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

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].

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

Part I

Introduction

Part II

Risk

Chapter 1

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.

Financial Crisis

In the Financial Services[1] industry, lots of effort is spend calculating 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.

They get expressed in ways like this:

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

In the financial crisis, though, these models of risk didn't turn out to be much use. Although there are lots of conflicting explanations of what happened, one way to look at it is this: - Liquidity difficulties (i.e. amount of cash you have for day-to-day running of the bank) caused some banks to not be able to cover their interest payments. - This caused credit defaults (the thing that **Credit Risk** measures were meant to guard against) even though the banks *technically* were solvent. - That meant that, in time, banks got bailed out, share prices crashed and there was lots of Quantitative Easing[4].
- All of which had massive impacts on the markets in ways that none of the **Market Risk** models foresaw.

All the **Risks** were correlated[5]. That is, they were affected by the *same underlying events*, or *each other*.

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

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*.

In this section, I am going to try and show you some of the geography 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*.

Three Basic Areas Of Risk

tbd; is this enough?

Figure 1.1: Risk Types

In general, you will definitely have at least 3 main **areas** of risk:

- **Product Risks**: Risks affecting the *product you're building*, such as **Feature Risk** and **Dependency Risk**
- **Staff Risks**: Risks to do with the people or organisations *building the product*, such as **Coordination Risk** and **Agency Risk**
- **Customer Risks**: Risks to do with the *consumers* of the product.

None of the risk categories we're going to look at fit *exactly* into these areas, and some of them exist at the **intersection** of these types: - **Feature Risk** is about the **Customer** and **Product** fit. - **Complexity Risk** is a problem between the **Staff** and the **Product** they are building. - **Communication Risk** occurs at the intersection of **Customer**, **Product** and **Staff**.

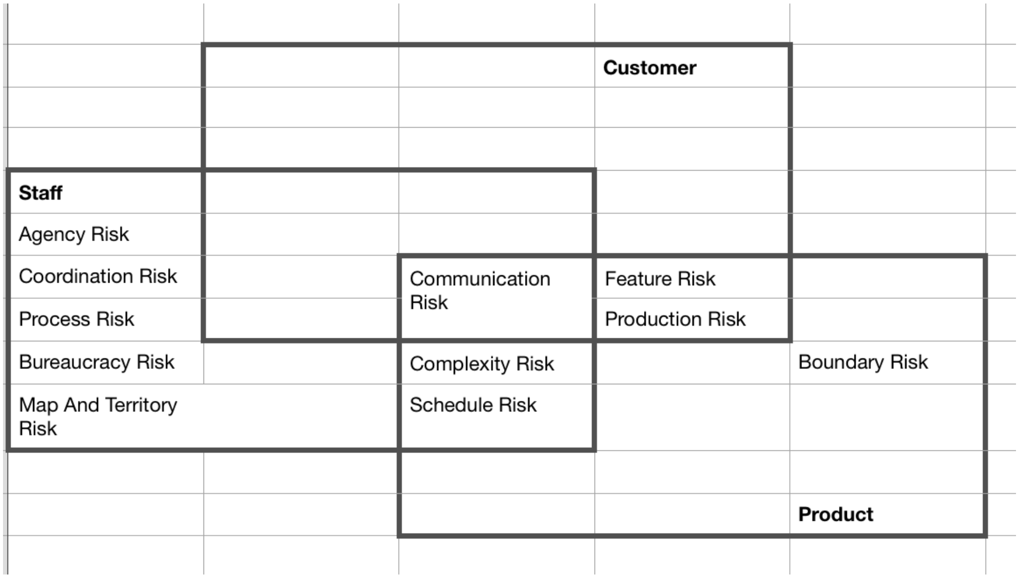


Figure 1.2: Risk Types 2

Our Tour Itinerary

tbd

Risk	Areas
Feature Risk	Customer, Product
Complexity Risk	Product, Staff
Communication Risk	Customer, Product, Staff
Dependency Risk	Product, Customer, Staff
Software Dependency Risk	Product, Staff
Process Risk	Staff
Schedule Risk	Product, Staff
Boundary Risk	Product
Agency Risk	Staff
Coordination Risk	Staff
Production Risk	Customer, Product
Map And Territory Risk	Staff

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.

Let's get started with **Feature Risk**.

Chapter 2

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*.

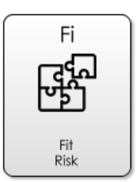
Eventually, this will come down to lost money, business, acclaim, or whatever else reason you are doing your project for.

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:

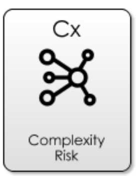
- What are we missing?
- How can we be *even better*?

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 3

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:

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.

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:

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)
```

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

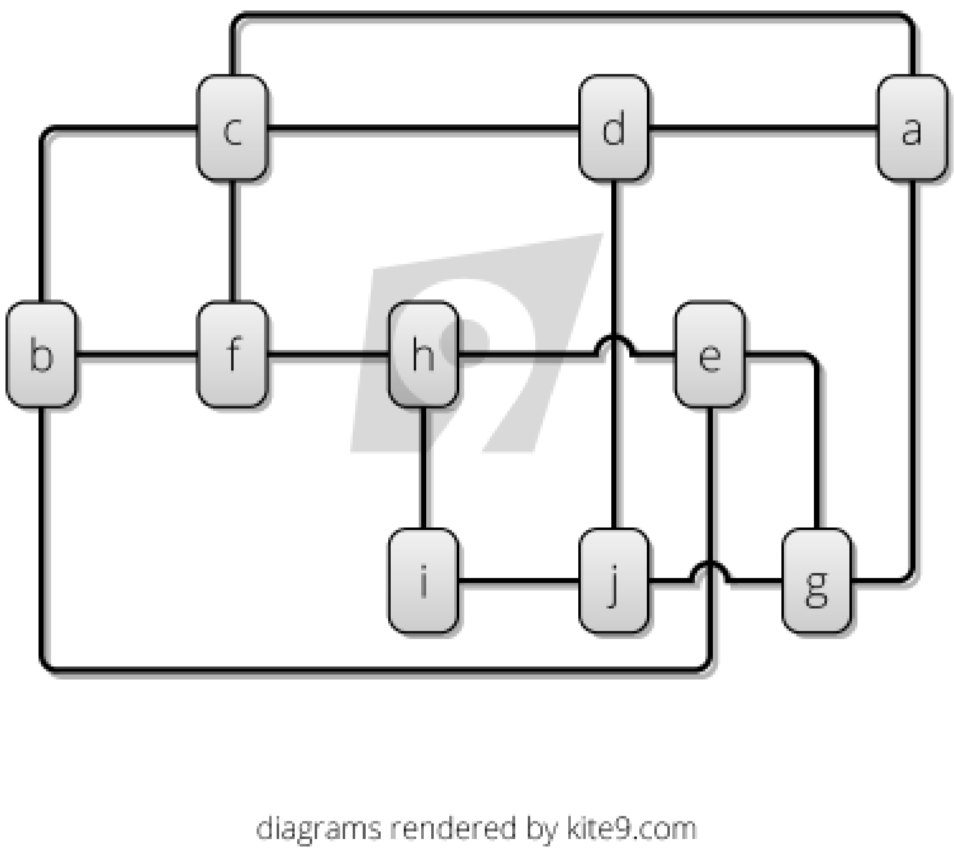


Figure 3.1: Graph 1

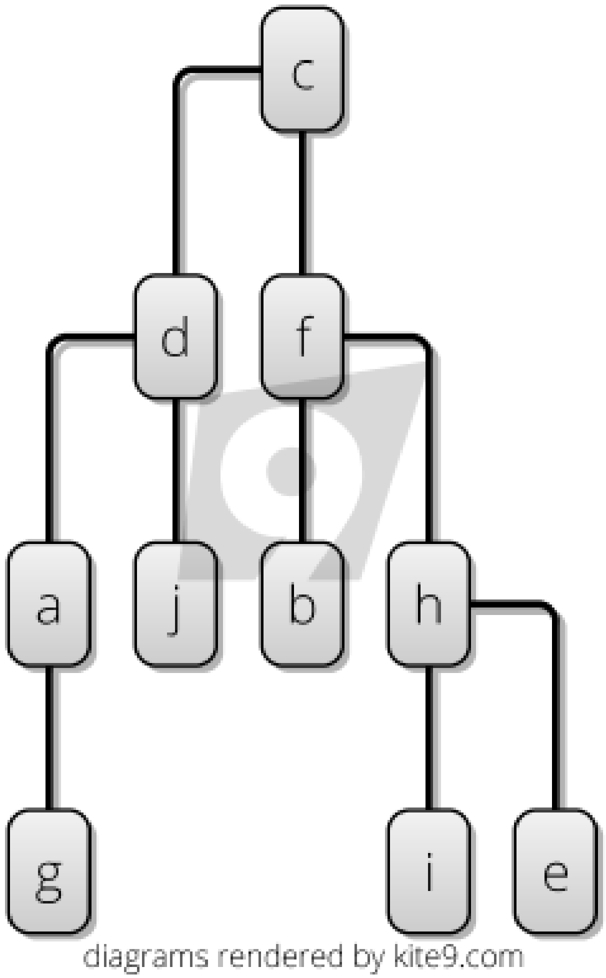


Figure 3.2: Graph 2

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**.

Complexity As Mass

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**.

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

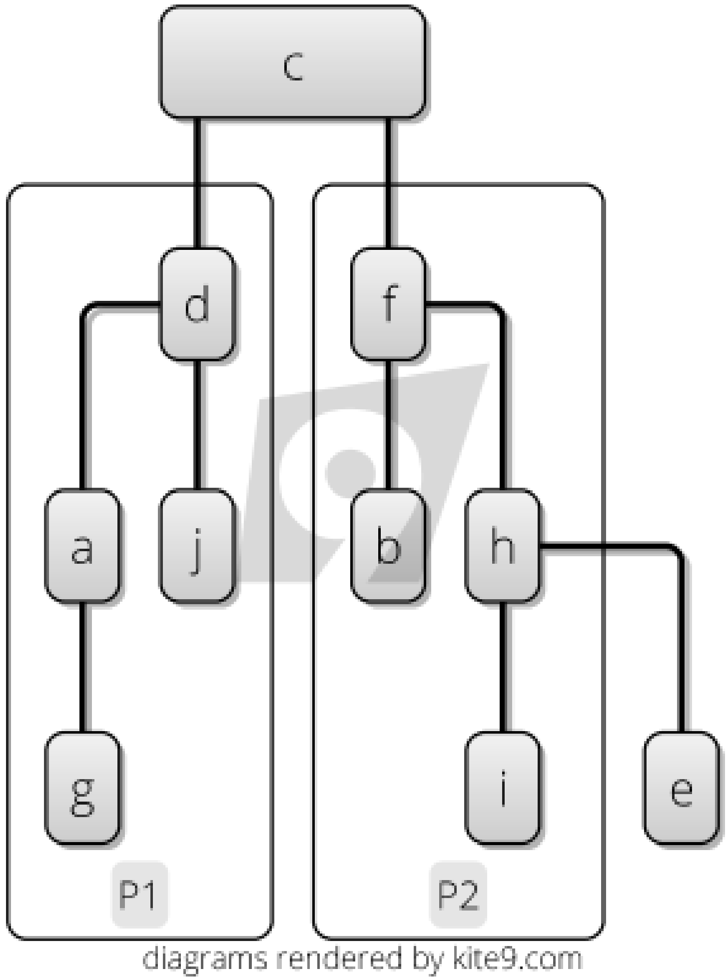


Figure 3.3: Complexity 3

> "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=Xo232kyTsOo&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 **Performance 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.

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

Chapter 4

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.

Communication 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

- **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.

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).

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

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 time-zone to someone else on your team, there is *automatic* **Channel Risk**, because instantaneous communication is only available for a few hours' a day.

When channels are **poor-quality**, less communication occurs. People will try to communicate just the most important information. But, it's often impossible to know a-priori what constitutes "important". This is why **Extreme Programming** recommends the practice of **Pair Programming** 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.

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.

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

- 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.

Communication Marketing

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

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

[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

resolve to that same address each time: *They have multiple IP addresses for `google.com`*. But, for the rest of the `curl` request, I'm now set to just use this one.

2. IP - Internet Protocol

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

Each packet consists of two things:

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

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

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

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:

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**, 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

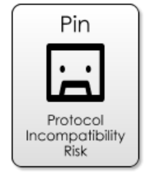
Generally, any time where you have different parts of a system communicating with each other, and one part can change incompatibly with another you have **Protocol Risk**.

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**:

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

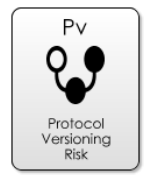
Protocol Incompatibility Risk



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[16], RAW[17] or JPEG[18], whilst we also have SVG[19] for vector graphics, GIF[20] for images and animations and PNG[21] 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**.

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**, 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

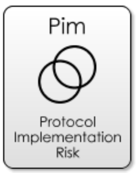
[16] https://en.wikipedia.org/wiki/TIFF
[17] https://en.wikipedia.org/wiki/Raw_image_format
[18] https://en.wikipedia.org/wiki/JPEG
[19] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[20] https://en.wikipedia.org/wiki/GIF
[21] https://en.wikipedia.org/wiki/Portable_Network_Graphics

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.

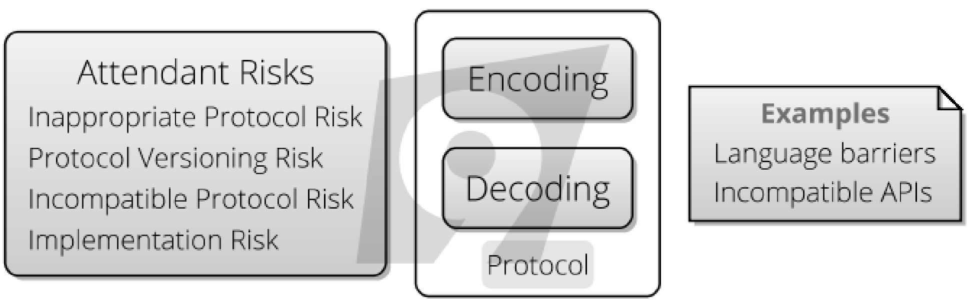


Figure 4.1: Communication Protocols Risks

Messages

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

Internal Model Assumption Risk

When we construct messages in a conversation, we have to make judgements about what the other person already knows. When talking to children, it's often hard work because they *assume* that you have knowledge of everything they do. This is called Theory Of Mind[22]: the appreciation that your knowledge is different to other people's, and adjusting you messages accordingly.

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

When teaching, this is called The Curse Of Knowledge[23]: 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[24], 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[25]" 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'
```

Abstraction and 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*[26]

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.

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

Abstraction and 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?

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?

Message Risk

Internal Models

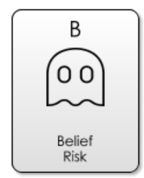
So finally, we are coming to the root of the problem: communication is about transferring ideas and concepts from one **Internal Model** to another.

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[27] and preventing man-in-the-middle attacks[28]), 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?

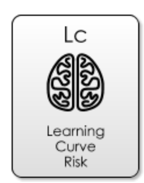
Even if the receiver trusts the communicator, they may not trust the message. Let's look at some reasons for that:



- Weltanschauung (World View)[29]: The ethics, values and beliefs in the receiver's **Internal Model** may be incompatible to those from the sender.
- Relativism[30] 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[31] 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**.

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[32]" or "Information Overload[33]", where

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

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 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*[34]

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[35])*
- 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.

Internal Model Risks

Communication Risk Wrap Up

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

Communication 2

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

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

Chapter 5

Coordination Risk

Coordination Risk is the risk that, despite a group of people (or processes) having the same **Goal In Mind** they can fail to coordinate on a way to meet this goal and end up making things worse. **Coordination Risk** is embodied in the phrase "Too Many Cooks Spoil The Broth": more people, opinions or actors often make results worse.

In this section, we're going to work on the assumption that everyone has a common **Goal**, but in reality, people often have their own agendas. We'll come to that in the section on **Agency Risk** later.

More Communication Risk?

You might think that this is just another type of **Communication Risk** problem, and that's often a part of it, but even with synchronized **Internal Models**, coordination risk can occur. Imagine the example of people all trying to madly leave a burning building. They all have the same information (the building is on fire). If they coordinate, and leave in an orderly fashion, they might all get out. If they don't, and there's a scramble for the door, more people might die.

Alternatively, even with a cluster of stateless server processes, some coordination is required to decide which server processes which request.

But generally, **Coordination Risk** occurs most commonly where people have different ideas about how to achieve a goal, and they have different ideas because they have different evaluations of the **Attendant Risk**. As we saw in the section on **Communication Risk**, we can only hope to synchronize **Internal Models** if there are high-bandwidth **Channels** available for communication.

Decision Making

So **Coordination Risk** is worse on projects with more members, and worse in organizations with more staff. If you are engaged in a solo project, do you suffer from **Coordination Risk** at all? Maybe: sometimes, you can feel "conflicted" about the best way to solve a problem. And weirdly, usually *not thinking about it* helps. Sleeping too. (Rich Hickey calls this "**Hammock Based Development**"). This is probably because, unbeknownst to you, your subconscious is furiously communicating internally, trying to resolve these conflicts itself, and will let you know when it's come to a resolution.

So, **Coordination Risk** is at it's core about resolving **Internal Model** conflicts, and arriving at consensus.

Vroom and Yetton introduced a model of group decision making which looks something like this:

!image tbd

On the left, you have the *least* consultative styles, and on the right, the *most*. On the left, decisions are made with just the leader's **Internal Model** but moving right, the **Internal Models** of the rest of the team are increasingly brought into play.

The decisions on the left are faster, but don't do much for mitigating **Coordination Risk**. The ones on the right take longer, (incurring **Schedule Risk**) but mitigate more **Coordination Risk**. Group decision-making inevitably involves everyone *learning*, and improving their **Internal Models**.

The trick is to be able to tell which approach is suitable at which time. Everyone is expected to make decisions *within their realm of expertise*: you can't have developers continually calling meetings to discuss whether they should be using an **Abstract Factory** or a [Factory Method], this would waste time. The critical question is therefore, "what's the biggest risk?" - Is the **Coordination Risk** greater? Are we going to suffer **Dead End Risk** if the decision is made wrongly? What if people don't agree with it? Poor leadership has an impact on **Morale** too. - Is the **Schedule Risk** greater? If you have a 1-hour meeting with eight people to decide a decision, that's *one man day* gone right there: group decision making is *expensive*.

With Processes

Almost the same model can be used with software processes.

tbd: use bitcoin as an example here.

Specialization / Abstraction Risk

One common way groups and larger organizations aim to mitigate **Coordination Risk** is via **Abstraction**: teams and organizations can be arranged along functional lines, with *interfaces* between their different parts. This means the different functions can *use* each other without *understanding* each other.

On a team level, this might mean that you have one developer doing "UI", another working on "billing" and so on. In a larger organisation you might have a "marketing" team or "accounts" team, or divisions by product.

As we saw before **Abstraction brings it's own risks**. A key one being that if team members are specialized, you can end up with "bottlenecks" in an organisation (see **Critical Chain**). This is covered in more detail in the **Dependency Risk** section.

Bottlenecks in one part of a team mean that other members will be under-utilised. This is the trade-off between **Fungibility** (people are jack-of-all-trades) and **Specialization** (people understand one small area well). Specialism pays off except in highly dynamic situations, where it becomes necessary for people to re-skill, with attendant **Learning Curve Risk**. But software is *often* highly dynamic: **Extreme Programming** avoids specialization with it's insistence on **Pair Programming**, for example.

Specialization is a type of complexity too: a homogeneous team of people presents fewer **Scheduling** problems, and

Another advantage to specialization is that people have domains of responsibility, which makes the **Decision Making** approach easier to choose. Individuals and teams generally know when a decision can't be made at their level, and that they need to escalate.

Staff Risk

If **Coordination Risk** is about trying to mitigate differences in **Internal Models**, then it's worth considering how varied people's models can be: - Different skill levels - Different experiences - Expertise in different areas - Preferences - Personalities

The job of harmonzing this on a project would seem to fall to the team leader, but actually people are self-organising to some extent. This process is called **Team Development**, after **Tuckman**, and can be encouraged with orthogonal practices such as **Team Building exercises** (generally, submitting everyone to extreme experiences in order to bond them together).

With enough communication bandwidth and entente, a team motivated will self-organise code reviews, information exchange and improve their practices. But **Staff Risks** sometimes cannot be resolved without escalation:

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

… and so on.

Experiments showed that rather than t

Resource Coordination Risk

split brain (left hand right hand)

People change their minds when they have evidence of new information, and quickly forget what they *previously thought* about things.

geographic risk

large organisation risks?

Chapter 6

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 *anything*. 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**. - 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. - Then, 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.
- Then, we'll look at some of the specific problems around working with other people or businesses in **Agency Risk**. - Finally, we'll wrap up this analysis with a closer look at **Boundary Risk** and Dead-End Risk. - A short wrap-up tbd.

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 thermal vents (tbd), 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 ou 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 allows tbd.

Event Dependencies

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

Arguably, the event dependencies are the simplest to express, too: usually, a *time* and a *place*. For example, "I can't start shopping until the supermarket opens at 9am", or "I must catch my bus to work at 7:30am". In the first example, you can't *start* something until a particular event happens. In the latter example, you must *be ready* for an event at a particular time.

Events Mitigate Risk…

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

- By taking the bus, we are mitigating our own **Schedule Risk**: we're (hopefully) reducing the amount of time we're going to spend on the activity of getting to work.

- 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. Events are a mitigation for **Coordination Risk**. 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**. However, as we saw in **A Simple Scenario**, this means we pick up **Attendant Risks**. Although you might be able to think of a few more, in this section, we're going to look at the following ones:

So, we're going to look at Dependency Risk for our toy events 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**
- Fit Risk and Baggage
- Dead-End Risk and **Boundary Risk**

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, miss 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 anyway, I've already paid for it.

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**. - A [Single Point Of Failure] in a system - Are there bugs in **B** that are going to prevent it working correctly in all circumstances?

Reliability of an overall system is constrained by the reliability. How does re

Is it a known unknown? You know you might be going the wrong way.

(This might sound unlikely, but I've made several career decisions off the back of this)

Dependency and reliability Pinto https://en.wikipedia.org/wiki/Reliability_engineering FECMA FEMA https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis Diagram of a distributed software system - where can failures hide? SPOFs.

Scarcity Risk

Let's get back to the bus: 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 tbd. - **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] adjust prices in real-time by so-called [Surge Pricing]. - **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. For exampl - **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.)

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. Recall from the discussion on **Marketing Communications** which started like this:

- 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?

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 Thinking", that is, ignoring what else is going on in other parts of the company or "not invented here" syndrome. These are all varieties of communication risk.

Ironically, *more communication* might not be the answer - if channels are provided where

Silo thinking

Lots of beginners don't see dependency risk: they have the dependencies round the wrong way in their heads. eg. sunny with jenkins and environment variables. You can't solve a problem if you are working down the line of dependencies but your issue is with an earlier one

dependency ijection - invisibile dependencies

Dependencies Are Abstractions

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 page on Feature Risk. There, we considered two problems: - The feature (or now, dependency) doesn't provide all the functinality 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.

diagram tbd

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](Complexity-Risk#Dead-End-Risk], which we looked at before.

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

We're also going to look at [Boundary Risk] in more detail later, but I want to introduce it here. Here are some examples: - If I choose to program some software in Java, I will find it hard to integrate libraries from other languages. The dependencies available to Java software are different to those in Javascript, or C#. Having gone down a Java route, there are *higher risks* associated with choosing incompatible technologies. Yes, I can pick dependencies that use C# (still), but I am now facing greater complexity risk than if I'd just chosen to do everything in C# in the first place. - If I choose one database over another, I am *limited to the features of that database*. This is not the same as a dead-end: I can probably build what I want to build, but the solution will be "bounded" by the dependency choices I make. One of the reasons we have standards like [JDBC] is to mitigate [Dead End Risk] around databases, so that we can move to a different datbase 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] for example, arrange work according to the capacity of the resources (i.e. dependencies) available, but also the *dependencies between the tasks*. If task B requires the outputs of task A, then clearly task A comes first and task B starts after it finishes. We'll look at this more in **Process Risk**.

We'll look in more detail in the *practices* part, later. But now let's get into the specifics with **Software Dependency Risk**.

Chapter 7

Process Risk

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

But first, what exactly is a process?

tbd. definition

Elaboration

In the software development world, and the business world generally, processes usually involve *forms*. If you're filling out a form (whether on paper or on a computer) then you're involved in a process of some sort, whether an "Account Registration" process, "Loan Application" process or "Consumer Satisfaction Survey" process.

Later in this section we'll look at

The Purpose Of Process

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

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

- **Coordination Risk**: You can often use process to help people coordinate. For example, a production Line is a process where work being done by one person is pushed to the next person when it's done. A meeting booking process ensures that people will all attend a meeting together at the same place and time, and that a room is available for it.

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

Evolution Of Business Process

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

tbd

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

tbd image.

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

tbd

2. Either, B co-opts other people to help, gets given a team (T), or someone else forms a team T containing B to get the job done "properly". T has control of A (it might be a resource, some source of complexity, whatever). However, it needs to supply the company with A reliably and responsibly, otherwise there will be problems. They can't simply sit on the resource and do nothing, but at the same time, so they try and please all of their clients as far as possible. This is a good deal for their clients, but they end up absorbing a lot of risk.

tbd

3. T organises bureaucratically, so that there is a controlled process (P) by which A can get done. Like a cell, they have arranged a protective barrier around themselves, the strength of which depends on the power conferred to them by control of A. P probably involves filling in a form (or following some other **Protocol**). They can now deal with

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

tbd

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

tbd

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

tbd

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

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

> In this conception, you can see how the structure of an organisation (the teams and processes within it, the heirarchy of control) will 'evolve' from the resources of the organisation and the strategy it pursues. Processes evolve to meet the needs of the organisation,

- [MInzberg, strategy safari]

In each step, the actors involved have been acting in good faith: they are working to mitigate risk in the organisation. The **Process Risk** that accretes along the way is an *unintended consequence*: There is no guarantee that the process that arises will be humane and intuitive. Many organisational processes end up being baroque or Kafkaesque, forcing unintuitive contortions on their users. Dealing with complex processes is a **Communication Risk** because you have to translate your requirements into the language of the process.

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

tbd - parkinson's law

An Example - Release Process

For many years I have worked in the Finance Industry, and it's given me time to observe how, across an entire industry, process can evolve, both in response to regulatory pressure but also because of organisational maturity, and mitigating risks:

1. Initially, I could release software by logging onto the production accounts with a password that everyone knew, and deploy software or change data in the database.

2. The first issue with this is bad actors. How could you know that the numbers weren't being altered in the databases? Production auditing came in so that at least you could tell *who was changing what*, in order to point the blame later.
3. But, there was still plenty of scope for deliberate or accidental damage. I personally managed to wipe production data on one occasion by mistaking it for a development environment. Eventually, passwords were taken out of the hands of developers and you needed approval to "break glass" to get onto production.

4. Change Requests were introduced. This is another approval process which asks you to describe what you want to change in production, and why you want to change it. In most places, this was quite an onerous process, so the unintended consequence was that release cadence was reduced.
5. The change request software is generally awful, making the job of raising change requests tedious and time-consuming. Therefore, developers would *automate* the processes for release, sometimes including the process to write the change request. This allowed them to improve release cadence, at the expense of owning more code.
6. Auditors didn't like the fact that this automation existed, because effectively, that meant that developers could get access to production with the press of a button, effectively taking you back to step 1. So auditing of Change Requests had to happen.

… and so on. tbd.

Process Risks

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

- **Reliability Risk**: If *people* are part of the process, there's the chance that they forget to follow the process itself, and miss steps or forget your request completely. The reliability is related to the amount of Complexity Risk the process is absorbing.
- [Visibility Risk]: Usually, processes (like other dependencies) trade Complexity Risk for visibility: it's often not possible to see how far along a process is to completion. Sometimes, you can do this to an extent. For example, when I send a package for delivery, I can see roughly how far it's got on the tracking website. But, this is still less-than-complete information, and is a representation of reality.

- Process Fit Risk/ **Dead-End Risk**: You have to be careful to match the process to the outcome you want. Much like choosing a **Software Dependency**, initiating a process has no guarantee that your efforts won't be wasted and you'll be back where you started from. The chances of this happening increase as you get further from the standard use-case for the process, and the sunk cost increases with the length of time the process takes to report back.
- **Agency Risk**: Due to Parkinson's Law, above.

Operational Risk

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

tbd - Wikipedia definition

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

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

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

Counterparty Risk

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

> tbd.

Money is *changing hands* between you and the supplier of the process, and often, the money doesn't transfer at the same time as the process is performed. Let's look at an example: Instead of hosting my website on a server in my office, I could choose to host my software project with an online provider. I am trading Complexity Risk for Counterparty Risk, because now, I have to care that the supplier is solvent. There's a couple of ways this could go wrong: They may *take my payment*, but then turn off my account. Or, they could go bankrupt, and leave me with the costs of moving to another provider (this is also Dead-End Risk).

Mechanisms like *insurance* and *guarantees* help reduce this risk:

Feedback Loops

Operational Risk is usually incurred for outliers: processes tend to work well for the common cases, because *practice makes perfect*. Processes are really tested when unusual situations occur. Having mechanisms to deal with edge-cases can incur Complexity RiskComplexity-Risk), so often it's better to try and have clear boundaries of what is "in" and "out" of the process' domain.

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

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

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

Sign-Offs

Often, Processes will include sign-off steps. The Sign-Off is an interesting mechanism: by signing off on something for the business, people are usually in some part staking their reputation on something being right. Therefore, you would expect that sign-off involves a lot of **Agency Risk**: people don't want to expose themselves in career-limiting ways. Therefore, the bigger the risk they are being asked to swallow, the more cumbersome and protracted the sign off process. Often, Sign Offs boil down to a balance of risk for the signer: on the one hand, personal risk from signing off, on the other, the risk of upsetting the rest of the staff waiting for the sign-off, and the [Dead End Risk] of all the effort gone into getting the sign off if they don't.

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

Dependencies - A Quick Review

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

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

What does the risk look like?	Software Dependency	Process	Event	Person
Dependency Risk[1]	Software Dependency Risk[2]	Process Risk[3]	Deadline Risk[4]	[Key-Man Risk][5]
———————— —-	——————————— ——-	————————— —–	————————— —-	——————————— ——-
Risks Mitigated	[Complexity Risk][6] from having to do the process yourself	[Complexity Risk][6]	[Coordination Risk][7]	[Resource Coordination Risk][8]

What does the risk look like?	Software Dependency	Process	Event	Person
Communication Risk	Understanding the API, using it	Filling in forms wrongly.	Communicating the right place and time to everyone	- Being misunderstood - Instructions not followed
[Invisibility Risk]	Understanding the API, but not the implementation. Debugging is harder.	You don’t know how far along the work might be	- Will people show up? - Will it be what you expect?	
Feature Risk	The product might not work how you expected. If features are missing you might be stuck.	Might not cater to your exact use case	Is this: -What you want -When you want it?	Do the available staff have the skills you need?
[Dead End Risk]				

1
2
3
4

Chapter 8

Schedule Risk

Schedule Risk is the fundamental risk 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: If you were completely immortal, 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 deadlines. 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, and voice-controlled devices like Alexa, Cortana and (cough) Siri. Peak iPad was in 2014, according to this graph[6].

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**.

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[7] and KPI's (Key Performance Indicators)[8]. 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. Compare with this quote:

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

[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://www.statista.com/statistics/269915/global-apple-ipad-sales-since-q3-2010/
[7] https://en.wikipedia.org/wiki/SMART_criteria
[8] https://en.wikipedia.org/wiki/Performance_indicator

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.

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.

In startup circles, this "amount of time you can afford it" is called the "runway": 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 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 / Turnover Risk

If a startup has a **Runway**, then the chances are that the founders and staff do too, as this article explores[9]. 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** on some of them. You also can't rely on getting the **best staff for failing projects**.

Student Syndrome

Student Syndrome[10] is, according to Wikipedia:

[9] https://www.entrepreneur.com/article/223135
[10] https://en.wikipedia.org/wiki/Student_syndrome

> "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*[11]

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 **Prioritisation**.

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*[12]

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[13], to negative reviews:

> "… 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*[14]

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[15] 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.

[11] https://en.wikipedia.org/wiki/Student_syndrome
[12] https://www.goodreads.com/quotes/458856-my-dear-here-we-must-run-as-fast-as-we
[13] https://en.wikipedia.org/wiki/Duke_Nukem_Forever
[14] https://en.wikipedia.org/wiki/Duke_Nukem_Forever
[15] https://en.wikipedia.org/wiki/Visual_Basic

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.

Having looked at both Time and Money components of risk, let’s look at something equally fundamental, **Complexity Risk**.

Doing things slowly.

beating rush hour

Chapter 9

Agency Risk

Coordinating a team is difficult enough when everyone on the team has a single **Goal**. But, people have their own goals, too. Sometimes, the goals harmlessly co-exist with the team's goal, but other times they don't.

This is **Agency Risk**. This term comes from finance and refers to the situation where you (the "principal") entrust your money to someone (the "agent") in order to invest it, but they don't necessarily have your best interests at heart. They may instead elect to invest the money in ways that help them, or outright steal it.

> "This dilemma exists in circumstances where agents are motivated to act in their own best interests, which are contrary to those of their principals, and is an example of moral hazard." - Principal-Agent Problem, *Wikipedia*[1]

The less visibility you have of the agent's activities, the bigger the risk. However, the whole *point* of giving the money to the agent was that you would have to spend less time and effort managing it. Hence the dilemma. So, **Agency Risk** flourishes where there is **Invisibility Risk**.

Agency Risk clearly includes the behaviour of Bad Actors[2]. But, this is a very strict definition of **Agency Risk**. In software development, we're not lending each other money, but we are being paid by the project sponsor, so they are assuming **Agency Risk** by employing us.

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

CV Building

This is when someone decides that the project needs a dose of "Some Technology X", but in actual fact, this is either completely unhelpful to the project (incurring large amounts of **Complexity Risk**), or merely less useful than something else.

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

It's very easy to spot CV building: look for choices of technology that are incongruently complex compared to the problem they solve, and then challenge by suggesting a simpler alternative.

Consultancies

When you work with an external consultancy, there is *always* more **Agency Risk** than with a direct employee. This is because as well as your goals and the employee's goals, there is also the consultancy's goals.

This is a good argument for not using consultancies, but sometimes the technical expertise they bring can outweigh this risk.

Also, try to look for *hungry* consultancies: if you being a happy client is valuable to them, they will work at a discount (either working cheaper, harder or longer or more carefully) as a result.

The Hero

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

Heroes put in more hours and try to rescue projects single-handedly, often cutting corners like team communication and process in order to get there.

Sometimes, projects don't get done without heroes. But other times, the hero has an alternative agenda than just getting the project done:

- A need for control, and for their own vision.
- A preference to work alone.
- A desire for recognition and acclaim from colleagues.
- For the job security of being a Key Man[4].

A team *can* make use of heroism, but it's a double-edged sword. The hero can becomes **a bottleneck** to work getting done, and because want to solve all the problems themselves, they **under-communicate**.

Devil Makes Work

Heroes can be useful, but *underused* project members are a nightmare. The problem is, people who are not fully occupied begin to worry that actually, the team would be better off without them, and then wonder if their jobs are at risk.

The solution to this is "busy-work": finding tasks that, at first sight, look useful, and then delivering them in an over-elaborate way (Gold Plating[5]) that'll keep them occupied. This will leave you with more **Complexity Risk** than you had in the first place.

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

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

Pet Projects

> A project, activity or goal pursued as a personal favourite, rather than because it is generally accepted as necessary or important. - Pet Project, *Wiktionary*[6]

Sometimes, budget-holders have projects they value more than others without reference to the value placed on them by the business. Perhaps the project has a goal that aligns closely with the budget holder's passions, or its related to work they were previously responsible for.

Working on a pet project usually means you get lots of attention (and more than enough budget), but due to **Map and Territory Risk**, it can fall apart very quickly under scrutiny.

Morale Risk

> Morale, also known as Esprit de Corps is the capacity of a group's members to retain belief in an institution or goal, particularly in the face of opposition or hardship - Morale, *Wikipedia*[7]

Sometimes, the morale of the team or individuals within it dips, leading to lack of motivation. **Morale Risk** is a kind of **Agency Risk** because it really means that a team member or the whole team isn't committed to the **Goal**, may decide their efforts are best spent elsewhere. **Morale Risk** might be caused by:

- External factors: Perhaps the employees' dog has died, or they're simply tired of the industry, or are not feeling challenged.
- If the team don't believe a goal is achievable, they won't commit their full effort to it. This might be due to to a difference in the evaluation of the risks on the project between the team members and the leader.
- If the goal isn't considered sufficiently worthy, or the team isn't sufficiently valued.
- In military science, a second meaning of morale is how well supplied and equipped a unit is. This would also seem like a useful reference point for IT projects. If teams are under-staffed or under-equipped, this will impact on motivation too.

Hubris & Ego

It seems strange that humans are over-confident. You would have thought that evolution would drive out this trait but apparently it's not so:

> "Now, new computer simulations show that a false sense of optimism, whether when deciding to go to war or investing in a new stock, can often improve your chances of winning." - Evolution of Narcissism, *National Geographic*[8]

[6] https://en.wiktionary.org/wiki/pet_project
[7] https://en.wikipedia.org/wiki/Morale
[8] https://news.nationalgeographic.com/news/2011/09/110914-optimism-narcissism-overconfidence-hubris-evolution-science-nature/

In any case, humans have lots of self-destructive tendencies that *haven't* been evolved away, and we get by.

Development is a craft, and ideally, we'd like developers to take pride in their work. Too little pride means lack of care, but too much pride is *hubris*, and the belief that you are better than you really are. Who does hubris benefit? Certainly not the team, and not the goal, because hubris blinds the team to hidden risks that they really should have seen.

Although over-confidence might be a useful trait when bargaining with other humans, the thesis of everything so far is that **Meeting Reality** will punish your over-confidence again and again.

Perhaps it's a little unfair to draw out one human characteristic for attention. After all, we are riddled with biases[9]. There is probably an interesting article to be written about the effects of different biases on the software development and project management processes. This task is left as an exercise for the reader.

tbd chapter link

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

Chapter 10

Map And Territory Risk

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

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

Sat-nav blunder sends Asda van crashing down narrow footpath

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

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

How about that?
News » UK News »
Technology News »
Andrew Hough »

In How About That?

Pictures of the day

Figure 10.1: Sat Nav Crash - Telegraph Newspaper

In the picture shown here, the driver *trusted* the SatNav to such an extent that he didn't pay attention to the road-signs around him, and ended up getting stuck.

This wasn't borne of stupidity, but experience: *so many times* the SatNav had been right, that the driver stopped questioning its fallibility. But SatNavs are pretty reliable, this is kind of excusable. People are happy to make this mistake with far less reliable systems because often it's a shortcut to having to do any real thinking.

Metrics

The simplest type of **Map And Territory Risk** occurs like this: someone finds a useful new metric that helps in evaluating performance. It might be:

- **SLOC (Source Lines Of Code)**: i.e. the number of lines of code each developer writes per day/week whatever.
- **Function Points**: the number of function points a person on the team completes, each sprint.
- **Code Coverage**: the number of lines of code exercised by unit tests
- **Response Time**: the time it takes to respond to an emergency call, say
- **Release cadence**: number of releases a team performs, per month, say.

With some skill, they may be able to *correlate* this metric against some other more abstract measure of success. For example: - "quality is correlated with more releases" - "user-satisfaction is correlated with SLOC" - "response time is correlated with revenue"

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

But *correlation* doesn't imply *causation*. The cause might be different:
- quality and number of releases might both be down to the simplicity of the product. - user satisfaction and SLOC might both be down to the calibre of the developers. - response time and revenue might both be down to clever team planning.

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

Drinking The Kool-Aid

The next problem comes when metrics start being used for more than just indicators, but as measures of performance or targets: - If a team is *told* to do lots of releases, they will perform lots of releases *at the expense of something else*. - If team members are promoted according to SLOC, they will make sure their code takes up as many lines as possible. - In the UK, when ambulances were asked to respond to all emergency calls within a short window, cars and bicycles were employed as ambulances too [tbd].

You are probably nodding your head at these examples. *Of course* SLOC is a terrible measure performance! We're not that stupid anymore. The problem is, it's not so much the *choice* of metric, but the fact that *all* metrics merely approximate reality with a few numbers.

The map is *always* simpler than the territory, therefore there can be no perfect metrics.

The Onion Of Bullshit

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

- My team had been tasked with building automated "smoke tests" of an application. But this was bullshit. We only needed to build these *at all* because the application was so complex. The reason it was so complex was…
- The application was being designed within a "Framework" constructed by the department. However, the framework was only being used by this one application. Building a "reuasable" framework which is only used by a single application is bullshit. But, we had to do this because…
- The organisational structure was created along a "matrix", with "business function" on one axis and "functional area" on another. Although we were only building the application for a single business function, it was expected to cater with all the requirements from the an entire "functional area". This was bullshit too, because
- The matrix structure was largely the legacy of a recent merger with another department. As **Conway's Law** predicts, our software therefore had to reflect this structure. But this was bullshit because
- The matrix structure didn't represent reality in any useful way. It was designed to pacify the budget committee at the higher level, and try to demonstrate attributes such as *control* and *governance*. But this was bullshit too, because
- The budget that was given to our department (Risk) was really based on how much fear the budget holders currently had of the market regulators. But this was bullshit too, because
- At a higher level, the executives had realised that Investment Banking wasn't one of the banks strategic strengths, and was working to close it all down anyway.

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

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

It feels like there's no way back from that. All of life might well be a big **Map and Territory** illusion. But let's analyse just a bit: - At each layer of the onion, the objectives changed. But, they impacted on the objectives of the layer below. - Therefore, it seems like the more layers you have, the less likely it is that your objectives become inconsistent between the lower and higher levels. - On a new project, it seems like a good idea to model this stuff: does the objective of the work you're about to undertake "align" with the objectives at a higher level? If not, the project might well be quite temporary: Before I left, I was able to eject most of the "framework" elements of the project, and massively simplify the architecture, thus obviating the need for the smoke tests.

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

Inadequate Equilibria

Inadequate Equilibria is a book by Eleizer Yudkovsky, who looks at how **Map and Territory Risk** can break not just departments, but entire societal systems. Here is one example involving *academics* and *grantmakers* in academia:

- It's not very apparent which scientists are better than which other scientists.
- One proxy is what they've published (scientific papers) and where they've published (journals).
- Universities want to attract research grants, and the best way to do this is to have the best scientists.
- Because "best" isn't measureable, they use the proxy.
- Therefore, immense power rests in the hands of the journals, since they can control the money-proxy.
- Therefore, journals are able to charge large amounts of money to universities for subscriptions.

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

tbd

Head In The Sand

Introduce Rapid Development example here?

how to pick projects

how to spot vanity projects

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

Head in the sand

Bullshit jobs

Biases

Why is this relevant?

human biases.

- showing progress

- the release
- less wrong

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

Do biases go in here? YES!!

Part III

Practices

Chapter 11

Coding

What Is It

Coding is the main practice that identifies us as working on a *software project*: Actually entering instructions in a language that the computer will understand, be it Java, C++, Matlab, Excel or *whatever*. It is transferring the ideas in your head into steps that the computer can understand, or, as Wikipedia has it:

> "…actual writing of source code." – Wikipedia, *Computer Programming*[1]

Often, this can be called "programming", "hacking" or "development", although that latter term tends to connotate more than just programming work, such as **Requirements Capture** or **Documentation**, but we're considering those separately on different pages.

How It Works

In **Development Process** we introduced the following diagram to show what is happening when we do some coding. Let's generalize a bit from this diagram:

- We start with a **Goal In Mind** to implement *something*.
- We build an **Internal Model** of how we're going to meet this goal (though coding, naturally)
- Then, we find out how well our idea stands up when we **Meet Reality** and try it out in our code-test-run-debug cycle.
- As we go, the main outcome is that we change reality, and create code, but along the way, we discover where our **Internal Model** was wrong, in the form of surfacing **Hidden Risks**.

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

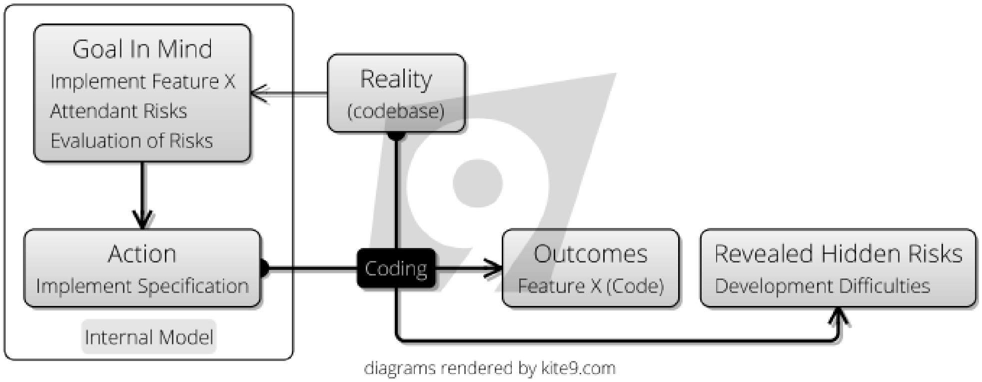


Figure 11.1: Coding

Examples

As with any **Practice**, we are coding to minimize **Attendant Risks**. We might want…

- **To Build** or improve some features which our clients will find useful. - ***Feature Risk***
- **To Automate** some process that takes too long or is too arduous. - ***Process Risk***
- **To Explore** how our tools, systems or dependencies work (also called Hacking[2]). - ***Dependency Risk***
- **To Refactor** our codebase, to reduce complexity. - ***Complexity Risk***
- **To Clarify** our product, making our software more *presentable* and *easier to understand*. - ***Communication Risk***

… and so on. As usual, the advice is to *reduce risk* in the most meaningful way possible, all the time. This might involve coding *or it might not*.

Where It's Used

Since the focus of this site is on *software methodologies*, you shouldn't be surprised to learn that *all* of the methodologies use **Coding** as a central focus.

Variations

Building Features

Most commonly, the reason we are **Coding** is same as the one in the **Development Process** page: we want to put features in the hands of our customers.

That is, we believe our clients don't have the features they need to see in the software, and we have **Feature Risk**.

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

By coding, we are mitigating **Feature Risk** in exchange for **Complexity Risk** in terms of the extra code we now have on the project, and **Schedule Risk**, because by spending time or money coding we now have less time or money to do other things. Bill Gates said:

> "Measuring programming progress by lines of code is like measuring aircraft building progress by weight." - Bill Gates

And you can see *why* this is true: the more code you write, the more **Complexity Risk** you now have on the project, and the more **Dead End Risk** you've picked up in case it's wrong. This is why **The Agile Manifesto** stresses:

> "Simplicity -the art of maximizing the amount of work not done- is essential." Agile Manifesto[3]

Prototyping

Users often have trouble *conceiving* of what they want in software, let alone *explaining* that to developers in any meaningful way.

Let's look at how that can happen.

Imagine for a moment, that there was such a thing as **The Perfect Product**, and a **User** wants to build it with a **Coder**: - The **Perfect Product** might be *conceptually elusive*, and it might take several attempts for the **User** to find it's form. ***Conceptual Integrity Risk*** - It might be hard for the **User** to *communicate* the idea of it in writing or words: where do the buttons go? What do they do? What are the key abstractions? ***Communication Risk*** - It might be hard too, for the **Coder** to work with this description. Since his **Internal Model** is different from the **User**'s, they have different ideas about the *meaning* of what the **User** is communicating. ***Communication Risk*** - Then, implementing the idea of whatever is in the **Coder**'s **Internal Model** takes *effort*, and therefore involves **Schedule Risk**. - Finally, we have a feedback loop, so the **User** can improve their **Internal Model** and see the previously unforeseen **Hidden Risks**. - Then, you can go round again.

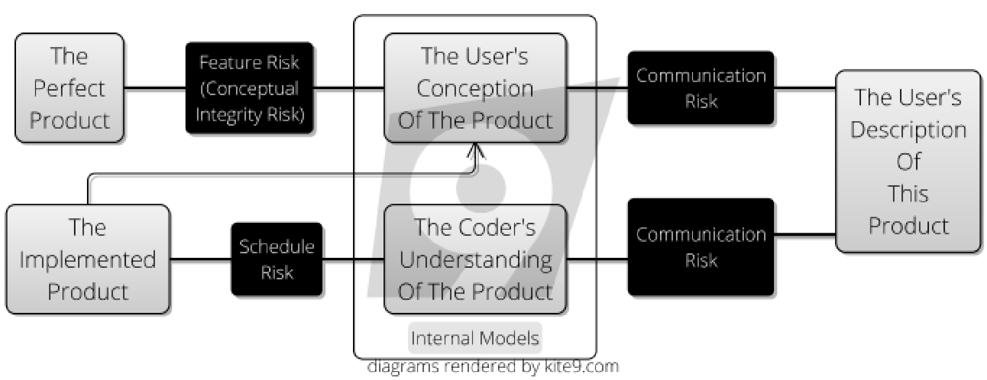


Figure 11.2: Coding Communication Risks

[3]http://agilemanifesto.org/

The problem here is that this is a very *protracted feedback loop*. This is mitigated by prototyping, because that's all about shortening the feedback loop as far as possible:
- By working together, you mitigate **Communication Risk**. - By focusing on one or two elements (such as UI design), you can minimize **Schedule Risk**. - By having a tight feedback loop, you can focus on *iteration*, try lots of ideas, and work through **Conceptual Integrity Risk**.

One assumption of Prototyping is that **Users** can iterate towards **The Perfect Product**. But it might not be so: the Conceptual gap between their own ideas and what they really *need* might prove too great.

After all, bridging this gap is the job of the **Designer**:

> "It's really hard to design products by focus groups. A lot of times, people don't know what they want until you show it to them." — Steve Jobs

SkunkWorks

The SkunkWorks[4] approach is one small step up from **Prototyping**. Wikipedia describes this as:

> A group within an organization given a high degree of autonomy and unhampered by bureaucracy, with the task of working on advanced or secret projects

The idea here is *again* to minimize the length of the feedback loop, and focus on **Design** to combat **Conceptual Integrity Risk**. It was in this kind of small, secret team that the iPhone was invented[5].

To give some idea of the **Conceptual Integrity Risk** involved, initially, the team were building a *tablet* using the multi-touch technology that the iPhone introduced to the world, but pivoted towards the phones after the failure of the "Apple Phone" collaboration with Motorola.

Scott Forstall picked a small, secret team from within the ranks of Apple. By doing this, he mitigated **Communication Risk** and **Coordiation Risk** *within his team*, but having fewer people in the room meant more **Throughput Risk**.

By having more people involved, the feedback loop will be longer than the two-man prototyping team, but that's the tradeoff you get for mitigating those other risks.

Specialization

One of the problems with a **SkunkWorks** approach is that you end up with more **Coordination Risk** than you'd like, as your different skunk-teams end up with different **Internal Models** and different **Goals**.

In large companies, this is called Silo Mentality[6] - the tendency for lines of business to stop communicating and sharing with one another. As you can imagine, this leads to a more **Complex** and **bureaucratic** structure than would be optimal.

[4] https://en.wikipedia.org/wiki/Skunk_Works
[5] https://www.networkworld.com/article/2159873/smartphones/apple-s-iphone--the-untold-story.html
[6] https://en.wikipedia.org/wiki/Information_silo

But this can happen within a single coding team, too: by splitting up and working on different pieces of functionality within a project, the team *specialises* and becomes expert in the parts it has worked on. This means the team members have different **Internal Models** of the codebase.

This is *perfectly normal*: we *need* people to have different opinions and points-of-view. We *need* specialisation, it's how humanity has ended up on top[7]. It's better to have a team who, between them all, know a codebase really well, than a group of people who know it poorly.

The reason for this is explained again by the first diagram in this section: the closer our **Internal Model** matches **Reality**, the fewer **Hidden Risks** we will meet, and the easier we'll have it.

The downside of specialization is **Coordination Risk**:
- If your payroll expert is off ill for a week, progress on that stops. - Work is rarely evenly spread out amongst the different components of a project for long. - If work temporarily dries up on a specific component, what do the component owners do in the meantime? - What if the developer of a particular component makes *the wrong assumptions* about other parts of the system or tool-set?

Pair Programming / Mob Programming

In the main, **Review** is the main way to mitigate **Coordination Risk**. For example: - **Code Reviews - Stand Up Meetings - Presentations & Demos - Training**

Pair Programming however *combines* the review with the process of coding: there are now two heads at each terminal. What does this achieve?
- Clearly, we mitigate **Key-Man Risk** as we've got two people doing every job.
- Knowledge is transferred at the same time, too, mitigating **Specialist Risk**.
- Proponents also argue that this mitigates **Complexity Risk**, as the software will be better quality. - Since the pair spend *so much time together*, the communication is very *high bandwidth*, so this mitigates **Communication Risk**

But, conversely, there is a cost to **Pair Programming**: - Having two people doing the job *one person could do* intimates **Schedule Risk**. - Could the same **Complexity Risk** be mitigated just with more regular **Code Reviews**? - Sometimes, asking members of a team to work so closely together is a recipe for disaster. **Team Risk** - Not every pair programmer "shares" the keyboard time evenly, especially if ability levels aren't the same. - There is only one **Feedback loop**, so despite the fact you have two people, you can only **Meet Reality** serially.

Mob Programming goes one stage further and suggests that we can write better software with *even more people around the keyboard*. So, what's the right number? Clearly, the usual trade-off applies: are you *mitigating* more risk than you're *gaining*?

Offshoring / Remote Teams

Pairing and **Mobbing** as mitigations to **Coordination Risk** are easiest when developers are together in the same room. But it doesn't always work out like this. Teams spread in different

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

locations and timezones naturally don't have the same **communication bandwidth** and you *will* have more issues with **Coordination Risk**.

In the extreme, I've seen situations where the team at one location has decided to "suck up" the extra development effort themselves rather than spend time trying to bring a new remote team up-to-speed. More common is for one location to do the development, while another gets the **Support** duties.

When this happens, it's because somehow the team feel that **Coordination Risk** is more unmanageable than **Schedule Risk**.

There are some mitigations here: video-chat, moving staff from location-to-location for face-time, frequent **show-and-tell**, or simply modularizing accross geographic boundaries, in respect of **Conway's Law**:

> "organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations." - *M. Conway*[8]

When we add **Outsourcing** into the mix, we also have to consider **Agency Risk**: the consultancy you've hired is *definitely* more interested in keeping themselves solvent than solving your business problems.

Team Size

As team sizes grow, **Coordination Risk** grows fast.

To see this, let's consider a made-up situation where all the developers are equal, and we can mitigate **Coordination Risk** at the cost of a 1-hour presentation each per week.

How many man-hours of presentations do we need?

Team Size	Hours Of Presentations	Man-Hours In Presentations
1	0	0
2	2	4
3	3	9
4	4	16
5	5	25
6	6	36
7	7	49

Adding the 7th person to the team (ceteris paribus) does absolutely *nothing* for productivity, it makes matters worse. Assuming everyone works a 40-hour week, we're now 9 hours worse off than before.

This is a *toy example*, but is it better or worse than this in reality? If the new developers are arriving on an existing project, then 1 hour-per-week of training by the existing team members might be conservative.

This is why we get Brooks' Law[9]:

[8] https://en.wikipedia.org/wiki/Conway%27s_law
[9] https://en.wikipedia.org/wiki/Brooks%27s_law

> "adding human resources to a late software project makes it later". - Fred Brooks, *The Mythical Man-Month*[10]

You can see that this law is founded in an appreciation of **Coordination Risk**. But the argument from **Coordination Risk** *adds nuance*, and explains when this is true and when it isn't.

Too Many Cooks

Sometimes, you have *too many developers* on a project. This is not a blessing. As with **Student Syndrome**, having too many resources means that:

> "Work expands so as to fill the time available for it's completion" - **Parkinson's Law**

One of the reasons for this is that *Developers love to develop* and it is, after all, their job. If they *aren't* developing, then are they still needed? This is **Agency Risk**: people who are worried about their jobs will often try to *look busy*, and if that means creating some drama on the project, then so be it.

Sadly, this usually occurs when a successful project is nearing delivery. Ideally, you want to be *decreasing* the amount of change on a project as it gets closer to key **Delivery Dates**. This is because the risk of **Missing the Date** is greater than the risk of **some features not being ready**.

In the past, I've found it helpful to down-size the team by temporarily moving developers into other less-fortunate teams, reducing both **Coordination Risk** and **Agency Risk** at the same time.

This can require some guts to do: you have to overcome your own ego (wanting to run a big team) for the sake of your project.

Automating

One of the key ways to measure whether your team is doing *useful work* is to look at whether, in fact, it can be automated. And this is the spirit of **DevOps** - the idea that people in general are poor at repeatable tasks, and anything people do repeatedly *should* be automated.

Repetitive work of any kind is a **Process Risk**, and can be mitigated at the expense of attendant **Complexity Risk** and **Schedule Risk**.

Since this is a trade-off, you have to be careful about how you *weigh* the **Process Risk**: clearly, it exists *into the future*.

You are making a bet that acting now will pay off in decreased **Process Risk** over the lifetime of the project. This is a hard thing to judge: - How much **Process Risk** are we carrying, week-to-week? (A good way to answer this is to look at past failures). - How much **Complexity Risk** will we pick up? - How much **Schedule Risk** (in spent developer effort) will we pick up? - How long will the mitigation last before the process changes again?

[10] https://en.wikipedia.org/wiki/Brooks%27s_law

Tool Use

In general, unless the problem is somehow *specific to your circumstances* it may well be better to skip direct coding and pick up some new tools to help with the job.

Tools are a different trade off to automation. You are mitigating **Process Risk** or **Feature Risk** in return for: - New **Dependency Risk** on the new tool. - **Communication Risk** because now the team has to understand the tool. - **Schedule Risk** in the time it takes to learn and integrate the tool. - **Complexity Risk** because your project necessarily becomes more complex for the addition of the tool.

Tools in general are *good* and *worth using* if they offer you a better risk return than you would have had from not using them.

But, this is a low bar - some tools offer *amazing* returns on investment. The **Silver Bullets** article describes in general some of these: - Assemblers - Compilers - Garbage Collection - Type Systems - Libraries - Build Tools - etc.

A *really good tool* offers such advantages that not using it becomes *unthinkable*: Linux is heading towards this point. For Java developers, the JVM is there already.

Picking new tools and libraries should be done **very carefully**: you may be stuck with your choices for some time. Here is a **short guide that might help**.

Refactoring

The term "refactoring" probably stems from the mathematical concept of *(Factorization)[https://en.wikipedia.org/wiki/Factorization]*. Factorizing *polynomial equations* or *numbers* means to identify and make clear their distinct components.

tbd: SoC

Most coders use the phrase "refactoring", and intuitively understand what it is. It shouldn't have been hard to find a clear explanation for this page, but sadly it was. There are some very woolly definitions of "refactoring" around, such as:

> "**Refactoring (n)**: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"" – Refactoring.com[11]

What do "easier to understand" (which makes sense) and "cheaper to modify" mean? Let's try and be more specific. With Refactoring, we are trying to:

- Mitigate **Communication Risk** by making the *intent* of the software clearer. This can be done by breaking down larger functions and methods into smaller ones with helpful names, and naming elements of the program clearly, and
- Mitigate **Complexity Risk** by employing *abstraction* and *modularization* to remove duplication and reduce cross-cutting concerns. By becoming less complex, the code has less **Inertia**.

On **Refactoring**, Kent Beck says:

[11] https://www.refactoring.com

> "If a programmer sees a one-minute ugly way to get a test working and a ten-minute way to get it working with a simpler design, the correct choice is to spend the ten minutes." – Kent Beck, *Extreme Programming Explained*

This is a bold, not-necessarily-true assertion. How does that ratio stack up when applied to *hours* or *days*? But you can see how it's motivated: Kent is saying that the nine extra minutes of **Schedule Risk** are *nothing* compared to the carry cost of **Complexity Risk** on the project.

Risks Mitigated / Attendant Risks

tbdd

Attendant Risks

tbd

See Also

Risk First Applied

- Do the riskiest bits first.
- Try and map out the risk landscape
- Examine Boundary Risk and Dead End Risk issues: is this decision going to limit you

Chapter 12

Design

What Is It

Design is what you do every time you think of an action to mitigate a risk. And **Big Design Up Front** is where you do a lot of it in one go, for example:

- Where you think about the design of all (or a set of) the requirements in one go, in advance.
- Where you consider a *set of **Attendant Risks*** all at the same time.

Compare with "little" design, where we consider just the *next* requirement, or the *most pressing* risk.

Although it's fallen out of favour in Agile methodologies, there are benefits to doing this *sometimes*.

How It Works

As we saw in **Meet Reality**, "Navigating the **Risk Landscape**", meant going from a position of high risk, to a position of lower risk. **Agile Design** is much like **Gradient Descent**: each day, one small step after another *downwards in risk* on the **Risk Landscape**.

But the problem with this is you can get trapped in a **Local Minima**, where there are *no* easy steps to take to get you to where you want to be. Here is a **real life example**. This is **Dead End Risk**.

In these cases, you have to *widen your horizon* and look at where you want to go: and this is the process of *design*. You're not necessarily now taking steps on the **Risk Landscape**, but imagining a place on the **Risk Landscape** where you want to be, and checking it against your **Internal Model** for validity.

Examples

Feedback Loops & Mitigated Risks

The feedback loop for any design is **Review and Sign Off**.

Too Many Cooks

By allowing lots of stakeholders to review and **agree to a design**, or select from alternatives, we try to reconcile the needs of lots of stakeholders *early on* in a project.

Visibility Risk

To allow for *discussion and understanding* of the project between multiple parties. This may extend to design being *marketing material* to help explain the project to potential clients or budget-holders.

Technical-Debt

To ensure an overall aesthetic or architectural integrity, avoiding the **Technical-Debt** that you might accrue by building the wrong things first.

Dead End Risk

Often, by thinking big-picture we can avoid building components that *seem* like a good next step, but actually aren't.

Attendant Risks

Building architects appreciate that their *plans might change*: Roman ruins might be discovered underneath the site, or the supporting wall might not be as sound as originally thought. The more effort you put into a design, the more will be wasted if it's wrong. So, how deep should you go? The answer as usual, is keep designing while it is reducing your overall project risk.

- The design might itself take a long time to complete **Schedule Risk**.
- People *stop thinking* **once they have a design**, even when reality *obviously* deviates from what the design assumed. But the whole point of a plan is that it's easier to change than the thing you are doing the plan for.

- If your plan starts to become as detailed as the code would be (but doesn't run) then you've made the mistake of *overspecification*, and you are creating **Technical Debt**.

> Everyone has a great plan until they get hit in the nose - Mike Tyson Fail to plan and you plan to fail - Eisenhower?

Risk first design example ; building the research indexer

Chapter 13

Prioritisation

What Is It

Prioritisation is a key process in trying to focus on building *useful* stuff first. It could look like:

- **A Sprint Planning Meeting**: Deciding on the most important things for the team to build in a time period.
- **Phased Delivery**: Breaking a large project into smaller-scoped projects.
- **A Backlog**: Having tasks or stories in delivery order in a queue.
- **Task Decomposition**: Breaking down larger units of a task into smaller items. Often, **Requirements** come *bundled together* and need to be broken down so that we work on just the most vital parts, as in
- **Identifying the MVP**: Trying to cast out *all* non-essential functionality.

Prioritisation relies on not delivering all the functionality in one go. But it tends to be a spectrum:

- **Big Bang**: Delivering all the functionality in a single go.
- **Cycles, or Phases**: Splitting a large project into smaller chunks.
- **Sprints**: Delivering with a fixed cadence, e.g. every month or week.
- **Continuous Delivery**: Delivering functionality one-piece-at-a-time.

Usually, risk is mitigated by **Prioritisation**. But sometimes, it's not appropriate: When Finland changed from driving on the right side of the road to the left, (in order to be in line with the rest of Europe) the changeover *had* to be **Big Bang** and the whole country changed **overnight**.

How It Works

There are several ways you can prioritise work:

- **Largest Mitigation First**: What's the thing we can do right now to reduce our **Attendant Risk** most? This is sometimes hard to quantify, given **Hidden Risk**, so maybe an

easier metric is…
- **Biggest Win**: What's the best thing we can do right now to reduce **Attendant Risk** for least additional **Schedule-Risk**? (i.e. simply considering how much *work* is likely to be involved)
- **Dependency Order**: Sometimes, you can't build Feature A until Feature B is complete. Prioritisation helps to identify and mitigate **Dependency Risk**.

By prioritising, you get to **Meet Reality** *sooner* and *more frequently* and in *small chunks*.

Feedback Loops & Risks Mitigated

Review

This one way in which a particular prioritisation **Meets Reality**

- Developers might tell you that the ordering incurs **Dependency Risk** or **Coordination Risk** if everyone is going to end up working on the same components.
- Product Owners might tell you that you're not tackling the right **Feature Risk**.
- If you're trying to work out what the **MVP** is, prioritisation might help your investors determine whether the project is worth **funding**.

Production Risk

Breaking a large delivery down into lots of small releases has an impact on **Production Risk**:

- Usually, lots of small releases allows you to *practice* the release process while the project is relatively unimportant. This experience allows you to figure out automation and reduce the **Process Risk** of releasing too.
- Smaller, higher-cadence releases also reduce **Visibility Risk**, because users don't have large amounts of change to get accustomed to all-in-one-go.

Schedule Risk

If you're able to do **Continuous Delivery**, and have de-risked the release process, then you can eliminate some **Schedule Risk**, because you'll know you can hit any date with *something*. The risks of what you deliver on that date are then Feature Risk rather than **Schedule Risk**.

Attendant Risks

Dependency Risk

The biggest risk to phased delivery is that you try and build functionality **now** that actually relies on things scheduled to be built **later**.

Schedule Risk

Sometimes, releases have a *cost* associated with them in terms of time and bureaucracy to perform them. Obviously, then, the more releases you'll do, the less time you'll spend doing *other stuff*, like building functionality. The trick to doing frequent releases is therefore to ensure they are *low cost*, and this means **automation**. But, building automation adds schedule risk too.

Complexity Risk

If you are replacing an old system with a new one, incrementally replacing functionality is a good way to go when the system is complex. However, this means that you're going to have two systems running at the same time, which is inevitably **more complex** than just one system.

PLanning

- also gannt chart
- critical path
- roadmap
- dependency analysis

Discuss the tool Duncan and I used to determine whether a release date was feasible.

- planning using risk

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

– estimating: holding the risks in your hand and saying, which is heavier?

Risk first planning: break down the goal into the biggest risks3

Chapter 14

Requirements Capture

What Is It

Requirements Capture is not a single technique, but a broad category of techniques such as:

- **Interviews**
- Focus Groups
- User Stories
- Use Cases

How It Works

Whatever exact methodology you are using, the aim is to meet **stakeholders** and try to capture their **Internal Models** in written form. This has a few effects:

- The **Internal Models** of the development team can be enriched with this new information.
- You are capturing at a moment-in-time what people *thought*.

- As with any **Documentation**, you can to some extent reconcile the **Internal Models** of a wide variety of stakeholders and implementers.

Variations

Structured Requirements

Use Cases

See also: **Terms Of Reference**

Feedback Loops / Risks Mitigated

Requirements Capture is itself a process of **Meeting Reality**, and in a limited way: rather than speculatively building a piece of software and trying it out on the world, Requirements Capture allows us, cheaply, to go and see what the world *thinks* it wants, which is much *cheaper*, but perhaps less accurate.

Feature-Risk

Asking people what they want is often a way to reduce **Feature Risk** by stopping you building the wrong thing.

[

The feedback loop for any design is *review*. You can also follow review with **Sign Off**.

Attendant Risks

Steve Jobs - people don't know what they want until they see it.

Elizer Yodowski - what exactly is a MVP.

Chapter 15

Testing

What Is It

Most forms of testing are about isolating a particular *characteristic* of your system, and exploring it from a risk perspective. It could be:

- **Performance Testing** addresses the risk of **not being able to support all the users**
- **Usability Testing** tries to see whether people struggle to make sense of your software, usually because the assumptions of their **Internal Models** differ from those embedded in the system, or that the system isn't adequately **transparent** about it's own model.
- **Security Testing** addresses the risk that your software could be used against you or its users **by hackers**.
- **Integration Testing**: Where we test how the software works as-a-whole, and test that it will work **with other systems**
- **Corridor Testing**: Asking a few, random people to use the system-under-test, in order to see **if it confuses them, or not**.
- **User Acceptance Testing**: Asking users to review new features, and make sure that they actually **do what is required**
- **Regression Testing**: Making sure changes in new versions of the system haven't **broken functionality**

How It Works

The whole purpose of testing is to **meet reality** early, ahead of putting software in front of real users, where you face **Production Risks**, like reputation damage and financial penalties.

Given this, the best approach to test planning should be risk-based: consider which risks you want to mitigate, and test accordingly:

- Identify Risks
- Evaluate Risks
- Prioritise Risks

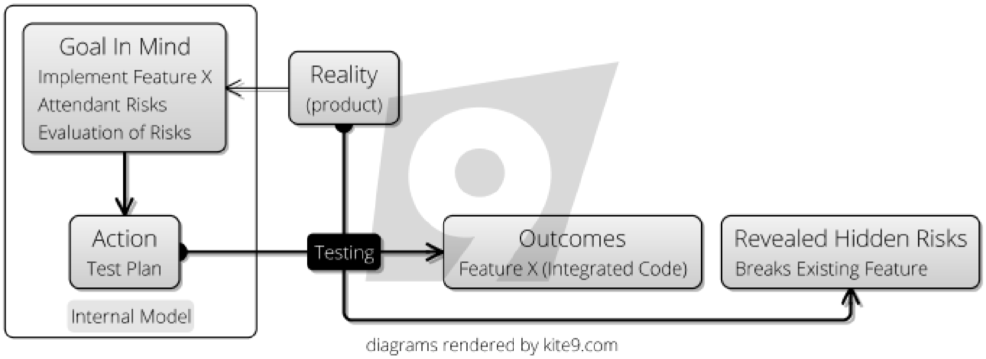


Figure 15.1: Testing Process

- Plan tests from the top of the priority list down.

Examples

This should work at *every level* within a project. If you are building a new feature, you should consider:

- Is it going to connect to third-party systems? If so, I should build System Integration Tests[1] to cover the **Dependency Risk** associated with this, and the chance that in the future, the interface will change.
- Does my code do what I expect? I probably should build a Unit Test[2] to mitigate **Complexity Risk**.
- Will users understand the software I build for them? I should probably do some Beta Testing[3] or Corridor Testing[4] to mitigate **Visiblity Risk**.
- To go live, am I going to need some piece of real-world paperwork? Test the process ahead-of-time to expose all the **Hidden Risks**

Where It's Used

- **Waterfall** initially was conceived with a long, manual testing phase to be performed on the *whole system* after development
- **Extreme Programming** championed the use of Unit Tests[5] in order to test individual subsystems, as well as having an **On-Site Customer** to act as a testing resource when needed.

[1]https://en.wikipedia.org/wiki/System_integration_testing
[2]https://en.wikipedia.org/wiki/Unit_testing
[3]https://en.wikipedia.org/wiki/Software_testing#Beta_testing
[4]https://www.usability.gov/what-and-why/glossary/corridor-testing.html
[5]https://en.wikipedia.org/wiki/Unit_testing

Variations

Automated Tests

Often, the decision of whether to automate a test will be based on whether or not it can be expressed *objectively*. For example, checking that a REST endpoint "returns the right error code" is *objective*, and is therefore a candidate for automation.

Automated tests look roughly the same, irrespective of the scope they are trying to test.
- We have a **System Under Test**, which may be a single class, or a whole executable.
- We have some **Input Conditions** for the test, and some **Expectations**. - When the test is executed, we compare the actual outputs with the expected ones, giving us **The Result**.

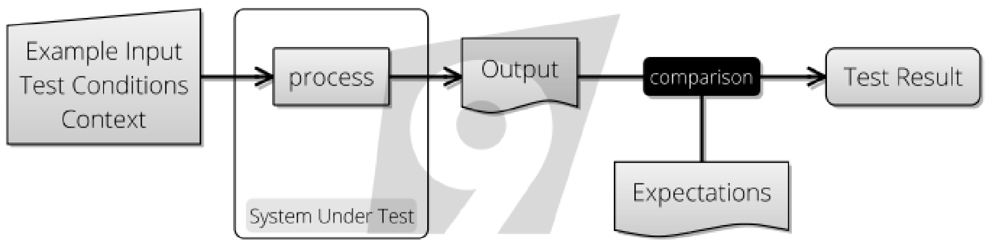


Figure 15.2: Testing Process

A useful way to think about automated testing is that it turns the **System Under Test** into a Pure Function[6]: This means that for a specific set of inputs, the system will produce a specific output, reliably, every time.

Getting complex systems to behave as pure functions can be costly, but there are techniques to help with this such as Mocking[7]. However, if you try to devise as much of your software in a pure-functional way to start with, automated testing is much easier.

Automated Testing has an interesting effect on managing **Complexity Risk**: Although you may initially write a Unit Test (say) to mitigate the risk of **having implemented a feature wrongly**, you are also given insurance against future change breaking that feature. That is to say, they are *regression tests*. However, implementing tests like this is better than building regression tests, **as discussed here**.

– how do automated tests mitigate complexity risk?

Manual Tests

Manual Testing is, at some level, essential if your product is to be used by humans. Although UI-Automation tools such as Selenium[8] allow you to script browser interactions, they cannot reliably catch every problem.

[6]https://en.wikipedia.org/wiki/Pure_function
[7]https://en.wikipedia.org/wiki/Mock_object
[8]https://docs.seleniumhq.org

For example, ensuring the UI "looks ok and doesn't glitch" is entirely *subjective*: you'll need to express this in a manual test. Manual Tests are often described in Test Plans[9] and Test Scripts[10] in order to ensure repeatability, and manage **Process Risk**.

Since manual tests carry much higher per-use cost to run, there is a tendency to want to save this cost by doing *fewer releases*. After all, fewer releases means less manual testing, but this may increase **Process Risk**.

How do you decide whether to keep a test manual, or automate? The more *automated* a test is, the more cheaply it can be re-used. However, the process of automation can take longer, and so adds **Schedule Risk**. Whether or not it's worth automating is to some extend going to depend on how much you **value future time**.

White-Box and Black-Box Testing

In the initial conception, Black-Box Testing[11] ignores the *implementation details* of a component and tests the interface only.

White-box testing however considers the components within the box, and how they interact with one another in order to define the tests. This is *fair enough* if, for some reason, you are unable to test the components individually for some reason: knowing how something is implemented gives you an insight into *where the bugs will hide*, and therefore, where the risks lie.

Testing Level

However, if possible, it's better to break open the white box and test the components *themselves*. This means you end up having "higher" and "lower" level tests, depending on the scope of the **System Under Test**. There are several advantages to this:

- First, tests become less "brittle": the smaller the **System Under Test**, the less **Context** it needs to operate, therefore the more insulated it is to changes in other parts of the system. As a counter-example, if *all* of your tests run over the whole system, and the authentication system changes, does that break all the tests? This is an argument from **Complexity-Risk**.
- Tests at the "whole system" level are usually longer-running since they require starting up the whole system, and also require more data and context to run. This is an argument both from **Complexity-Risk** and **Process Risk**.

Expanding on this then, the Testing Pyramid[12] idea is that lower level, automated tests which run quickly should be common, while there should be fewer of the more expensive "whole system" level tests.

Finally, since manual tests are run by people (who are comparatively slow and costly), these should be the *rarest* kind of test.

[9] https://en.wikipedia.org/wiki/Test_plan
[10] https://en.wikipedia.org/wiki/Test_script
[11] https://en.wikipedia.org/wiki/Black-box_testing
[12] https://martinfowler.com/bliki/TestPyramid.html

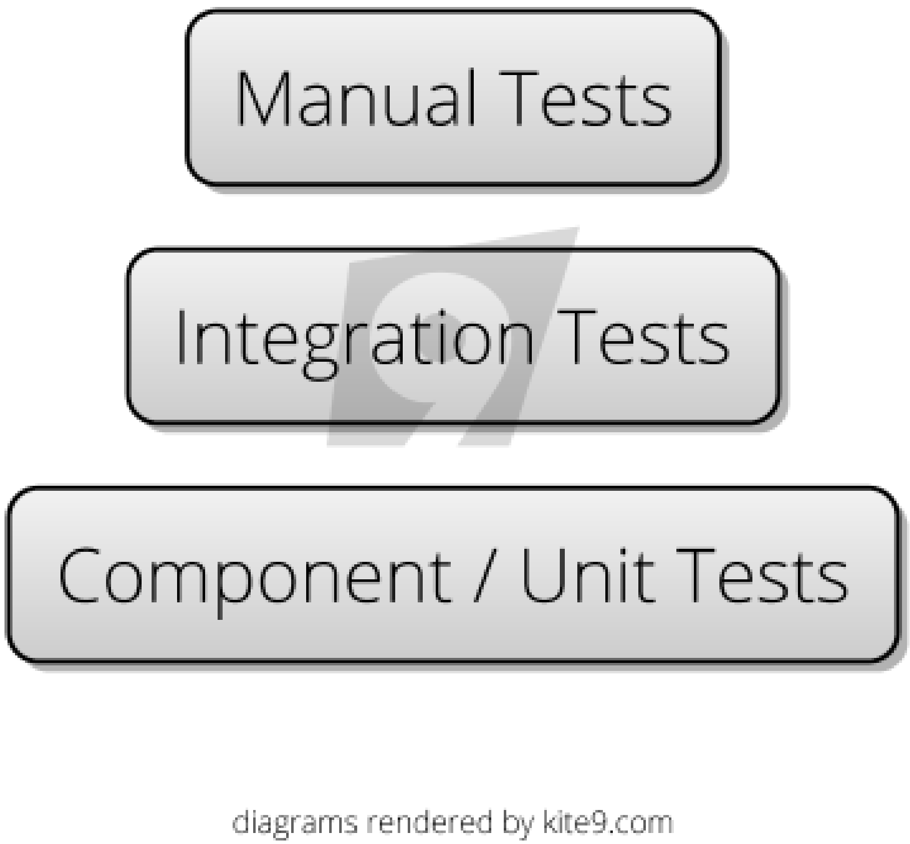


Figure 15.3: Testing Pyramid

Testing Team

Sometimes, testing is handled by external teams (possibly in other locales). This is often done as a **cost-saving measure**, but comes with some penalties such as: - Increased **Bureacratic Risk** in terms of having to engage with an external company. - **Agency Risk** because the testing team are a *business in their own right*, who might be more interested in the goal of making money from you than shipping your product. - Obvious **Coordination Risk** in trying to arrange work in other teams, buildings, timezones or countries, and not having control on exactly which staff are dealing with your product. - **Visibility Risk** because at some level, the testing team need to understand *what your software is for*.

Test-Driven Development

Also called test-first development, the idea here (from **Extreme Programming**) is that you write the tests before the code, in order that you think up-front about the requirements of the software you are writing. The aim of this is to minimize **Complexity Risk** via preventing developers from Gold Plating[13], and getting them to do **The Simplest Thing That Can Possibly Work**.

Additionally, by having test *fail* before they *pass*, you mitigate the risk of writing a "null" test (see below).

Code Coverage

Code Coverage tools are a useful way of showing you which parts of your software might contain bugs due to lack of testing, which is really useful in the **Risk Evaluation** phase of test-planning.

Sometimes code coverage spawns its own **Map And Territory Risks** though, where people forget that the goal should be mitigating overall project risk (via delivering functionality and so forth) and start to believe that the goal is delivering 100% code coverage. Writing tests to cover every `get()` method is a fools' errand which increases the overall **codebase complexity** for no real reduction in **Feature Risk**.

Worse still is that having 100% code coverage does not guarantee an absence of bugs, or that the code will do what the users wanted it to do. **Feature Risk** is always there.

Risks Mitigated

There are so many different types of testing and this guide is not meant to be exhaustive. Instead, here is a table covering some of the main types of testing and the risks they mitigate:

Risk	Mitigation
Boundary Risk	System Integration TestingCI DeploymentUser Acceptance Testing

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

Risk	Mitigation
Dependency Risk	Integration TestingSystem Integration Testing
Production Risk	Performance Testing / Load TestingNon-Functional TestingDisaster Recovery TestingSecurity estingSmoke / Sanity Testing
Software Risk	Unit TestingComponent TestingEnd-To-End TestingFunctional Testing
Feature Risk	Browser-Based TestingAccessibility TestingAcceptance Testing (UAT)Beta Testing
Visibility Risk	Usability TestingCorridor Testing
Complexity Risk	Unit TestingAutomated Acceptance testingIntegration Testing

Attendant Risks

Firstly, it can be easy to fool yourself with tests: just because your tests pass does *not* mean your code is perfect. Vigilance is required against **Map And Territory Risk**:

- Do the tests explore the behaviour of the system the same way the users will?
- Can you be sure you haven't written a "null test", one that passes when it should fail?
- Have you covered the "cracks" between the different parts of the system? Just because all the *components* of a bicycle are fine, it doesn't mean that the *bike itself will work*.

Second, Testing is a double-edged sword. While it allows you to mitigate various **Feature Risks**, by adding test-code to your project you are necessarily increasing the **complexity**. Maintaining tests is hard work, and if you're not careful, *running* tests can take time and slow down builds and add delay through **Process Risk**.

Third, if you are **exploring functionality** in order to flush out requirements, understand user behaviour or figure out performance characteristics, then there is *no point in building tests* yet: what you are doing is exploratory at best and the extra code will **slow you down**.

For these reasons, focus on writing the *smallest number of tests that mitigates the risks*.

See Also

Risk Based Agile Testing[14] by Martin Ivison, which covers a lot of this ground in much more detail.

[14]https://www.amazon.co.uk/Risk-Driven-Agile-Testing-risk-based-effective-ebook/dp/B06XGL4CDL/ref=sr_1_1?ie=UTF8&qid=1521908627&sr=8-1&keywords=risk+based+agile+testing

Part IV

Methodologies

Chapter 16

Methodologies

Thinking is hard. And worrying about Risk constantly would be *exhausting*.

Life is too short to go around considering Risk Management over everything you do.

Luckily, our brains do it for us automatically and subconsiously: Sometimes a voice inside will cry out Wait! before we walk out into a road or try to pick up a hot teapot.

Habit and Experience

These subconsious reactions are borne of two things: **habit** (we drill our children from an early age in crossing the road to embed road safety) and **experience** (after picking up lots of too-hot kitchenware, you don't do it again).

In this section (on Methodology), we're going to focus on how **habit** can help us short-cut the Risk Management process, whilst in the next we'll look at **experience**.

When it sticks, a methodology embeds a set of practices in a team to such an extent that they become *habit*: the team following the methodology from feature to feature or release to release without question.

A Pattern Language

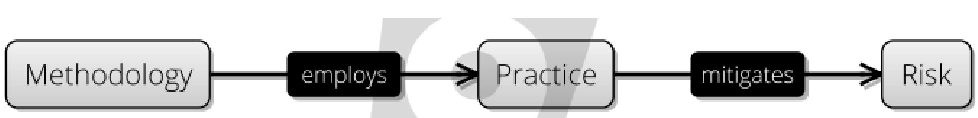


Figure 16.1: A Pattern Language

It stands to reason that if **all software is about risk management**, then we can examine methodologies *themselves* in terms of how their practices mitigate risk, and change the balance of risk on projects.

With that in mind, we are going to examine several methodologies, and break them down into their key *practices*. For each practice, we will look at which **attendant risks** it mitigates, and what **attendant risks** it incurs.

!Show similarity between pattern and practice

So *Methodology* exists as a way

ceremony practices bureacratic overhead

a point of religion.

The questions we want to ask in this section are as follows:

-How do frameworks change the risk landscape?

What are the risks in choosing a framework?

How does choosing a framework (at all) modify our risk landscape?

How should we choose a framework, then?

Evolution of software

There are more methodologies than stars in the sky, and it's not useful to look at all of them. Instead, we're going to pick a few *archetypes* and leave it at that.

So, let's start at the beginning then, with **Waterfall**.

It's the same steps, but it's Sizing those steps:

Agile is per-feature delivery, Waterfall is a bunch of features.

But, a lot of the practices end up being the same.

Chapter 17

Waterfall

Waterfall[1] is a linear, stepwise approach to the processes involved in delivering a software system, and it really represents a family of methodologies, such as RUP[2] or SSADM[3].

Major Practices

The specifics differ from one formulation to another, but generally speaking the process looks something like this:

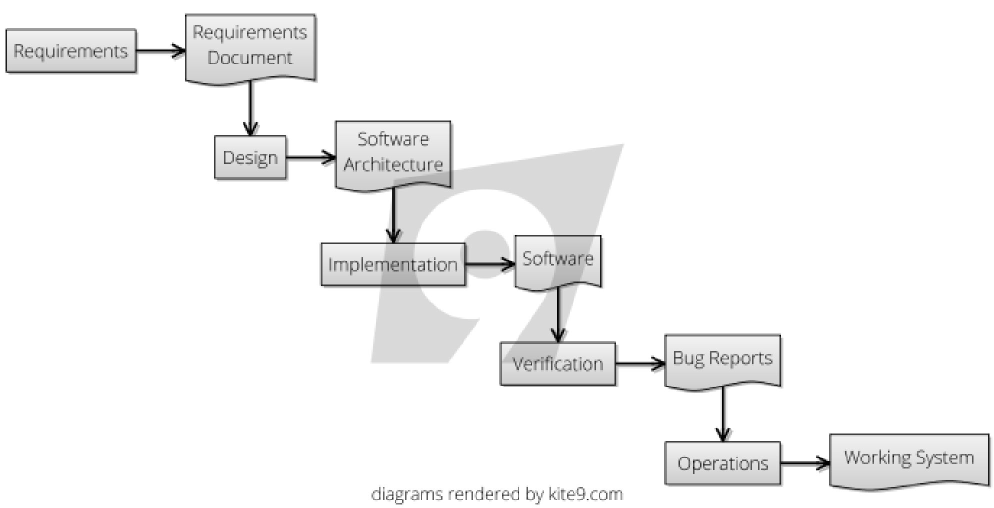


Figure 17.1: Waterfall Methodology

[1] https://en.wikipedia.org/wiki/Waterfall_model
[2] https://en.wikipedia.org/wiki/Rational_Unified_Process
[3] https://en.wikipedia.org/wiki/Structured_systems_analysis_and_design_method

As shown in the diagram above, the software process is broken into distinct stages, usually including:

- **Requirements Capture**
- **Design**
- **Implementation**
- **Verification**
- **Delivery** and **Operations**
- **Sign Offs** at each stage

Variations

- **Prototyping**: Picking a particularly high-risk part of the project (such as UI elements) and delivering it first.

- **Business Case**: Adding a stage in the at the start of the project to perform some benefits calculations.
- **Cycles**: Delivering in multiple, incremental stages.

Risks Mitigated

1. Cost Of Implementation

It's likely that the Waterfall-Style methodologies were inspired by the construction industry, wherein we try to **Design Up Front** in order to avoid the cost of re-work: once concrete is poured, it's expensive to change it again, compared to the cost of updating the design in a diagram.

Also, when Waterfall was originally conceived, automated testing techniques were not well established. If you expect to perform a large **manual testing cycle** for each release, then clearly, doing fewer releases looks cheaper on paper.

But, while *in principle*, Waterfall aims to *contain* the cost of implementation. However, in practice, because of **Requirements Drift**, **Student Syndrome** and **Complexity Risk**, the schedules get more inaccurate the larger the project.

2. Lots Of Stakeholders

In any construction project, there are likely to be lots of stakeholders - landowners, neighbours, government, clients and so on.

Waterfall tries to mitigate this risk by getting **Sign-Offs** as it goes along.

Additionally, by putting in the work at the planning and design stage, hopefully this means lots of staff can work together and not interfere with each other when the time for construction comes.

4. Agency Risk

Because of it's step-wise delivery and reduction in visibility risk, Waterfall documentation can be used as the basis for **contracted delivery**, and this is useful in situations where you are employing 3rd parties or putting work to tender.

This is very different from the way **Agency Risk** is mitigated in, say **Scrum**, which relies on the **On Site Customer** to police the implementation team.

5. Bureaucratic Risk

Where projects can get tied up in lots of red tape, a Waterfall process can supply enough gravitas in the form of documentation and ceremony in order to appease bureaucracy, in a way that **Lean** or **Agile** methods do not.

Additionally, because a **plan** can be based on the **Design**, you can include bureaucratically-onerous tasks in the plan and work on these in parallel.

Attendant Risks

1. Complexity Risk

One of the biggest problems in sticking to a **Design**, rather than letting the design evolve, is that you are not going to be practicing **Refactoring** in order to keep down

2. Production Risk

The fewer different **phases or cycles** in your project, the fewer times you will **Meet Reality**