r1: Risk-First Software Development

Rob Moffat

This is part of the risk first series.

sdfds f sd sd f sdf

Published by.

On.

Stuff;

Dedicated to blan

Contents

This is a depressing book.

It's part of 2, but in this one you only get to meet the bad guy.

Preface

Welcome to Risk-First

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

One perspective is that they are individual software methodologies[1], offering different viewpoints on how to build software.

However, here, we are going to consider a second 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*.

Goal

Hopefully, after reading through some of the articles here, 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 software methodologies[2] 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 is *letting you down*, and the vocabulary to argue for when it's a good idea to deviate from it.

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**.)

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.

[1] https://en.wikipedia.org/wiki/Software_development_process#Methodologies
[2] https://en.wikipedia.org/wiki/Software_development_process#Methodologies

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.

Lastly, although this is a Wiki[3], it's not meant to be an open-ended discussion of software techniques like Ward's Wiki[4]. In order to be concise and useful, discussions need to be carried out by Opening an Issue[5].

if waterfall applies the principles from building, and lean applies the principles from manufacture, risk First applies the principles from finance.

[3]https://en.wikipedia.org/wiki/Wiki
[4]http://wiki.c2.com
[5]https://github.com/risk-first/website/issues

Part I

Introduction

Chapter 1

A Simple Scenario

Hi.

Welcome to the Risk-First Wiki.

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.

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

A Simple Scenario

Lets for a moment forget about software completely, and think about *any endeavor 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.

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.

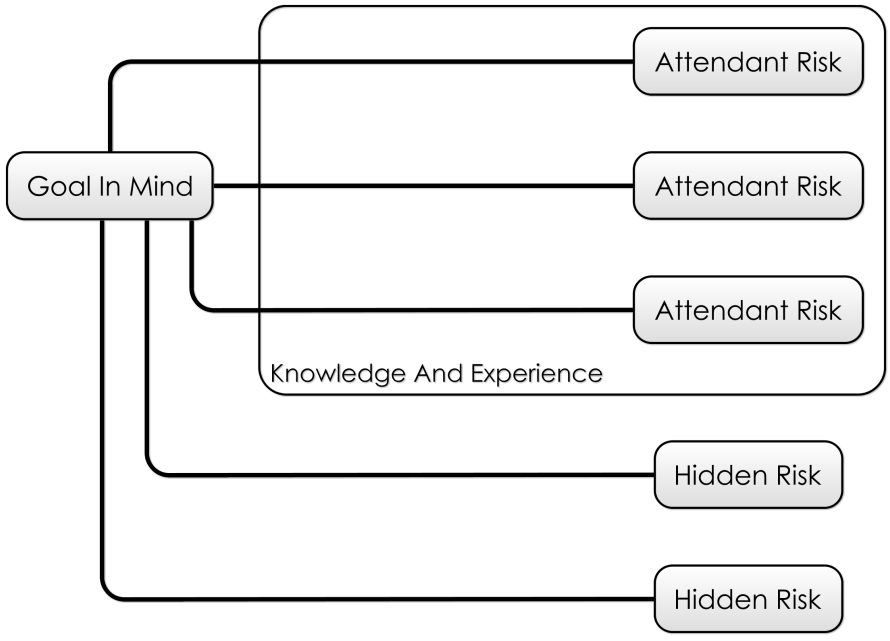


Figure 1.1: Goal In Mind

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].

Different people will evaluate the risks differently. (That is, worry about them more or less.)

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

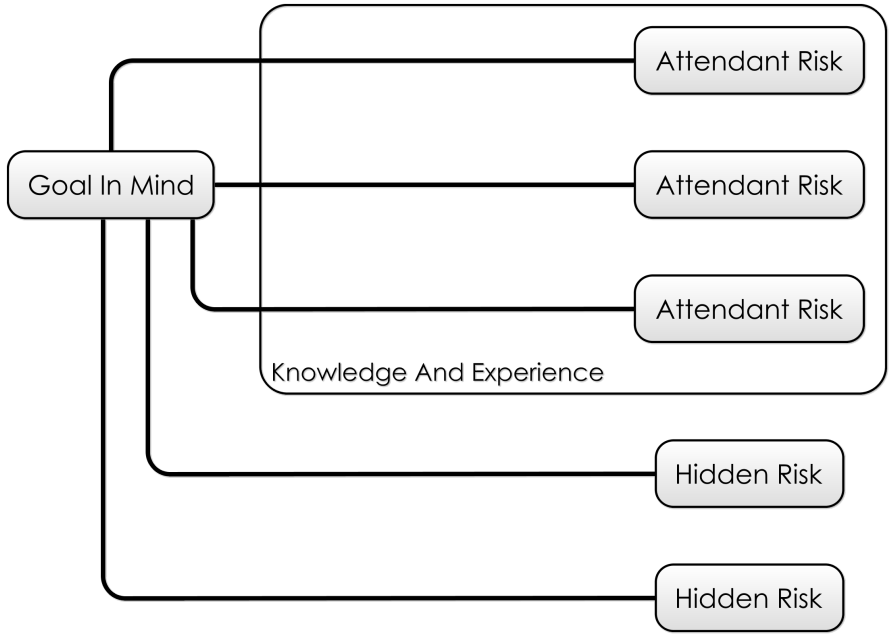


Figure 1.2: Goal In Mind

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; 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.

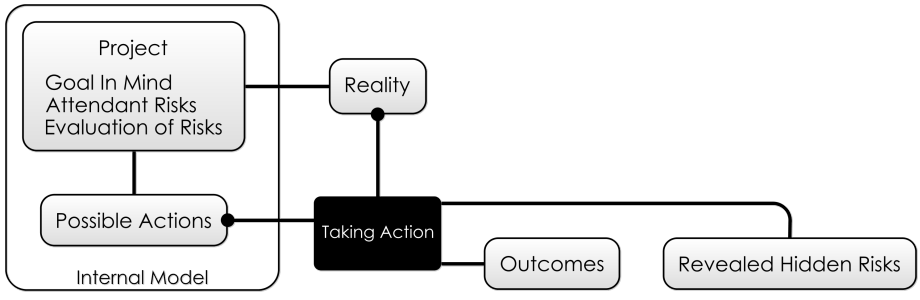


Figure 1.3: Reality

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.

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 section** 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.

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*.

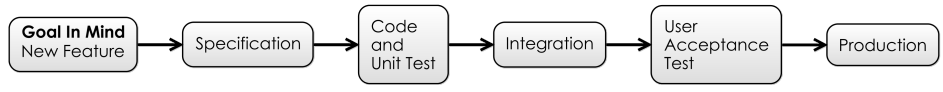


Figure 2.1: Development Process

What's happening here? Why these steps?

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.

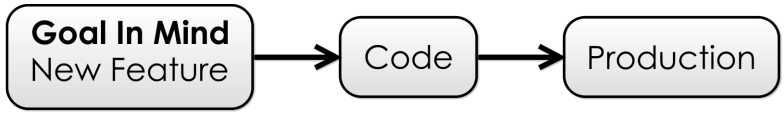


Figure 2.2: Development Process

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.

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.

Figure 2.3: Development Process - Hidden Risks

-or-

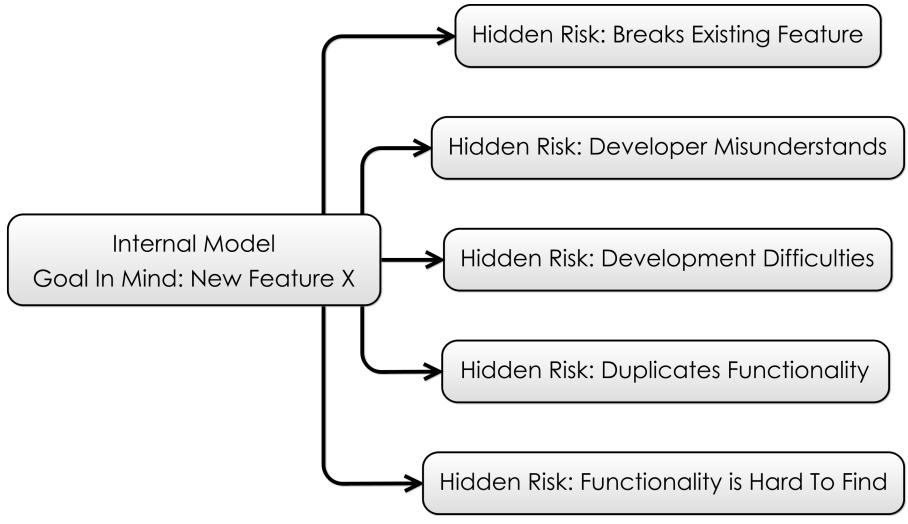


Figure 2.4: Development Process - Hidden Risks

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:

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.

Hopefully, after this stage, our **Internal Model** might look something like this:

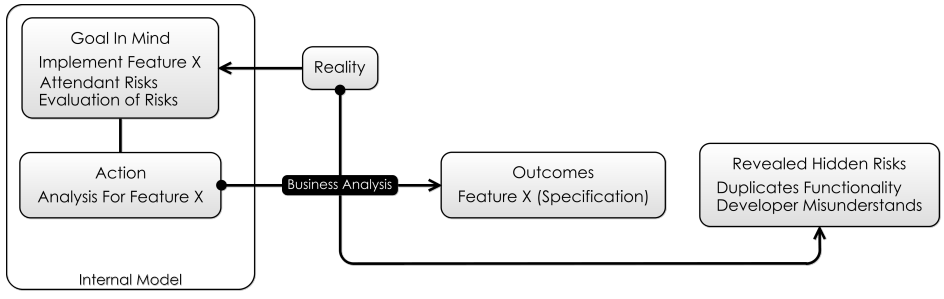


Figure 2.5: BA Specification

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. After its brush with reality, the **goal in mind** has *evolved* from being **Feature X (Idea)** to **Feature X (Specification)**.

Code And Unit Test

The next stage for our feature, **Feature X (Specification)** 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.

To a large extent, this is the whole point of *type safety*: to ensure that your **Internal Model** stays consistent with the reality of the codebase. If you add code that doesn't fit the reality of the codebase, you'll know about it with compile errors.

The same thing is true of writing unit tests: again you are testing your **Internal Model** against the reality of the system being built, running in your development environment. Hopefully, this will surface some new hidden risks, and again, because the **goal in mind** has met reality, it is changed, to **Feature X (Code)**.

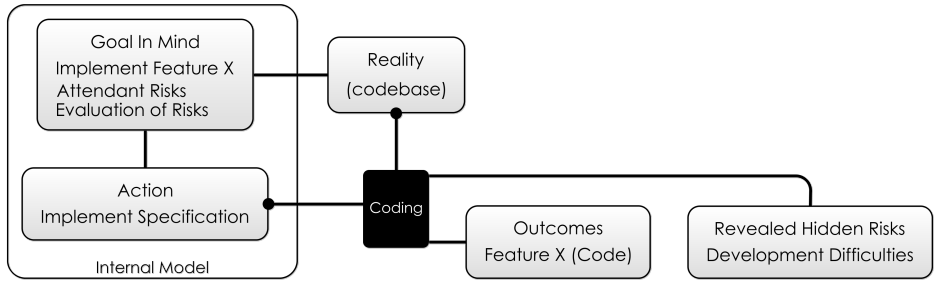


Figure 2.6: Coding Process

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**

UAT

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**:

Observations

A couple of things:

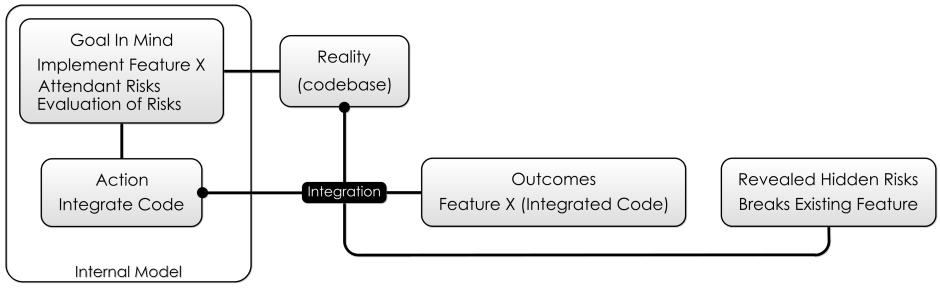


Figure 2.7: Integration

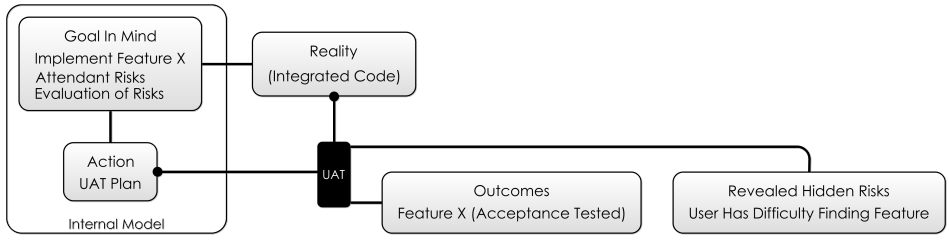


Figure 2.8: UAT

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*.

Conclusion?

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

Chapter 3

All Risk Management

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

In the **last section**, 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 toolbox, the RAID Log[1], and observe how risk-centric it is.

RAID Log

Many project managers will be familiar with the RAID Log[2]. 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.

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

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

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 Mitigates Risk

The reason you are *taking* an action is to mitigate a risk. For example, if you're coing 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*.

Every Action Carries Risk.

- How do you know if the action will get completed?
- Will it overrun on time?
- Will it lead to yet more actions?

Consider *coding a feature* (as we did in the earlier **Development Process** section). 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 mindset 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?

What To Do?

It makes it much easier to tackle the RAID log if there's only one list: all you do is pick the worst risk on the list, and deal with it. (In **Risk Theory** we look at how to figure out which one that is).

OK, so maybe that *works* for a RAID log (or a Risk log, since we've thrown out the others), but does it scale to a whole project?

In the next section, **Software Project Scenario** I will make a slightly stronger case for the idea that it does.

Chapter 4

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*.

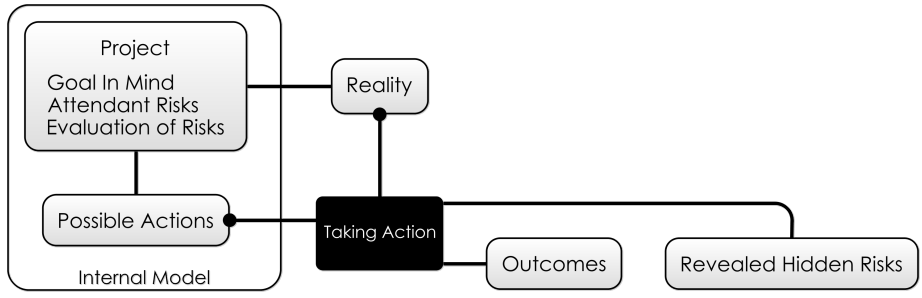


Figure 4.1: Reality

Goal In Mind

How should we decide how to spend our time today?

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

If we want to take the right actions, we need to have a good **Internal Model**.

Sometimes, we will know that our model is deficient, and our time should be spend *improving* it, perhaps by talking to our clients, or the support staff, or other developers, or reading.

But let's say for example, today our **Goal In Mind** is to grow our user base.

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.

Evaluating The Risks

Next, we can look at each of these risks and consider the threat they represent. Usually, when **evaluating a risk** we consider both it's **impact** and **likelihood**.

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 there are **bugs**, does it mean that their pictures don't end up on the internet? Does it mean that they have to restart the program? Does it mean that they'll waste time, or that they end up thinking they have insurance but haven't?

- 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?

Outcomes

As part of evaluating the risks, we can also *predict* the negative outcomes if these risks materialise and we don't take action.

- Losing Revenue
- Legal Culpability
- Losing Users
- Bad Reputation
- etc.

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

We've seen in the example of hacks on LinkedIn and Ashley Maddison[1] that passwords were not held as hashes in the database. (A practice which experienced developers mainly would see as negligent).

How does our model explain what happened here?

- It's possible that *at the time of implementing the password storage*, hashing 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.
- It's also possible that for the developers in question this was a **Hidden Risk**, and they hadn't even considered it.
- However, as the number of users of the sites increased, the risk increased too, but there was no re-evaluation of the risk otherwise they would have addressed it. This was a costly *failure to update the **Internal Model***.

Possible Action

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[2] are for: they *tell you about the **Attendant Risks*** and further, try to provide some evaluation of them to guide your actions.

[1] https://www.acunetix.com/blog/articles/password-hashing-and-the-ashley-madison-hack/
[2] https://www.owasp.org/index.php/Top_10-2017_Top_10

Actions

So, this gives us a guide for one potential action we could take *today*. But on it's own, this isn't helpful: we would need to consider this action against the actions we could take to mitigate the other risks. Can we answer this question:

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

That is, we consider for each possible action:

- The Impact and Likelihood of the **Attendant Risks** it mitigates
- The Cost of the Action

For example, it's worth considering that if we're just starting this project, risks 1-4 are *negligible*, and we're only going to spend time building functionality or improving our understanding of the market. (Which makes sense, right?)

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**.

Whether we do this explicitly or not, we are still individually following this model.

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

Chapter 5

Risk Theory

Here, I am going to recap on some pre-existing knowledge about risk, generally, in order to set the scene for the next section on **Meeting Reality**.

Risk Registers

In the previous section **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, Analyze, Plan Response, Monitor and Control.

This is Wikipedia's example:

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

[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 5.1: Wikipedia 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 **Production Risk** later on by a greater amount.

Graphical Analysis

The Wikipedia page[2] also includes this wonderful diagram showing you risks of a poorly run barbecue party:

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).

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

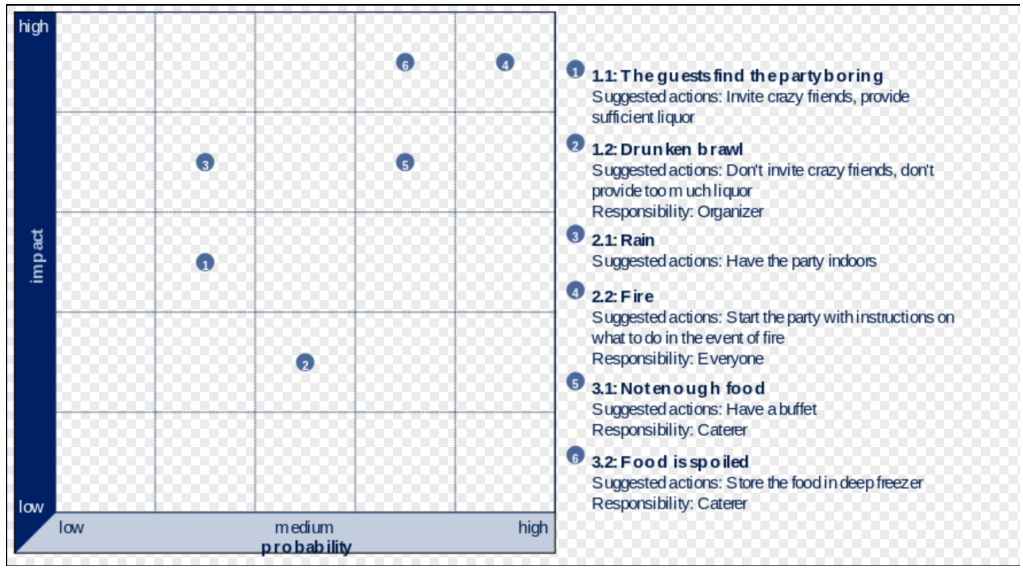


Figure 5.2: Wikipedia Risk Register

Unknown Unknowns

In Wikipedia's example, this ficticious 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

Donald Rumsfeld's famous Known Knowns[3] is also a helpful conceptualization.

Risk And Uncertainty

Arguably, this site uses the term 'Risk' wrongly: most literature suggests risk can be measured[4] 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

[3] https://en.wikipedia.org/wiki/There_are_known_knowns
[4] https://keydifferences.com/difference-between-risk-and-uncertainty.html

Risk[5] or Reputational risk[6] which are also not really a-priori measurable.

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

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 minimize the "Downside") 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. 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[7].

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*. Or at least, the way it was being implemented.

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

[5] https://en.wikipedia.org/wiki/Operational_risk
[6] https://www.investopedia.com/terms/r/reputational-risk.asp
[7] https://en.wikipedia.org/wiki/Sunk_costs

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.

tbd.

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.

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** and **Net Present Value**. 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** by Steve McConnell we have the following diagram:

tbd. redraw this.

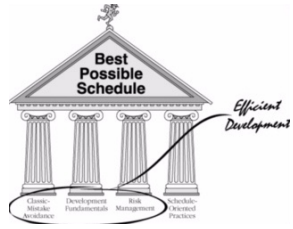


Figure 5.3: Rapid Development Pillars

And, this is *fine*, McConnel 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:

tbd. risk-first diagram.

Net Present Risk

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)

tbd. image from wikipedia. text from wikipedia.

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:

chart of discounting into the future tbd.

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

> Net Present Risk is tbd.

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 £50 in 5 year's time.
- Risk **B**, which will cost you £70 in 8 year's time.
- Risk **C**, which will cost you £120 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:

tbd, see numbers

On this basis, the biggest risk is **B**, at about #45. If we *reduce* the discount factor to 3%, we get a different result:

tbd, see numbers.

Now, risk **C** is bigger.

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

- Risk **A** costs £20 to mitigate
- Risk **B** costs £50 to mitigate
- Risk **C** costs £100 to mitigate

Which is the best deal?

Well, under the 6% regime, only Risk **A** is worth mitigating, because you spend £20 today to get rid of #40 of risk (today). The NPR is positive at around £20, whereas for **B** and **C** mitigations it's under water.

tbd.

Under a 3% regime, risk **A** and **B** are *both* worth mitigating, as you can see in this graph:

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 means that anything with a horizon over a week is irrelevant. Regimes of such **hyperinflation** 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: £80 cost, happening *tomorrow*
- Risk B: £500 cost, happening in *5 days*.

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

tbd.

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. For publically-listed companies you can also

- more pressure, heavier discounting pooh bear procrastination

Is This Scientific?

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 at collected data about IT projects, but **techniques and tools change**.

- 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?"

Reality is Reality, **so let's meet it**.

Chapter 6

Meeting Reality

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

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.

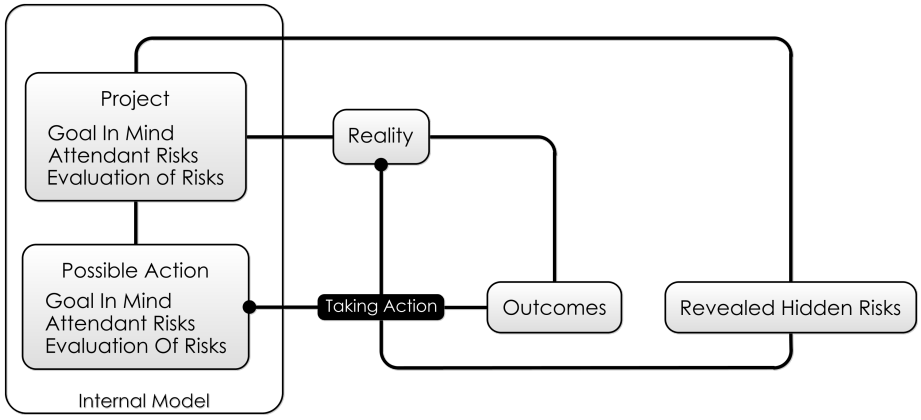
And, in **Development Process** we looked at how we can meet with reality in *different forms*: Analysis, Testing, Integration and so on, and saw how the model could work in each stage of a project.

Finally, in **Software Project Scenario** we looked at how we could use this model on a day-to-day basis to inform what we should do next.

So, 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?

"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**.

Sometimes, in solving one problem, 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.

An Example: Automation

diagram of how automation reduces process risk, but increases complexity?

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.

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**.

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 overburdened 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*.

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".

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

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 should leave it out.

Which is right?

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

- 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** 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**.

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 is a cycle around our model.
- Each cycle, you'll expose new **Hidden Risks**, changing your **Internal Model**.
- Preferably, each cycle should reduce the overall **Attendant Risk** of the **Goal**

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

Surely, the faster you can do this, the better?
Let's investigate…

Chapter 7

Cadence

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

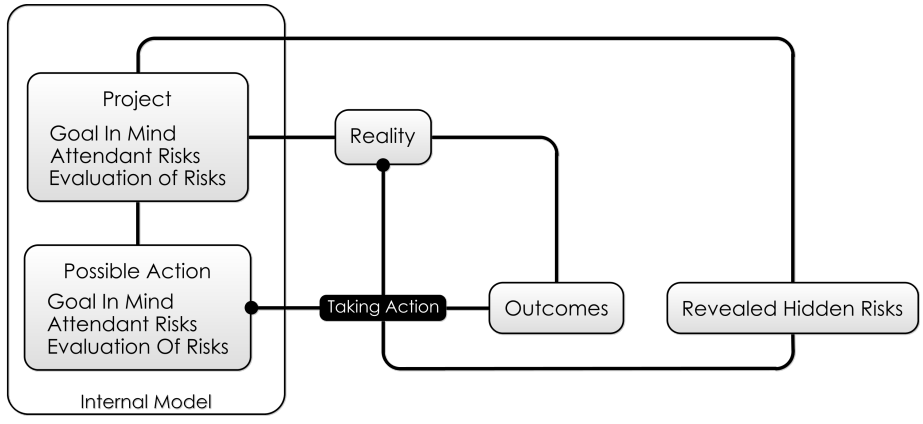


Figure 7.1: Reality 2

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** (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[1] hints at this truth:

- **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.

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

Production

You could take this section 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 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*.

Recap

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.
- 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.
- 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**.
- 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

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.

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 **Bureaucratic 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.

Aftermath

Hopefully, this type of conversation will feel familiar. It should. There's nothing groundbreaking 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.

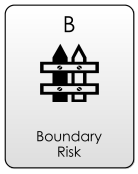
So with the groundwork out of the way, let's get on to Part 2 and investigate **The Risk Landscape**.

Part II

Risk

Chapter 9

Boundary Risk



AND DEAD END RISK

Boundary Risk is an emergent risk, which exists at the intersection of **Complexity Risk**, **Dependency Risk** and **Communication Protocol 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.

Musical Instruments - ergonomics and boundary risk.

used to be boundary risk was the risk that problems occur because you can't test across boundaries.

Integration And Translation

If we are going to use a software tool as a dependency, we have to use it's **API**:

tbd

Essentially, the API is a **protocol**: it's the language that the tool understands. If you want to work with it, you have to use it's protocol, it won't come to you. This is where **Boundary Risk** really starts.

Let's take a look at a hypothetical project structure:

tbd.

In this design, we have included 3 dependencies, `a`, `b`, `c`. As you can see, `Our Code` is orchestrating the flow of information between them: - First, it receives something from `a`, using the **Protocol** of `a`. - Then, it **Translates** this into the **Protocol** of `b`, retrieving something back from `b`. - Then, it **Translates** that into the **Protocol** of `c`.

You could say we are doing **Integration** of the different dependencies, or **Translation** between those dependencies. 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 Risk**. But each of these systems 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 `b` won't understand. - Maybe `b` works on some streaming API basis, that is incompatible with `a`. - Maybe `c` runs on Windows, whereas `a` and `b` run on Linux.

Boundary Risk Defined

Wherever we integrate complex dependencies, we 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**.

We can mitigate attendant **Boundary Risk** by trying to choose the simplest dependencies for any job, and also the smallest number of dependencies. Let's look at some examples:

- `mkdirp` is an `npm` 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 standard 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 minimize 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**, **Redux** or **Angular** 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.

So **Boundary Risk** is the attendant **Complexity** required to integrate **Dependencies**. Let's look at some examples.

Examples

Drupal and WordPress

On the face of it, **WordPress** and **Drupal** should be very similar: - They are both **Content Management Systems** - They both use a **LAMP (Linux, Apache, MySql, PHP) Stack** - They were both started around the same time. - They are both Open-Source, and have a wide variety of plugins.

However, 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**, they do it in very different ways.

Alternatively, I could have picked on Team City and Jenkins here, or Maven and Gradle. In all cases, the choice of plugins I have is dependent on the platform I've chosen, despite the fact that the platforms are solving pretty much the same problem. If I want to

This is a crucial determinant of **Boundary Risk**: given the same problems, people will approach them and solve them in different ways. And, this will impact the 'shape' of the abstractions, and the APIs that you end up with. **Boundary Risk** emerges from the solution, as the solution gets more complex and opinionated.

In all these cases,

Nowadays, the **WordPress** user base is huge: approximately tbd 30% of all websites are hosted with **WordPress**. For **Drupal** it's tbd. Because **WordPress** is so popular, it has attracted an **ecosystem** of plugin developers, who have bent the platform to more purposes, and expanded the boundary of it's functionality. The **WordPress** APIs are now much more complex than they were originally, in order to support this vast ecosystem of plugins.

TALK ABOUT EVOLUTION

Java

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 tbd).

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 an API *better*. - There is **Red Queen Risk**: our requirements evolve with time.
Android Apps weren't even a thing when Java 3 came out, for example, yet they are all written with this.
- The art of good design is to afford the greatest increase in functionality with the smallest increase in boundary possible, and this usually means **Refactoring**. But, this is at odds with **Backward Compatibility**

Each new version has a larger boundary than the one before, and this increases the API **Complexity Risk** as there is more functionality to deal with. But, on the plus side, this increased **Boundary Risk** is offset by the decrease in the **Complexity Risk** in programs depending on it: all that extra functionality should mean you have less code to write, right?

java is popular because it defeated boundary risk. js does the same.

Encapsulation is the main

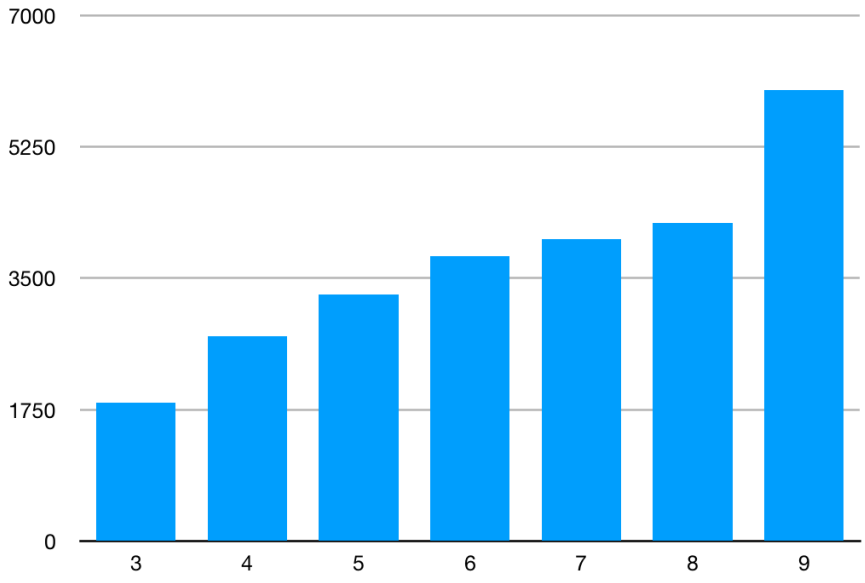


Figure 9.1: Java Public Classes By Version (3-9)

Boundary-Crossing

Sometimes, technology comes along that allows us to cross boundaries effectivey, and there are a number of ways that this can happen:

ASCII

ASCII fixed the different-character-sets boundary risk by being a standard that others could adopt.
UTF continues this work to ensure we don't have to worry about **Translation** from one encoding to another. This mitigates **Boundary Risk** by standards.

Maven / Semver

Maven is a Java build tool in which you can specify a project declaratively, including details of all its dependencies. Maven will then manage the downloading of these dependencies at compile time.
Semantic Versioning is a technique for ameliorating the problems of different dependencies being on different version numbers.

C

The C programming language provided a way to get the same programs compiled against different architectures, therefore providing some *portability* to code. This essentially reduced the **Boundary Risk** of the system architectures (although this was a leaky abstraction)

Java

Java provided something similar to C: instead of a *compile anywhere* ethos, it was a *run anywhere* ethos. Java code could run anywhere where there was a **Java Virtual Machine** installed. Again, this *abstracted* away the details of the lower level architecture.

Microservices

Microservices: using HTTP as the boundary.

Virtualization / Docker

Complex Boundaries

. - many of the concepts are the same, but it's a nightmare to map between the two.

However, sometimes, tech comes along to bridge the gap: C, Java, Containerization (Docker)

Vendor Lock-In

Trying to create a complex, fractal surface. User requirements are fractal in nature.

Executable Boundary Risk

- protocols issue - this is boundary risk too.

Add configuration risk

Should this be called contextual risk?

deployment risk

long command line

Testing in Production

platforms - compiling code for different platofrms, the jvm. javascript, other languages

protocol risk

One of the ways to sort boundary risk is with abstraction, but this doesn't work with languages - is the surface of a langauge too large?

- some languages compile back ot javascript

C# and Java, for example.

They deliberately are different in order to accentuate boundary risk/

Boundary risk moves on. e.g. the JVM abstracted away *platform*. IP protocol abstracted away a lot of different network types.

we used to worry about disks, and file formats and even ASCII wasn't a common format, everyone used something different.

Now, the boundary is AWS/Azuretc . Databases are still different *a bit*, but you're foolish for using the weird features. Tom's argument for using Oracle features.

Should I abstract away a technology? Does this gain anything

Boundary risk - filesystems and databases.

SQL, Linux, Java, Bash, Servlets, Junit, Gradle, Maven, Team City, HTML, CSS, Browsers, Javascript, SVN, GIT..

Why this?

It's like a fitness landscape. We're evolving

Evolution of dependencies

Red Queen Risk again

boundary risk is also like dead-end risk.. consider the point from the risk landscape

But originally, boundary risk referred to the fact that the boundaries were around the software.

this kind of made sense: running on your pc would be different than running on an xbox, linode etc. Muneer building the extractor, using ASP.net.. turned out we don't deploy .net/ASP only Java

So what exactly is boundary risk?

The incompatibilities of different worlds... that can't be spanned.

phantomjs / chrome headless etc.

the boundary can be too complex to overcome

java/scala

Sometimes the boundary is simple, other times complex sometimes its complexity is hidden

Clojure can interop with Java because the complexity of the boundary is *simple*: all that needs to be provided is a way to call methods on java objects and get return values.

Scala and Java have a complex relationship because Scala creates it's own complex boundary, and so does Java. It's *almost* impossible for interop here. Why is one so different from the other?

The boundaries really exist: building things in a java-only world means less boundary risk Emacs is entirely written in lisp, and your extensions to it are also lisp. This means less boundary risk, which in turn makes some tasks easier to do

Clearly, from this analysis,

Second, you can't always be sure that a dependency now will always have the same guarantees in the future: - **Ownership changes** (e.g. Oracle[1] buys Sun[2] who own Java[3] for example) - **Licensing changes**. (e.g. Oracle[4] buys **Tangosol** who make Coherence[5] for example) - Security updates not applied. - **Better alternatives become available**: As a real example of this, I began a project in 2016 using **Apache Solr**. However, in 2018, I would probably use Elastic-Search[6]. In the past, I've built websites using **Drupal** and then later converted them to use **Wordpress**.

Stuck In the Middle Pattern - Scala / Raj.

machine publishers ## Versioning

Need to rethink this:
- it's about decision making.

Boundary Risk Defined - Exists at confluence of Dependency Risk, Complexity Risk and Protocol Risk - - Complexity Of the Abstraction (abstractions still are complex).
- Translation and Integration - Boundary Risk Defined: boundaries that exist due to the dependencies' requirements. - Emergent Boundary Risk (discussion of surface areas)

Some Examples - Wordpress / Drupal: boundary around each ecosystem - Team City / Jenkins:
- Java API Surface Area - C# - AWS / Azure Etc. (Vendor Lock-In) - How Containerization mitigates boundary risk - Tom's Argument for using Oracle features

Boundary Risk Mitigated - Standards and Common Protocols (JDBC, ASCII, XML, JSON, HTTP (Microservices)) - New Abstractions (Maven, Languages: C, Java) - Translation Services (Docker, Virtualization)

Wicked Problems In Boundary Risk - Scala (type system) - Browser Testing (then, testing generally) - Testing Across Boundaries In General

- Object relational impedance mismatch Boundary risk around companies, (Contracts), abstraction boundaries. bigger the organisation, the more risk that you don't know what's happening outside it

walled gardens

[1] http://oracle.com
[2] http://sun.com
[3] https://en.wikipedia.org/wiki/Java_%28programming_language%29
[4] http://oracle.com
[5] https://en.wikipedia.org/wiki/Oracle_Coherence
[6] https://en.wikipedia.org/wiki/Elasticsearch

Chapter 10

Process Risk

Process Risk, as we will see, is the risk you take on whenever you embark on a process. This is the final part of our analysis of different **Dependency Risks**, so at the end of this section we're going to summarise what we've learned about **Dependency Risk** so far.

But first, what exactly is a process?

tbd. definition

Elaboration

In the software development world, and the business world generally, processes usually 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.

Later in this section we'll look at

The Purpose Of Process

Process exists to mitigate other kinds of risk, and for this reason, we'll be looking at them again in **Practices**. In this section, we'll look mainly at how you can deal with **Process Risk** where you are a client of the process. However, in the later section we'll look at how you can use process design to your advantage mitigating risk on your own project.

But 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 is a process where work being done 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**.
- **Operational Risk**: Operational Risk is the risk of people not doing their job properly (tbd). 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.

Evolution Of Business Process

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

tbd

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 lifecycle of how that can happen:

tbd image.

1. Person `B` in a company starts doing `A`. A is really useful! B gets busy. No one cares. But then, B goes on holiday. A doesn't get done, and people now care: the **Dependency Risk** is apparent.

tbd

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 has control of A (it might be a resource, some source of complexity, whatever). However, it needs to supply the company with A reliably and responsibly, otherwise there will be problems. They can't simply sit on the resource and do nothing, but at the same time, so they try and please all of their clients as far as possible. This is a good deal for their clients, but they end up absorbing a lot of risk.

tbd

3. T organises bureaucratically, so that there is a controlled process (P) by which A can get done. 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**). They can now deal with

requests on a first-come-first-served basis: **Resource Risk** and **Dependency Risk** are externalized.

tbd

4. But, there are abuses of A: people either misuse it, or use it too much. People do things in the wrong order. T reacts and sets up sign-off, authorization or monetary barriers around B, increasing the bureauratic load involved in using A. But, also by requiring these things, they move risk *out of* their team.

tbd

5. There are further abuses of A: bureaucratic load increases to match, increasing the amount of *process* to use A. This corresponds to greater **Process Risk** for clients of T.

tbd

6. Person C, who has experience working with team T acts as a middleman for customers requiring some variant of A. They are able to help navigate the bureaucratic process (deal with Process Risk). The cycle potentially starts again, except with process risk being dealt with by someone else.

In each step, you can see how the organisation evolves to mitigate risk around the use (and misuse) of A: First, **Dependency Risk**, then **Coordination Risk**, then **Dependency Risk** and finally, the **Process Risk** of the process that 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 heirarchy 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]

In each step, 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 Kafkaesque, forcing unintuitive contortions on their users. Dealing with complex processes is a **Communication Risk** because you have to translate your requirements into the language of the process.

But [Parkinson's Law] 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 section):

tbd - parkinson's law

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 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. tbd.

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 your request completely. The reliability is related to the amount of **Complexity Risk** the process is absorbing.
- [Visibility 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.

- Process Fit Risk/ **Dead-End Risk**: You have to be careful to match the process to the outcome you want. Much like choosing a **Software Dependency**, 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, above.

Operational Risk

When processes fail, this is called *Operational Risk*:

tbd - Wikipedia definition

This is a very specific name for **Reliability Risk** with regard to processes. In the UK each year, X number of people are killed in car accidents. If you regard driving a car from A to B as a process, then you could say that car accidents are Operational Risk[1]. Why do we tolerate such costly operational risk in the UK. Could it be reduced? Well, yes. There are lots of ways. One way is that we could just reduce the speed limit.

It is interesting that we *don't* do that: although we know the driving process fails, and fails in a way that is costly to human lives, as a society we value the freedom, the economic efficiency and time savings that come from not mitigating this operational risk. Changing the speed limit would have it's own risks, of course: there would be a complicated transition to manage. However, if ten times as many people were killed in car accidents, and it was shown that reducing the speed limit would help, maybe it would be done. The **Operational Risk** would outweigh the **Schedule Risk**.

The point of this is that we *accept* **Operational Risk** as we go. However, if opportunities rise to mitigate it, which don't leave us with a net risk increase elsewhere, we'll make those improvements.

Counterparty 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 Counterparty Risk:

> tbd.

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 Counterparty 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* and *guarantees* help reduce this risk:

Feedback Loops

Operational Risk is usually incurred for outliers: processes tend to work well for the common cases, because *practice makes perfect*. Processes are really tested when unusual situations occur. Having mechanisms to deal with edge-cases can incur Complexity Risk[2]Complexity-Risk),

[1] problem%20of%20_delivering%20on%20our%20promises_%20as%20a%20dependency.

[2] to%20do%20with%20the%20number%20of%20independent%20components%20in%20the%20system,%20and%20their%20interactions.

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.

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 risk from signing off, on the other, the risk of upsetting 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 signoffs 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.

Bureaucratic Risk

tbd.

Dependencies - A Quick Review

Dependency of any kind is a choice in which you are trying to choosing a position of lower **Attendant Risk** than you would have without the dependency.

So, we've looked at different types of dependencies.

Chapter 11

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. Hence the dilemma. So, **Agency Risk** flourishes where there is **Invisibility Risk**.

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.

Let's look at some examples of borderline **Agency Risk** situations, in order to sketch out where the domain of this risk lies.

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.

[1] https://en.wikipedia.org/wiki/Principal–agent_problem
[2] https://en.wiktionary.org/wiki/bad_actor

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.

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.

The Hero

> "The one who stays later than the others is a hero." - Hero Culture, *Ward's Wiki*[3]

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.

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[4].

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**.

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[5]) that'll keep them occupied. This will leave you with more **Complexity Risk** than you had in the first place.

[3]http://wiki.c2.com/?HeroCulture
[4]https://en.wikipedia.org/wiki/Key_person_insurance
[5]https://en.wikipedia.org/wiki/Gold_plating_(software_engineering)

Even if they don't worry about their jobs, doing this is a way to stave off *boredom*.

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*[6]

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.

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*[7]

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 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.

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*[8]

[6] https://en.wiktionary.org/wiki/pet_project

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

[8] https://news.nationalgeographic.com/news/2011/09/110914-optimism-narcissism-overconfidence-hubris-evolution-science-nature/

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[9]. There is probably an interesting article 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.

On The Take?

tbd

tbd chapter link

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

Chapter 12

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.

In this section and beyond, we're going to use the term **Agent**, which refers to anything with *agency* in a system to decide it's own fate. That is, an **Agent** has an **Internal Model**, and can **take actions** based on it. In this section, we're going to consider Agents at several different levels (because of **Scale Invariance**) . We'll look at: - People - Teams - Organisations - Processes (in software) - and Organisms

... and we'll consider how **Coordination Risk** is a problem at each scale.

In this section, we're going to work on the assumption that the Agents can agree on a common **Goal**, but in reality it's not always the case, and we'll analyse that more in the section on **Agency Risk** later.

Also, in the section on **Map And Territory Risk**, we'll look at how, even when **Coordination** issues are solved, we can end up with less-than-optimal results.

But for now, let's crack on and examine where **Coordination Risk** comes from.

Problems Of Coordination

Here are some classic problems of coordination:

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 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:

- 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. Organizing on such a huge scale becomes self-defeating, and so instead at some level we end up with competition instead of coordination. 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. We'll address in more depth later.
4. **Resource Allocation**: 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 tbd, and ensuring that packages get delivered around the world in good time. 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**, 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 losing 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.

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 risk. Here, however, we're looking

at the situation where there is competition for those dependencies: other parties want to use them in a different way.

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[1], 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.

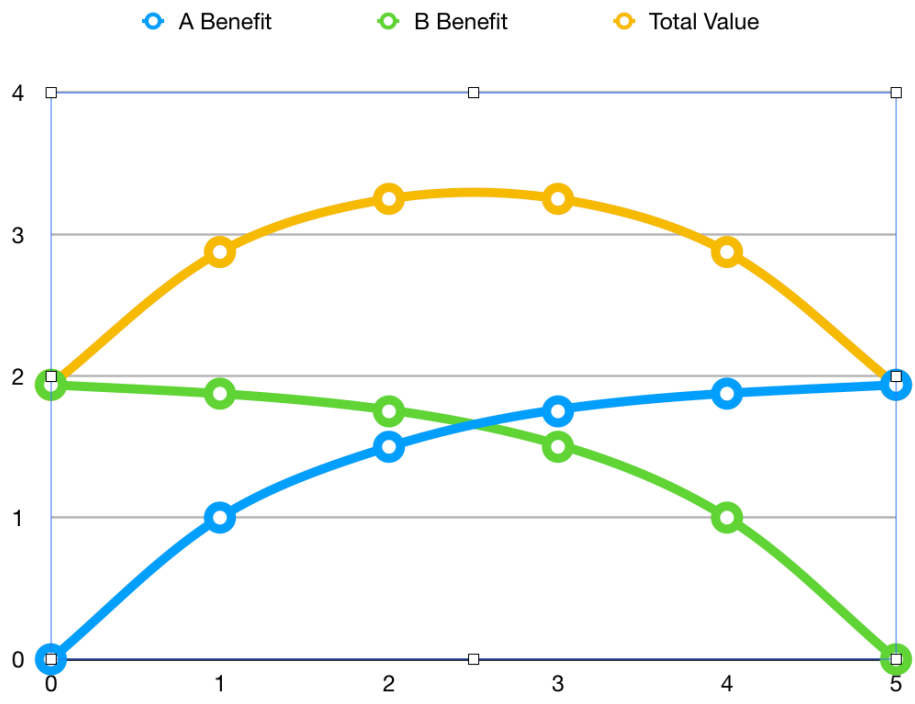


Figure 12.1: Sharing Resources. 5 units are available, and the X axis shows A's consumption of the resource. B gets whatever remains. Total benefit is maximised somewhere in the middle

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.

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**.

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

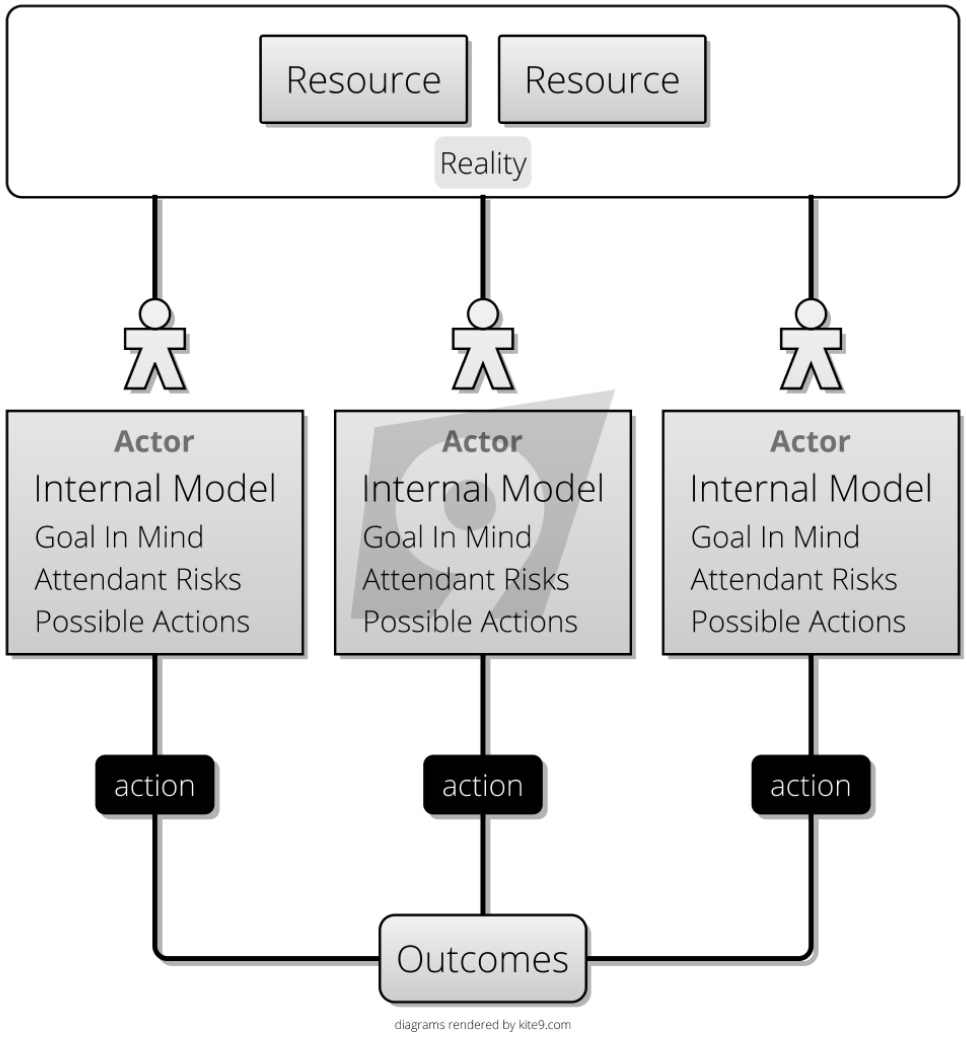


Figure 12.2: Coordination Risk 2

The only way that the Agents can move away from competition towards coordination is via **Communication**, and this is where their 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 (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 section on **Communication Risk**, we can only hope to synchronize **Internal Models** if there are high-bandwidth **Channels** available for communication.

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 Based Development**"). This is probably 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 introduced a model of group decision making which looks something like this:

In their original formulation, Vroom and Yetton introduced five different ways of making decisions as a team, which 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.

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[2]	Autocratic, top-down

Type	People Involved In Decision	Opinions	Channels Of Communication	Coordination Risk	Description
AII	1	1	s + i (Messages from informants)		Autocratic, with information flow up.
CI	1	1 + r	s + i + r (Opinions)		Individual Consultations
CII	1	1 + r	s + i + r2		Group Consultation
GII	r	1 + r	s + i + r2	Maximum Communication Risk[3]	Group Consultation with voting

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.

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** or a **Factory Method**, 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**:

2

3

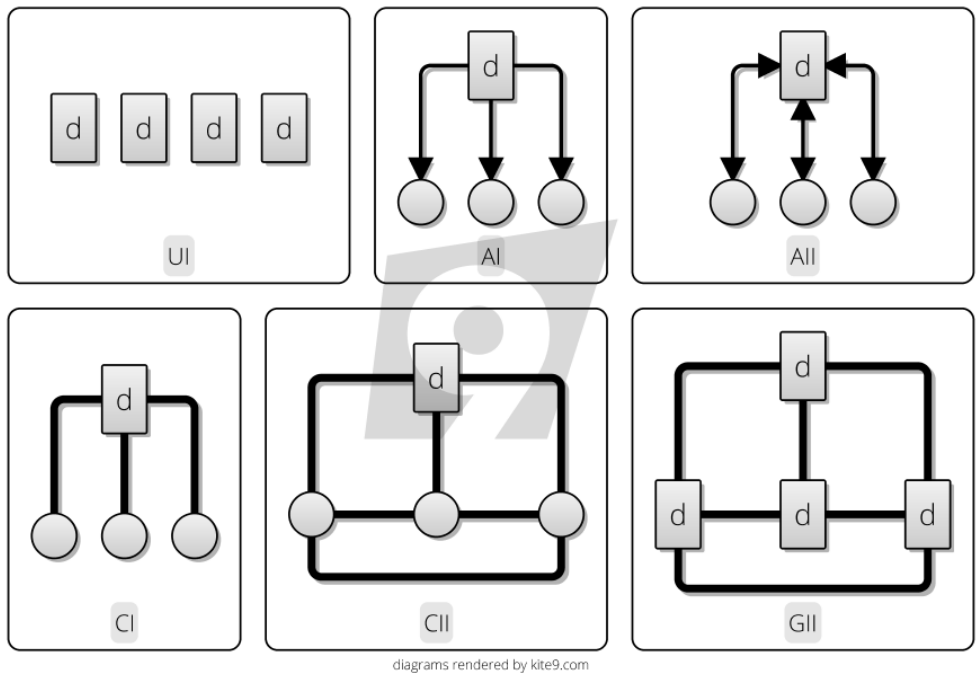
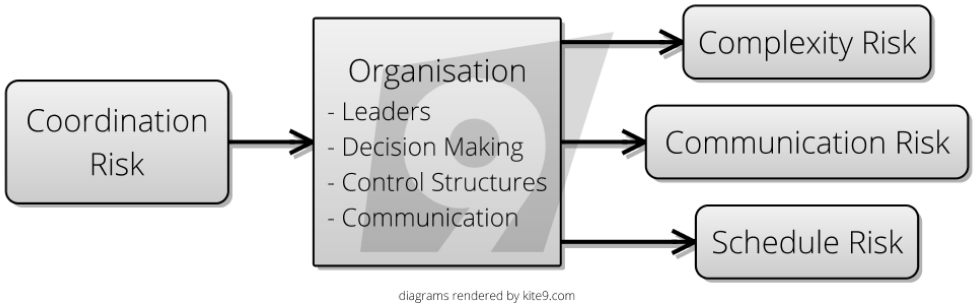


Figure 12.3: Vroom And Yetton Decision Making Styles. "d" indicates authority in making a decision. Thin lines with arrow-heads show information flow, whilst thick lines show *opinions* being passed around.



– add specialization

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**, 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*. In the human body, we have various systems[4]:

- The **Respiratory System** which is responsible for ensuring that **Red Blood Cells** are replenished with Oxygen, as well as disposing of Carbon Dioxide.

- The **Digestive System** which is responsible for extracting nutrition from food and putting them in our **Blood Plasma**.
- The **Circulatory System** which is responsible for moving blood cells to all the rest of the body.
- The **Nervous System** which is responsible for collecting information from all the parts of the body, dealing with it in the **Brain** and issuing commands.
- The **Motor System** 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 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 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**

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

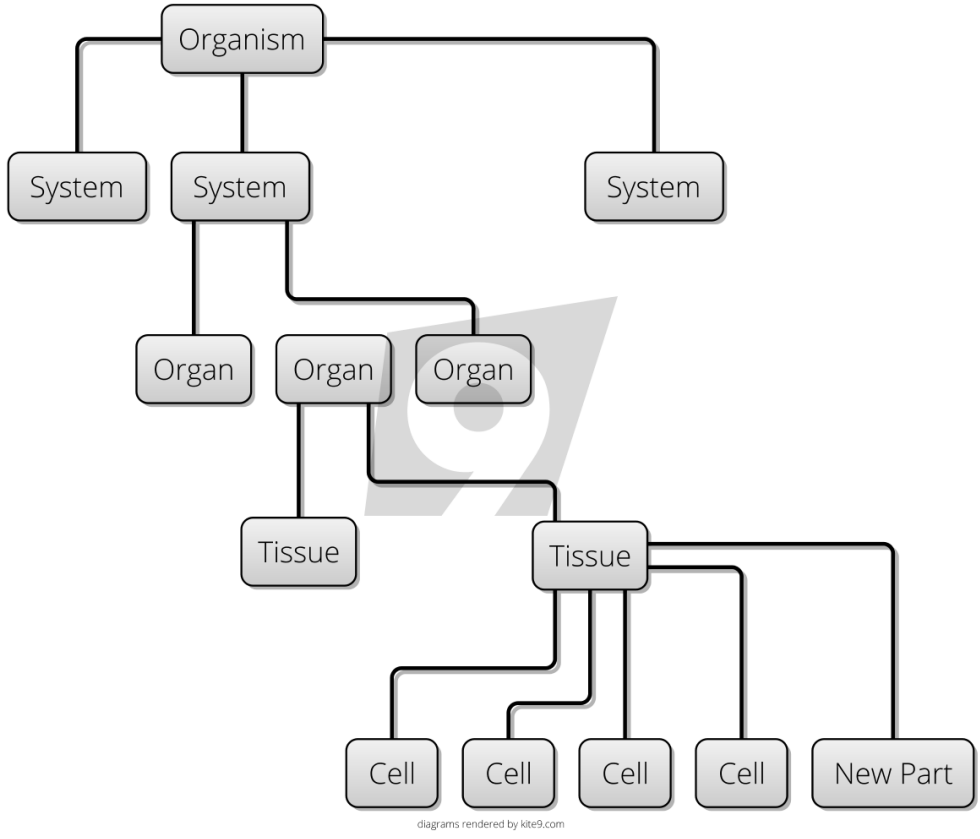


Figure 12.4: Hierarchy of Function in the Human Body

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 acceed 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**, 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** 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.

As we saw in **Process Risk**, *evolution* is the driving force taking steps on the risk landscape tbd.

Large Organisations

Working in a large organisation often feels like being a cell in a larger organism. 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, so we need something simpler. As we saw in the **Complexity Risk** section, 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. So we end up with a hierarchy of groups like this:

tbd diagram of this.

Staff Risk

Staff in a team are both **Agents** and **Resources** at the same time. The team **depends** on them 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.

If **Coordination Risk** is about trying to mitigate differences in **Internal Models**, then 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**:

after **Tuckman**,

They can be encouraged with orthogonal practices such as **Team Building exercises** (generally, submitting everyone to extreme experiences in order to bond them together). With enough communication bandwidth and entente, a motivated team will self-organise code reviews, information exchange and improve their practices. But **Staff Risks** sometimes cannot be resolved without escalation:

- People leave, taking their **Internal Models** and expertise with them **Key Man Risk**.
- People often require external training, to understand new tools and techniques **Learning-Curve Risk**
- People can get protective about their knowledge in order to protect their jobs **Agency Risk**.
- Where there are mixed ability levels, senior developers might not help juniors as it "slows them down"
- People don't get on.

... and so on.

new starters

norming, storming etc.

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.

As before, in order to face **Coordination Risk** in software, we need multiple Agents all working together, so **Coordination Risks** (such as race conditions or deadlock) only really occurs in multi-threaded software where there is resource competition.

CAP Theorem

The **CAP Theorem** has a lot to say about **Coordination Risk**. Imagine talking to a distributed database, where your request (read or write) can be handled by one of many agents.

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.

Since *any* agent can receive the read or write, it's 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, and what the last value written was.

CAP Theory states that this is a **Trilemma**. 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 a diagram how this plays out. In this, the last write (3) was sent to a node which is now *isolated*, and can't be communicated with, due to network failure. What do you get back?

tbd image

CA: show how the system wouldn't be partition tolerant if the last write was on an isolated node. CP: waits until the isolated node comes back AP: you can return some value back, but it won't necessarily be the last one.

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. This explains in part why *countries* are often created along geographic bounds:

Immutability

Immutability (or write-only data structures) are often presented as a solution to many of the problems of multi-agent systems. After all, if values in the system aren't *changing*, then memory is not a scarce resource, and we avoid **race conditions**. However, we *still* have to contend with **Coordination Risk**. Let's look at two examples.

First, **BitCoin** (BTC) is a write-only **distributed ledger**, where agents *compete* to mine BTC, but also at the same time record transactions on the ledger. But there is *huge* **Coordination Risk** in BTC, because it is pretty much outright competition. If someone beats you to completing a piece of work, then your work is wasted. For this reason, BTC agents *coordinate* into **mining consortia**, so they can avoid working on the same problems at the same time. Nevertheless, the performance of BTC is **highly questionable**, and this is because it is entirely competitive. In **CAP** terms, BitCoin is tbd.

Second, **git** is a write-only ledger of source changes. However, as we already discussed, where different agents make incompatible changes, someone has to decide how to resolve the conflicts so that we have a single source of truth. The **Coordination Risk** just *doesn't go away*. Git is an AP system.

Monitoring

tbd. talk about invisibility risk again.

Communication Is For Coordination

Earlier in this section, we questioned whether **Coordination Risk** was just another type of **Communication Risk**. However, it should be clear after looking at the examples of competi-

tion, 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 needs to communicate.

The whole point of communication is for coordination.

In the next section, **Map And Territory Risk**, we're going to look at some new ways in which systems can fail, despite their attempts to coordinate.

Chapter 13

Map And Territory Risk

As we discussed in the section on **Abstraction Risk**, our understanding of the world is entirely informed by the names we give things and the abstractions we create.

Map And Territory Risk is the recognition that there is a danger that we come to believe the abstractions are more real than reality itself. It comes from the expression "Confusing the Map for the Territory". That is believing the abstraction is reality, instead of reality itself.

Sat-nav blunder sends Asda van crashing down narrow footpath

An Asda supermarket delivery van driver in Lancashire crashed his vehicle on a narrow footpath after his satellite navigation sent him the wrong way.

A van got stuck on a narrow footpath when the driver took a wrong turn while blindly following his sat-nav. Photo: MEN

How about that?

News » UK News »
Technology News »
Andrew Hough »

In How About That?

Pictures of the day

Figure 13.1: Sat Nav Crash - Telegraph Newspaper

In the picture shown here, 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: *so many times* the SatNav had been right, that the driver stopped questioning its fallibility. But SatNavs are pretty reliable, this is kind of excusable. People are happy to make this mistake with far less reliable systems because often it's a shortcut to having to do any real thinking.

Metrics

The simplest type of **Map And Territory Risk** occurs like this: 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
- **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" - "response time is correlated with revenue"

Because the *thing on the left* being is immediate and easier to measure than *the thing on the right*, it becomes used as a proxy (or, Map) for the thing they are really interested in (the Territory).

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.

When you have easy go-to metrics based on accidental or incidental correlations, **Hidden Risk** mounts up. By relying on the metrics, you're not really *seeing* the reality. The devil is in the detail.

Drinking The Kool-Aid

The next problem comes when metrics 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, when ambulances were asked to respond to all emergency calls within a short window, cars and bicycles were employed as ambulances too [tbd].

You are probably nodding your head at these examples. *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.

The Onion Of Bullshit

Map-And-Territory Risk "trickles down" through an organisation, in what I term "The Onion Of Bullshit". In which successive layers of the organisational heirarchy imposed worse and worse. Here's how this came about in a bank I worked at:

- 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...
- 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 "reuasable" framework which is only used by a single application is bullshit. But, we had to do this because...
- 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
- The matrix structure was largely the legacy of a recent merger with another department. As **Conway's Law** predicts, our software therefore had to reflect this structure. But this was bullshit because
- 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
- The budget that was given to our department (Risk) was really based on how much fear the budget holders currently had of the market regulators. But this was bullshit too, because
- At a higher level, the executives had realised that Investment Banking wasn't one of the banks strategic strengths, and was working to close it all down anyway.

When faced with so many mis-aligned objectives, it seemed completely hopeless to concentrate on the task at hand. But then, my colleague Gavin was able to nihilistically complete the onion by adding a final layer:

- It's all about chasing money, which is bullshit, because life is bullshit.

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 of the onion, 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? If not, the project might well be quite temporary: Before I left, I was able to eject most of the "framework" elements of the project, and massively simplify the architecture, thus obviating the need for the smoke tests.

So far, we've considered what happens when a team *has been told* to optimise around a particular objective. But it's not a great stretch from here to a point where people are optimising the metric at the expense of doing what they know is best for the project. Or, optimising a metric for personal gain because that metric is more visible than other (perhaps more important) qualities. This is **Agency Risk** which we'll look at in the next section.

Inadequate Equilibria

Inadequate Equilibria is a book by Eleizer Yudkovsky, who looks at how **Map and Territory Risk** can break not just departments, but entire societal systems. Here is 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 measureable, 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.

So, publication in prestigious journals is a *metric* which is open to abuse, as we saw earlier.

- Everyone can see this system is broken, but no-one can change it because they are all trapped within it.
- The onion is a better situation - at least the people at the top can do something.

Picking Projects

tbd

Head In The Sand

Introduce Rapid Development example here?

how to pick projects

how to spot vanity projects

how to spot where the Goal In Mind is hopelessly ill-thought-through. following the rules more important than getting things done.

Head in the sand

Bullshit jobs

Biases

It's tempting to think that **Map And Territory Risk** is something that happens to someone else, and that you are going to be immune to it.

Unfortunately, science is here to prove you wrong - our brains are *filled* with biases which

Why is this relevant?

human biases.

- showing progress
- the release
- less wrong

Is this really a risk?? Why is this here? To shore up the scheduling thing?

Do biases go in here? YES!!

Any time your internal model is inconsistent with the world outside. Should we call this internal model risk?

Machine biases: machines only store what we put in them, which is not necessarily reality either

Chapter 14

Operational Risk

In this section on [Operational Risks], we're going to take our head out of the clouds a bit and start considering the realities of running software systems in the real world. After all, **Coordination Risk** and **Map And Terriory Risk** got a bit theoretical. Here, we're going to look at [Operational Risks], including **Security Risk** and Reputational Risk: real-world concerns for anyone running a business.

But before we go there, let's try and recap on where we've come so far. So far, we've been looking at risks to *systems in general*:

-
-
-
- **Coordination Risk** by taking on Communication Risk[1].

Here is a diagram that shows how these different elements line up:

But, although this is starting to look complicated, in reality, things are *worse*. We have to contend with **Bad Actors** from *within* and *outside* the system too, and, we have to consider our software a component of a larger *system in the world*, which consists of our *customers* depending on us. Our system is embedded within a context which we can't hope to fully understand:

-
-
-
-

[1] how%20they%20mitigate

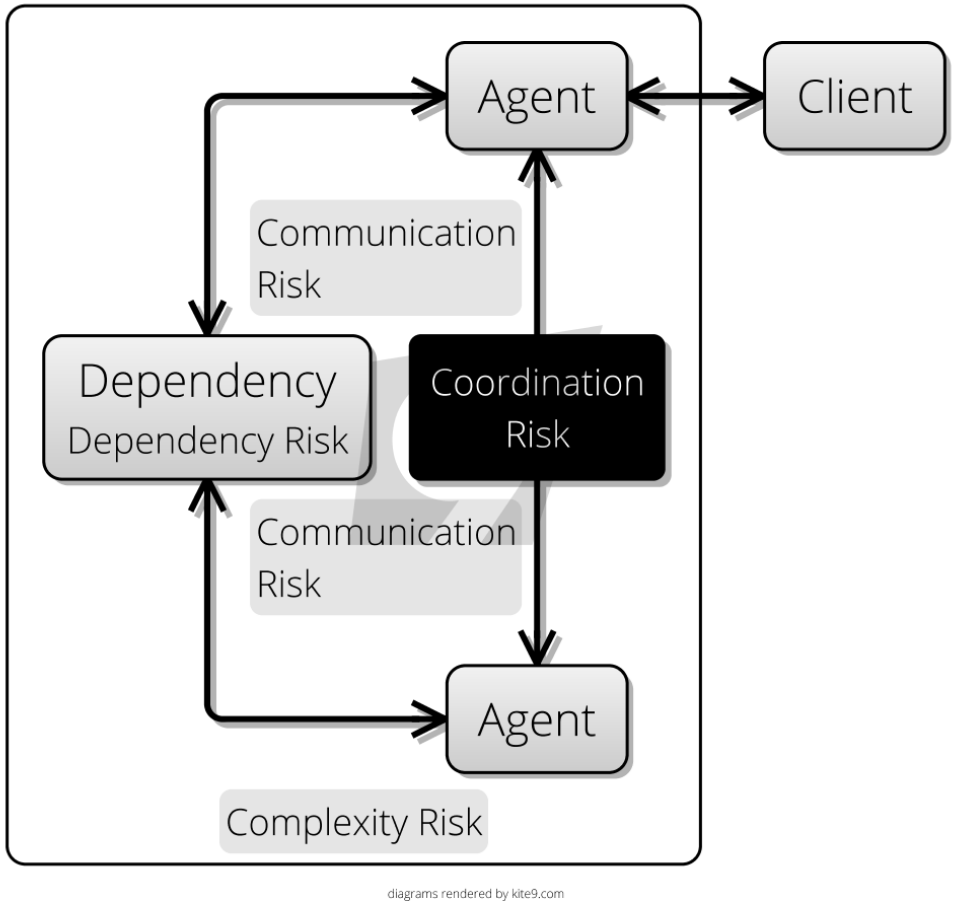


Figure 14.1: Systemic View of Risks

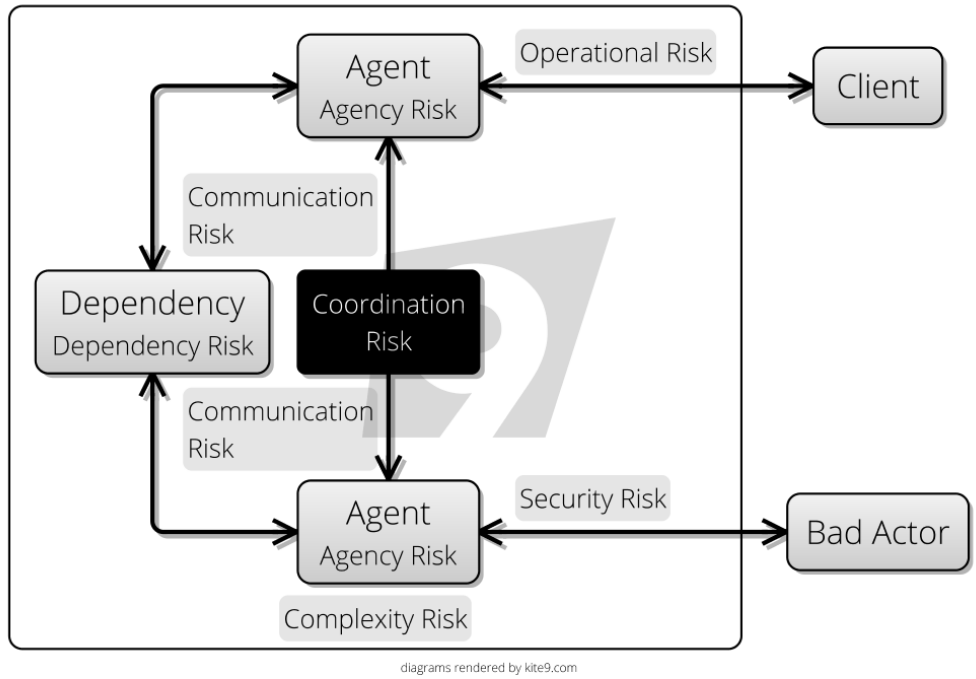


Figure 14.2: Systemic View of Risks (2)

Introducing Operational Risk Management

The *software systems* we build and put into production don't just consist of software. tbd

Operational Risk is a very broad church, and encompasses pretty much all the real-world things that can go wrong with your enterprise. Let's try and define what we mean by Operational Risk[2]:

> "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*[3]

Here are some examples of Operational Risk[4]: - Freak weather conditions affecting ability of staff to get to work, interrupting the developmemt and support teams. - A data-centre going off-line, causing your customers to lose access. - 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. - Hackers breaking into the system and bringing your service down. - Reputational damage caused when staff are rude to the customers. - Regulatory change, which means you have to adapt your business model.

... basically, a long laundry-list of everything that can go wrong due to operating in the Real World. *Ideally*, when we design our software system, we design in features to *strengthen* it

[2] problem%20of%20_delivering%20on%20our%20promises_%20as%20a%20dependency.
[3] https://en.wikipedia.org/wiki/Operational_risk_management
[4] problem%20of%20_delivering%20on%20our%20promises_%20as%20a%20dependency.

against **Operational Risks** such as these. *However*, we don't get these for free: they come at the cost of extra [Complexity].

For example, as we saw in Development Process, a lot of the processes we put in place (e.g. **Continuous Integration** or various types of [Testing]) are there to mitigate **Operational Risk** problems caused by *releasing buggy software*.
Unit Testing increases our **Kolmogorov Complexity** (as there is more code) but we accept this extra complexity as the price for mitigating this Operational Risk[5].
User Acceptance Testing increased our [Schedule Risk], but again reduced the likelihood of bugs making it into production.

Mitigating Operational Risk

Operational Risks, then are things that happen *to* our carefully constructed, theoretical system, as shown in this diagram:

diagram: our system -> event -> weakness -> effect -> reaction -> recovery -> adaptation

(our system: meeting reality, deployment) (event: detection) (weakness: minimization (see Security risk)) (effect: what happens, how much chaos ensues) (reaction: monitoring, etc) (recovery: how we fix it) (adaptation: how the system changes in the future)

– something like this.

Therefore, one of the best defences against **Operational Risk** is dealing with the issues quickly once they happen. This requires:

Good **Feedback Loops** in the form of **Monitoring** and rapid response to issues.

Meeting Reality

So in this second model, we are now considering that the world is a dangerous, untrustworthy place where *bad things happen*, either deliberately or accidentally. And, since we don't have a perfect understanding of the world, most of the **Production Risk** we face is **Hidden Risks**.

Putting software into production is **Meeting Reality** in the fullest way possible. The more contact we can give our system with the outside world, the more **Hidden Risks** will materialize. If we observe these and take action to mitigate them, then our system can get stronger. cybernetics, antifragile tbd.

It is tempting to delay **Meeting Reality** as long as possible, to "get your house in order". 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 becomes greater than pressure to improve tbd

A Risk-First reframing of this might be the balance between:

- The perceived Reputational Risk, **Feature Risk** and **Operational Risk** of going to production (pressure to improve)

[5] problem%20of%20_delivering%20on%20our%20promises_%20as%20a%20dependency.

- The perceived **Schedule 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:

Meet Reality...	Techniques
Sooner **More Frequently** **Sprints** **In Smaller Chunks**	Limited Early-Access Programs, Beta Programs, Soft Launches [Continuous Delivery],

Modular Releases Microservices Feature Toggles Trial Populations | |**With Feedback** |User Communities, Support Groups, Monitoring, Logging, Analytics|

External Events and Bad Actors

We're familiar with the concept of taking steps on the [Risk Landscape], wherein we **take action** to move to a position where the **Attendant Risks** are more acceptable to us. However, now we have to contend with the idea that external events *also* change the risk landscape too. If there is sudden bad weather, we might have a risk of a power-cut, and the losses that might entail to productivity or sales. If there is a change of government, that might impact the contracts we've written, or the security of our servers or staff.

Being *in production* is accepting that the **Risk Landscape** is a volatile, uncaring place. But it's worse than that, since we also have to contend with [Bad Actors], who are deliberately out to exploit weaknesses in the systems we build.

Ordinarily, when we transact with a [Dependency], it should be the case that after the transaction, there is value on both sides of the transaction. This could be, *you do my accounting*, I pay you money. On both sides, financial risks are reduced. If the price is too high, or too low, we see one or other side getting the better deal, and *capturing an unfair share of the value*.

With a [Bad Actor], we're in a situation more like a zero-sum game: value is *taken* from one party and *transferred* to the other. These are exactly the dependency relationships that societies *don't* condone: there is net *zero* or *negative* value in the transaction.

- Regulatory Risk Legal Risk (Pestle?)

Security Risk

Complex systems (ones which contain multiple, interacting parts, like the ones in the above diagrams) have to contend with their external environments, and try to minimize the ways in which they get interrupted from outside either by *Bad Actors* or external events. In the tbd

Interestingly, security is handled in very similar ways at all sorts of levels:

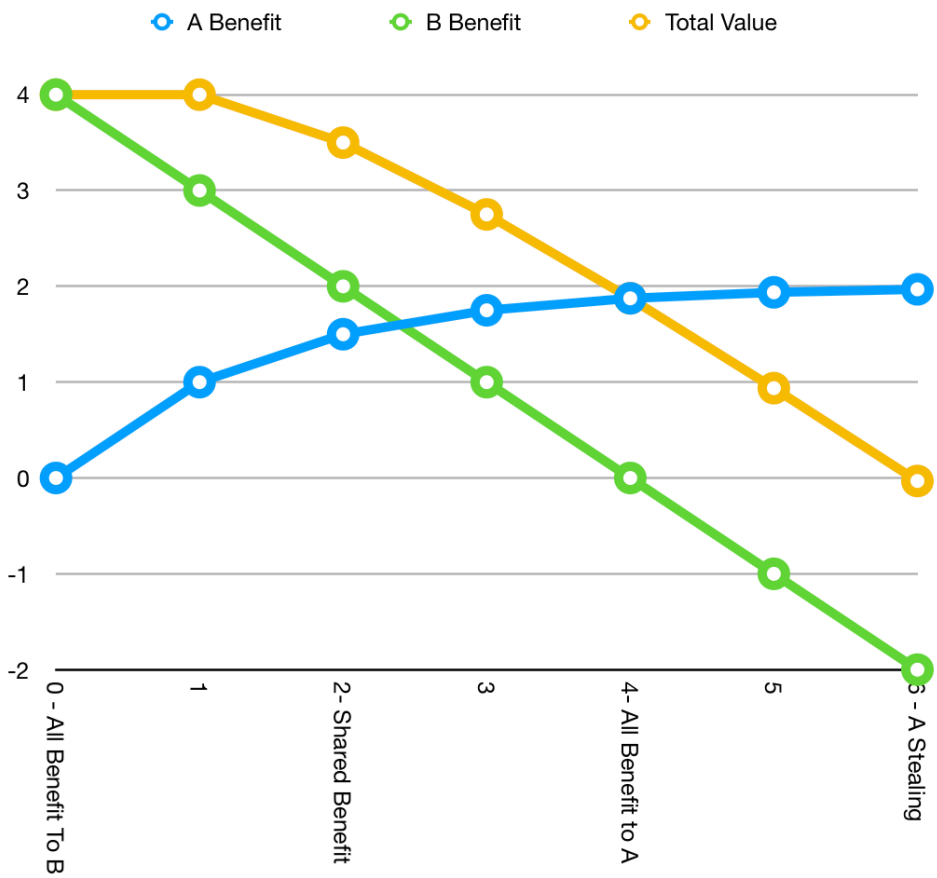


Figure 14.3: In this diagram, B does a deal with A, the value of the product B gets from A is always 4, but the price varies with the X axis. When the price is zero, B captures all the value, but as the price increases, it becomes a worse and worse deal for B, and the net value heads towards zero or negative.

- **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. diagram version 3.

Mitigating **Security Risk** is a trade-off. You can spent *a lot* of time and effort on this, only to never face the

secrets: how to mitigate this

Effect / Impact

Sometimes, it's possible to measure the impact of Operational Risks. For example, if a software system fails, and leaves customers unable to access it, this can have a measurable financial impact in lost revenues or damages. Car recall example tbd. - fight club

Impact is usually proportional to some of the below variables:

- Number of customers affected.
- Number of transactions affected.
- Size of the transactions
- Length of time systems were affected.

stuff that can go wrong in production

changing stuff in production is harder than changing it in test, as you have to *migrate*.

all the costs of breaking stuff, and damaging the running of the business.

reputation damage (you only get one chance to make a first impression)

- You don't know all the ways the software will get used in production.
- Different browsers, versions of code, accessiblilty.
- CAn you support all the users? IS there enough kit? WIll you know?

Correlation - Upgrades (tell story of Research upgrade that went wrong because we were upgrading at the same time as an outage) - Single points of failure.

reputational damage

Reaction & Recovery

Reliability Risk

```
- Feedback Loops
   - Bug reports, feedback
   - Quality of feedback
   - Internal Controls
     - Agency Risk meets Production Risk (bad actors, controls)



  - Contingency Planning
  - Disaster Recovery
        - Performance Degradation / Runaway processes  (Performance Risk)

        - Support (trade off - promptness vs ability)

  Sometimes, the reaction of the company makes things worse - streisand effect? others?

  - Poor monitoring, visibility risk meets operational risk (otherwise, it doesn't matter - good exam
  - Correlation  (need a good example here)
  - Monitoring Tools and Logs
```

Prevention

- How we learn from our mistakes
- You can't know everything
- Reality changes anyway

Performance Risk

There is a lot more to Operational Risk. Here, we've touched on it, and sketched the edges of it enough for it to be familiar and fit in our framework.

Reputational Risk

High-Profile Cases

Maturity

https://math.nist.gov/IFIP-UQSC-2011/slides/Oberkampf.pdf https://www.bsimm.com/framework/intellige
models.html

Chapter 15

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** our collection on some solid [Mounting Boards], and do some work in classifying what we've seen.

tbd collecting image

Some Observations

Your Feature Risk is Someone Else's Dependency Risk

In the **Feature Risk** section, we looked at the problems of *supplying a dependency to someone else*: you've got to satisfy a demand [Market Risk], and ensure a close fit with requirements [Conceptual Integrity Risk]. The section on **Production 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** sections, 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** *is* **Dependency Risk**: they're two sides of the same coin. In a dependency, you're a client, whereas feature risk, you're the supplier.

Coordination Risk

- similar to _threading/deadlocking issues

One thing that should be apparent is that there are similarities in the risks between all the kinds of

Production risk == security risk??

Boehm.. OWASP..

How much do compilers do for you?

1. Classifying Risks

- Dependencies and Features are the same
  - Dependency risks are all 2-sided. (Counterparty risk)
- Communication is a Dependency
- Fit Risk / Communication Risk - buckets
  - Fit risk example from work: choosing the right format.
  - Why standards can never be perfect.
  - Mention Kanban = the control is the physical object
- Expected Requirement Coverage - diagrams 1 & 2.
- Dependencies and Coordination
- Need to do this again, now.
- In The Bin
  - CapsLock: complexity, not using tools.
  - Configuration Tool (Complexity, feature fit, bugs in hibernate, 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)

2. What's Gone Before

- An attempt to categorize all the ways in which software projects go wrong.
- Boehm.

3. What's To Come

- risk based debugging.
- risk based coding.

4. On to Part 3.

- I'm a better coder for knowing this.
- We're all naturalists now.

Part III

Glossary

Chapter 16

Glossary

Abstraction

Goal In Mind

Internal Model

The most common use for **Internal Model** is to refer to the model of reality that you or I carry around in our heads. You can regard the concept of **Internal Model** as being what you *know* and what you *think* about a certain situation.

Obviously, because we've all had different experiences, and our brains are wired up differently, everyone will have a different **Internal Model** of reality.

Alternatively, we can use the term **Internal Model** to consider other viewpoints: - Within an organisation, we might consider the **Internal Model** of a *team of people* to be the shared knowledge, values and working practices of that team. - Within a software system, we might consider the **Internal Model** of a single processor, and what knowledge it has of the world. - A codebase is a team's **Internal Model** written down and encoded as software.

An internal model *represents* reality: reality is made of atoms, whereas the internal model is information.

Meet Reality

Risk

Attendant Risk

Hidden Risk

Mitigated Risk

Take Action