We have looked at processes like the **Transform Process** in the chapter on Process Risk. The healthy functioning of this process is the domain of Operations Management, and in the diagram this involves the following tasks:

- **Control**: Ensuring that the Operation is working according to its design. This includes quality control.
- **Improvement**: Improving the operation in response to changes in the **Environment** and the **Operational Strategy**, detecting failure and recovering from it.
- **Planning**: This covers aspects such as capacity planning, forecasting and project planning.
- **Design**: Ensuring that the design of the product and the transform process itself fulfils an **Operational Strategy**.

23.3 Mitigating Operational Risk

We've spent a lot of time looking at the varieties of Dependency Risk on a software project. But in the "real world" of Operational Risk we have to consider that these dependencies will fail in any number of unusual ways, and we can't be ready for all of them.

For this reason, the toolbox of mitigations for Operatational Risk is somewhat different to that for regular dependencies. Here we're going to focus on four *basic strategies*, and show how they align with the activities described above.

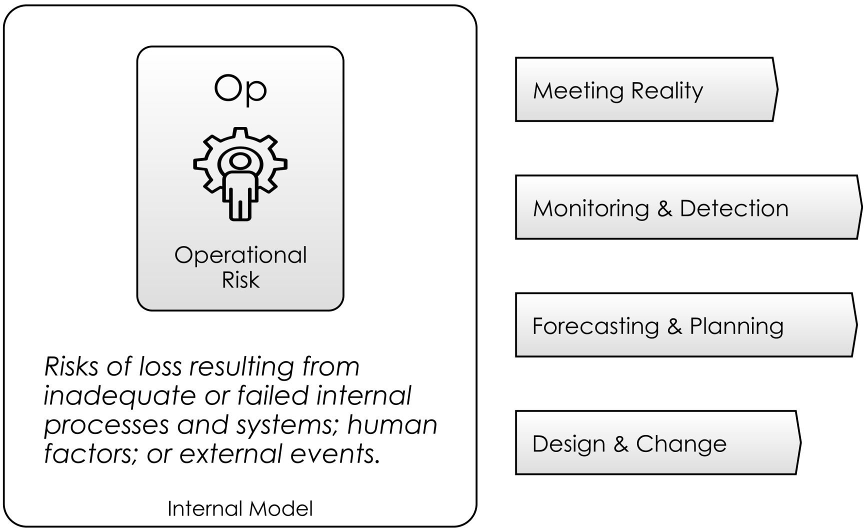


Figure 23.2: Diagram of Four Strategies to Mitigate Operational Risk

As shown in this diagram, these are Meeting Reality, Monitoring & Detection, Design & Change and Forecasting & Planning.

23.4 Meeting Reality

Once exposed to the real world, no system is perfect. This means we must design-in ways in which the systems we build can improve and change. Since we don't have a perfect understanding of the world, most of the Operational Risk we face is Hidden Risks.

Reputational Risk

Our production systems are Meeting Reality all the time, and in order to mitigate Operational Risk we need to take the most advantage of this as possible. However, conversely, Operational Risk includes **Reputational Risk**, which gives us pause: we don't want to destroy good will created for our organisation, this is very hard to rebuild.

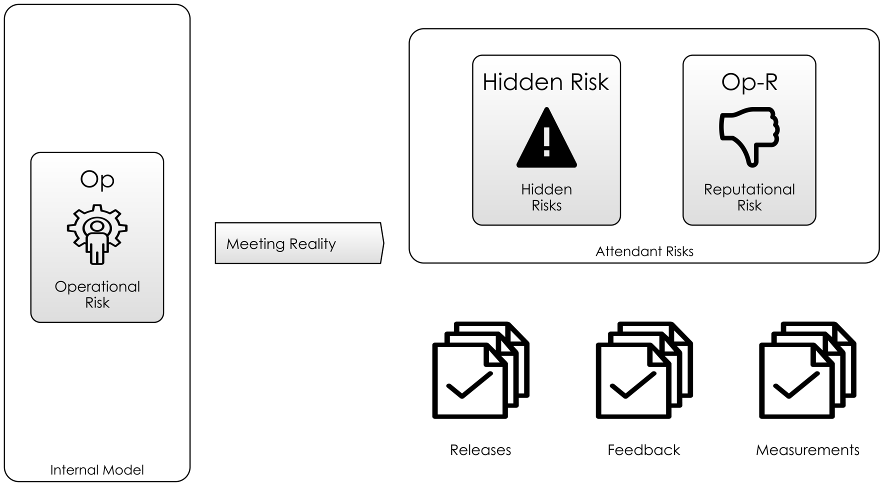


Figure 23.3: Taking action against Operational Risk by Meeting Reality

So there is a tension between "you only get one chance to make a first impression" and "gilding the lilly" (perfectionism). In the past I've seen this stated as:

> Pressure to ship vs pressure to improve

A Risk-First re-framing of this might be the balance between:

- The perceived Reputational Risk, Feature Risk and Operational Risk of going to production (pressure to improve).
- The perceived Scarcity Risks (such as funding, time available, etc) of staying in development (pressure to ship).

The "should we ship?" decision is therefore a complex one. In Meeting Reality, we discussed that it's better to do this "sooner, more frequently, in smaller chunks and with feedback". We can meet Operational Risk *on our own terms* by doing so:

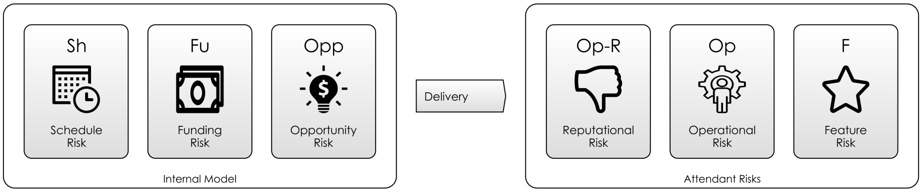


Figure 23.4: Balance of Risks from Delivering Software

Meet Reality...	Techniques
Sooner	Quality Control Processes, Limited Early-Access Programs, Beta Programs, Soft Launches, Business Continuity Testing
More Frequently	Continuous Delivery, Sprints
In Smaller Chunks	Modular Releases, Microservices, Feature Toggles, Trial Populations
With Feedback	User Communities, Support Groups, Monitoring, Logging, Analytics

23.5 Monitoring and Detection

Monitoring & Detection is the second strategy for mitigating Operational Risk, and in terms of practices, broadly corresponds to Slack's *Improvement* category, above.

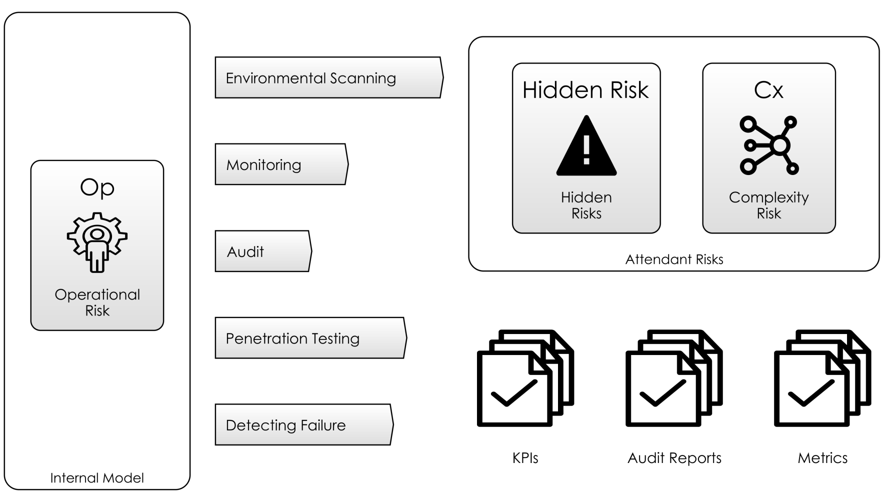


Figure 23.5: Monitoring And Detection

Since Humans and machines have different areas of expertise, and because Operational Risks are often novel, it's often not optimal to try and automate everything. A good operation will consist of a mix of human and machine

Humans Are...	Machines Are...
Good at adaptation	Good at consistency
Expensive at scale	Cheap at scale
Reacting and Anticipating	Recording

The aim is to build a human-machine operational system that is Homeostatic[4]. This is the property of living things to try and maintain an equilibrium (for example, body temperature or blood glucose levels), but also applies to organisations at any scale. The key is to build systems with feedback loops, even though this leads to more complex systems overall.

Performance Risk

As we saw in Map and Territory Risk, it's very easy to fool yourself, especially around Key Performance Indicators (KPIs) and metrics. Good Operations Management is about going beyond this and looking for trouble. Large organisations have Audit[5] functions precisely to guard against their own internal failing Processes and Agency Risk. Audits could be around software tools, processes, practices, quality and so on. Practices such as Continuous Improvement[6] and Total Quality Management[7] also figure here.

The Operational Context

- *Environmental Scanning* is all about trying to determine which changes in the environment are going to impact your operation. Here, we are trying to determine the level of Dependency Risk we face for external dependencies, such as *suppliers*, *customers* and *markets*. Tools like PEST[8] are relevant here, as is
- Penetration Testing[9] is looking for security weaknesses within the operation. See OWASP[10] for examples.
- Vulnerability Management[11] is keeping up-to-date with vulnerabilities in Software Depenendencies.

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

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

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

[7] https://en.wikipedia.org/wiki/Total_quality_management

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

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

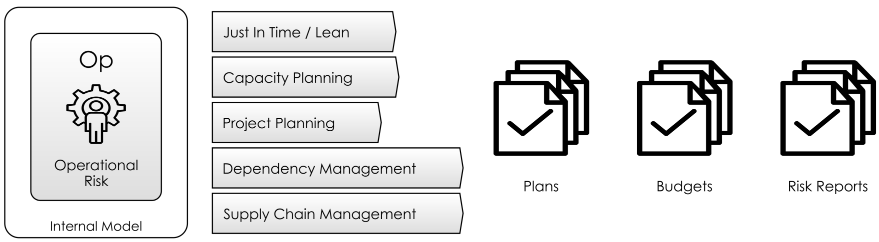
[10] https://en.wikipedia.org/wiki/OWASP

[11] https://en.wikipedia.org/wiki/Vulnerability_management

23.6 Forecasting and Planning

Dependencies are not just things we *use*: For a system to run well, it needs to carefully manage unreliable dependencies, and ensure their safety and availability. In the example of the humans, say, it's the difference between Hunter-Gathering[12] (picking up food where we find it) and Agriculture[13].

Forecasting and Planning then is a strategy we can bring to bear on dependency management, and this usually falls to the more human end of the operation.



.

23.7 Design and Change

You might think that for an IT operation, tasks like Planning and Design belong within the Development function within an organisation. But there is (and always has been) significant overlap because it's important that we design software that allows it to be managed effectively. In recent years, the "DevOps" movement has brought this relationship into sharper focus.

Since our operation exists in a world of Red Queen Risk and Feature Drift, we would expect that the output of our Forecasting and Planning activities would result in changes to our operation.

In a way, we are now back to where we started from, identifying Dependency Risk, Feature Risk and Complexity Risk that hinders our operation, and mitigating it through tasks like *software development*. Our safari of risk is finally complete, it's time to look back and what we've seen in Staging and Classifying.

[12] `https://en.wikipedia.org/wiki/Hunter-gatherer`

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

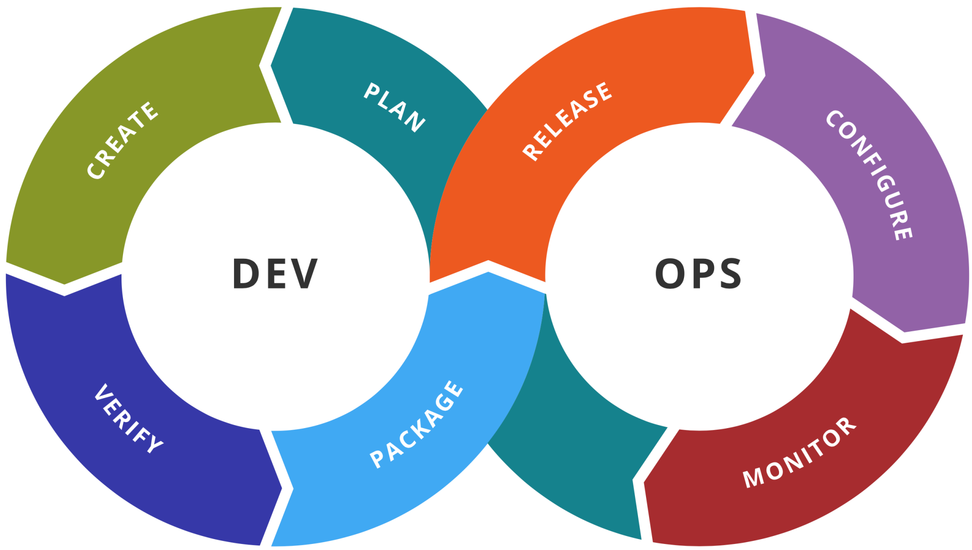


Figure 23.6: DevOps Activities: Development and Operations activities overlap one-another (Credit: Kharnagy, Wikipedia)

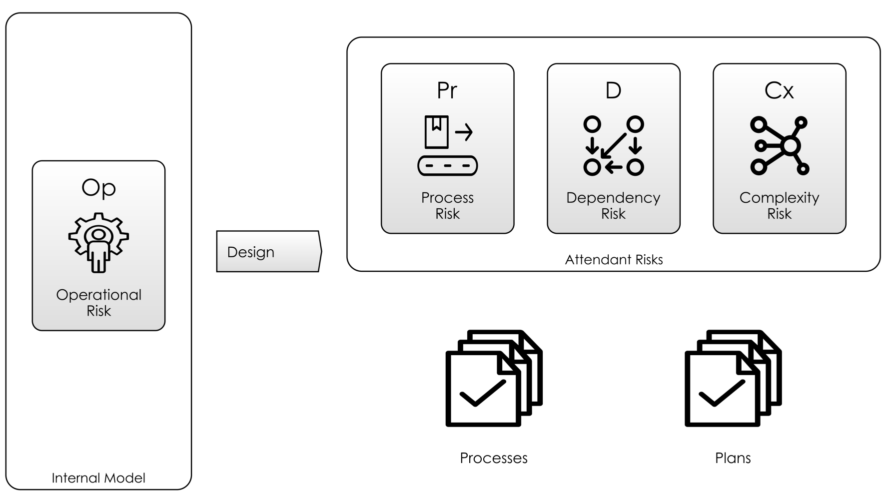


Figure 23.7: Design and Change Activities

CHAPTER 24

Staging And Classifying

Our tour is complete.

We've collected on this journey around the Risk Landscape a (hopefully) good, representative sample of Risks and where to find them. But if we are good collectors, then before we're done we should Stage[1] our specimens and do some work in classifying what we've seen.

If you've been reading closely, you'll notice that a number of themes come up again and again within the different chapters. For example, concepts like **Fit**, **Abstraction**, **Evolution**. Although we've been looking at patterns of risk across software projects, it's time to look at the *patterns within the patterns*.

24.1 The Power Of Abstractions

Abstraction appears as a concept continually throughout the book, whether we are looking at Communication, Complexity Metrics, Map and Territory Risk or how it causes Boundary Risk. And, so far, we've looked at some complicated examples of abstractions, such as network protocols, dependencies on technology or Business Processes.

There's a good reason for this repetition. Abstraction is at the heart of *everything we do within software*. So, let's now *generalize* what is happening with abstraction, but have in mind a really simple example: having a name for something. So, at the simplest end, you might be simply *naming a pattern* of behaviour we see in the real world, such as "Binge Watching" or "Remote Working", or naming a category of insects as "Beetles".

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

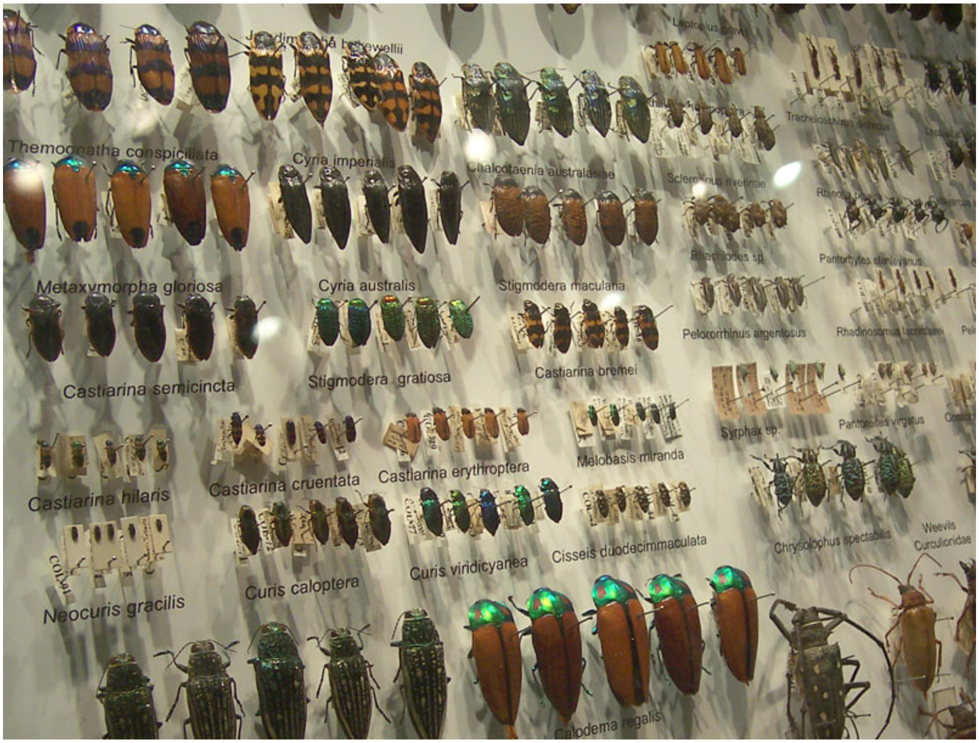


Figure 24.1: Staged and Classified Beetle Collection, (Credit: Fir0002, Wikipedia)

Using An Existing Abstraction means:

- **Mitigating Feature Risk:** Because the abstraction is providing you with something *useful*.
- **Living with Dependency Risk:** We depend on a word in our language, or a function in our library, or a service on the Internet. But all of these things are *unreliable*. The word might not communicate what you want it to, or be understood by the audience, the function might not work, the service might be down. Also, language *changes* and *evolves*, and the words you are using now might not always mean what you want them to mean. Software too changes and evolves. We've seen this in Red Queen Risk and Feature Drift Risk.
- **Accepting Communication Risk.** : Because if you are using the abstraction in conversation, the people you are using it with *need to understand it too*.
- **Accepting Map and Territory Risk**: Because the abstraction is a simplification, and not the actual thing itself.

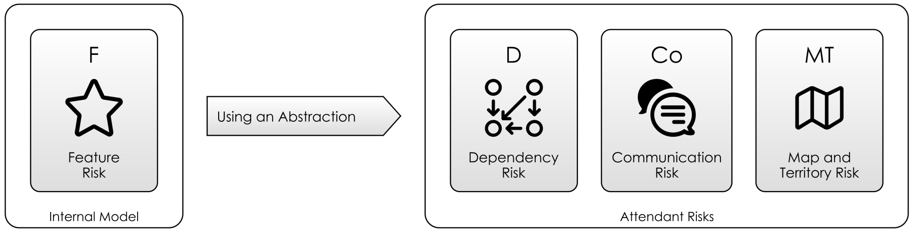


Figure 24.2: Depending on an Abstraction

Inventing A New Abstraction means:

- **Mitigating Feature Risk.** These abstractions are Features in the sense that other people can choose to use them, if they fit their requirements.
- **Creating a Protocol.** At the very simplest level (again), this is just introducing *new words to a language*. Therefore, we create Protocol Risk: what if the person we are communicating with *doesn't* know this word?
- **Increasing Complexity Risk.** Because, the more words we have, the more complex the language is.
- **Creating Boundary Risk.** By naming something, you *implicitly* create a boundary, because the world is now divided into "things which *are* X" and "things which *are not* X". Sometimes, this abstraction may literally end up having a physical boundary to enforce this division (such as, "My Property / Not My Property"). *Boundary Risk is created by abstractions.*

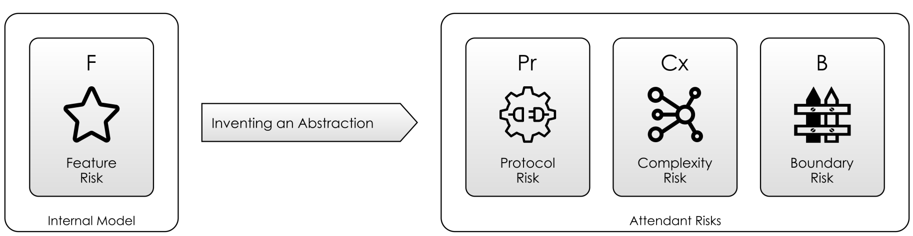


Figure 24.3: Inventing an Abstraction

Choosing Between Abstractions means:

- **Overcoming a Learning Curve**: Because you have to *learn* a name in order to use it (whether a function, a dog, or the name of someone at a

party).
- **Accepting Boundary Risks.** Just using *a single word* means accepting the whole *ecosystem* of the language the word is in. Using *French words* means the Boundary Risk of the French Language.
- **Accepting Map And Territory Risk.** Because the word refers to the *concept* of the thing, and *not the thing itself*.

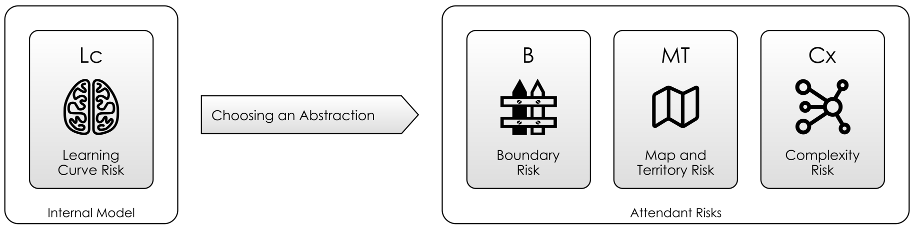


Figure 24.4: Choosing an Abstraction

24.2 Your Feature Risk is Someone Else's Dependency Risk

In the Feature Risk chapter, we looked at the problems of *supplying a dependency to someone else*: you've got to satisfy a demand Market Risk, and service a segment of the user community Feature Access Risk. The chapter on Operational Risk went further, looking at specific aspects of being the supplier of an IT service as a *dependency*.

However, over the rest of the Dependency Risk chapters, we looked at this from the point of view of *being a client to someone else*: you want to find trustworthy, reliable dependencies that don't give up when you least want them to.

So Feature Risk and Dependency Risk are *two sides of the same coin*. You face Dependency Risk when you're a client, Feature Risk when you're the supplier.

Further, to *use* a dependency requires the client and the supplier to communicate. And, this entails *learning protocols* and other types of Communication Risk. You have to learn to use it. Maybe it learns you. This requires changes to your internal model.

These relationships of features/dependencies are the basis of Supply Chains and the world-wide network of goods and services that forms the modern economy. The incredible complexity of this network *should* mean incredible

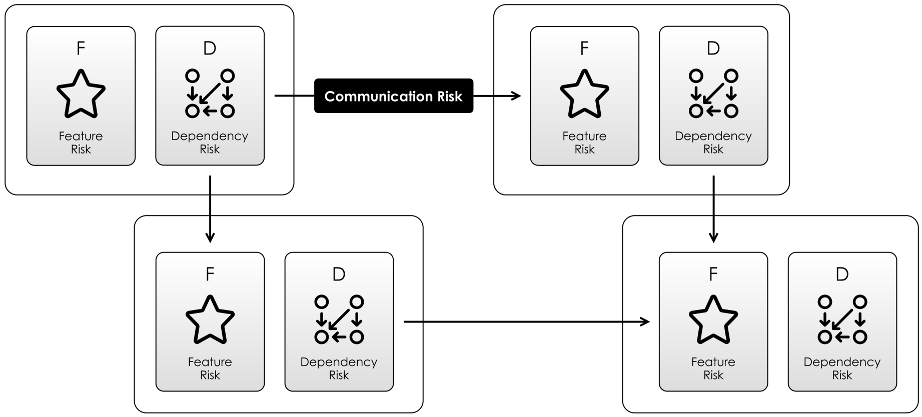


Figure 24.5: Features And Dependencies

Complexity Risk, too, but because humans are good at Coordinating and managing our dependencies, and we rely on market mechanisms to help us allocate dependencies efficiently.

24.3 The Original Risk

As we discussed in Dependency Risk, *depending on things* is necessary for life, whether it is oxygen, food or sunlight. Minimising Dependency-Risk is therefore the goal of all life. For some species, this also means dealing with Feature Risk: there's no point trying to supply a fruit that no creature will eat, for example.

Our problems really start when we try to Coordinate with the dependencies themselves or each other. As discussed in the chapter on Coordination, coordination is the root of Communication Risk, as without coordination, we don't have to care about what the world is trying to tell us, or what we are trying to tell the world.

- Communication Risk then begets Map and Territory Risk, because
- Communication Risk also means Complexity Risk, because now we have built a communication graph, and we saw how to calculate how complex that is.
- As discussed in Boundary Risk, this is at a confluence of Dependency Risk, Communication Risk and Complexity Risk.

The below diagram shows how this causality plays out.

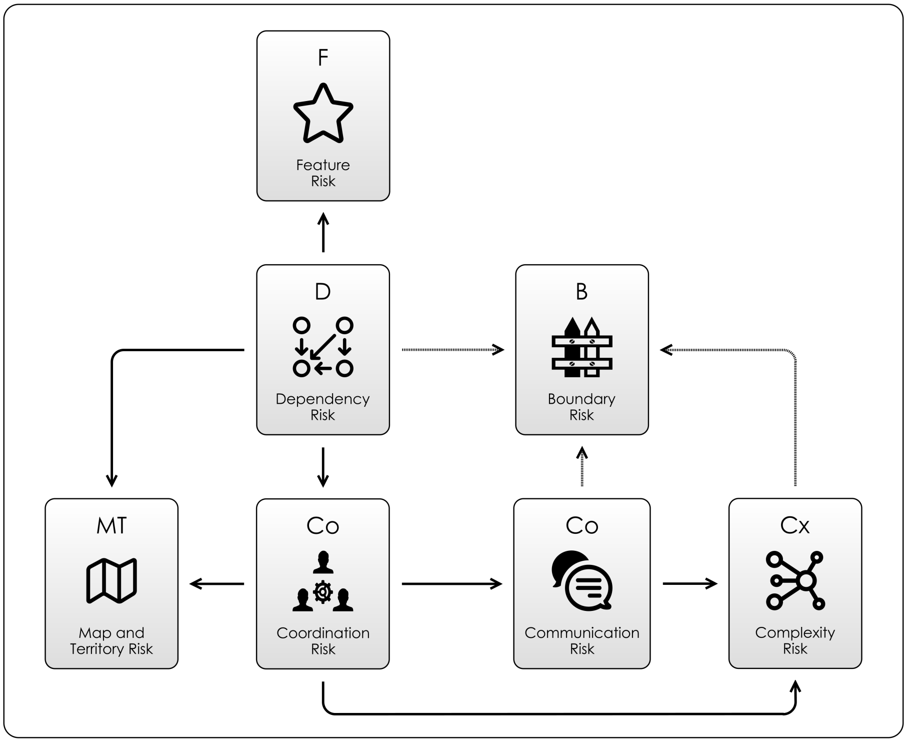


Figure 24.6: The Origin Of Risk

24.4 Towards A Periodic Table Of Risks

As we said at the start, Risk-First is all about developing *A Pattern Language*. We can use the terms like "Feature Risk"_ or "Learning Curve Risk" to *explain* phenomena we see on software projects. If we want to De-Risk our work, we need to be able to explain what the risks are, and what we expect to do about them.

The diagram below compiles all of the risks we've seen so far on the journey across the risk landscape. Just like a periodic table, there are perhaps others left to discover. Please help by reporting back what you see.

In the next chapter, we will be looking at some Stories Of Failure, and trying to apply the patterns we've learnt.

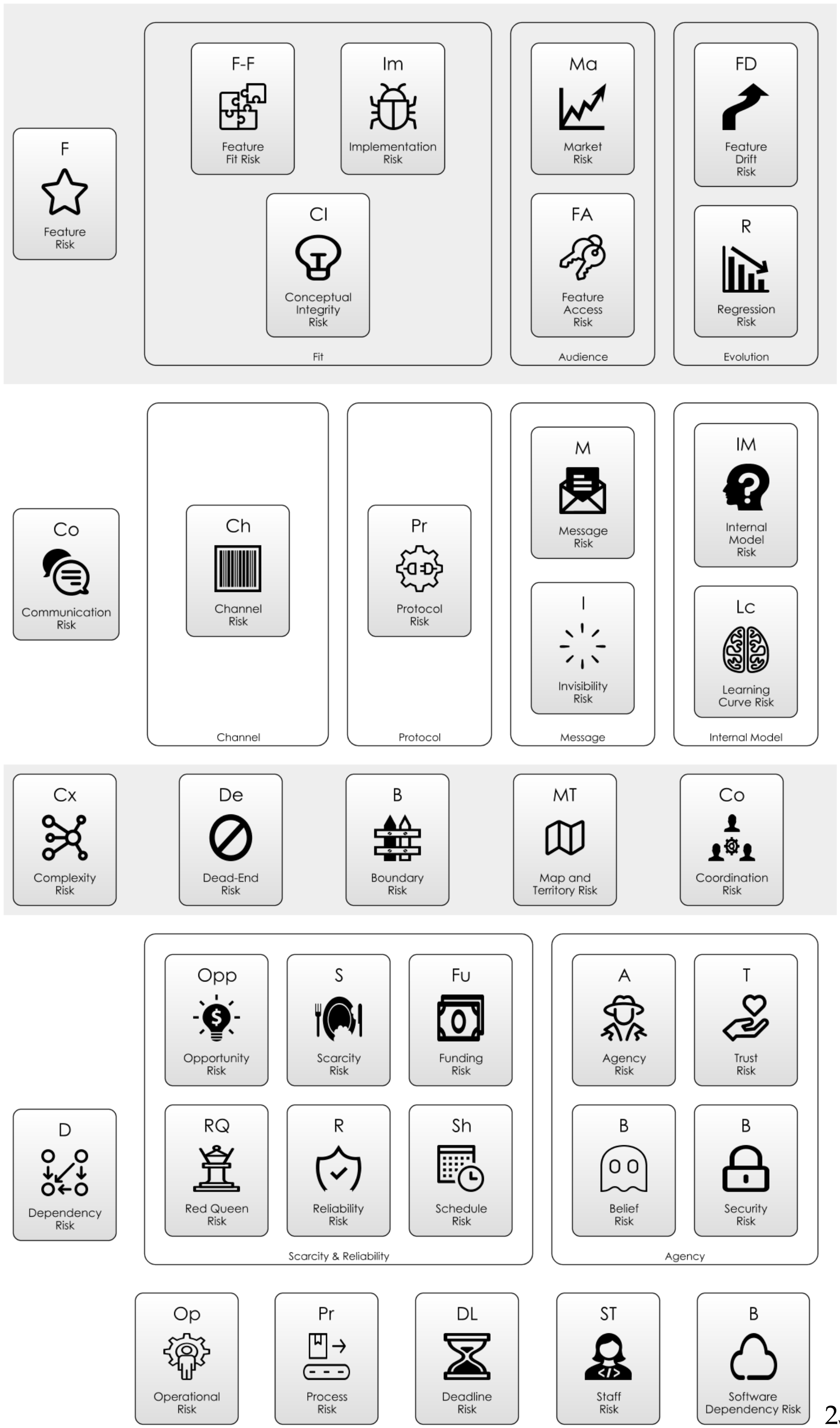


Figure 24.7: Periodic Table of Risks

CHAPTER 25

Stories Of Failure

2. Failure Modes

- Understanding Failure: what exactly does it mean to fail?
- Personal Failures
  - CapsLock: complexity, not using tools.
  - Configuration Tool (Complexity, feature fit, bugs in hibernate (dependency risk, then dead-end risk), difficulty mapping domain model)
  - Wide Learning (Funding, but also complexity), did we know what we were building? Agency risk
  - AreAye - needless complexity XMLBox
  - Agora: Notes / Typing. (Complexity Risk) Archipelago
  - PDC: website redesign. funding. i.e. schedule risk
  - Hawk: complexity risk in the software. but actually, they made it work. offshoring.
  - Dark: market/feature fit?
  - J10: marketing / market fit / Complexity in spades. algorithmic complexity
  - DSL: complexity (code generation). complexity = layers. team dynamics.
  - REF: complexity. agency risk. failure of goals. m&t.
  - REF Testing: complexity risk. communication risk?
  - HSC: Trader Comments: feature fit.
  - HSC: Takeover of Symph: Complexity (of change)
  - TT: Feature Fit
- Boehm.

https://wwwx.cs.unc.edu/~welch/class/comp145/media/docs/Boehm_Term_NE_Fail.pdf

https://www.worksoft.com/top-software-failures-of-2017-so-far

https://sites.hks.harvard.edu/m-rcbg/ethiopia/Publications/Top%2010%20Reasons%20Why%

JC Example 1: Compensation Workbench, rules to uplift payrolls annually based on certain definitions. e.g tied into performance gradiing, min wage. Was done off s/s, out of HR system. Analysis done on S/s. But wanted to do it in the HR database. It took a lot longer than it should. Underestimated the complexity. Excel could be infinitely complex, but if you standardize, you lose that. The customer hadn't got experience of big IT projects - scope creep. No real sense of prioritisation and focusing on what was iportant. PM was nice, but didn't understand delivery management. (i.e. managing risks and issues). Had short timescale, lead delivery resource underestimated. 2m exercise. Lasted 5m in the end, but not with full functionality.

Part III

Application

CHAPTER 26

Coming Next

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

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

3. What's To Come

- risk based debugging.
- risk based coding.

CHAPTER 27

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.

27.1 The Purpose Of Estimating

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

1. **To allow for the creation of *events*.** As we saw in Deadline Risk, if we can put a date on something, we can mitigate lots of Coordination Risk. Having a *release date* for a product allows whole teams of people to coordinate their activities in ways that hugely reduce the need for Communication. Much like "attack at dawn" allows disparate units of an army to avoid lots of the Coordination Risk inherent in "attack on my signal". This is a *good reason for estimating*, because by using events you are mitigating risk. This is often called a *hard deadline*.

2. **To allow for the estimation of the Pay-Off of an action.** This is a *bad reason for estimating*, as we will discuss in detail below. But briefly, the main issue is that Pay-Off isn't just about figuring out Schedule Risk - you should be looking at all the other Attendant Risks of the action too.

27.2 How Estimates Fail

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

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

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

> How long will it take to deliver X

Is the following:

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

... and so on.

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

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

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

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

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

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

Let's try and pull this apart:

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

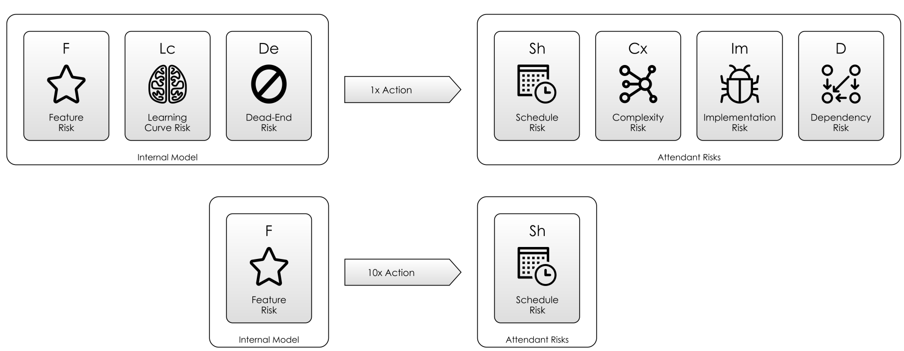


Figure 27.1: 1x Task vs 10X Task

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

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

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

How to Be a "10X Developer"

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

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

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

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

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

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

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

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

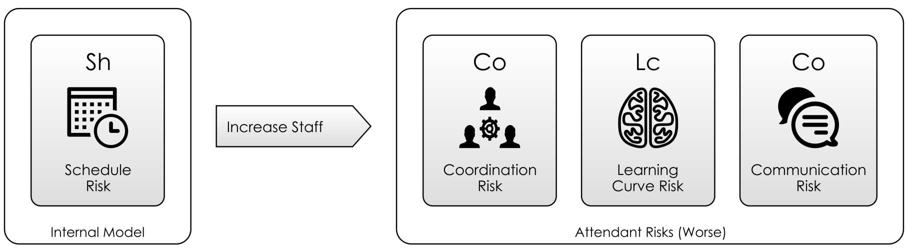


Figure 27.2: Brooks' Law, Risk-First Style

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

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

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

Old Saw #3: Parkinson's Law

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

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

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

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

A Case Study

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

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

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

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

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

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

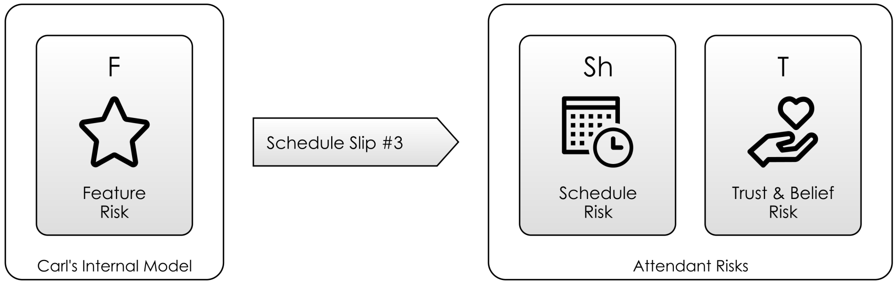


Figure 27.3: Carl's Schedule Slip increases Trust and Belief Risks

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

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

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

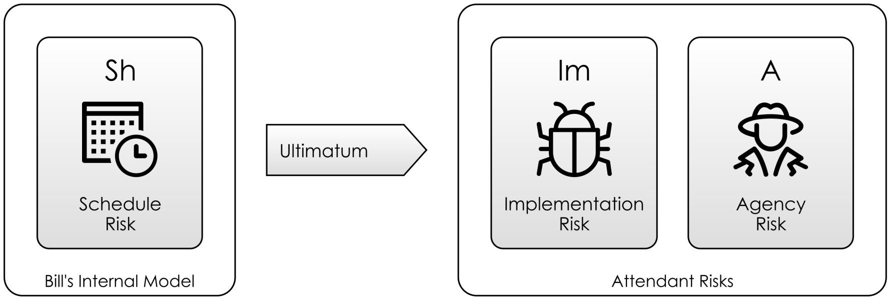


Figure 27.4: Bill's Ultimatum

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

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

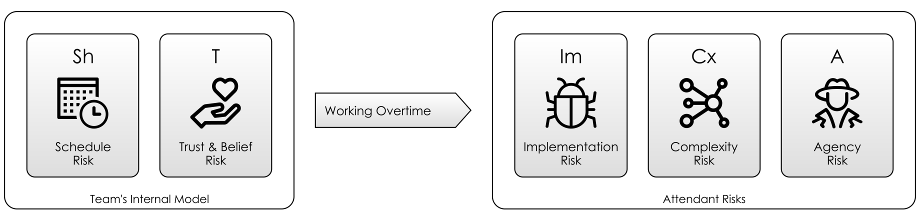


Figure 27.5: Team Response

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?

27.5 Agile Estimation

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

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

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

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

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

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

27.6 Risk-First Estimating

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

- The 10X developer saw, and the difference made by experience implies that a lot of the effort on a project comes from Learning Curve Risk and Dead End Risk.
- The lesson from "Quality, Speed, Cost" is that actually, we need to be considering *all* risks, not just some arbitrary milestones on a project plan. Project plans can always be gamed, and you can always leave

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

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

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

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

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

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

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

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

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

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

- "I think this task is problematic because it's going to massively increase our Dependency Risk to add a new database here."
- "I don't think we should have component A interacting with component B because it'll introduce extra Communication Risk which we will always be tripping over."
- "I worry we might not understand what the sales team want and are facing Implementation Risk. How about we try and get agreement on a specification?"

Figure 27.6: Journey via the Central Line

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.

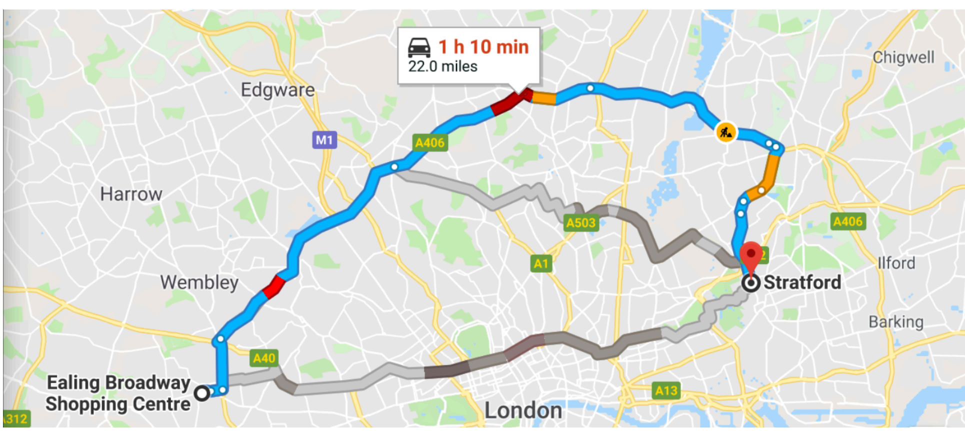


Figure 27.7: Journey by Car

In terms of risk, they all have different profiles. You're often delayed in the car, by some amount. The tube is *generally* reliable, but when it breaks down or is being repaired it might end up quicker to walk.

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

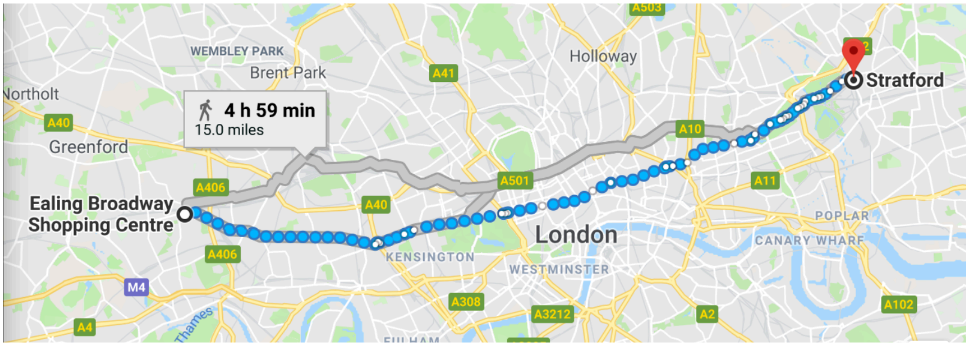


Figure 27.8: Journey on Foot

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

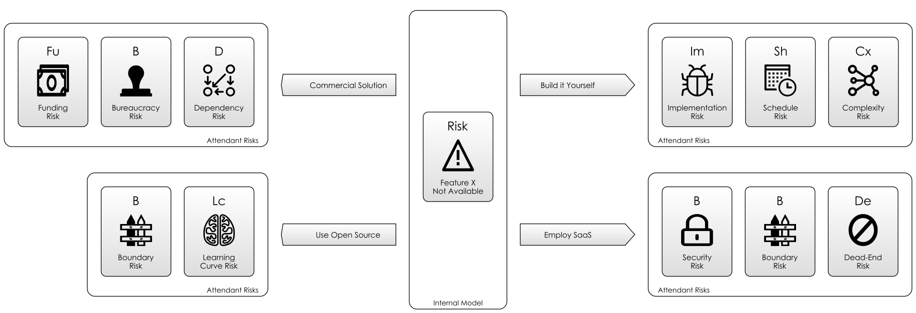


Figure 27.9: Possible Moves On The Risk Landscape

Because the modern Risk Landscape is so complex:

- There can be orders of magnitude difference in *time*, with very little difference in destination.
- If it's Schedule Risk you're worried about, *Code Yourself* isn't a great solution (for the whole thing, anyway). "Take the tube" and at least partly use something someone built already. There are probably multiple alternatives you can consider.
- If no one has built something similar already, then why is that? Have you formulated the problem properly?

- Going the wrong way is *so much easier*.
- Dead-Ends (like a broken Central Line) are much more likely to trip you up.
- You need to keep up with developments in your field. Read widely.

Tip #3: Meet Reality Early on the Biggest Risks

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

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

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

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

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

- *Having a vocabulary* (like the one Risk-First provides) allows us to *at least talk about these*. e.g. "I believe there is a Dead End Risk that we might not be able to get this software to run on Linux."
- Build mock-ups:
  - UI wireframes allow us to bottom out the Communication Risk of the interfaces we build.
  - Spike Solutions allow us to test algorithms and approaches before making them part of the main development.
- Don't pick delivery dates far in the future. Collectively work out the biggest risks with your clients, and then arrange the next possible date to demonstrate the mitigation.
- Do actions *early* that are *simple* but are nevertheless show-stoppers. They are as much a source of Hidden Risk as more obviously tricky actions.

Tip #4: Talk Frankly About All The Risks

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

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

> I think it will take about 9 months, but that's just a rough estimate at this point," Carl said. "That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. I need to show the board something by then or I'm worried they will lose confidence in me and this project

> OK," said Carl. "But I'm really concerned we have huge Feature Fit Risk. The task of understanding the requirements and doing the design is massive.

> Well, in my head it's actually pretty simple, " said Bill. "Maybe I don't have the full picture, or maybe your idea of what to build is more complex than I think it needs to be. That's a massive risk right there and I think we should try and mitigate it right now before things progress. Maybe I'll need to go back to the board if it's worse than I think.

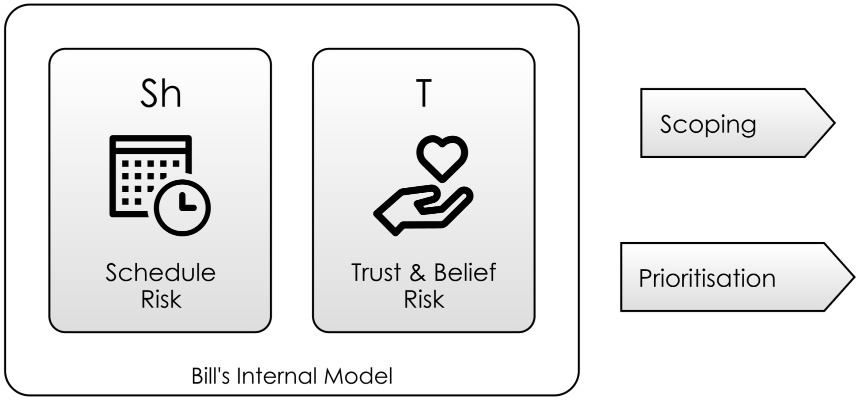


Figure 27.10: Identifying The Action

Tip #5: Picture Worrying Futures

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

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

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

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

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

This is the essence of De-Risking.