Risk-First Software Development

Volume 1: The Menagerie

Rob Moffat

Risk-First: The Menagerie

By Rob Moffat

ISBN: 9781717491855

Credits

tbd

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

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

Thanks to:

Books In The Series

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

Online

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

Published By

`Kite9 Ltd.`
`14 Manor Close`

Colchester
CO6 4AR

Contents

Preface

Welcome to Risk-First!

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

Why

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

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

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

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

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

This book considers that perspective: that building software is all about *managing risk*, and that these methodologies are acknowledgements of this

fact, and they differ because they have *different ideas* about which are the most important *risks to manage*.

What This Is

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

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

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

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

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

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

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

[2] `https://tvtropes.org`

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

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

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

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

Patterns For Practitioners

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

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

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

Towards a "Periodic Table"

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

What This is Not

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

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

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

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

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

Who

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

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

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

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

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

For Developers

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

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

For Project Managers and Team Leads

All too often, Project Managers don't have a full grasp of the technical details of their projects. And this is perfectly normal, as the specialization belongs

below them. However, projects fail because risks materialize, and risks materialize because the devil is in those details.

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

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

How

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

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

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

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

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

Where

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

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

A Note on References

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

On to The Executive Summary

[7] https://github.com

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

[9] https://wikipedia.org

Executive Summary

1. There are Lots of Ways of Running Software Projects

There are lots of different ways to look at a project 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 and RAG status reporting are used. These techniques should be familiar to project managers and developers everywhere.

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

For example:

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

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

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

For example:

- If we do a Code Review, we are partly trying to minimise the risks of bugs slipping through into production, and also manage the Key-Man 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 to dealing with different risks, the same is true of the methodologies we use on our projects. We can use a Risk-First approach to examine the different methodologies, and see which risks they address.

For example:

- **Agile** methodologies prioritise 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 tradeoffs with all of these choices.

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

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

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

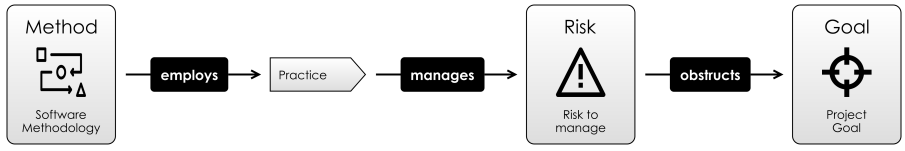

Figure 1: Methodologies, Risks, Practices

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 Taxonomy of Software Risk, and look at which specific tools you can use to mitigate each kind of risk.

Then, we look at different software practices, and how they 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

Part II

Risk

Part III

Tools & Practices

CHAPTER 1

Estimates

In this chapter, we're going to put a Risk-First spin on the process of Estimating. But, in order to get there, we first need to start with understanding *why* we estimate. We're going to look at some "Old Saws" of software estimation and what we can learn from them. Finally, we'll bring our Risk-First menagerie to bear on de-risking the estimation process.

1.1 The Purpose Of Estimating

Why bother estimating at all? There are two reasons why estimates are useful:

1. **To allow for the creation of *events*.** As we saw in Deadline Risk, if we can put a date on something, we can mitigate lots of Coordination Risk. Having a *release date* for a product allows whole teams of people to coordinate their activities in ways that hugely reduce the need for Communication. "Attack at dawn" allows disparate army units to avoid the Coordination Risk inherent in "attack on my signal". This is a *good reason for estimating*, because by using events you are mitigating Coordination Risk. This is often called a *hard deadline*.

2. **To allow for the estimation of the Pay-Off of an action.** This is a *bad reason for estimating*, as we will discuss in detail below. But briefly, the main issue is that Pay-Off isn't just about figuring out Schedule Risk - you should be looking at all the other Attendant Risks of the action too.

1.2 How Estimates Fail

Estimates are a huge source of contention in the software world:

> "Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time."
>
> —Software Development Effort Estimation, *Wikipedia*[1].

In their research "Anchoring and Adjustment in Software Estimation", Aranda and Easterbrook[2] asked developers split into three groups (A, B and Control) to give individual estimates on how long a piece of software would take to build. They were each given the same specification. However:

- Group A was given the hint: "I admit I have no experience with software, but I guess it will take about two months to finish".
- Group B were given the same hint, except with *20* months.

How long would members of each group estimate the work to take? The results were startling. On average:

- Group A estimated 5.1 months.
- The Control Group estimated 7.8 months.
- Group B estimated 15.4 months.

The anchor mattered more than experience, how formal the estimation method, or *anything else*. *We can't estimate time at all*.

1.3 Is Risk To Blame?

Why is it so bad? The problem with a developer answering a question such as:

> "How long will it take to deliver X?"

Seems to be the following:

- The developer and the client likely don't agree on exactly what X is, and any description of it is inadequate anyway (Invisibility Risk).

[1] https://en.m.wikipedia.org/wiki/Software_development_effort_estimation

[2] http://www.cs.toronto.edu/%7Esme/papers/2005/ESEC-FSE-05-Aranda.pdf

- The developer has a less-than-complete understanding of the environment he will be delivering X in (Complexity Risk and Map And Territory Risk).
- The developer has some vague ideas about how to do X, but he'll need to try out various approaches until he finds something that works (Boundary Risk and Learning-Curve Risk).
- The developer has no idea what Hidden Risk will surface when he starts work on it.
- The developer has no idea what will happen if he takes too long and misses the date by a day/week/month/year (Schedule Risk).

. . . and so on.

The reason the estimate of *time* is wrong is because every action attempts to mitigate risk and the estimate of *risk* is wrong.

So what are we to do? It's a problem as old as software itself, and in deference to that, let's examine the estimating problem via some "Old Saws".

1.4 Old Saw No. 1: The "10X Developer"

> "A 10X developer is an individual who is thought to be as productive as 10 others in his or her field. The 10X developer would produce 10 times the outcomes of other colleagues, in a production, engineering or software design environment."
>
> —10X Developer, *Techopedia*[3]

Let's try and pull this apart:

- How do we measure this "productivity"? In Risk-First terms, this is about taking action to *transform* our current position on the Risk Landscape to a position of more favourable risk. A "10X Developer" then must be able to take actions that have much higher Pay-Off than a "1X Developer". That is, mitigating more existing risk, and generating less Attendant Risk.
- It stands to reason then, that someone taking action *faster* will leaving us with less Schedule Risk.
- However, if they are *more expensive*, they may leave us with greater Funding Risk afterwards.

[3] https://www.techopedia.com/definition/31673/10X-developer

- But, Schedule Risk isn't the only risk being transformed: The result might be bugs, expensive new dependencies or spaghetti-code complexity.
- The "10X" developer *must* also leave behind less of these kind of risks too.
- That means that the "10X Developer" isn't merely faster, but *taking different actions*. They are able to use their talent and experience to see actions with greater pay-off than the 1X Developer.

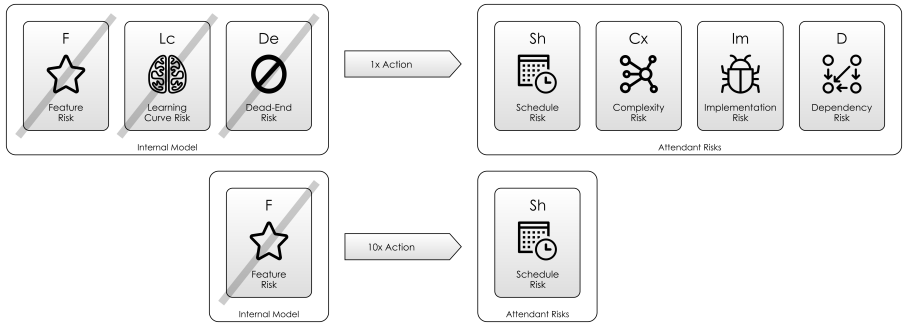

Figure 1.1: 1X Task vs 10X Task

Does the "10X Developer" even exist? Crucially, it would seem that such a thing would be predicated on the existence of the "1X Developer", who gets "1X" worth of work done each day. It's not clear that there is any such thing as an average developer who is mitigating risk at an average rate.

Even good developers have bad days, weeks or projects. Taking Action is like placing a bet. Sometimes you lose and the Pay-Off doesn't appear:

- The Open-Source software you're trying to apply to a problem doesn't solve it in the way you need.
- A crucial use-case of the problem turns out to change the shape of the solution entirely, leading to lots of rework.
- An assumption about how network security is configured turns out to be wrong, leading to a lengthy engagement with the infrastructure team.

The easiest way to be the "10X developer" is to have *done the job before*. If you're coding in a familiar language, with familiar libraries and tools, delivering a cookie-cutter solution to a problem in the same manner you've done several

times before, then you will be a "10X Developer" compared to *you doing it the first time* because:

- There's no Learning Curve Risk, because you already learnt everything.
- There's no Dead End Risk because you already know all the right choices to make.

1.5 Old Saw No. 2: Quality, Speed, Cost: Pick Any Two

> "The Project Management Triangle (called also the Triple Constraint, Iron Triangle and Project Triangle) is a model of the constraints of project management. While its origins are unclear, it has been used since at least the 1950s. It contends that:
>
> 1. The quality of work is constrained by the project's budget, deadlines and scope (features).
>
> 2. The project manager can trade between constraints.
>
> 3. Changes in one constraint necessitate changes in others to compensate or quality will suffer."
>
> —Project Management Triangle, *Wikipedia*[4]

From a Risk-First perspective, we can now see that this is an over-simplification. If *quality* is a Feature Fit metric, *deadlines* is Schedule Risk and *budget* refers to Funding Risk then that leaves us with a lot of risks unaccounted for:

- I can deliver a project in very short order by building a bunch of screens that *do nothing* (accruing *stunning* levels of Implementation Risk as I go).
- Or, by assuming a lottery win, the project's budget is fine. (Although I would have *huge* Funding Risk because *what are the chances of winning the lottery?*.)
- Brooks' Law contradicts this by saying you can't trade budget for deadlines:

> "Brooks' law is an observation about software project management according to which"adding human resources to a late software project makes it later".
>
> —Brooks Law, *Wikipedia*[5]

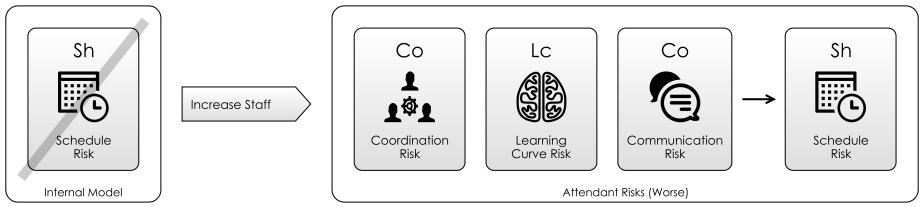

Figure 1.2: Brooks' Law, Risk-First Style

Focusing on the three variables of the iron triangle isn't enough. You can game these variables by sacrificing others: we need to be looking at the project's risk *holistically*.

- There's no point in calling a project complete if the dependencies you are using are unreliable or undergoing rapid change
- There's no point in delivering the project on time if it's an Operational Risk nightmare, and requires constant round-the-clock support and will cost a fortune to *run*. (Working on a project that "hits it's delivery date" but is nonetheless a broken mess once in production is too common a sight.)
- There's no point in delivering a project on-budget if the market has moved on and needs different features.

Old Saw No. 3: Parkinson's Law

We've already looked at Parkinson's Law in the chapter on Agency Risk, but lets recap:

> "Parkinson's law is the adage that 'work expands so as to fill the time available for its completion'."
>
> —Parkinson's Law, *Wikipedia*[6]

Let's leave aside the Agency Risk concerns this time. Instead, let's consider this from a Risk-First perspective. *Of course* work would expand to fill the time available: *Time available* is an *absence of Schedule Risk*, it's always going to be sensible to exchange free time to reduce more serious risks.

This is why projects will *always* take at least as long as is budgeted for them.

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

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

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

A Case Study

Let's look at a quick example of this in action, taken from Rapid Development by Steve McConnell[7]. At the point of this excerpt, Carl (the Project Manager) is discussing the schedule with Bill, the project sponsor:

> "I think it will take about 9 months, but that's just a rough estimate at this point," Carl said. "That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. We absolutely need to bring that system in within 6 months. Can you do it in 6?" (1)

Later in the story, the schedule has slipped twice and is about to slip again:

> . . . At the 9-month mark, the team had completed detailed design, but coding still hadn't begun on some modules. It was clear that Carl couldn't make the 10-month schedule either. He announced the third schedule slip number— to 12 months. Bill's face turned red when Carl announced the slip, and the pressure from him became more intense. (2)

At point (2), Carl's tries to mitigate Feature Risk by increasing Schedule Risk, although he knows that Bill will trust him less for doing this, as shown below:

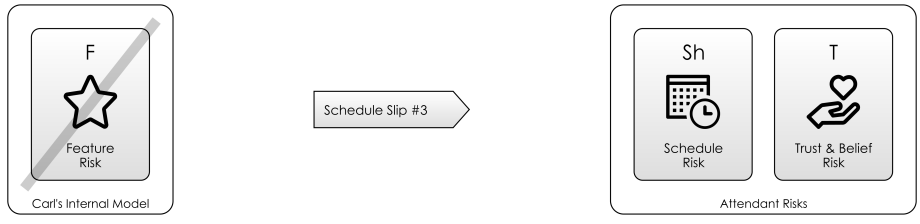

Figure 1.3: Carl's Schedule Slip increases Trust and Belief Risks

> Carl began to feel that his job was on the line. Coding proceeded fairly well, but a few areas needed redesign and reimplementation. The team hadn't coordinated design details in those areas well, and some of their implementations conflicted. At the 11-month oversight-committee meeting, Carl announced the

[7] http://amzn.eu/d/eTWKOsK

> fourth schedule slip— to 13 months. Bill became livid. "Do you have any idea what you're doing?" he yelled. "You obviously don't have any idea! You obviously don't have any idea when the project is going to be done! I'll tell you when it's going to be done! It's going to be done by the 13-month mark, or you're going to be out of a job! I'm tired of being jerked around by you software guys! You and your team are going to work 60 hours a week until you deliver!" (3)

At point (3), the schedule has slipped again, and Bill has threatened Carl's job. Why did he do this? Because *he doesn't trust Carl's evaluation of the Schedule Risk*. By telling Carl that it's his job on the line, he makes sure Carl appreciates the Schedule Risk. However, forcing staff to do overtime is a dangerous ploy: it could disenfranchise the staff, or cause corners to be cut:

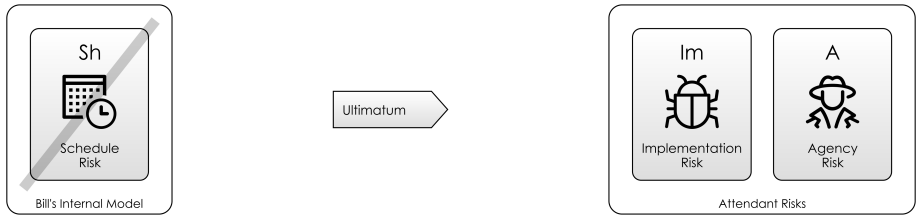

Figure 1.4: Bill's Ultimatum

> Carl felt his blood pressure rise, especially since Bill had backed him into an unrealistic schedule in the first place. But he knew that with four schedule slips under his belt, he had no credibility left. He felt that he had to knuckle under to the mandatory overtime or he would lose his job. Carl told his team about the meeting. They worked hard and managed to deliver the software in just over 13 months. Additional implementation uncovered additional design flaws, but with everyone working 60 hours a week, they delivered the product through sweat and sheer willpower. " (4)
>
> —McConnell, Steve, *Rapid Development*[8]

At point (4), we see that Bill's gamble worked (for him at least): the project was delivered on time by the team working overtime for two months. This

[8] http://amzn.eu/d/eTWKOsK

was lucky - it seems unlikely that no-one quit and that the code didn't descend into a mess in that time.

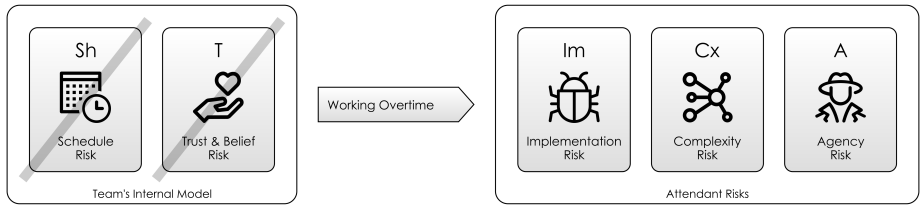

Figure 1.5: Team Response

Despite this being a fictional (or fictionalised) example, it rings true for many projects. What *should* have happened at point (1)? Both Carl and Bill estimated incorrectly... Or did they?

1.6 Agile Estimation

One alternative approach, must espoused in DevOps/Agile is to pick a short-enough period of time (say, two days or two weeks), and figure out what the most meaningful step towards achieving an objective would be in that time. By fixing the time period, we remove Schedule Risk from the equation, don't we?

Well, no. First, how to choose the time period? Schedule Risk tends to creep back in, in the form of something like Man-Hours[9] or Story Points[10]:

> "Story points rate the relative effort of work in a Fibonacci-like format: 0, 0.5, 1, 2, 3, 5, 8, 13, 20, 40, 100. It may sound counter-intuitive, but that abstraction is actually helpful because it pushes the team to make tougher decisions around the difficulty of work."
>
> —Story Points, *Atlassian*[11]

Second, the strategy of picking the two-day action with the greatest Pay-Off is *often good*. (After all, this is just Gradient Descent[12], and that's a perfectly good way for training Machine Learning[13] systems.) However, just like following

[9] https://en.wikipedia.org/wiki/Man-hour

[10] https://www.atlassian.com/agile/project-management/estimation

[11] https://www.atlassian.com/agile/project-management/estimation

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

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

a river downhill from the top of a mountain will *often* get you to the sea, it probably won't take the shortest path, and sometimes you'll get stuck at a lake.

The choice of using gradient descent means that you have given up on Goals: Essentially, we have here the difference between "Walking towards a destination" and "Walking downhill". Or, if you like, a planned economy and a market economy. But, we don't live in *either*: everyone lives in some mixture of the two: our governments *have plans* for big things like roads and hospitals, and taxes. Other stuff, they leave to the whims of supply and demand. A project ends up being the same.

1.7 Risk-First Estimating

Let's figure out what we can take away from the above experiences:

- **From the "10X Developer" Saw:** the difference made by experience implies that a lot of the effort on a project comes from Learning Curve Risk and Dead End Risk.
- **From "Quality, Speed, Cost":** we need to be considering *all* risks, not just some arbitrary milestones on a project plan. Project plans can always be gamed, and you can always leave risks unaccounted for in order to hit the goals.
- **From the Parkinson's Law:** giving people a *time budget*, you absolve them from Schedule Risk... at least until they realise they're going to overrun. This gives them one less dimension of risk to worry about, but means they end up taking all the time you give them, because they are optimising over the remaining risks.
- Finally, the lesson from Agile Estimation is that *just iterating* is sometimes not as efficient as *using your intuition and experience* to find a more optimal path.

How can we synthesise this knowledge, along with what we've learned into something that makes more sense?

Tip #1: Estimating Should be About *Estimating Pay Off*

For a given action / road-map / business strategy, what Attendant Risks are we going to have:

- What bets are we making about where the market will be?

- What Communication Risk will we face explaining our product to people?
- What Feature Fit risks are we likely to have when we get there?
- What Complexity Risks will we face building our software? How can we avoid it ending up as a Big Ball Of Mud?
- Where are we likely to face Boundary Risks and Dead End Risks

Instead of the Agile Estimation being about picking out a story-point number based on some idealised amount of typing that needs to be done, it should be about surfacing and weighing up risks. e.g:

- "Adding this new database is problematic because it's going to massively increase our Dependency Risk."
- "I don't think we should have component A interacting with component B because it'll introduce extra Communication Risk which we will always be tripping over."
- "I worry we might not understand what the sales team want and are facing Feature Implementation Risk. How about we try and get agreement on a specification?"

Tip #2: The Risk Landscape is Increasingly Complex: Utilise This

Figure 1.6: Journey via the Central Line

If you were travelling across London from Ealing (in the West) to Stratford (in the East) the *fastest* route might be to take the Central Line. You could do it via the A406 road, which would take a *bit* longer. It would *feel* like you're mainly going in completely the wrong direction doing that, but it's much faster than cutting straight through London and you don't pay the congestion charge.

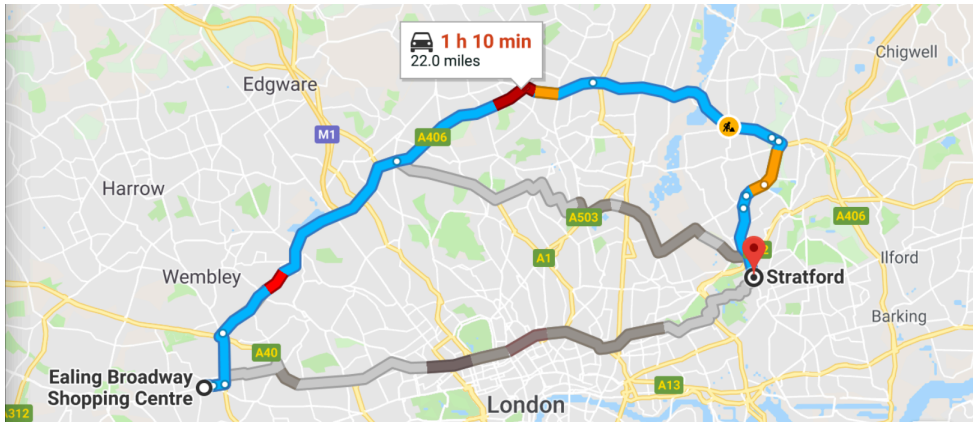

Figure 1.7: Journey by Car

In terms of risk, they all have different profiles. You're often delayed in the car, by some amount. The tube is *generally* reliable, but when it breaks down or is being repaired it might end up quicker to walk.

If you were doing this same journey on foot, it's a very direct route, but would take five times longer. However, if you were making this journey a hundred years ago, that might be the way you chose (horseback might be a bit faster).

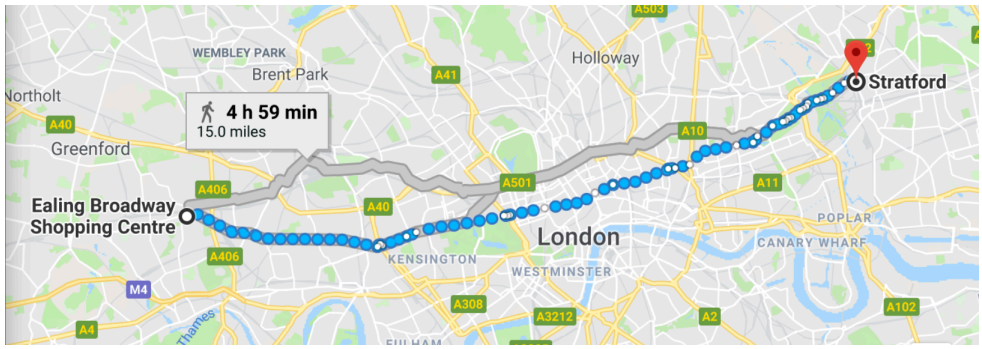

Figure 1.8: Journey on Foot

In the software development past, *building it yourself* was the only way to get anything done. It was like London *before road and rail*. Nowadays, you are bombarded with choices. It's actually *worse than London* because it's not even a two-dimensional geographic space and there are multitudes of different routes and acceptable destinations. Journey planning on the software Risk Landscape is an optimisation problem *par excellence*.

Because the modern Risk Landscape is so complex:

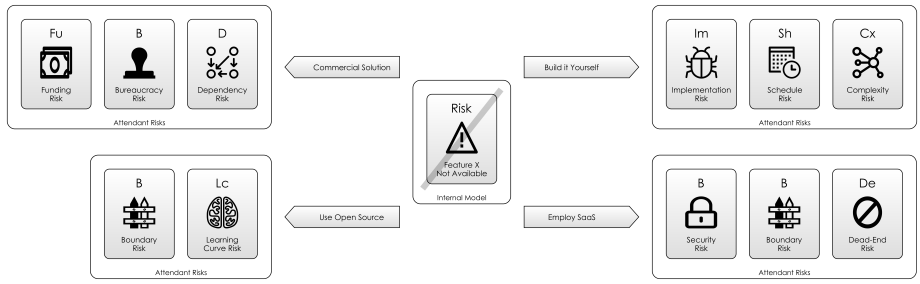

Figure 1.9: Possible Moves On The Risk Landscape

- There can be orders of magnitude difference in *time*, with very little difference in destination.
- If it's Schedule Risk you're worried about, *Code Yourself* isn't a great solution (for the whole thing, anyway). "Take the tube" and at least partly use something someone built already. There are probably multiple alternatives you can consider.
- If no one has built something similar already, then why is that? Have you formulated the problem properly?
- Going the wrong way is *so much easier*.
- Dead-Ends (like a broken Central Line) are much more likely to trip you up.
- You need to keep up with developments in your field. Read widely.

Tip #3: Meet Reality Early on the Biggest Risks

In getting from A to B on the Risk Landscape, imagine that all the Attendant Risks are the stages of a journey. Some might be on foot, train, car and so on. In order for your course of action to work, all the stages in the journey have to succeed.

Although you might have to make the steps of a journey in some order, you can still mitigate risk in a different order. For example, checking the trains are running, making sure your bike is working, booking tickets and taxis, and so on.

The *sensible* approach would be to test the steps *in order from weakest to strongest*. This means working out how to meet reality for each risk in turn, in order from biggest risk to smallest.

Often, a *strategy* will be broken up into multiple actions. *Which are the riskiest actions?* Figure this out, using the Risk-First vocabulary and the best expe-

rience you can bring to bear, then, perform the actions which Pay Off the biggest risks first.

As we saw from the "10X Developer" Saw, Learning Curve Risk and Dead End Risk, are likely to be the biggest risks. How can we front-load this and tackle these earlier?

- *Having a vocabulary* (like the one Risk-First provides) allows us to *at least talk about these*. e.g. "I believe there is a Dead End Risk that we might not be able to get this software to run on Linux."
- Build mock-ups:
  - UI wireframes allow us to bottom out the Communication Risk of the interfaces we build.
  - Spike Solutions allow us to de-risk algorithms and approaches before making them part of the main development.
  - Test the market with these and meet reality early.
- Don't pick delivery dates far in the future. Collectively work out the biggest risks with your clients, and then arrange the next possible date to demonstrate the mitigation.
- Do actions *early* that are *simple* but are nevertheless show-stoppers. They are as much a source of Hidden Risk as more obviously tricky actions.

Tip #4: Talk Frankly About All The Risks

Let's get back to Bill and Carl. What went wrong between points `(1)` and `(2)`? Let's break it down:

- **Bill *wants* the system in 3-4 months.** It doesn't happen.
- **He says it "must be delivered in 6 months", but this doesn't happen either.** However, the world (and the project) doesn't end: *it carries on*. What does this mean about the truth of his statement? Was he deliberately lying about the end date, or just espousing his view on the Schedule Risk?
- **Carl's original estimate was 9 months.** Was he working to this all along? Did the initial brow-beating over deadlines at point `(1)` contribute to Agency Risk in a way that *didn't* happen at point `(2)`?
- **Why *did* Bill get so angry?** His understanding of the Schedule Risk was, if anything, *worse* than Carl's. It's not stated in the account, but it's likely the Trust Risk moved upwards: Did his superiors stop trusting him? Was his job at stake?

- **How could including this risk in the discussion have improved the planning process?** Could the conversation have started like this instead?

"I think it will take about 9 months, but that's just a rough estimate at this point," Carl said. "That's not going to work," Bill said. "I was hoping you'd say 3 or 4 months. I need to show the board something by then or I'm worried they will lose confidence in me and this project".

"OK," said Carl. "But I'm really concerned we have huge Feature Fit Risk. The task of understanding the requirements and doing the design is massive."

"Well, in my head it's actually pretty simple," said Bill. "Maybe I don't have the full picture, or maybe your idea of what to build is more complex than I think it needs to be. That's a massive risk right there and I think we should try and mitigate it right now before things progress. Maybe I'll need to go back to the board if it's worse than I think."

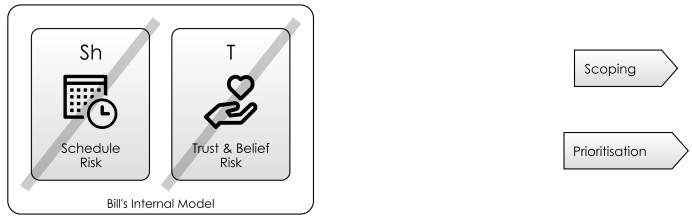

Figure 1.10: Identifying The Action

Tip #5: Picture Worrying Futures

The Bill/Carl problem is somewhat trivial (not to mention likely fictional). How about one from real life? On a project I was working on in November some years ago, we had two pieces of functionality we needed: Bulk Uploads and Spock Integration. (It doesn't really matter what these are). The bulk uploads would be useful *now*. But, the Spock Integration wasn't due until January. In the Spock estimation meeting I wrote the following note:

> "Spock estimates were 4, 11 and 22 until we broke it down into tasks. Now, estimates are above 55 for the whole piece. And worryingly, we probably don't have all the tasks. We know we need bulk uploads in November. Spock is January. So, do bulk uploads?"

The team *wanted* to start Bulk Uploads work. After all, from these estimates it looked like Spock could easily be completed in January. However, the question should have been:

> "If it was February now, and we'd *got nothing done*, what would our biggest risk be?"

Missing Bulk Uploads wouldn't be a show-stopper, but missing Spock would be a huge regulatory problem. *Start work on the things you can't miss.*

This is the essence of De-Risking.

Glossary

Abstraction The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems in order to more closely attend to other details of interest.

Feedback Loop

Internal Model

Taking Action

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

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

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

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

Meet Reality

Pay-Off

Risk

Risk Landscape

Goal In Mind

Initial Risk

Attendant Risk

Hidden Risk

Mitigated Risk