Quick Summary

1. There are Lots of Ways to Run Software Projects

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

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

None of these is necessarily more "right" than another: they are suitable on different projects at different times.

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

2. We Can Look at Projects in Terms of Risks

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

Commonly, tools such as RAID logs[1] and RAG status[2] reporting are used. These techniques should be familiar to project managers and developers everywhere.

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

For example:

[1]`https://www.projectmanager.com/blog/raid-log-use-one`

[2]`https://pmtips.net/blog-new/what-does-rag-status-mean`

- A story about improving the user login screen can be seen as reducing *the risk of users not signing up*.
- A task about improving the health indicators could be seen as mitigating *the risk of the application failing and no-one reacting to it*.
- Even a task as basic as implementing a new function in the application is mitigating *the risk that users are dissatisfied and go elsewhere*.

One assertion of Risk-First is that **every action you take on a project is to manage a risk.**

3. We Can Break Down Risks on a Project Methodically

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

For example:

- In manufacturing, *tolerances* allow for calculating the likelihood of defects in production.
- In finance, projects and teams are structured around monitoring risks like *credit risk*, *market risk* and *liquidity risk*.
- *Insurance* is founded on identifying particular risks and providing financial safety-nets for when they occur, such as death, injury, accident and so on.

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

With this in place, we can:

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

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

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

For example:

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

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

5. Different Methodologies are for Different Risk Profiles

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

For example:

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

Although many developers have a methodology-of-choice, the argument here is that there are trade-offs with all of these choices.

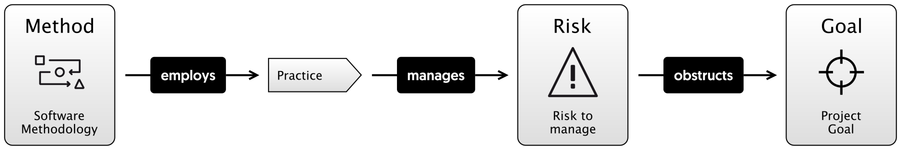

Figure 1: Methodologies, Risks, Practices

"Methodologies are like *bicycles*, rather than *religions*. Rather than simply *believing*, we can take them apart and see how they work. "

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

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

How do we take this further?

One idea explored is the *Risk Landscape*: although the software team can't remove risk from their project, they can take actions that move them to a place in the Risk Landscape where the risks on the project are more favourable than where they started.

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

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

For example:

- If we are introducing a **Sign-Off** in our process, we have to balance the risks it *mitigates* (coordination of effort, quality control, information sharing) with the risks it *introduces* (delays and process bottlenecks).
- If we build in **Redundancy**, this mitigates the risk of a *single point of failure*, but introduces risks around *synchronizing data* and *communication* between the systems.

- If we introduce **Process**, this may make it easier to *coordinate as a team* and *measure performance* but may lead to bureaucracy, focusing on the wrong goals or over-rigid interfaces to those processes.

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

Still interested? Then dive into reading the introduction.

Part I

Introduction

CHAPTER 1

A Simple Scenario

In this chapter, I'm going to introduce some terms for thinking about risk.

For a moment forget about software completely, and think about *any endeavour at all* in life. It could be passing a test, mowing the lawn or going on holiday. Choose something now. I'll discuss from the point of view of "cooking a meal for some friends", but you can play along with your own example.

1.1 Goal In Mind

Now, in this endeavour, we want to be successful. That is to say, we have a **Goal In Mind**: we want our friends to go home satisfied after a decent meal, and not to feel hungry. As a bonus, we might also want to spend time talking with them before and during the meal. So, now to achieve our Goal In Mind we *probably* have to do some tasks.

Since our goal only exists *in our head*, we can say it is part of our **Internal Model** of the world. That is, the model we have of reality. This model extends to *predicting what will happen*.

If we do nothing, our friends will turn up and maybe there's nothing in the house for them to eat. Or maybe, the thing that you're going to cook is going to take hours and they'll have to sit around and wait for you to cook it and they'll leave before it's ready. Maybe you'll be some ingredients short, or maybe you're not confident of the steps to prepare the meal and you're worried about messing it all up.

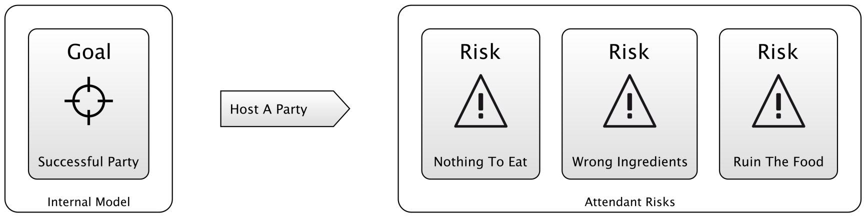

Figure 1.1: Goal In Mind, with the risks you know about

1.2 Attendant Risk

These *nagging doubts* that are going through your head are what I'll call the Attendant Risks: they're the ones that will occur to you as you start to think about what will happen.

When we go about preparing for this wonderful evening, we can choose to deal with these risks: shop for the ingredients in advance, prepare parts of the meal and maybe practice the cooking in advance. Or, we can wing it, and sometimes we'll get lucky.

How much effort we expend on these Attendant Risks depends on how big we think they are. For example, if you know there's a 24-hour shop, you'll probably not worry too much about getting the ingredients well in advance (although, the shop *could still be closed*).

1.3 Hidden Risks

Attendant Risks are risks you are aware of. You may not be able to exactly *quantify* them, but you know they exist. But there are also **Hidden Risks** that you *don't* know about: if you're poaching eggs for dinner, perhaps you didn't know that fresh eggs poach best. Donald Rumsfeld famously called these kinds of risks "Unknown Unknowns":

> "Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know. And if one looks throughout

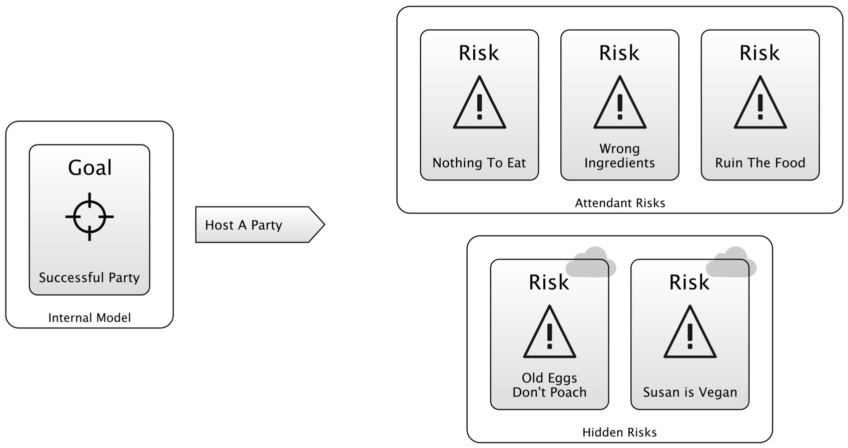

Figure 1.2: Goal In Mind, the risks you know about and the ones you don't

> the history of our country and other free countries, it is the latter category that tend to be the difficult ones."
>
> —Donald Rumsfeld, *Wikipedia*[1]

Different people evaluate risks differently, and they'll also *know* about different risks. What is an Attendant Risk for one person is a Hidden Risk for another.

Which risks we know about depends on our **knowledge** and **experience**, then. And that varies from person to person (or team to team).

1.4 Meeting Reality

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

Our model is forced to Meet Reality, and the model changes, forcing us to deal with these risks, as shown in Figure 1.3. Whenever we try to *do something* about a risk, it is called Taking Action. Taking Action *changes* reality, and with it your Internal Model of the risks you're facing. That's because it's only by

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

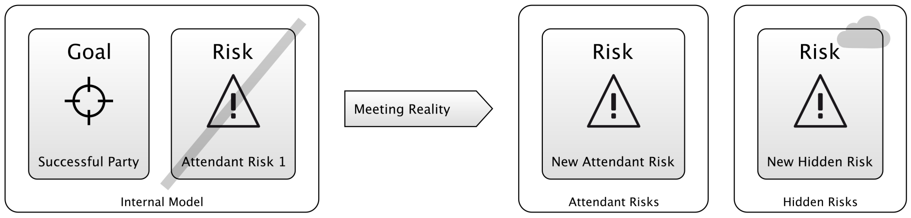

Figure 1.3: How Taking Action affects Reality, and also changes your Internal Model

interacting with the world that we add knowledge to our Internal Model about what works and what doesn't. Even something as passive as *checking the shop opening times* is an action, and it improves on our Internal Model of the world.

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

1.5 On To Software

Here, we've introduced some new terms that we're going to use a lot: Meet Reality, Attendant Risk, Hidden Risk, Internal Model, Taking Action and Goal In Mind. And, we've applied them in a simple scenario.

But Risk-First is about understanding risk in software development, so let's examine the scenario of a new software project, and expand on the simple model being outlined above: instead of a single person, we are likely to have a team, and our model will not just exist in our heads, but in the code we write.

On to Development Process. . .

CHAPTER 2

Development Process

In the previous chapter we introduced some terms for talking about risk (such as Attendant Risk, Hidden Risk and Internal Model) via a simple scenario.

Now, let's look at the everyday process of developing *a new feature* on a software project, and see how our risk model informs it.

2.1 A Toy Process

Let's ignore for now the specifics of what methodology is being used - we'll come to that later. Let's say your team have settled for a process something like the following:

1. **Specification**: a new feature is requested somehow, and a business analyst works to specify it.
2. **Code And Unit Test**: a developer writes some code, and some unit tests.
3. **Integration**: they integrate their code into the code base.
4. **UAT**: they put the code into a User Acceptance Test (UAT) environment, and user(s) test it.
5. . . . All being well, the code is **Released to Production**.

Can't We Improve This?

Is this a *good* process? Probably, it's not that great: you could add code review, a pilot phase, integration testing, whatever.

Also, the *methodology* being used might be Waterfall, it might be Agile. We're not going to commit to specifics at this stage.

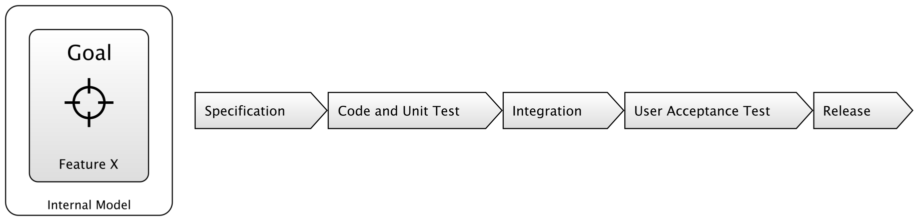

Figure 2.1: A Simple Development Process

For now though, let's just assume that *it works for this project* and everyone is reasonably happy with it.

We're just doing some analysis of *what process gives us*.

Minimising Risks - Overview

I am going to argue that this entire process is *informed by software risk*:

1. We have *a business analyst* who talks to users and fleshes out the details of the feature properly. This is to minimize the risk of **building the wrong thing**.
2. We *write unit tests* to minimize the risk that our code **isn't doing what we expected, and that it matches the specifications**.
3. We *integrate our code* to minimize the risk that it's **inconsistent with the other, existing code on the project**.
4. We have *acceptance testing* and quality gates generally to **minimize the risk of breaking production**, somehow.

A Much Simpler Process

We could skip all those steps above and just do this:

1. Developer gets wind of new idea from user, logs onto production and changes some code directly.

We can all see this might end in disaster, but why?

Two reasons:

1. You're Meeting Reality all-in-one-go: all of these risks materialize at the same time, and you have to deal with them all at once.

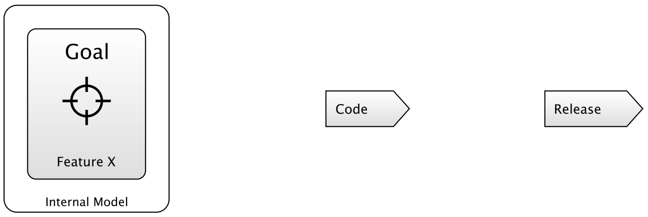

Figure 2.2: A Dangerous Development Process

2. Because of this, at the point you put code into the hands of your users, your Internal Model is at its least-developed. All the Hidden Risks now need to be dealt with at the same time, in production.

2.2 Applying the Process

Let's look at how our process should act to prevent these risks materializing by considering an unhappy path, one where at the outset, we have lots of Hidden Risks. Let's say a particularly vocal user rings up someone in the office and asks for new **Feature X** to be added to the software. It's logged as a new feature request, but:

- Unfortunately, this feature once programmed will break an existing **Feature Y**.
- Implementing the feature will use some api in a library, which contains bugs and have to be coded around.
- It's going to get misunderstood by the developer too, who is new on the project and doesn't understand how the software is used.
- Actually, this functionality is mainly served by **Feature Z**. . .
- which is already there but hard to find.

Figure 2.3 shows how this plays out.

This is a slightly contrived example, as you'll see. But let's follow our feature through the process and see how it meets reality slowly, and the Hidden Risks are discovered:

Specification

The first stage of the journey for the feature is that it meets the Business Analyst (BA). The *purpose* of the BA is to examine new goals for the project

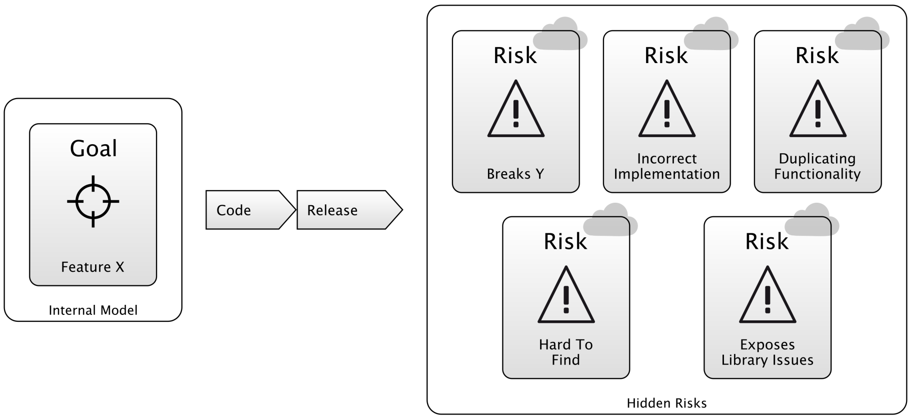

Figure 2.3: Development Process - Exposing Hidden Risks

and try to integrate them with *reality as they understand it*. A good BA might take a feature request and vet it against his Internal Model, saying something like:

- "This feature doesn't belong on the User screen, it belongs on the New Account screen"
- "90% of this functionality is already present in the Document Merge Process"
- "We need a control on the form that allows the user to select between Internal and External projects"

In the process of doing this, the BA is turning the simple feature request *idea* into a more consistent, well-explained *specification* or *requirement* which the developer can pick up. But why is this a useful step in our simple methodology? From the perspective of our Internal Model, we can say that the BA is responsible for:

- Trying to surface Hidden Risks
- Trying to evaluate Attendant Risks and make them clear to everyone on the project.

In surfacing these risks, there is another outcome: while **Feature X** might be flawed as originally presented, the BA can "evolve" it into a specification, and tie it down sufficiently to reduce the risks. The BA does all this by simply *thinking about it*, *talking to people* and *writing stuff down*.

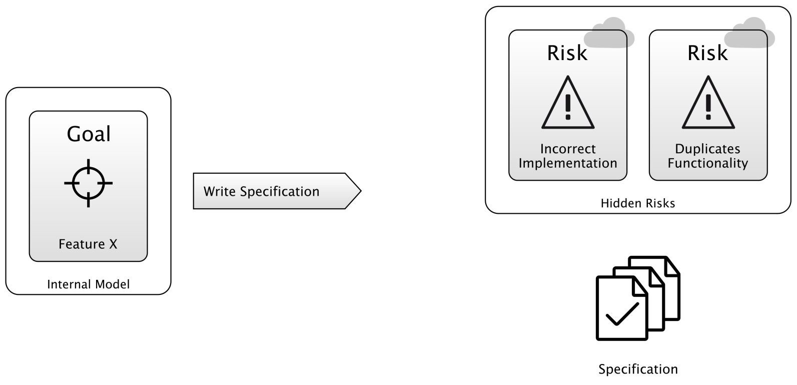

Figure 2.4: BA Specification: exposing Hidden Risks as soon as possible

This process of evolving the feature request into a requirement is the BA's job. From our Risk-First perspective, it is *taking an idea and making it Meet Reality*. Not the *full reality* of production (yet), but something more limited.

Code And Unit Test

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

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

At this point, let's stop and discuss the visual grammar of the Risk-First Diagrams we've been looking at. A Risk-First diagram shows what you expect to happen when you Take Action. The action itself is represented by the shaded, sign-post-shaped box in the middle. On the left, we have the current state of the world, on the right is the anticipated state *after* taking the action.

The round-cornered rectangles represent our Internal Model, and these contain our view of Risk, whether the risks we face right now, or the Attendant Risks expected after taking the action. In Figure 2.5, taking the action of

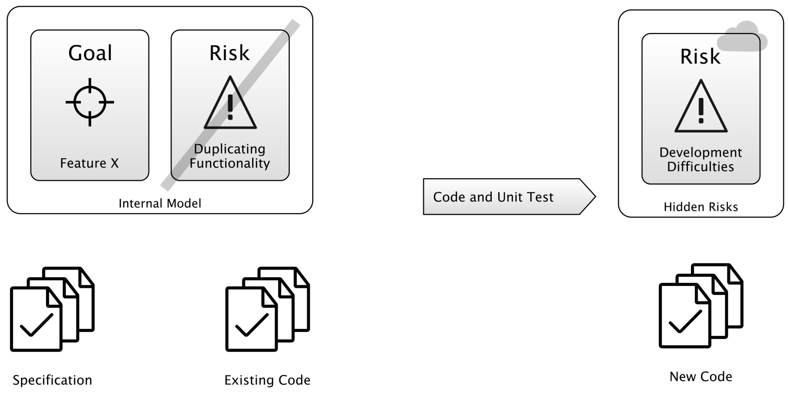

Figure 2.5: Coding Process: exposing more hidden risks as you code

"coding and unit testing" is expected to mitigate the risk of "Duplicating Functionality".

Beneath the internal models, we are also showing real-world tangible artifacts. That is, the physical change we would expect to see as a result of taking action. In Figure 2.5, the action will result in "New Code" being added to the project, needed for the next steps of the development process.

Integration

Integration is where we run *all* the tests on the project, and compile *all* the code in a clean environment, collecting together the work from the whole development team.

So, this stage is about meeting a new reality: the clean build.

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

User Acceptance Test

Next, User Acceptance Testing (UAT) is where our new feature meets another reality: *actual users*. I think you can see how the process works by now. We're just flushing out yet more Hidden Risks.

- Taking Action is the *only* way to create change in the world.
- It's also the only way we can *learn* about the world, adding to our Internal Model.

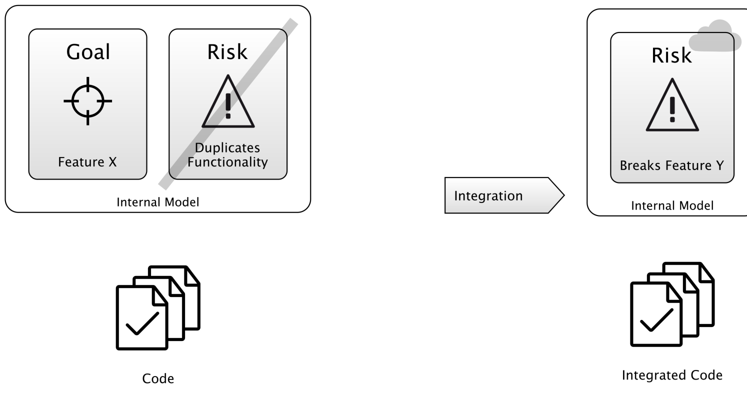

Figure 2.6: Integration testing exposes Hidden Risks before you get to production

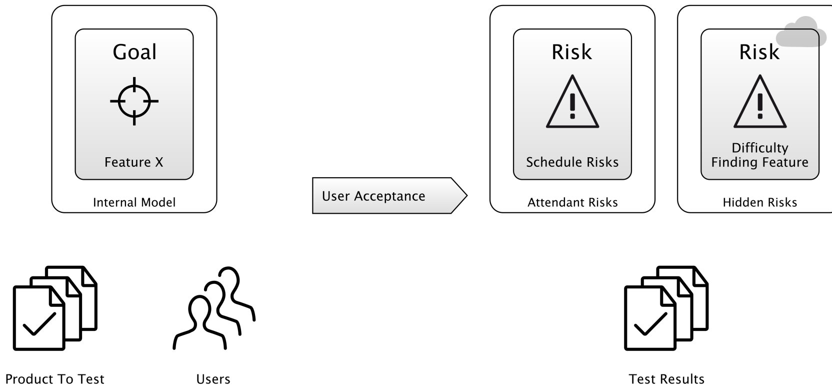

Figure 2.7: UAT - putting tame users in front of your software is better than real ones, where the risk is higher

- In this case, we discover a Hidden Risk: the user's difficulty in finding the feature. (The cloud obscuring the risk shows that it is hidden).
- In return, we can *expect* the process of performing the UAT to delay our release (this is an attendant schedule risk).

2.3 Observations

First, the people setting up the development process *didn't know* about these *exact* risks, but they knew the *shape that the risks take*. The process builds "nets" for the different kinds of Hidden Risks without knowing exactly what they are.

Second, are these really risks, or are they *problems we just didn't know about*? I am using the terms interchangeably, to a certain extent. Even when you know you have a problem, it's still a risk to your deadline until it's solved. So, when does a risk become a problem? Is a problem still just a schedule-risk, or cost-risk? We'll come back to this question presently.

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

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

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

CHAPTER 3

Meeting Reality

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

3.1 Revisiting the Model

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

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

It should be no surprise to see that there is a *recursive* nature about this: the actions we take each day have consequences, they expose new hidden risks which inform our Internal Model and at the same time change reality in some way. As a result, we then have to take *new actions* to deal with these new risks.

So, let's see how this kind of recursion looks on our model.

3.2 Navigating the "Risk Landscape"

Figure 3.1 shows *just one possible action*, in reality, you'll have choices. We often have multiple ways of achieving a Goal In Mind.

What's the best way?

I would argue that the best way is the one which mitigates the most existing risk while accruing the least attendant risk to get it done.

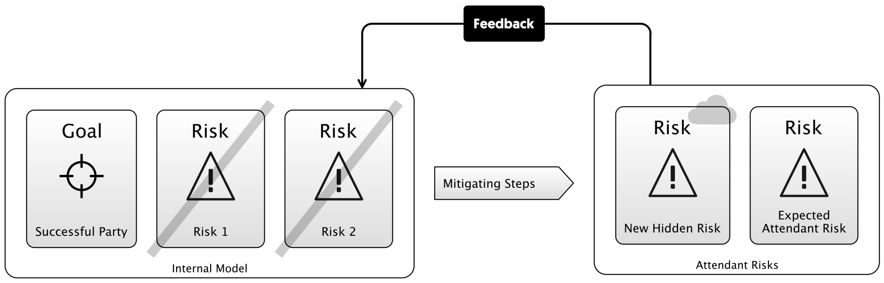

Figure 3.1: Taking actions changes reality, but changes your model of the risks too

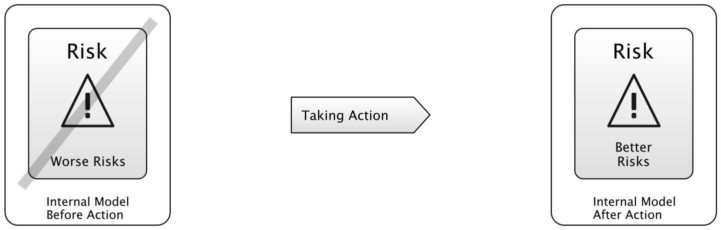

Figure 3.2: Navigating The Risk Landscape

Ideally, when you take an action, you are trading off a big risk for a smaller one. Take Unit Testing for example. Clearly, writing Unit Tests adds to the amount of development work, so on its own, it adds Schedule Risk. However, if you write *just enough* of the right Unit Tests, you should be short-cutting the time spent finding issues in the User Acceptance Testing (UAT) stage, so you're hopefully trading off a larger Schedule Risk from UAT and adding a smaller Schedule Risk to Development. There are other benefits of Unit Testing too: once written, a suite of unit tests is almost cost-free to run repeatedly, whereas repeating a UAT is costly as it involves people's time.

You can think of Taking Action as moving your project on a "Risk Landscape": ideally, when you take an action, you move from some place with worse risk to somewhere more favourable.

Sometimes, you can end up somewhere *worse*: the actions you take to manage a risk will leave you with worse Attendant Risks afterwards. Almost certainly, this will have been a Hidden Risk when you embarked on the action, otherwise you'd not have chosen it.

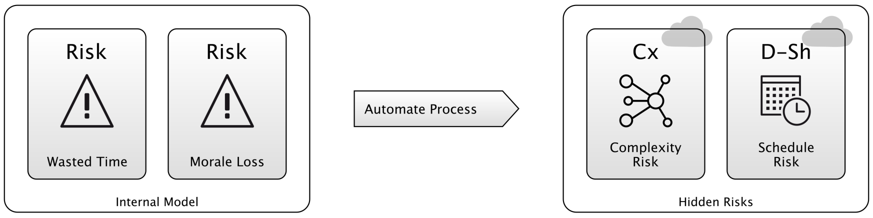

Figure 3.3: Hidden Risks of Automation

An Example: Automation

For example, *automating processes* is very tempting: it *should* save time, and reduce the amount of boring, repetitive work on a project. But sometimes, it turns into an industry in itself, and consumes more effort than it's worth.

Another Example: MongoDB

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

Other options were available, but the developer chose MongoDB because of their *existing familiarity* with it: therefore, they felt that the Hidden Risks of MongoDB were *lower* than the other options, and disregarded the others' opinions.

This turned out to be a mistake: The internal bureacracy eventually proved too great, and MongoDB had to be abandoned after much investment of time.

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

3.3 Payoff

We can't know in advance how well any action we take will work out. Therefore, Taking Action is a lot like placing a bet.

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

Payoff then is our judgement about whether we expect an action to be worthwhile: are the risks we escape *worth* the attendant risks we will encounter? We should be able to *weigh these separate risks in our hands* and figure out whether the Payoff makes a given Action worthwhile.

The fruits of this gambling are revealed when we meet reality, and we can see whether our bets were worthwhile.

3.4 The Cost Of Meeting Reality

Meeting reality *in full* is costly. For example, going to production can look like this:

- Releasing software
- Training users
- Getting users to use your system
- Gathering feedback

All of these steps take a lot of effort and time. But you don't have to meet the whole of reality in one go. But we can meet it in a limited way which is less expensive.

In all, to de-risk, you should try and meet reality:

- **Sooner**: so you have time to mitigate the hidden risks it uncovers.
- **More Frequently**: so the hidden risks don't hit you all at once.
- **In Smaller Chunks**: so you're not over-burdened by hidden risks all in one go.
- **With Feedback**: if you don't collect feedback from the experience of meeting reality, hidden risks *stay hidden*.

In Development Process, we performed a UAT in order to Meet Reality more cheaply and sooner. The *cost* of this is that we delayed the release to do it, adding risk to the schedule.

3.5 Practice 1: YAGNI

As a flavour of what's to come, let's look at YAGNI, an acronym for You Aren't Gonna Need It:

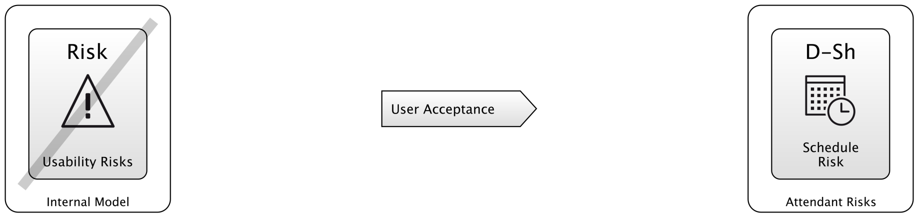

Figure 3.4: Testing flushes out Hidden Risk, but increases Schedule Risk

> YAGNI originally is an acronym that stands for "You Aren't Gonna Need It". It is a mantra from Extreme Programming that's often used generally in agile software teams. It's a statement that some capability we presume our software needs in the future should not be built now because "you aren't gonna need it".
>
> —YAGNI, *Martin Fowler*[2]

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

But, there is always the opposite opinion: You *Are* Gonna Need It[3]. As a simple example, we often add log statements in our code as we write it (so we can trace what happened when things go wrong), though following YAGNI strictly says we shouldn't.

Which is right?

Now, we can say: do the work *if there is a worthwhile Payoff*.

- Logging statements are *good*, because otherwise, you're increasing the risk that in production, no one will be able to understand *how the software went wrong*.
- However, adding them takes time, which might introduce Schedule Risk.

So, it's a trade-off: continue adding logging statements so long as you feel that overall, the activity pays off reducing overall risk.

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

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

3.6 Practice 2: Do The Simplest Thing That Could Possibly Work

Another mantra from Kent Beck (originator of the Extreme Programming[4] methodology), is "Do The Simplest Thing That Could Possibly Work", which is closely related to YAGNI and is an excellent razor for avoiding over-engineering. At the same time, by adding "Could Possibly", Kent is encouraging us to go beyond straightforward iteration, and use our brains to pick apart the simple solutions, avoiding them if we can logically determine when they would fail.

Our risk-centric view of this strategy would be:

- Every action you take on a project has its own Attendant Risks.
- The bigger or more complex the action, the more Attendant Risk it'll have.
- The reason you're taking action *at all* is because you're trying to reduce risk elsewhere on the project
- Therefore, the biggest Payoff is likely to be the one with the least Attendant Risk.
- So, usually this is going to be the simplest thing.

So, "Do The Simplest Thing That Could Possibly Work" is really a helpful guideline for Navigating the Risk Landscape, but this analysis shows clearly where it's left wanting:

- *Don't* do the simplest thing if there are other things with a better Payoff available.

3.7 Summary

So, here we've looked at Meeting Reality, which basically boils down to taking actions to manage risk and seeing how it turns out:

- Each Action you take is a step on the Risk Landscape
- Each Action exposes new Hidden Risks, changing your Internal Model.
- Ideally, each action should reduce the overall Attendant Risk on the project (that is, puts it in a better place on the Risk Landscape

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

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

CHAPTER 4

Just Risk

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

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

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

4.1 RAID Log

Many project managers will be familiar with the RAID Log. It's simply four columns on a spreadsheet: **Risks**, **Actions**, **Issues** and **Decisions**.

Let's try and put the following Risk into the RAID Log:

> "Debbie needs to visit the client to get them to choose the logo to use on the product, otherwise we can't size the screen areas exactly."

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

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

4.2 Let's Go Again

This is a completely made-up example, deliberately chosen to be hard to categorise. Normally, items are more one thing than another. But often, you'll have to make a choice between two categories, if not all four.

This *hints* at the fact that at some level it's all about risk:

4.3 Every Action Attempts to Mitigate Risk

The reason you are *taking* an action is to mitigate a risk. For example:

- If you're coding up new features in the software, this is mitigating Feature Risk (which we'll explore in more detail later).
- If you're getting a business sign-off for something, this is mitigating the risk of everyone not agreeing on a course of action (a Coordination Risk).
- If you're writing a specification, then that's mitigating the type of "Incorrect Implementation Risk" we saw in the last chapter.

4.4 Every Action Has Attendant Risk

- How do you know if the action will get completed?
- Will it overrun, or be on time?
- Will it lead to yet more actions?
- What Hidden Risk will it uncover?

Consider *coding a feature* (as we did in the earlier Development Process chapter). We saw here how the whole process of coding was an exercise in learning what we didn't know about the world, uncovering problems and improving our Internal Model. That is, flushing out the Attendant Risk of the Goal In Mind.

And, as we saw in the Introduction, even something *mundane* like the Dinner Party had risks.

4.5 An Issue is Just A Type of Risk

- Because issues need to be fixed. . .
- And fixing an issue is an action. . .
- Which, as we just saw also carries risk.

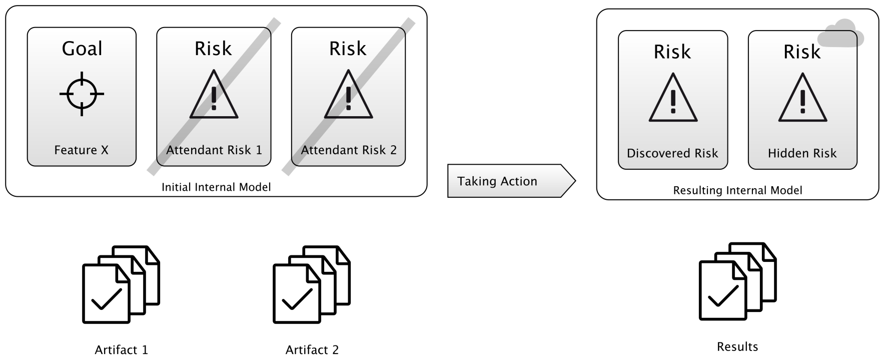

Figure 4.1: Risk-First Diagram Language

One retort to this might be to say: "an issue is a problem I have now, whereas a risk is a problem that *might* occur." I am going to try and break that mind-set in the coming pages, but I'll just start with this:

- Do you know *exactly* how much damage this will do?
- Can you be sure that the issue might not somehow go away?

Issues then, just seem more "definite" and "now" than *risks*, right? This classification is arbitrary: they're all just part of the same spectrum, they all have inherent uncertainty, so there should be no need to agonise over which column to put them in.

4.6 Goals Are Risks Too

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

Goals live inside our Internal Model, just like Risks. It turns out, that functionally, Goals and Risks are equivalent. For example, The Goal of "Implementing Feature X" is equivalent to mitigating "Risk of Feature X not being present".

Let's try and back up that assertion with a few more examples:

Goal	Restated As A Risk
Build a Wall	Mitigate the risk of something getting in / out

Goal	Restated As A Risk
Land a man on the moon	Mitigate the risk of looking technically inferior during the cold war
Move House	Mitigate the risks/problems of where you currently live

There is a certain "interplay" between the concepts of risks, actions and goals. After all, on the Risk Landscape they correspond to a starting point, a movement, and a destination. From a redundancy perspective, any one of these can be determined by knowing the other two.

Psychologically, humans are very goal-driven: they like to know where they're going, and are good at organising around a goal. However, by focusing on goals ("solutionizing") it's easy to ignore alternatives. By focusing on "Risk-First", we don't ignore the reasons we're doing something.

4.7 Every Decision is About Payoff

Sometimes, there will be multiple moves available on the Risk Landscape and you have to choose one.

- There's the risk you'll decide wrongly.
- And, making a decision takes time, which could add risk to your schedule.
- And what's the risk if the decision doesn't get made?

Let's take a hypothetical example: you're on a project and you're faced with the decision - release now or do more testing?

Obviously, in the ideal world, we want to get to the place on the Risk Landscape where we have a tested, bug-free system in production. But we're not there yet, and we have funding pressure to get the software into the hands of some paying customers. The table below shows an example:

Risk Managed	Action	Attendant Risk	Payoff
Funding Risk	Go Live	Reputational Risk, Operational Risk	MEDIUM
Implementation Risk	User Acceptance Test	Worse Funding Risk, Operational Risk	LOW

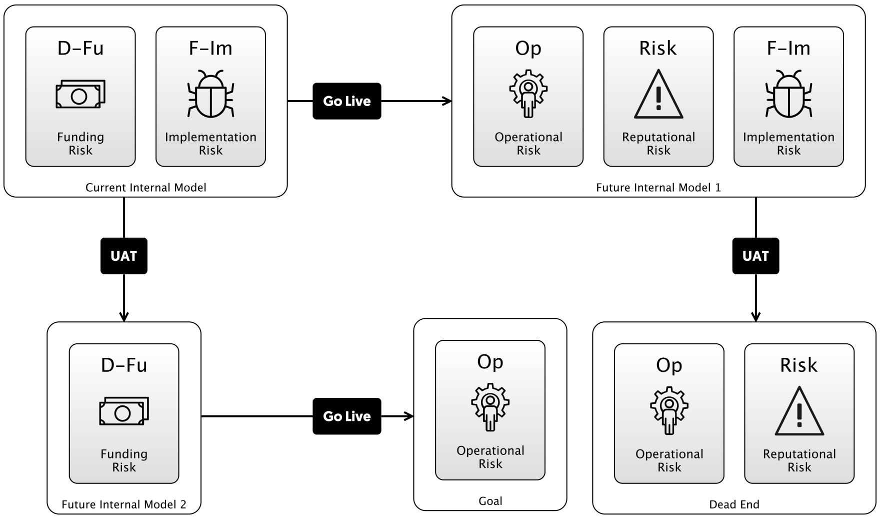

Figure 4.2: UAT or Go Live: where will you end up?

This is (a simplification of) the dilemma of lots of software projects - *test further*, to reduce the risk of users discovering bugs (Implementation Risk) which would cause us reputational damage, or *get the release done* and reduce our Funding Risk by getting paying clients sooner.

In the above table, it *appears* to be better to do the "Go Live" action, as there is a greater Payoff. The problem is, actions are not *commutative*, i.e. the order you do them in counts.

Figure 4.2 shows our decision as *moves on the Risk Landscape*. Whether you "Go Live" first, or "UAT" first makes a difference to where you will end up. Is there a further action you can take to get you from the "Dead End" to the "Goal"? Perhaps.

Failure

So, when we talk about a project "failing", what do we mean?

Usually, we mean we've failed to achieve a goal, and since *goals are risks*, it is simply the scenario where we are overwhelmed by Attendant Risks: there is *no* action to take that has a good-enough Payoff to get us out of our hole.

4.8 What To Do?

It makes it much easier to tackle the RAID log if there's only one list. But you still have to choose a *strategy*: do you tackle the *most important* risk on the list, or the *most urgent*, or take the action with the biggest Payoff and deal with it?

In the next chapter, Evaluating Risk we'll look at some approaches to choosing what to do.

CHAPTER 5

Cadence

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

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

How *fast* should we go round this loop? The longer you leave your goal in mind, the longer it'll be before you find out how it really stacks up against reality.

Testing your goals in mind against reality early and safely is how you'll manage risk effectively, and to do this, you need to set up **Feedback Loops**. e.g.

- **Bug Reports and Feature Requests** tell you how the users are getting on with the software.
- **Monitoring Tools and Logs** allow you to find out how your software is doing in reality.

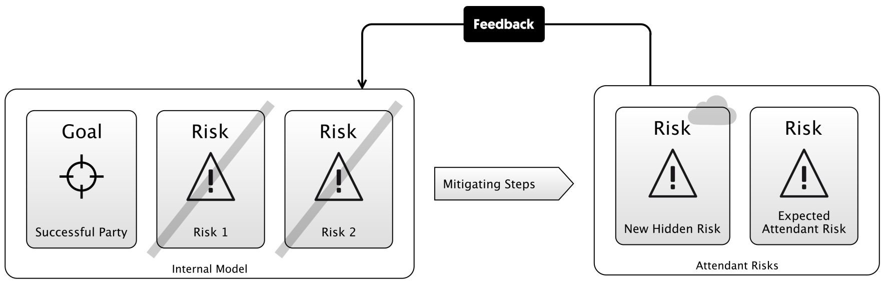

Figure 5.1: Meeting Reality: reality is changed and so is your internal model.

- **Dog-Fooding** i.e using the software you write yourself might be faster than talking to users.
- **Continuous Delivery**[1] is about putting software into production as soon as it's written.
- **Integration Testing** is a faster way of meeting *some* reality than continually deploying code and re-testing it manually.
- **Unit Testing** is a faster feedback loop than Integration Testing.
- **Compilation** warns you about logical inconsistencies in your code.

.. and so on.

Time / Reality Trade-Off

This list is arranged so that at the top, we have the most visceral, most *real* feedback loop, but at the same time, the slowest.

At the bottom, a good IDE can inform you about errors in your Internal Model in real time, by way of highlighting compilation errors . So, this is the fastest loop, but it's the most *limited* reality.

Imagine for a second that you had a special time-travelling machine. With it, you could make a change to your software, and get back a report from the future listing out all the issues people had faced using it over its lifetime, instantly.

That'd be neat, eh? If you did have this, would there be any point at all in a compiler? Probably not, right?

The whole *reason* we have tools like compilers is because they give us a short-cut way to get some limited experience of reality *faster* than would otherwise be possible. Because cadence is really important: the faster we test our ideas, the more quickly we'll find out if they're correct or not.

Development Cycle Time

Developers often ignore the fast feedback loops at the bottom of the list above because the ones nearer the top *will do*.

In the worst cases this means changing two lines of code, running the build script, deploying and then manually testing out a feature. And then repeating. Doing this over and over is a terrible waste of time because the feedback loop is so long and you get none of the benefit of a permanent suite of tests to run again in the future.

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

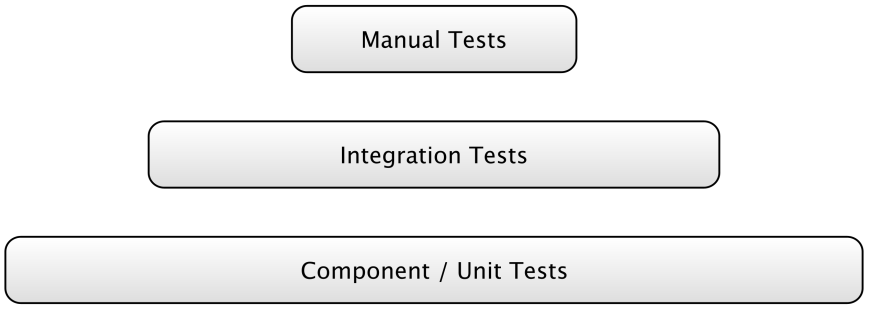

Figure 5.2: The Testing Pyramid

The Testing Pyramid[2] hints at this truth:

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

Production

You could take this chapter to mean that Continuous Delivery (CD) is always and everywhere a good idea. That's not a bad take-away, but it's clearly more nuanced than that.

Yes, CD will give you faster feedback loops, but even getting things into production is not the whole story: the feedback loop isn't complete until people have used the code, and reported back to the development team.

The right answer is to use multiple feedback loops, as shown in Figure 5.3.

In the next chapter De-Risking we're going to introduce a few more useful terms for thinking about risk.

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

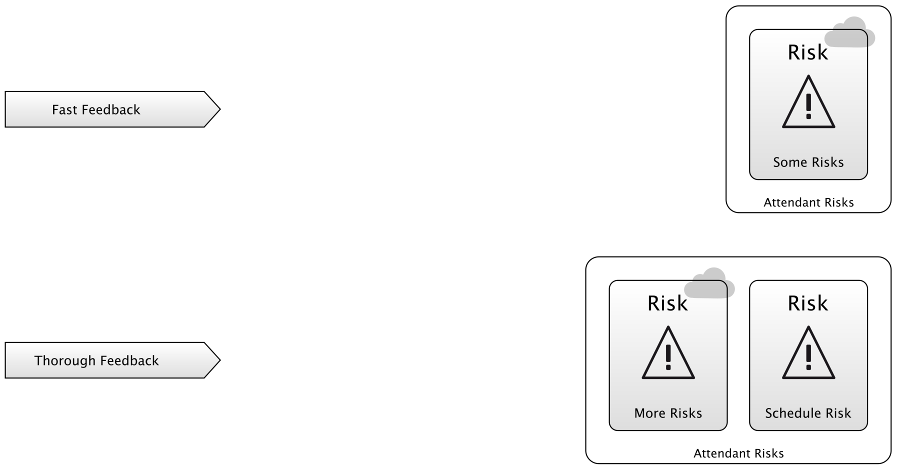

Figure 5.3: Different actions have different feedback loops

CHAPTER 6

One Size Fits No-One

Why are Software Methodologies[1] all different?

Previously, we made the case that any action you take on a software project is to do with managing risk, and the last chapter, A Conversation was an example of this happening.

Therefore, it stands to reason that software methodologies are all about handling risk too. Since they are prescribing a particular day-to-day process, or set of actions to take, they are also prescribing a particular approach to managing the risks on software projects.

6.1 Methodologies Surface Hidden Risks. . .

Back in the Development Process chapter we introduced a toy software methodology that a development team might follow when building software. It included steps like *analysis*, *coding* and *testing*. We looked at how the purpose of each of these actions was to manage risk in the software delivery process. For example, it doesn't matter if a developer doesn't know that he's going to break "Feature Y", because the *Integration Testing* part of the methodology will expose this hidden risk in the testing stage, rather than in let it surface in production (where it becomes more expensive).

6.2 . . . But Replace Judgement

But, following a methodology means that you are trusting something *other* than your own judgement to make decisions on what actions to take: perhaps

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

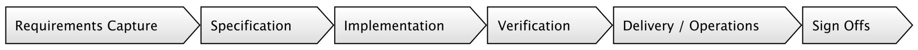

Figure 6.1: Waterfall Actions

the methodology recommends some activity which wastes time, money or introduces some new risk?

Following a software methodology is therefore an act of *trust*:

- Why should we place trust in any *one* methodology, given there are so many alternatives?
- Should there not be more agreement between them, and if not, why not?
- How can a methodology *possibly* take into account the risks on *my* project?

In this chapter, we're going to have a brief look at some different software methodologies, and try to explain *why* they are different. Let's start with Waterfall.

6.3 Waterfall

> "The waterfall development model originated in the manufacturing and construction industries; where the highly structured physical environments meant that design changes became prohibitively expensive much sooner in the development process. When first adopted for software development, there were no recognized alternatives for knowledge-based creative work."
>
> —Waterfall Model, *Wikipedia*[2]

Waterfall is a family of methodologies advocating a linear, stepwise approach to the processes involved in delivering a software system. The basic idea behind Waterfall-style methodologies is that the software process is broken into distinct stages, as shown in Figure 6.1. These usually include:

- Requirements Capture
- Specification

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

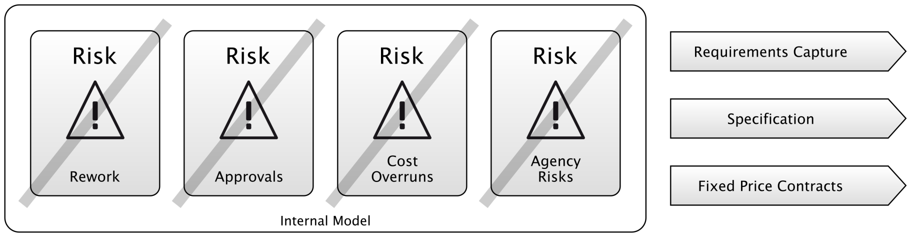

Figure 6.2: Waterfall, Specifications and Requirements Capture

- Implementation
- Verification
- Delivery and Operations
- Sign Offs at each stage

Because Waterfall methodologies are borrowed from *the construction industry*, they manage the risks that you would care about in a construction project, specifically, minimising the risk of rework, and the risk of costs spiralling during the physical phase of the project. For example, pouring concrete is significantly easier than digging it out again after it sets.

Construction projects are often done by tender which means that the supplier will bid for the job of completing the project, and deliver it to a fixed price. This is a risk-management strategy for the client: they are transferring the risk of construction difficulties to the supplier, and avoiding the Agency Risk that the supplier will "pad" the project and take longer to implement it than necessary, charging them more in the process. In order for this to work, both sides need to have a fairly close understanding of what will be delivered, and this is why a specification is created.

The Wrong Risks?

In construction this makes a lot of sense. But *software projects are not the same as building projects*. There are two key criticisms of the Waterfall approach when applied to software:

> "1. Clients may not know exactly what their requirements are before they see working software and so change their requirements, leading to redesign, redevelopment, and re-testing, and increased costs."

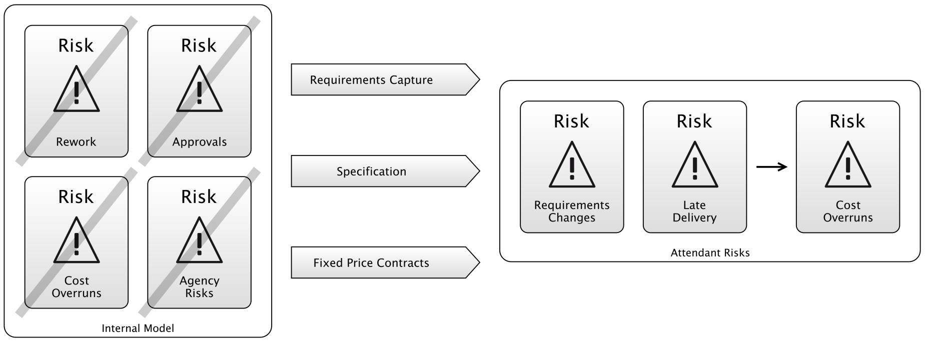

Figure 6.3: Waterfall, Applied to a Software Project

> "2. Designers may not be aware of future difficulties when designing a new software product or feature."
>
> —Waterfall Model, *Wikipedia*[3]

So, the same actions Waterfall prescribes to mitigate rework and cost-overruns in the building industry do not address (and perhaps exacerbate) the two issues raised above when applied to software.

As you can see in Figure 6.3, some of the risks on the left *are the same* as the ones on the right: the actions taken to manage them made no difference (or made things worse). The inability to manage these risks led to the identification of a "Software Crisis", in the 1970's:

> "Software crisis is a term used in the early days of computing science for the difficulty of writing useful and efficient computer programs in the required time... The software crisis was due to the rapid increases in computer power and the complexity of the problems that could not be tackled."
>
> —Software Crisis, *Wikipedia*[4]

6.4 Agile

The Software Crisis showed that, a lot of the time, up-front requirements-capture, specification and fixed-price bids did little to manage cost and schedule risks on software projects. So it's not surprising that by the 1990's, various

[3] `https://en.wikipedia.org/wiki/Waterfall_model#Supporting_arguments`

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

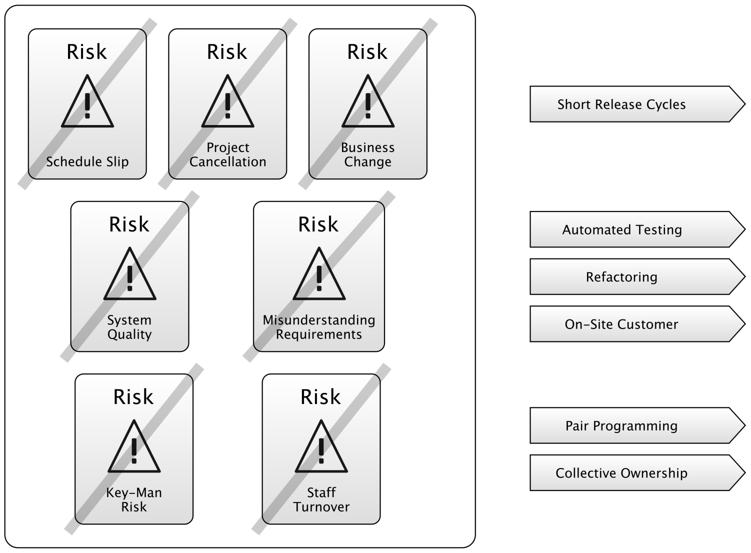

Figure 6.4: Risks, and the practices that manage them in Extreme Programming

different groups of software engineers were advocating "Agile" techniques which did away with those actions.

In Extreme Programming Explained[5], Kent Beck breaks down his methodology, 'Extreme Programming', listing the risks he wants to address and the actions with which he proposes to address them. Figure 6.4 summarises the main risks and actions he talks about. These are *different* risks to those addressed by Waterfall, so unsurprisingly, they lead to different actions.

6.5 Different Methodologies For Different Risks

Here are some high-level differences we see in some other popular methodologies:

- **Lean Software Development[6]**. While Waterfall borrows from risk management techniques in the construction industry, Lean Software Development applies the principles from Lean Manufacturing[7], which was developed at Toyota in the last century. Lean takes the view that the biggest risk in manufacturing is from *waste*, where waste is inventory,

[5] `http://amzn.eu/d/1vSqAWa`

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

[7] `https://en.wikipedia.org/wiki/Lean_manufacturing`

over-production, work-in-progress, time spent waiting or defects in production. Applying this approach to software means minimising work-in-progress, frequent releases and continuous improvement.

- **Project Management Body Of Knowledge (PMBoK)(https://en.wikipedia.org/wiki/Project** This is a formalisation of traditional project management practice. It prescribes best practices for managing scope, schedule, resources, communications, dependencies, stakeholders etc. on a project. Although "risk" is seen as a separate entity to be managed, all of the above areas are sources of risk within a project, as we will see in [Part 2.

- **Scrum**[8]. Is a popular Agile methodology. Arguably, it is less "extreme" than Extreme Programming, as it promotes a limited set, more achievable set of agile practices, such as frequent releases, daily meetings, a product owner and retrospectives. This simplicity arguably makes it simpler to learn and adapt to and probably contributes to Scrum's popularity over XP.

- **DevOps**[9]. Many software systems struggle at the boundary between "in development" and "in production". DevOps is an acknowledgement of this, and is about more closely aligning the feedback loops between the developers and the production system. It champions activities such as continuous deployment, automated releases and automated monitoring.

While this is a limited set of examples, you should be able to observe that the actions promoted by a methodology are contingent on the risks it considers important.

6.6 Effectiveness

> "All methodologies are based on fear. You try to set up habits to prevent your fears from becoming reality."
>
> —Extreme Programming Explained, *Kent Beck*[10]

The promise of any methodology is that it will help you manage certain Hidden Risks. But this comes at the expense of the *effort* you put into the practices of the methodology.

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

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

[10]http://amzn.eu/d/1vSqAWa

A methodology offers us a route through the Risk Landscape, based on the risks that the designers of the methodology care about. When we use the methodology, it means that we are baking into our behaviour actions to avoid those risks.

Methodological Failure

When we take action according to a methodology, we expect the Payoff, and if this doesn't materialise, then we feel the methodology is failing us. It could just be that it is inappropriate to the *type of project* we are running. Our Risk Landscape may not be the one the designers of the methodology envisaged. For example:

- NASA don't follow an agile methodology[11] when launching space craft: there's no two-weekly launch that they can iterate over, and the the risks of losing a rocket or satellite are simply too great to allow for iteration in production. The risk profile is just all wrong: you need to manage the risk of *losing hardware* over the risk of *requirements changing*.

- Equally, regulatory projects often require big, up-front, waterfall-style design: keeping regulators happy is often about showing that you have a well-planned path to achieving the regulation. Often, the changes need to be reviewed and approved by regulators and other stakeholders in advance of their implementation. This can't be done with an approach of "iterate for a few months".

- At the other end of the spectrum, Facebook used to have[12] an approach of "move fast and break things". This may have been optimal when they were trying mitigate the risk of being out-innovated by competitors within the fast-evolving sphere of social networking. *Used to have*, because now they have modified this to "move fast with stable infrastructure"[13], perhaps as a reflection of the fact that their biggest risk is no longer competition, but bad publicity.

6.7 Choosing A Methodology

There is value in adopting a methodology as a complete collection of processes: choosing a methodology (or any process) reduces the amount of

[11] https://standards.nasa.gov/standard/nasa/nasa-std-87398

[12] https://mashable.com/2014/04/30/facebooks-new-mantra-move-fast-with-stability/?europe=true

[13] https://www.cnet.com/news/zuckerberg-move-fast-and-break-things-isnt-how-we-operate-anymore/

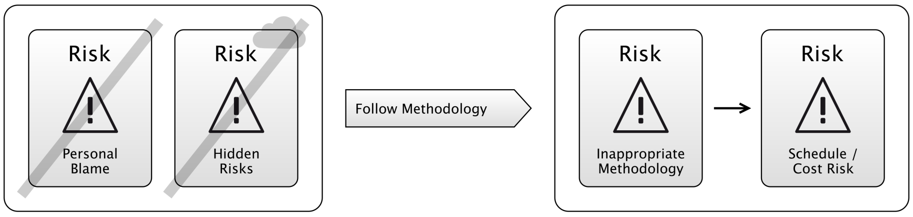

Figure 6.5: Inappropriate Methodologies create their own risks

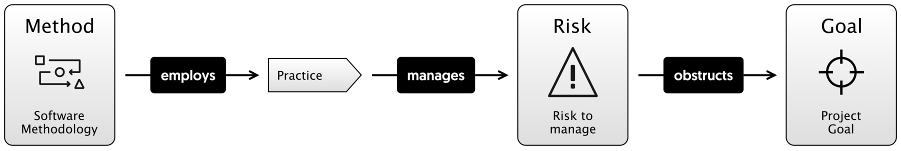

Figure 6.6: Methodologies, Actions, Risks, Goals

thinking individuals have to do, and it becomes *the process* that is responsible for failure, not the individual (as shown in Figure 6.5).

It's nice to lay the blame somewhere else. But, if we genuinely care about our projects, then it's critical that we match the choice of methodology to the risk profile of the project. We need to understand exactly what risks our methodology will help us with, which it won't, where it is appropriate, and where it isn't.

> "Given any rule, however 'fundamental' or 'necessary' for science, there are always circumstances when it is advisable not only to ignore the rule, but to adopt its opposite."
>
> Paul Feyerabend[14]

An off-the-shelf methodology is unlikely to fit the risks of any project exactly. Sometimes, we need to break down methodologies into their component practices, and apply just the practices we need. This requires a much more fine-grained understanding of how the individual practices work, and what they bring.

As Figure 6.6 shows, different methodologies advocate different practices, and different practices manage different risks. If we want to understand

[14]https://www.azquotes.com/author/4773-Paul_Feyerabend

methodologies, or choose practices from one, we really need to understand the *types of risks* we face on software projects. This is where we go next in Part 2.