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.

- I have this pattern
- looking at projects, working out how they fail. Understanding how failure happens, then avoiding it. "It's all about risk" - Ward C.
- it's all on the web.
- it's part 1 of 2.

[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.

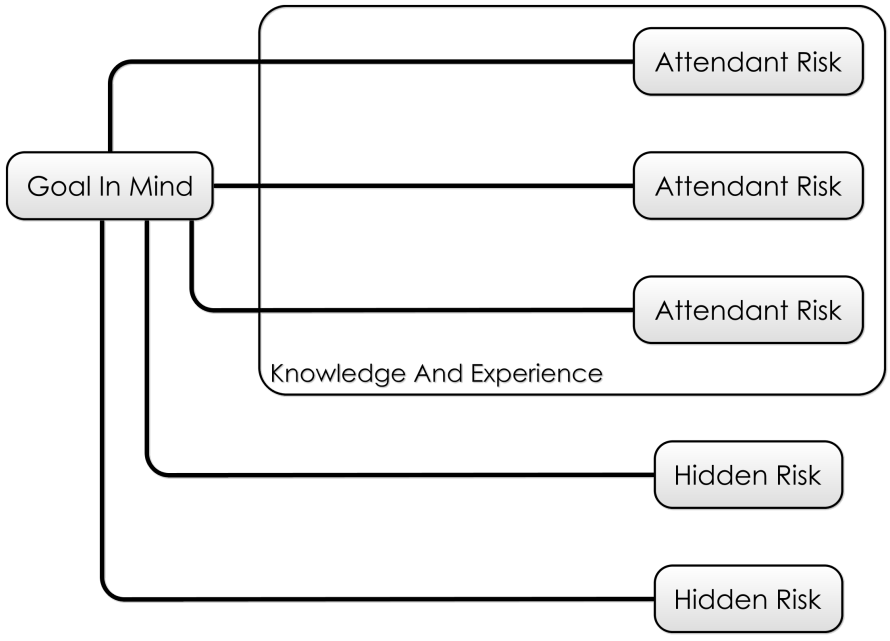


Figure 1.1: Goal In Mind

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

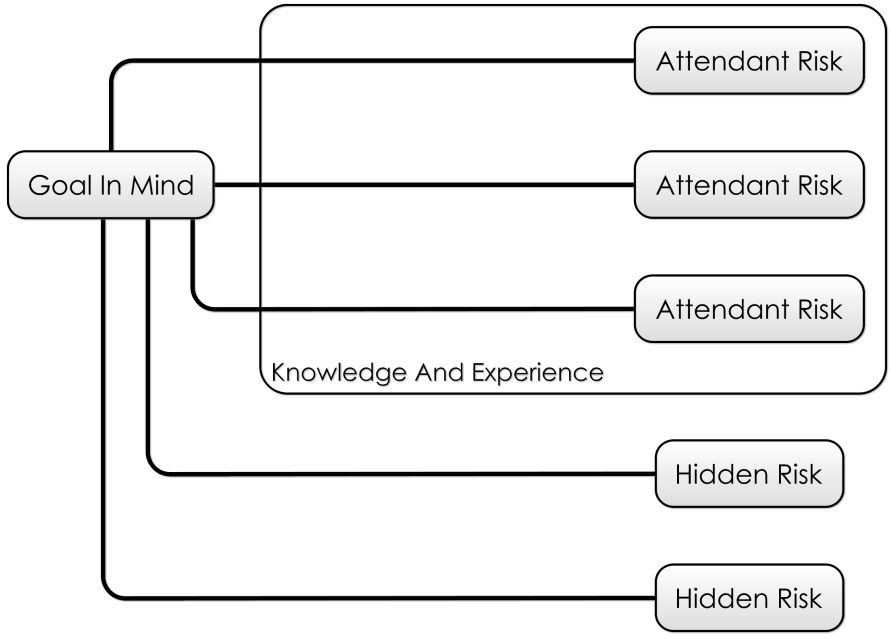


Figure 1.2: Goal In Mind

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.

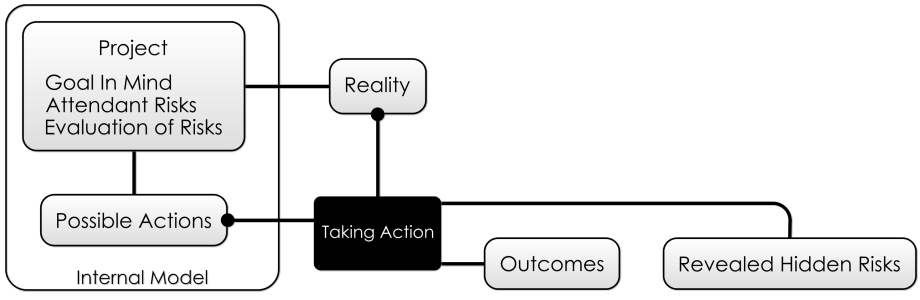


Figure 1.3: Reality

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*.

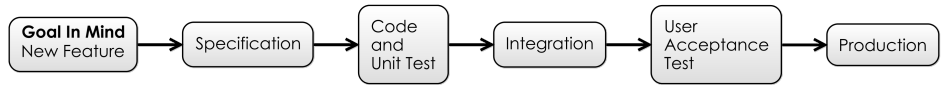


Figure 2.1: Development Process

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.

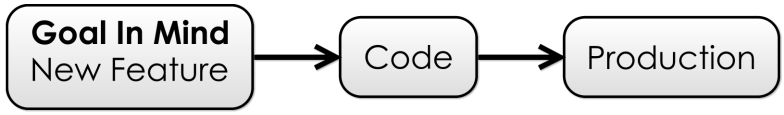


Figure 2.2: Development Process

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-

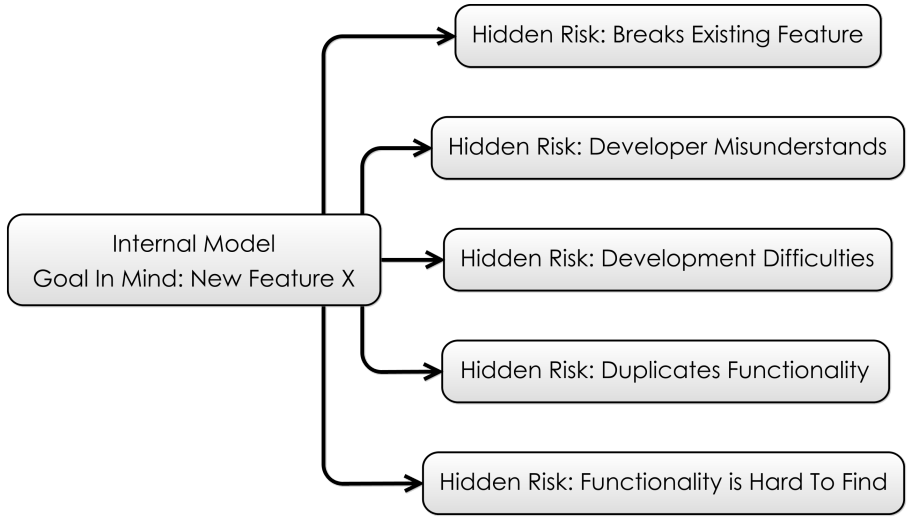


Figure 2.4: Development Process - Hidden Risks

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:

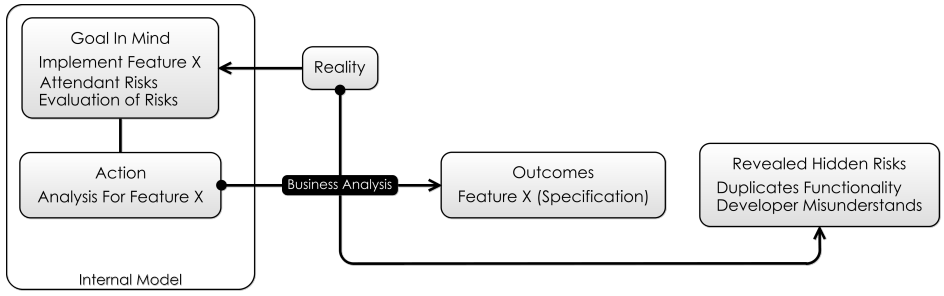


Figure 2.5: BA Specification

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)**.

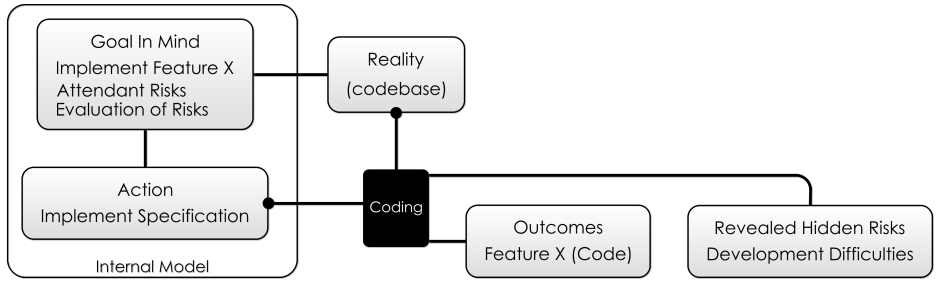


Figure 2.6: Coding Process

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**

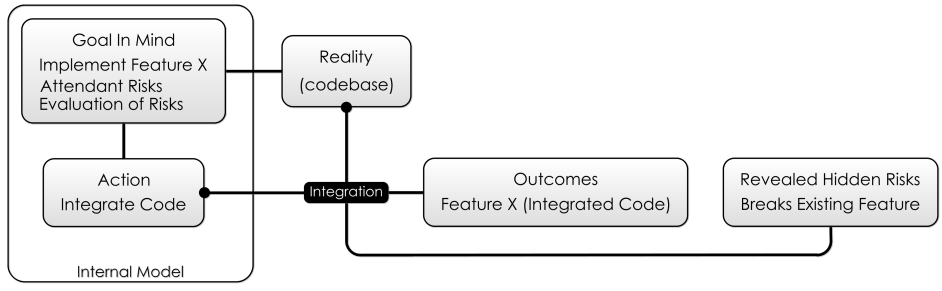


Figure 2.7: Integration

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**:

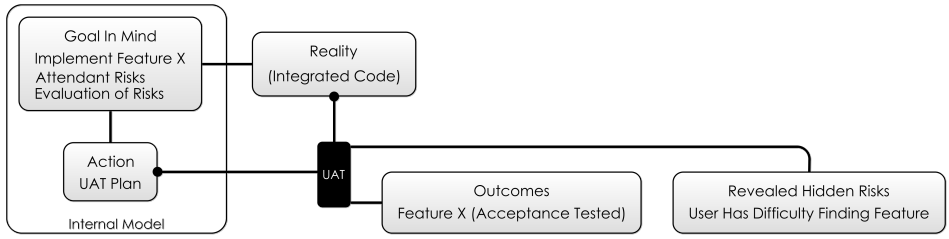


Figure 2.8: UAT

Observations

A couple of things:

First, the people setting up the development process *didn't know* about these *exact* risks, but they knew the *shape that the risks take*. The process builds "nets" for the different kinds of hidden risks without knowing exactly what they are. Part of the purpose of this site is to help with this and try and provide a taxonomy for different types of risks.

Second, are these really risks, or are they *problems we just didn't know about*? I am using the terms interchangeably, to a certain extent. Even when you know you have a problem, it's still a risk to your deadline until it's solved. So, when does a risk become a problem? Is a problem still just a schedule-risk, or cost-risk? It's pretty hard to draw a line and say exactly.

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

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

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*.

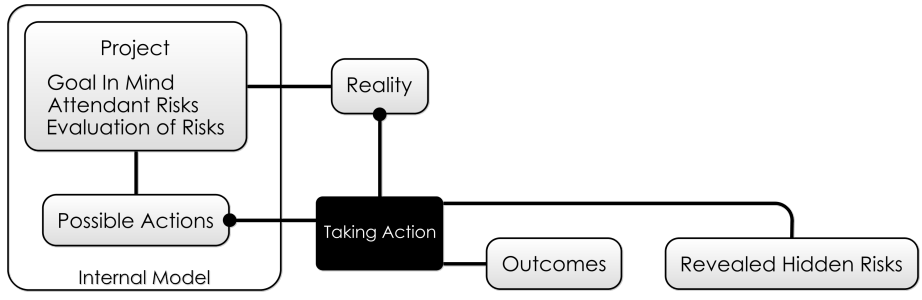


Figure 4.1: Reality

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

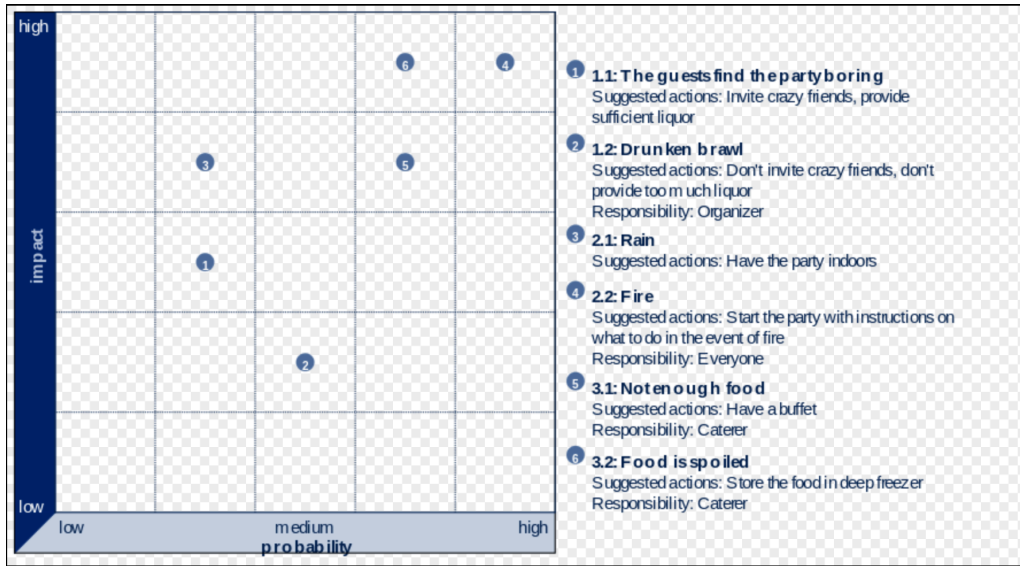


Figure 5.2: Wikipedia Risk Register

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.

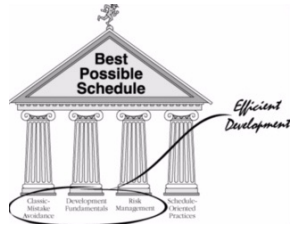


Figure 5.3: Rapid Development Pillars

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.

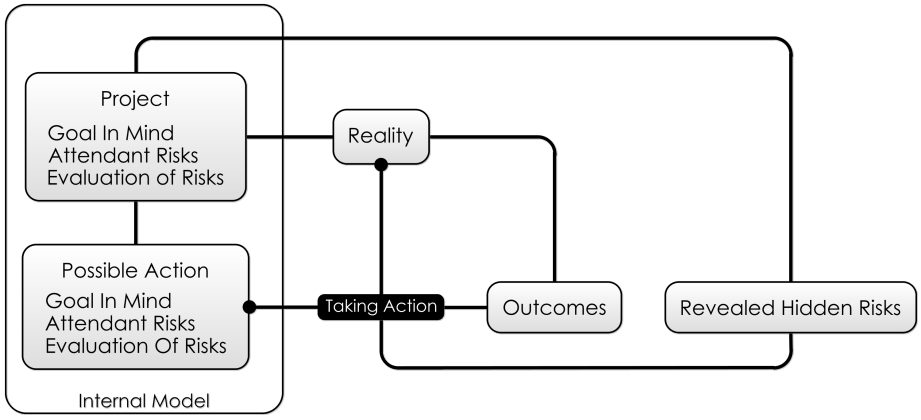
And, in **Development Process** we looked at how we can meet with reality in *different forms*: Analysis, Testing, Integration and so on, and saw how the model could work in each stage of a project.

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**

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

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

Chapter 7

Cadence

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

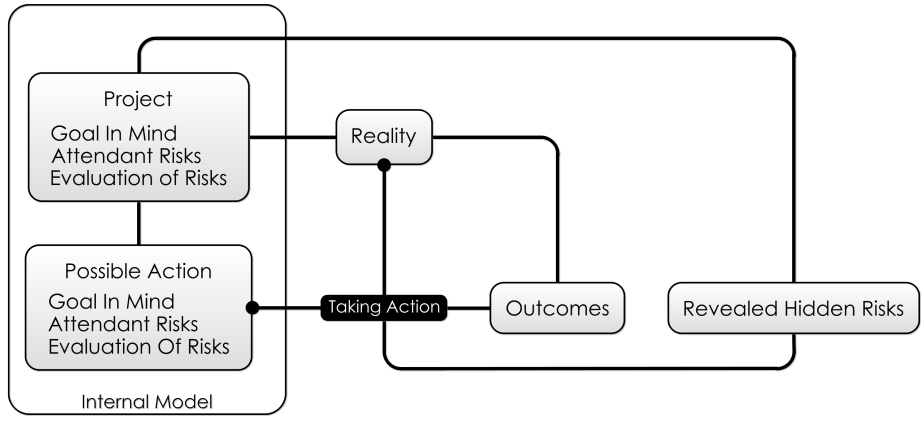


Figure 7.1: Reality 2

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 **Bureaucracy Risk**. I'm also worried that there are too many steps here, and we're going to discover loads of **Hidden Risks** as we go.

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

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

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

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

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

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

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

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

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

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

Risk Landscape

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

The Financial Crisis

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

- Market Risk[2]: the risk that the amount some asset you hold/borrow/have loaned is going to change in value.
- Credit Risk[3]: the risk that someone who owes you a payment at a specific point in time might not pay it back.
- Liquidity Risk[4]: the risk that you can't find a market to sell/buy something, usually leading to a shortage of ready cash.

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

They get expressed in ways like this:

> "we have a 95% chance that today we'll lose less than £100"

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

- Liquidity difficulties (i.e. amount of cash you have for day-to-day running of the bank) caused some banks to not be able to cover their interest payments.
- This caused credit defaults (the thing that Credit Risk[5] measures were meant to guard against) even though the banks *technically* were solvent.

[1] https://en.wikipedia.org/wiki/Financial_services
[2] https://en.wikipedia.org/wiki/Market_risk
[3] https://en.wikipedia.org/wiki/Credit_risk
[4] https://en.wikipedia.org/wiki/Liquidity_risk
[5] https://en.wikipedia.org/wiki/Credit_risk

- That meant that, in time, banks got bailed out, share prices crashed and there was lots of Quantitative Easing[6].
- All of which had massive impacts on the markets in ways that none of the Market Risk[7] models foresaw.

All the **Risks** were correlated[8]. That is, they were affected by the *same underlying events*, or *each other*.

The Risk Landscape Again

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

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

It's tempting to think of our **Risk Landscape** as being like a Fitness Landscape[9]. That is, you have a "cost function" which is your height above the landscape, and you try and optimise by moving downhill in a Gradient Descent[10] fashion.

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

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

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

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

Why Should We Categorize The Risks?

This is a "spotters' guide" to risks, not an in-depth encyclopedia.

[6] https://en.wikipedia.org/wiki/Quantitative_easing
[7] https://en.wikipedia.org/wiki/Market_risk
[8] https://www.investopedia.com/terms/c/correlation.asp
[9] https://en.wikipedia.org/wiki/Fitness_landscape
[10] https://en.wikipedia.org/wiki/Gradient_descent

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

As software developers, we can't hope to know the detailed specifics of the whole discipline of Complexity Theory[11], or Concurrency Theory[12]. But, we're still required to operate in a world where these things exist. So, we may as well get used to them, and ensure that we respect their primacy. We are operating in *their* world, so we need to know the rules.

We're all Naturalists Now

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

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

Our Tour Itinerary

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

Risk	Description
Feature Risk	When you haven't built features the market needs, or they contain bugs, or the market changes underneath you.
Complexity Risk	Your software is so complex it makes it hard to change, understand or run.
Communication Risk	Risks associated with getting messages heard and understood.
Dependency Risk	Risks of depending on other people, products, software, functions, etc. This is a general look at dependencies, before diving into specifics like...
Schedule Risk	Risks associated with having a dependency on time (or money).
Software Dependency Risk	When you choose to depend on a software library, service or function.

[11] https://en.wikipedia.org/wiki/Complexity_theory
[12] https://en.wikipedia.org/wiki/Concurrency_(computer_science)

Risk	Description
Boundary Risk	Risks due to making decisions that limit your choices later on. Sometimes, you go the wrong way on the **Risk Landscape** and it's hard to get back to where you want to be.
Process Risk	When you depend on a business process, or human process to give you something you need.
Agency Risk	Risks that staff have their own **Goals**, which might not align with those of the project or team.
Coordination Risk	Risks due to the fact that systems contain multiple agents, which need to work together.
Map And Territory Risk	Risks due to the fact that people don't see the world as it really is. (After all, they're working off different, imperfect **Internal Models**.)
Operational Risk	Software is embedded in a system containing people, buildings, machines and other services. Operational risk considers this wider picture of risk associated with running a software service or business in the real world.

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

So, let's get started with **Feature Risk**.

Chapter 10

Feature Risk

Feature Risk is the category of risks to do with features that have to be in your software. You could also call it **Functionality Risk**. It is the risk that you face by *not having features that your clients need*.

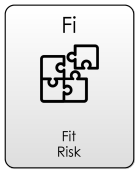
Eventually, this will come down to lost money, business, acclaim, or whatever else reason you are doing your project for.

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

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

Variations

Feature Fit Risk

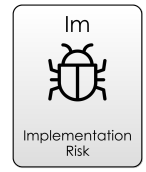


This is the one we've just discussed above: the feature that you (or your clients) want to use in the software *isn't there*. Now, as usual, you could call this an issue, but we're calling it a **Risk** because it's not clear exactly *how many* people are affected, or how badly.

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

Features Don't Work Properly

Feature Risk also includes things that don't work as expected: That is to say, bugs[1]. Although the distinction between "a missing feature" and "a broken feature" might be worth making in the development team, we can consider these both the same kind of risk: *the software doesn't do what the user expects*.



(At this point, it's worth pointing out that sometimes, *the user expects the wrong thing*. This is a different but related risk, which could be down to **Training** or **Documentation** or simply **Poor User Interface** and we'll look at that more in **Communication Risk**.)

Regression Risk

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

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

Market Risk

On the **Risk Landscape** page I introduced the idea of **Market Risk** as being the value that the market places on a particular asset. Since the product you are building is your asset, it makes sense that you'll face **Market Risk** on it:

> "Market risk is the risk of losses in positions arising from movements in market prices." - Market Risk, *Wikipedia*[2]

I face market risk when I own (i.e. have a *position* in) some Apple[3] stock. Apple's[4]'s stock price will decline if a competitor brings out an amazing product, or if fashions change and people don't want their products any more.

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

Conceptual Integrity Risk

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

[1]https://en.wikipedia.org/wiki/Software_bug
[2]https://en.wikipedia.org/wiki/Market_risk
[3]http://apple.com
[4]http://apple.com

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

Which leads to Greenspun's 10th Rule:

> "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp." - Greenspun's 10th Rule, *Wikipedia*[5]

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

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

Feature Access Risk

Sometimes, features can work for some people and not others: this could be down to Accessibility[6] issues, language barriers or localization.

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

Feature Drift Risk

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

The point is: **Requirements captured** *today* might not make it to *tomorrow*, especially in the fast-paced world of IT.

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

Fashion

Fashion plays a big part in IT, as this infographic on website design shows[7]. True, websites have got easier to use as time has gone by, and users now expect this. Also, bandwidth is greater

[5]https://en.wikipedia.org/wiki/Greenspun's_tenth_rule
[6]https://en.wikipedia.org/wiki/Accessibility
[7]https://designers.hubspot.com/blog/the-history-of-web-design-infographic

now, which means we can afford more media and code on the client side. However, *fashion* has a part to play in this.

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

Delight

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

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

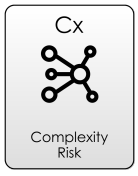
- What are we missing?
- How can we be *even better*?

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

Chapter 11

Complexity Risk

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



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

Kolmogorov Complexity

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

> "...is the length of the shortest computer program (in a predetermined programming language) that produces the object as output." - Kolmogorov Complexity, Wikipedia[2]

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

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

```
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

We might choose this representation:

```
function out() {                                          (7 symbols)
```

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

```
    return "abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd"          (45 symbols)
}                                                              (1 symbol)
```

... which contains **53** symbols, if you count `function`, `out` and `return` as one symbol each.

But, if we write it like this:

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

function out() {                                               (7 symbols)
    return ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD   (21 symbols)
}                                                              (1 symbol)
```

With this version, we now have **40** symbols. And with this version:

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

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

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

Abstraction

What's happening here is that we're *exploiting a pattern*: we noticed that `ABCD` occurs several times, so we defined it a single time and then used it over and over, like a stamp. Separating the *definition* of something from the *use* of something as we've done here is called "abstraction". We're going to come across it over and over again in this part of the book, and not just in terms of computer programs.

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

Trade-Off

But we could go further down into Code Golf[3] territory. This javascript program plays FizzBuzz[4] up to 100, but is less readable than you might hope:

```
for(i=0;i<100;)document.write(((++i%3?'':'Fizz')+
(i%5?'':'Buzz')||i)+"<br>")                                    (66 symbols)
```

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

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

is, the more difficulty users will have understanding it, and the more difficulty developers will have changing it.

Connectivity

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

> "...the minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other" - Connectivity, *Wikipedia*[5]

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

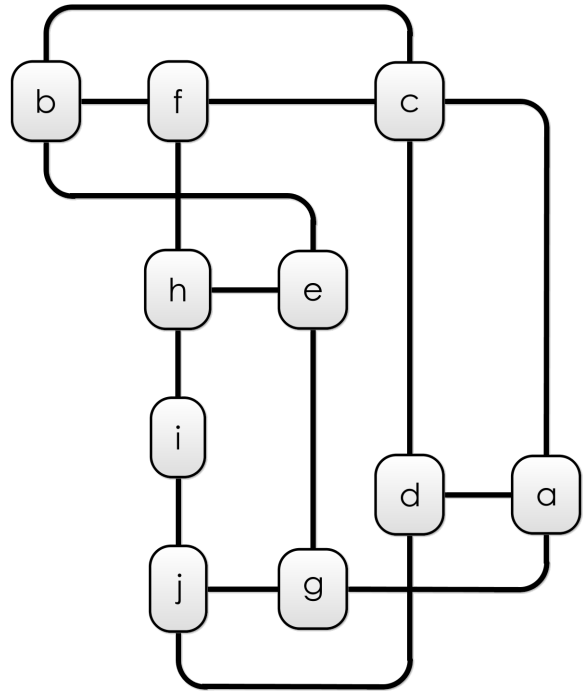


Figure 11.1: Graph 1

It has 10 vertices, labelled **a** to **j**, and it has 15 edges (or links) connecting the vertices together. If any single edge were removed from this diagram, the 10 vertices would still be linked together. Because of this, we can say that the graph is *2-connected*. That is, to disconnect any single vertex, you'd have to remove *at least* two edges.

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

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

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

a: b,c,d
b: c,f,e
c: f,d
d: j
e: h,j
f: h
g: j
h: i
i: j                                                    (39 symbols)
```

Let's remove some of those extra links:

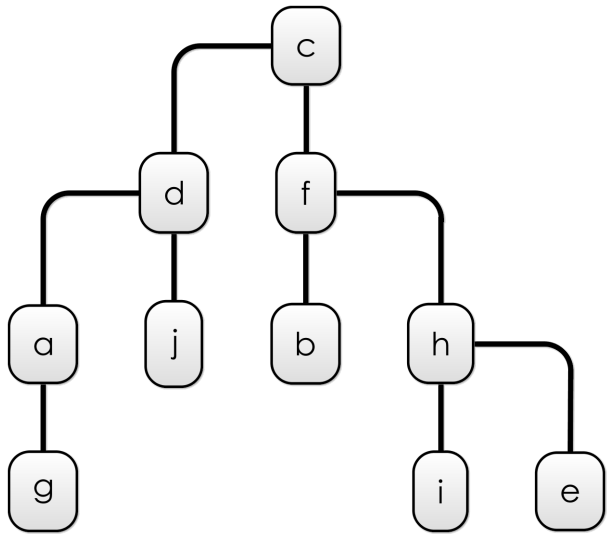


Figure 11.2: Graph 2

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

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

```
a: d,g
```

```
b: f
c: d,f
d: j
f: h
e: h
h: i                                                    (25 symbols)
```

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

Hierarchies and Modularization

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

```
edges = vertices - 1
```

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

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

On the downside, perhaps our messages have farther to go now: in the original **i** could send a message straight to **j**, but now we have to go all the way via **c**. But this is the basis of Modularization[6] and Hierarchy[7].

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

- **Organelles** - such as Mitochondria[8].
- **Cells** - such as blood cells, nerve cells, skin cells in the Human Body[9].
- **Organs** - like hearts livers, brains etc.
- **Organisms** - like you and me.

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

> "Spatial locality refers to the use of data elements within relatively close storage locations." - Locality Of Reference, *Wikipedia*[10]

[6] https://en.wikipedia.org/wiki/Modular_programming
[7] https://en.wikipedia.org/wiki/Hierarchy
[8] https://en.wikipedia.org/wiki/Mitochondrion
[9] https://en.wikipedia.org/wiki/List_of_distinct_cell_types_in_the_adult_human_body
[10] https://en.wikipedia.org/wiki/Locality_of_reference

Cyclomatic Complexity

A variation on this graph connectivity metric is our third measure of complexity, Cyclomatic Complexity[11]. This is:

```
Cyclomatic Complexity = edges – vertices + 2P,
```

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

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

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

More Abstraction

Although we ended up with our second graph having a **Cyclomatic Complexity** of 1 (the minimum), we can go further through abstraction, because this representation isn't minimal from a **Kolmogorov Complexity** point-of-view. For example, we might observe that there are further similarities in the graph that we can "draw out":

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

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

Complexity As Mass

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

Newton's Second Law states:

> "F = *m***a**, (Force = Mass x Acceleration)" - Netwon's Laws Of Motion, *Wikipedia*[12]

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

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

[11] https://en.wikipedia.org/wiki/Cyclomatic_complexity
[12] https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion

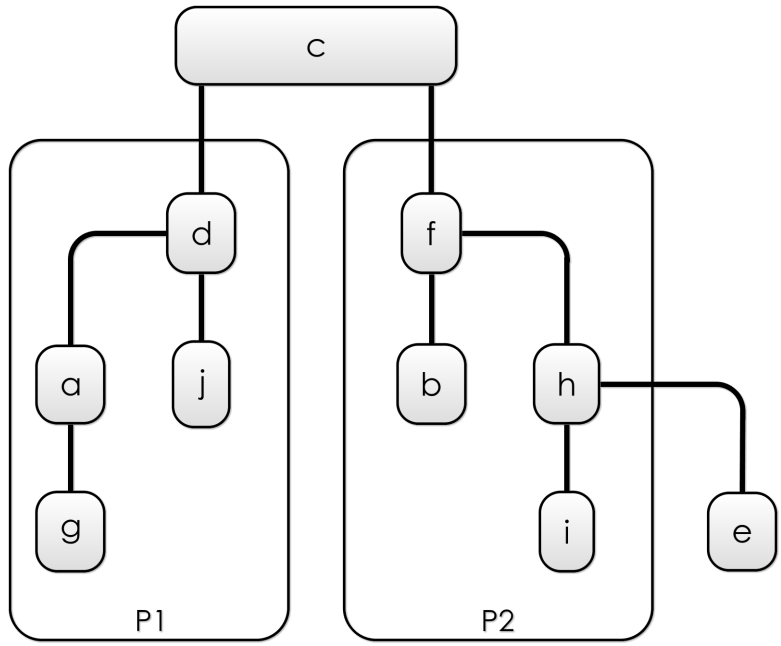


Figure 11.3: Complexity 3

> "mass is the quantitative or numerical measure of a body's inertia, that is of its resistance to being accelerated". - Inertia, *Wikipedia*[13]

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

But there is actually some underlying sense in which *this is real*, as discussed in this Veritasium[14] video. To paraphrase:

> "Most of your mass you owe due to $E=mc^2$, you owe to the fact that your mass is packed with energy, because of the **interactions** between these quarks and gluon fluctuations in the gluon field... what we think of as ordinarily empty space... that turns out to be the thing that gives us most of our mass." - Your Mass is NOT From the Higgs Boson, *Veritasium*[15]

I'm not an expert in physics, *at all*, and so there is every chance that I am pushing this analogy too hard. But, substituting quarks and gluons for pieces of software we can (in a very handwaving-y way) say that more complex software has more **interactions** going on, and therefore has more mass than simple software.

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

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

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

Technical Debt

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

> "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise." – Ward Cunningham, 1992[16]

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

---

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

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

[15] https://www.youtube.com/watch?annotation_id=annotation_3771848421&feature=iv&src_vid=X0232kyTsO0&v=Ztc6QPNUqls

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

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

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

Kitchen Analogy

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

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

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

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

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

Feature Creep

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

- **Essence**: *the difficulties inherent in the nature of the software.*
- **Accident**: *those difficulties that attend its production but are not inherent.*
  - Fred Brooks, *No Silver Bullet*[17]

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

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

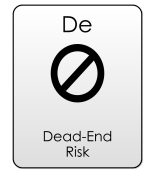
Therefore, Feature Creep[18] (or Gold Plating[19]) is a failure to observe this basic equation: instead of considering this trade off, you're building every feature possible. This has an impact on **Complexity Risk**, which in turn impacts **Communication Risk** and also **Schedule Risk**.

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

[17] https://en.wikipedia.org/wiki/No_Silver_Bullet
[18] https://en.wikipedia.org/wiki/Feature_creep
[19] https://en.wikipedia.org/wiki/Gold_plating_(software_engineering)

"Perfection is Achieved Not When There Is Nothing More to Add, But When There Is Nothing Left to Take Away" - Antoine de Saint-Exupery

Dead-End Risk



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

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

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

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

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

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

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

Source Control

Version Control Systems[20] like Git[21] are a useful mitigation of **Dead-End Risk**, because it means you can *go back* to the point where you made the bad decision and go a different way. Additionally, they provide you with backups against the often inadvertent **Dead-End Risk** of someone wiping the hard-disk.

[20] https://en.wikipedia.org/wiki/Version_control
[21] https://en.wikipedia.org/wiki/Git

The Re-Write

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

> There's a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is the interesting observation: they are probably wrong. The reason that they think the old code is a mess is because of a cardinal, fundamental law of programming: *It's harder to read code than to write it.* - Things You Should Never Do, Part 1, *Joel Spolsky*[22]

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

Where Complexity Hides

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

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

Memory Management

Memory Management is another place where **Complexity Risk** hides:

> "Memory leaks are a common error in programming, especially when using languages that have no built in automatic garbage collection, such as C and C++." - Memory Leak, *Wikipedia*[23]

Garbage Collectors[24] (as found in Javascript or Java) offer you the deal that they will mitigate the **Complexity Risk** of you having to manage your own memory, but in return perhaps give you fewer guarantees about the *performance* of your software. Again, there are times when you can't accommodate this **Operational Risk**, but these are rare and usually only affect a small portion of an entire software-system.

[22] https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
[23] https://en.wikipedia.org/wiki/Memory_leak
[24] https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

Protocols And Types

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

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

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

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

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

Then, we can say that the protocol has changed. This problem is so common, so endemic to computing that we've had compilers that check function arguments since the 1960's[25]. The point being is that it's totally possible for the compiler to warn you about when a protocol within the program has changed.

The same is basically true of Data Types[26]: whenever we change the **Data Type**, we need to correct the usages of that type. Note above, I've given the `javascript` example, but I'm going to switch to `typescript` now:

```
interface BInput {
    a: string,
    b: string,
    c: string,
    d: string
}

function b(in: BInput): string {
    return "whatever" // do something here.
}
```

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

Nevertheless, compilers and type-checking are so prevalent in software that clearly, you have to accept that in most cases, the trade-off has been worth it: Even languages like Clojure[27] have been retro-fitted with type checkers[28].

[25] https://en.wikipedia.org/wiki/Compiler
[26] https://en.wikipedia.org/wiki/Data_type
[27] https://clojure.org
[28] https://github.com/clojure/core.typed/wiki/User-Guide

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

Space and Time Complexity

So far, we've looked at a couple of definitions of complexity in terms of the codebase itself. However, in Computer Science there is a whole branch of complexity theory devoted to how the software *runs*, namely Big O Complexity[29].

Once running, an algorithm or data structure will consume space or runtime dependent on it's characteristics. As with Garbage Collectors[30], these characteristics can introduce **Performance Risk** which can easily catch out the unwary. By and large, using off-the-shelf data structures and algorithms helps, but you still need to know their performance characteristics.

The Big O Cheatsheet[31] is a wonderful resource to investigate this further.

Concurrency / Mutability

Although modern languages include plenty of concurrency primitives, (such as the java.util.concurrent[32] libraries), concurrency is *still* hard to get right.

Race conditions[33] and Deadlocks[34] *thrive* in over-complicated concurrency designs: complexity issues are magnified by concurrency concerns, and are also hard to test and debug.

Recently, languages such as Clojure[35] have introduced persistent collections[36] to alleviate concurrency issues. The basic premise is that any time you want to *change* the contents of a collection, you get given back a *new collection*. So, any collection instance is immutable once created. The tradeoff is again attendant **Performance Risk** to mitigate **Complexity Risk**.

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

Networking / Security

The last area I want to touch on here is networking. There are plenty of **Complexity Risk** perils in *anything* to do with networked code, chief amongst them being error handling and (again) **protocol evolution**.

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

---

[29] https://en.wikipedia.org/wiki/Big_O_notation
[30] https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
[31] http://bigocheatsheet.com
[32] https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html
[33] https://en.wikipedia.org/wiki/Race_condition
[34] https://en.wikipedia.org/wiki/Deadlock
[35] https://clojure.org
[36] https://en.wikipedia.org/wiki/Persistent_data_structure

> "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis." - Bruce Schneier, 1998[37]

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

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

tbd - next section.

costs associated with complexity risk

CHANGE is also more risky why?

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

Chapter 12

Communication Risk

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

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

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

A Model Of Communication

In 1948, Claude Shannon proposed this definition of communication:

> "The fundamental problem of communication is that of reproducing at one point, either exactly or approximately, a message selected at another point." - A Mathematical Theory Of Communication, *Claude Shannon*[1]

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

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

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

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

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

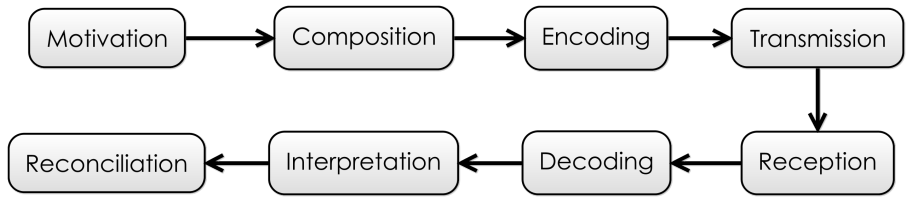


Figure 12.1: Communication Model

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

Approach To Communication Risk

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

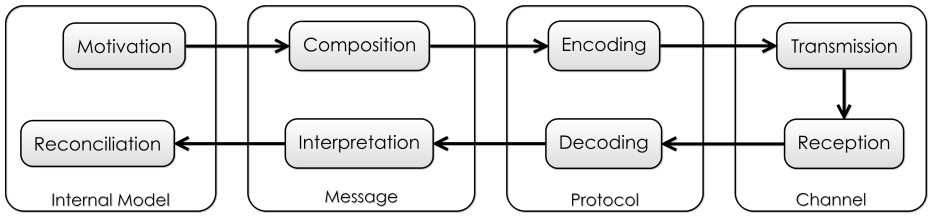


Figure 12.2: Communication Risk 2

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

going to examine four basic concepts in each of these settings:

- Channels[2], the medium via which the communication is happening.
- Protocols[3] - the systems of rules that allow two or more entities of a communications system to transmit information.
- Messages[4]: The information we want to convey.
- **Internal Models**: the sources and destinations for the messages. Updating internal models (whether in our heads or machines) is the reason why we're communicating.

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

Channels

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

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

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

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

Channel Risk

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

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

When channels are **poor-quality**, less communication occurs. People will try to communicate just the most important information. But, it's often impossible to know a-priori what

[2]https://en.wikipedia.org/wiki/Communication_channel
[3]https://en.wikipedia.org/wiki/Communication_protocol
[4]https://en.wikipedia.org/wiki/Message

constitutes "important". This is why **Extreme Programming** recommends the practice of **Pair Programming** and siting all the developers together: although you don't know whether useful communication will happen, you are mitigating **Channel Risk** by ensuring high-quality communication channels are in place.

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

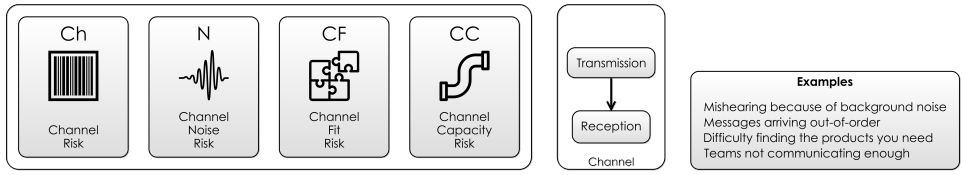


Figure 12.3: Communication Channels

Marketing Communications

When we are talking about a product or a brand, mitigating **Channel Risk** is the domain of Marketing Communications[5]. How do you ensure that the information about your (useful) project makes it to the right people? How do you address the right channels?

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

- The concept that there is such a thing as **D** which solves my problem isn't something I'd even considered.
- I'd like to use something like **D**, but how do I find it?
- There are multiple implementations of **D**, which is the best one for the task?
- I know **D**, but I can't figure out how to solve my problem in it.
- I've chosen **D**, I now need to persuade my team that **D** is the correct solution...
- ... and then they also need to understand **D** to do their job too.

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

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

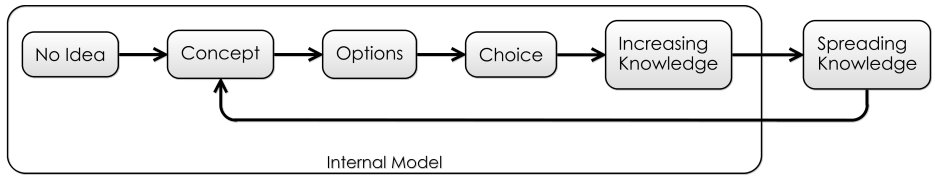


Figure 12.4: Communication Marketing

Protocols

In this section, I want to examine the concept of Communication Protocols[6] and how they relate to **Abstraction**.

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

This is going to involve (at least) six separate protocols, the top-most one being the HTTP Protocol[7]. As far as the HTTP Protocol is concerned, a *client* makes an `HTTP Request` at a specific URL and the `HTTP Response` is returned in a predictable format that the browser can understand.

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

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

1. DNS - Domain Name System

Then, the first thing that happens is this:

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

At this point, curl has used DNS[8] to *resolve* the address "google.com" to an IP address. This is some **Abstraction**: instead of using the machine's IP Address[9] on the network, `216.58.204.78`, I can use a human-readable address, `google.com`. The address `google.com` doesn't necessarily resolve to that same address each time: *They have multiple IP addresses for `google.com`.* But, for the rest of the `curl` request, I'm now set to just use this one.

[6] https://en.wikipedia.org/wiki/Communication_protocol
[7] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[8] https://en.wikipedia.org/wiki/Domain_Name_System
[9] https://en.wikipedia.org/wiki/IP_address

2. IP - Internet Protocol

But this hints at what is beneath the abstraction: although I'm loading a web-page, the communication to the Google server happens by IP Protocol[10] - it's a bunch of discrete "packets" (streams of binary digits). You can think of a packet as being like a real-world parcel or letter.

Each packet consists of two things:

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

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

3. 802.11 - WiFi Protocol

I ran this at home, using WiFi, which uses IEEE 802.11 Protocol[11], which allows my laptop to communicate with the router wirelessly, again using an agreed, standard protocol. But even *this* isn't the bottom, because this is actually probably specifying something like MIMO-OFDM[12], giving specifications about frequencies of microwave radiation, antennas, multiplexing, error-correction codes and so on. And WiFi is just the first hop: after the WiFi receiver, there will be protocols for delivering the packets via the telephony system.

4. TCP - Transmission Control Protocol

Anyway, the next thing that happens is this:

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

The second obvious **Abstraction** going on here is that `curl` now believes it has a TCP[13] connection. The TCP connection abstraction gives us the surety that the packets get delivered in the right order, and retried if they go missing. Effectively it *guarantees* these things, or that it will have a connection failure if it can't keep it's guarantee.

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

[10] https://en.wikipedia.org/wiki/Internet_Protocol
[11] https://en.wikipedia.org/wiki/IEEE_802.11
[12] https://en.wikipedia.org/wiki/MIMO-OFDM
[13] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

5. HTTP - Hypertext Transfer Protocol

Next, we see this:

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

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

- `(1)` says what version of HTTP we are using, and the path we're loading (`/preferences` in this case).

- `(2)` to `(4)` are *headers*. They are name-value pairs, separated with a colon. The HTTP protocol specifies a bunch of these names, and later versions of the protocol might introduce newer ones.

- `(5)` is an empty line, which indicates that we're done with the headers, please give us the response. And it does:

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

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

- The first line is the HTTP Status Code[14]. `301` is a code meaning that the page has moved.
- The next 9 lines are HTTP headers again (name-value pairs). The `Location:` directive tells us where the page has moved to. Instead of trying `http://google.com/preferences`, we should have used `http://www.google.com/preferences`.
- The lines starting `<HTML>` are now some HTML to display on the screen to tell the user that the page has moved.

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

6. HTML - Hypertext Markup Language

Although HTML[15] is a language, a language is also a protocol. (After all, language is what we use to encode our ideas for transmission as speech.) In the example we gave, this was a very simple page telling the client that it's looking in the wrong place. In most browsers, you don't get to see this: the browser will understand the meaning of the `301` error and redirect you to the location.

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

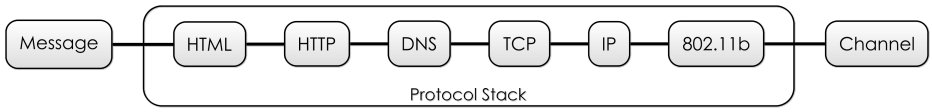


Figure 12.5: Protocol Stack

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

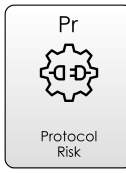
By having a stack of protocols, we are able to apply Separation Of Concerns[16], each protocol handling just a few concerns:

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

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

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

Protocol Risk

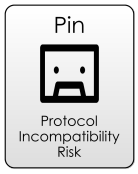


Generally, any time where you have different parts of a system communicating with each other, and one part can change incompatibly with another you have **Protocol Risk**.

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

Let's look at some types of **Protocol Risk**:

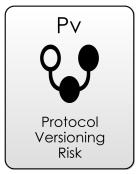
Protocol Incompatibility Risk



The people you find it *easiest* to communicate with are your friends and family, those closest to you. That's because you're all familiar with the same protocols. Someone from a foreign country, speaking a different language and having a different culture, will essentially have a completely incompatible protocol for spoken communication to you.

Within software, there are also competing, incompatible protocols for the same things, which is maddening when your protocol isn't supported. Although the world seems to be standardizing, there used to be *hundreds* of different image formats. Photographs often use TIFF[17], RAW[18] or JPEG[19], whilst we also have SVG[20] for vector graphics, GIF[21] for images and animations and PNG[22] for other bitmap graphics.

Protocol Versioning Risk



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

Protocol Complexity

tbd.

[15] https://en.wikipedia.org/wiki/HTML
[16] https://en.wikipedia.org/wiki/Separation_of_concerns
[17] https://en.wikipedia.org/wiki/TIFF
[18] https://en.wikipedia.org/wiki/Raw_image_format
[19] https://en.wikipedia.org/wiki/JPEG
[20] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[21] https://en.wikipedia.org/wiki/GIF
[22] https://en.wikipedia.org/wiki/Portable_Network_Graphics

Backward Compatibility

Backwards Compatibility mitigates **Protocol Versioning Risk**. Quite simply, this means, supporting the old format until it falls out of use. If a server is pushing for a change in protocol it either must ensure that it is Backwards Compatible with the clients it is communicating with, or make sure they are upgraded concurrently. When building web services[23], for example, it's common practice to version all APIs so that you can manage the migration. Something like this:

- Server publishes `/api/v1/something`.
- Clients use `/api/v1/something`.
- Server publishes `/api/v2/something`.
- Clients start using `/api/v2/something`.
- Clients (eventually) stop using `/api/v2/something`.
- Server retires `/api/v2/something` API.

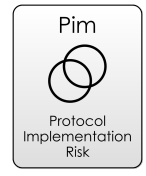
Forward Compatibility

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

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

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

Protocol Implementation Risk



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

Messages

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

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

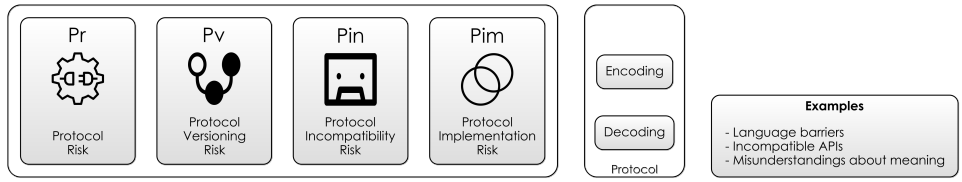
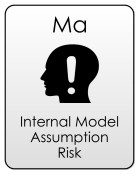


Figure 12.6: Communication Protocols Risks

Internal Model Assumption Risk



When we construct messages in a conversation, we have to make judgements about what the other person already knows. When talking to children, it's often hard work because they *assume* that you have knowledge of everything they do. This is called Theory Of Mind[24]: the appreciation that your knowledge is different to other people's, and adjusting you messages accordingly.

When teaching, this is called The Curse Of Knowledge[25]: teachers have difficulty understanding students' problems *because they already understand the subject*. For example, if I want to tell you about a new JDBC Driver[26], this pre-assumes that you know what JDBC is: the message has a dependency on prior knowledge.

Message Dependency Risk

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

This happens when messages get missed, or delivered out of order. In the past, TV shows were only aired once a week at a particular time. So writers were constrained plot-wise by not knowing whether their audience would have seen the previous week's episode. Therefore, often the state of the show would "reset" week-to-week, allowing you to watch it in *any* order.

The same **Message Dependency Risk** exists for computer software: if there is replication going on between instances of an application, and one of the instances misses some messages, you end up with a "Split Brain[27]" scenario, where later messages can't be processed because they refer to an application state that doesn't exist. For example, a message saying:

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

only makes sense if the application has previously had the message

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

[24] https://en.wikipedia.org/wiki/Theory_of_mind
[25] https://en.wikipedia.org/wiki/Curse_of_knowledge
[26] https://en.wikipedia.org/wiki/JDBC_driver
[27] https://en.wikipedia.org/wiki/Split-brain_(computing)

Misinterpretation Risk

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

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

> "The famous pipe. How people reproached me for it! And yet, could you stuff my pipe? No, it's just a representation, is it not? So if I had written on my picture "This is a pipe", I'd have been lying!" - Rene Magritte, of *The Treachery of Images*[28]

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

Invisibility Risk

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

Invisibility Risk In Software

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

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

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

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

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

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

Invisibility Risk In Conversation

Invisibility Risk is risk due to information not sent. But because humans don't need a complete understanding of a concept to use it, we can cope with some **Invisibility Risk** in communication, and this saves us time when we're talking. It would be *painful* to have conversations if, say, the other person needed to understand everything about how cars worked in order to discuss cars.

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

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

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

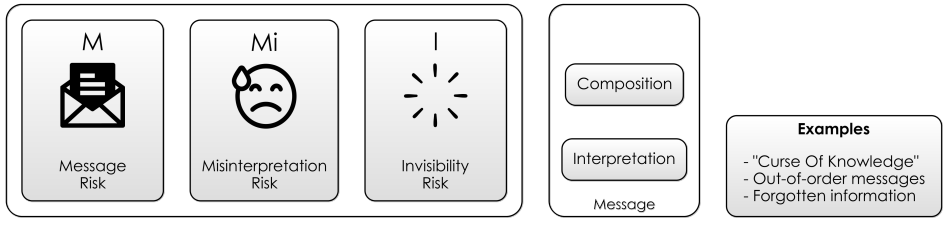


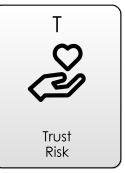
Figure 12.7: Message Risk

Internal Models

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

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

Trust Risk & Belief Risk



Although protocols can sometimes handle security features of communication (such as Authentication[29] and preventing man-in-the-middle attacks[30]), trust goes further than this, intersecting with **Agency Risk**: can you be sure that the other party in the communication is acting in your best interests?

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

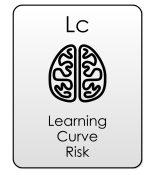
- Weltanschauung (World View)[31]: The ethics, values and beliefs in the receiver's **Internal Model** may be incompatible to those from the sender.
- Relativism[32] is the concept that there are no universal truths. Every truth is from a frame of reference. For example, what constitutes *offensive language* is dependent on the listener.
- Psycholinguistics[33] is the study of humans aquire languages. There are different languages and dialects, (and *industry dialects*), and we all understand language in different ways, take different meanings and apply different contexts to the messages.

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

Reputational Risk

tbd.

Learning-Curve Risk



If the messages we are receiving force us to update our **Internal Model** too much, we can suffer from the problem of "too steep a Learning Curve[34]" or "Information Overload[35]", where the messages force us to adapt our **Internal Model** too quickly for our brains to keep up.

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

Reading Code

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

> "If you ask a software developer what they spend their time doing, they'll tell you that they spend most of their time writing code. However, if you actually observe

[29] https://en.wikipedia.org/wiki/Authentication
[30] https://en.wikipedia.org/wiki/Man-in-the-middle_attack
[31] https://en.wikipedia.org/wiki/World_view
[32] https://en.wikipedia.org/wiki/Relativism
[33] https://en.wikipedia.org/wiki/Psycholinguistics
[34] https://en.wikipedia.org/wiki/Learning_curve
[35] https://en.wikipedia.org/wiki/Information_overload

what software developers spend their time doing, you'll find that they spend most of their time trying to understand code." - When Understanding Means Rewriting, *Coding Horror*[36]

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

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

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

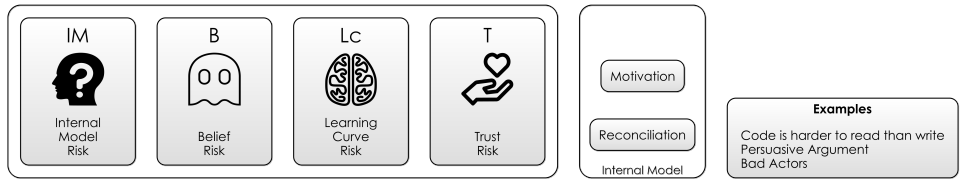


Figure 12.8: Internal Model Risks

Communication Risk Wrap Up

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

Since the purpose of Communication is to *coordinate our actions*, next it's time to look at **Coordination Risk**.

this seems complex tbd.

[36]https://blog.codinghorror.com/when-understanding-means-rewriting/
[37]https://en.wikipedia.org/wiki/Principle_of_least_astonishment

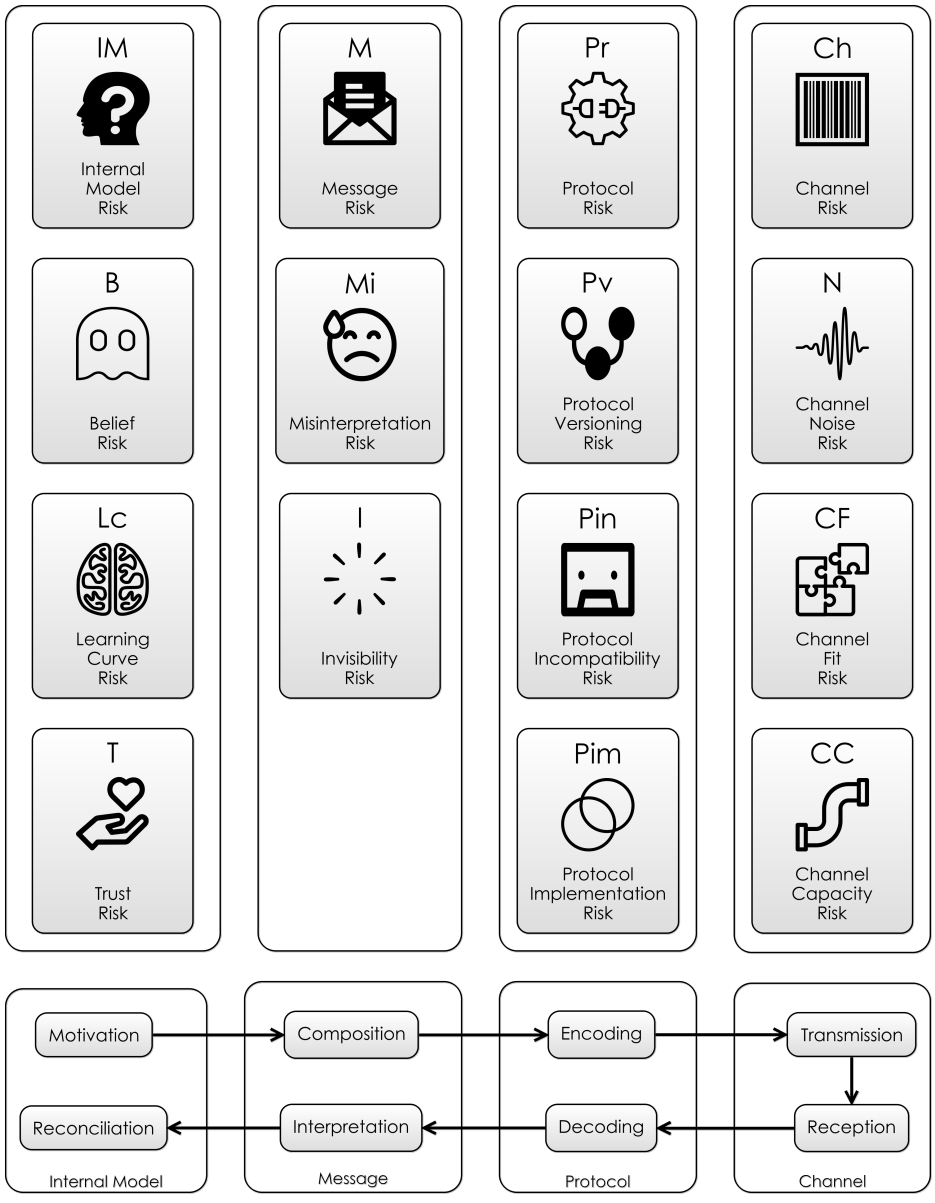


Figure 12.9: Communication 2

Chapter 13

Dependency Risk

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

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

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

- In this first section will look at dependencies *in general*, and specifically on *events*, and some of the variations on **Dependency Risk**.
- Next, we'll look at **Schedule Risk**, because time and money are key dependencies in any project.
- Then, we'll move on to look specifically at **Software Dependency Risk**, covering using libraries, software services and building on top of the work of others.
- Next, we'll take a closer look at **Boundary Risk** and **Dead-End Risk**. These are the risks you face in choosing the wrong things to depend on.
- After, we'll take a look at **Process Risk**, which is still **Dependency Risk**, but we'll be considering more organisational factors and how bureaucracy comes into the picture.
- Finally, we'll wrap up this analysis with a look at some of the specific problems around working with other people or businesses in **Agency Risk**.

Why Have Dependencies?

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

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

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

This tells us a lot about **Dependency Risk** right here:

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

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

Simple Made Easy

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

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

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

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

Event Dependencies

Let's start with dependencies on *events*.

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

[1] https://en.wikipedia.org/wiki/Hydrothermal_vent
[2] https://www.infoq.com/presentations/Simple-Made-Easy
[3] https://en.wikipedia.org/wiki/Division_of_labour

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

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

Events Mitigate Risk...

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

- By taking the bus, we are mitigating our own **Schedule Risk**: we're (hopefully) reducing the amount of time we're going to spend on the activity of getting to work.
- Events are a mitigation for **Coordination Risk**: A bus needn't necessarily *have* a fixed timetable: it could wait for each passenger until they turned up, and then go. (A bit like ride-sharing works). This would be a total disaster from a **Coordination Risk** perspective, as one person could cause everyone else to be really really late. Having a fixed time for doing something mitigates **Coordination Risk** by turning it into **Schedule Risk**. Agreeing a date for a product launch, for example, allows lots of teams to coordinate their activities.
- It's not entirely necessary to even take the bus: you could walk, or go by another form of transport. But, effectively, this just swaps one dependency for another: if you walk, this might well take longer and use more energy, so you're just picking up **Schedule Risk** in another way. If you drive, you have a dependency on your car instead. So, there is often an *opportunity cost* with dependencies. Using the bus might be a cheap way to travel. You're therefore imposing less **Dependency Risk** on a different scarce resource - your money.

But, Events Lead To Attendant Risk

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

Figure 13.1: Action Diagram showing risks mitigated by having an *event*

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

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

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

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

Let's look at each of these in turn.

Schedule Risk

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

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

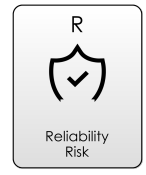
Each party can mitigate **Deadline Risk** with *slack*. That is, ensuring that the exact time of the event isn't critical to your plans:

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

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

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

Reliability Risk



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

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

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

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

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

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

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

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

Scarcity Risk

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

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

- **Buffers**: Smoothing out peaks and troughs in utilisation.
- **Reservation Systems**: giving clients information *ahead* of the dependency usage about whether the resource will be available to them.
- **Graceful degradation**: Ensuring *some* service in the event of over-subscription. It would be no use allowing people to cram onto the bus until it can't move.
- **Demand Management**: Having different prices during busy periods helps to reduce demand. Having "first class" seats means that higher-paying clients can get service even when the train is full. Uber[9] adjust prices in real-time by so-called Surge Pricing[10]. This is basically turning **Scarcity Risk** into a **Market Risk** problem.
- **Queues**: Again, these provide a "fair" way of dealing with scarcity by exposing some mechanism for prioritising use of the resource. Buses operate a first-come-first-served system, whereas emergency departments in hospitals triage according to need.
- **Pools**: Reserving parts of a resource for particular customers.
- **Horizontal Scaling**: allowing a scarce resource to flexibly scale according to how much demand there is. (For example, putting on extra buses when the trains are on strike, or opening extra check-outs at the supermarket.)

Much like **Reliability Risk**, there is science for it:

- Queue Theory[11] is all about building mathematical models of buffers, queues, pools and so forth.
- Logistics[12] is the practical organisation of the flows of materials and goods around things

[5] https://en.wikipedia.org/wiki/Reliability_engineering
[6] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis
[7] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis
[8] https://en.wikipedia.org/wiki/Ford_Pinto#Design_flaws_and_ensuing_lawsuits
[9] https://www.uber.com
[10] https://www.uber.com/en-GB/drive/partner-app/how-surge-works/
[11] https://en.wikipedia.org/wiki/Queueing_theory
[12] https://en.wikipedia.org/wiki/Logistics

like Supply Chains[13].

- And Project Management[14] is in large part about ensuring the right resources are avaiable at the right times. We'll be taking a closer look at that in the Part 3 sections on **Prioritisation** and the **Project Managment Body Of Knowledge**.

Communication Risk

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

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

Let's apply this to our Bus scenario:

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

Silo Mentality

Finding out about bus schedules is easy. But in a large company, **Communication Risk** and especially **Invisibility Risk** are huge problems. This tends to get called "Silo Mentality[15]", that is, ignoring what else is going on in other divisions of the company or "not invented here"[16] syndrome:

> "In management the term silo mentality often refers to information silos in organizations. Silo mentality is caused by divergent goals of different organizational units." - Silo Mentality, *Wikipedia*[17]

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

Silo Mentality is exacerbated by the problems you would face in *budgeting* if suddenly all the divisions in an organisation started providing dependencies for each other. This starts to require

[13]https://en.wikipedia.org/wiki/Supply_chain
[14]https://en.wikipedia.org/wiki/Project_management
[15]https://en.wikipedia.org/wiki/Information_silo#Silo_mentality
[16]https://en.wikipedia.org/wiki/Not_invented_here
[17]https://en.wikipedia.org/wiki/Information_silo#Silo_mentality

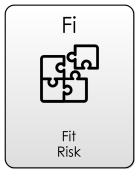
a change to organisational structure towards being a set of individual businesses marketing services to one another, rather than a division-based one. We'll look more closely at these kind of organisational issues in the **Coordination Risk** section.

Complexity Risk

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

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

Fit Risk



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

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

Dead-End Risk and Boundary Risk

When you choose something to depend on, you can't be certain that it's going to work out in your favour. Sometimes, the path from your starting point to your goal on the **Risk Landscape** will take you to dead ends: places where the only way towards your destination is to lose something, and do it again another way. This is **Dead End Risk**, which we looked at before.

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

Figure 13.2: Feature Fit: A Two-Dimensional Problem (at least)

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

- If I choose to program some software in Java, I will find it hard to integrate libraries from other languages. The dependencies available to Java software are different to those in Javascript, or C#. Having gone down a Java route, there are *higher risks* associated with choosing incompatible technologies. Yes, I can pick dependencies that use C# (still), but I am now facing greater complexity risk than if I'd just chosen to do everything in C# in the first place.
- If I choose one database over another, I am *limited to the features of that database*. This is not the same as a dead-end: I can probably build what I want to build, but the solution will be "bounded" by the dependency choices I make. One of the reasons we have standards like Java Database Connectivity (JDBC)[18] is to mitigate **Dead End Risk** around databases, so that we can move to a different database later.
- If I choose to buy a bus ticket, I've made a decision not to travel by train, even though later on it might turn out that the train was a better option. Buying the bus ticket is **Boundary Risk**: I may be able to get a refund, but having chosen the dependency I've set down a path on the risk landscape.

Managing Dependency Risk

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

There are *some* tools for managing dependency risk: Gantt Charts[19] for example, arrange work according to the capacity of the resources (i.e. dependencies) available, but also the *dependen-*

[18]https://en.wikipedia.org/wiki/Java_Database_Connectivity
[19]https://en.wikipedia.org/wiki/Gantt_chart

cies between the tasks. If task **B** requires the outputs of task **A**, then clearly task **A** comes first and task **B** starts after it finishes. We'll look at this more in **Process Risk**.

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

Chapter 14

Schedule Risk

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

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

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

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

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

Opportunity Risk

Opportunity Risk is really the concern that whatever we do, we have to do it *in time*. If we wait too long, we'll miss the Window Of Opportunity[1] for our product or service.

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

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

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

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

The iPad

For example, let's look at the iPad[2], which was introduced in 2010 and was hugely successful.

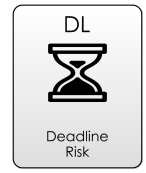
This was not the first tablet computer. Apple had already tried to introduce the Newton[3] in 1989, and Microsoft had released the Tablet PC[4] in 1999. But somehow, they both missed the Window Of Opportunity[5]. Possibly, the window existed because Apple had changed changed the market with their release of the iPhone, which left people open to the idea of a tablet being "just a bigger iPhone".

But maybe now, the iPad's window is closing? We have more *wearable computers* like the Apple Watch[6], and voice-controlled devices like Alexa[7] or Siri[8]. Peak iPad was in 2014, according to this graph[9].

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

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

Deadline Risk



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

What happens if you miss the deadline? It could be: - The funding on the project runs out, and it gets cancelled. - You have to go back to a budgeting committee, and get more money. - The team gets replaced, because of lack of faith.

.. or something else.

Deadlines can be set by an authority in order to *sharpen focus* and reduce **Coordination Risk**. This is how we arrive at tools like SMART Objectives[10] and KPI's (Key Performance Indicators)[11]. Time scales change the way we evaluate goals, and the solutions we choose.

In JFK's quote:

> "First, I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the Earth." - John F. Kennedy, 1961

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

[2] https://en.wikipedia.org/wiki/History_of_tablet_computers
[3] https://en.wikipedia.org/wiki/Apple_Newton
[4] https://en.wikipedia.org/wiki/Microsoft_Tablet_PC
[5] https://en.wikipedia.org/wiki/Window_of_opportunity
[6] https://en.wikipedia.org/wiki/Apple_Watch
[7] https://en.wikipedia.org/wiki/Amazon_Alexa
[8] https://en.wikipedia.org/wiki/Siri
[9] https://www.statista.com/statistics/269915/global-apple-ipad-sales-since-q3-2010/
[10] https://en.wikipedia.org/wiki/SMART_criteria
[11] https://en.wikipedia.org/wiki/Performance_indicator

Compare with this quote:

> "I love deadlines. I love the whooshing noise they make as they go by." - Douglas Adams[12]

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

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

Student Syndrome

Student Syndrome[13] is, according to Wikipedia:

> "Student syndrome refers to planned procrastination, when, for example, a student will only start to apply themselves to an assignment at the last possible moment before its deadline." - *Wikipedia*[14]

Arguably, there is good psychological, evolutionary and risk-based reasoning behind procrastination: the further in the future the **Deadline Risk** is, the more we discount it. If we're only ever mitigating our *biggest risks*, then deadlines in the future don't matter so much, do they? And, putting efforts into mitigating future risks that *might not arise* is wasted effort.

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

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

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

Funding Risk

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

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

[12] https://en.wikipedia.org/wiki/Douglas_Adams
[13] https://en.wikipedia.org/wiki/Student_syndrome
[14] https://en.wikipedia.org/wiki/Student_syndrome

In startup circles, this "amount of time you can afford it" is called the "Runway"[15]: you have to get the product to "take-off" before the runway ends. So you could term this component as "Runway Risk".

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

Staff Risk

If a startup has a "Runway", then the chances are that the founders and staff do too, as this article explores[16]. It identifies the following risks:

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

You need to consider how long your staff are going to be around, especially if you have Key Man Risk[17] on some of them. People like to have new challenges, or move on to live in new places, or simply get bored. The longer your project goes on for, the more **Staff Risk** you will have to endure, and you can't rely on getting the **best staff for failing projects**.

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

Red-Queen Risk

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

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

> "My dear, here we must run as fast as we can, just to stay in place. And if you wish to go anywhere you must run twice as fast as that." - Lewis Carroll, *Alice in Wonderland*[18]

The problem with software projects is that tools and techniques change *really fast*. In 2011, 3DRealms released Duke Nukem Forever after 15 years in development[19], to negative reviews:

[15]https://en.wiktionary.org/wiki/runway
[16]https://www.entrepreneur.com/article/223135
[17]https://en.wikipedia.org/wiki/Key_person_insurance#Key_person_definition
[18]https://www.goodreads.com/quotes/458856-my-dear-here-we-must-run-as-fast-as-we
[19]https://en.wikipedia.org/wiki/Duke_Nukem_Forever

> "... most of the criticism directed towards the game's long loading times, clunky controls, offensive humor, and overall aging and dated design." - *Duke Nukem Forever, Wikipedia*[20]

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

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

Schedule Risk and Feature Risk

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

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

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

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

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

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

Part III

Glossary

Chapter 15

Glossary

Abstraction

Feedback Loop

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