Risk-First

Software Development De-Risked

Volume 1: The Menagerie

Rob Moffat

Risk First: The Menagerie

By Rob Moffat

ISBN: tbd.

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"

Does that diminish it? If you have visited the TVTropes 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:

> tbd.

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, tbd, the tbd published a book called "Design Patterns: tbd". Which shows you patterns of *structure* within Object-Oriented programming:

tbd.

Patterns For Practitioners

This book 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, this book 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.)

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

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 mitigate risks, and we will uncover *exactly which risks are mitigated* and *how they do it*.

Where

All of the material for this book is available Open Source on github.com[1], and at the risk-first.org[2] 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[3] 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

[1] https://github.com

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

[3] 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. 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 - Continuous Integration, Unit Testing or Pair Programming, for example.

Software methodologies, then, are collections of tools and practices: "Agile", "Waterfall", "Lean" or "Phased Delivery" (for example) all suggest 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 a 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 mitigating 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 therefore, is that every action you take on a project is to mitigate some 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, reserves are commonly set aside for the risks of stock-market crashes, and teams are structured around monitoring these different risks.
- The insurance industry 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: describes the 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.
- Expose 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 Mitigate 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 mitigate the risks of bugs slipping through into production, and also mitigate the Key-Man Risk of knowledge not being widely-enough shared.
- If we write Unit Tests, we're also mitigating the risk of bugs going to production, but we're also mitigating against 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.

Different tools are appropriate for mitigating different types of risks.

5. Different Methodologies 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 mitigating the risk that requirements capture is complicated, error-prone and that requirements change easily.
- **Waterfall** takes the view that coding effort is an expensive risk, and that we should build plans up-front to avoid it.
- **Lean** takes the view that risk lies in incomplete work and wasted work, and aims to minimize 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.

We can place methodologies within a framework, and show how choice of methodology is contingent on the risks faced.

6. Driving Development With a Risk-First Perspective

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

Figure 1: Methdologies, Risks, Practices

How do we take this further?

The first idea we explore is that of 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 mitigate various risks. Beyond this we examine the question: *how can a Risk-First approach inform the use of this technique?*

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 have **Redundant Systems**, 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 choices* and weigh up *accepting* versus *mitigating* risks.

Still interested? Then dive into reading the introduction.

Part I

Introduction

Part II

Risk

CHAPTER 1

Feature Risk

Feature Risk is the category of software risk to do with features that have to be in your software. It is the risk that you face by *not having features that your clients need*.

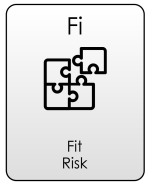
In a way, Feature Risk is very fundamental: if there were *no* feature risk, the job would be done already, either by you, or by another product, and the product would be perfect!

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

Not considering Feature Risk means that you might be building the wrong functionality, for the wrong audience or at the wrong time. And eventually, this will come down to lost money, business, acclaim, or whatever else reason you are doing your project for. So let's unpack this concept into some of it's variations.

1.1 Feature Fit Risk

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



- Risk that the needs of the client don't coincide with services provided by the supplier.

Figure 1.1: Feature Risk

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

Features Don't Work Properly

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

- Risk that the functionality you are providing doesn't match the features the client is expecting, due to poor or partial implementation.

Figure 1.2: Implementation Risk

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

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

1.2 Regression Risk

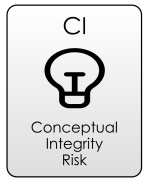
- Risk that the functionality you provide changes for the worse, over time.

Figure 1.3: Regression Risk

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

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

1.3 Conceptual Integrity Risk



- Risk that the software you provide is too complex, or doesn't match the expectations of your clients' internal models.

Figure 1.4: Conceptual Integrity Risk

Sometimes, users *swear blind* that they need some feature or other, but it runs at odds with the design of the system, and plain *doesn't make sense*. Often, the development team can spot this kind of conceptual

failure as soon as it enters the Backlog. Usually, it's in coding that this becomes apparent.

tbd: feature phones.

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

Which leads to Greenspun's 10th Rule:

> "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp." - Greenspun's 10th Rule, *Wikipedia*[2]

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

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

1.4 Feature Access Risk

- Risks due to some clients not having access to some or all of the features in your product.

Figure 1.5: Feature Access Risk

[2] https://en.wikipedia.org/wiki/Greenspun's_tenth_rule

Sometimes, features can work for some people and not others: this could be down to Accessibility[3] issues, language barriers or localization.

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

In Marketing, minimizing Feature Access Risk is all about Segmentation: trying to work out *who* your product appeals to, and tailoring it to that particular market, but for technologists, increasing Feature Access means increasing complexity: you have to deliver the software on more platforms, localized in more languages, with different configurations of features at different price-points. Mitigating Feature Acccess Risk therefore means increased effort and complexity (which we'll come to later).

Market Risk

Feature Access Risk is related, of course, to Market Risk, which I introduced on the Risk Landscape page as being the value that the market places on a particular asset. Since the product you are building is your asset, it makes sense that you'll face Market Risk on it:

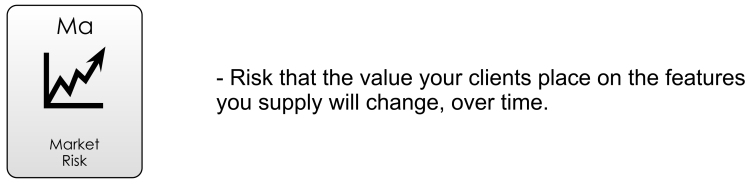


Figure 1.6: Market Risk

> "Market risk is the risk of losses in positions arising from movements in market prices." - Market Risk, *Wikipedia*[4]

[3] https://en.wikipedia.org/wiki/Accessibility

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

I face market risk when I own (i.e. have a *position* in) some Apple[5] stock. Apple's's stock price will decline if a competitor brings out an amazing product, or if fashions change and people don't want their products any more.

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

1.5 Feature Drift Risk

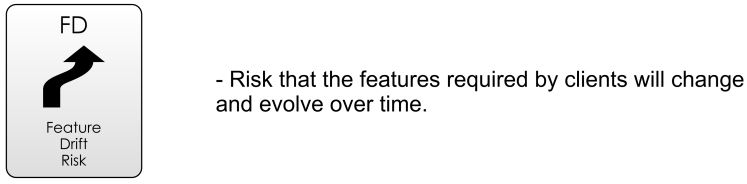


Figure 1.7: Feature Drift Risk

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

The point is: Requirements captured *today* might not make it to *tomorrow*, especially in the fast-paced world of IT. This is partly because the market *evolves* and becomes more discerning. This happens in several ways:

- Features present in competitor's versions of the software become *the baseline*, and they're expected to be available in your version.
- Certain ways of interacting become the norm (e.g. querty keyboards, or the control layout in cars: these don't change with time).

[5] `http://apple.com`

- Features decline in usefulness: *Printing* is less important now than it was, for example.

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

Fashion

Fashion plays a big part in IT, as this infographic on website design shows[6]. True, websites have got easier to use as time has gone by, and users now expect this. Also, bandwidth is greater now, which means we can afford more media and code on the client side. However, *fashion* has a part to play in this.

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

1.6 Delight

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

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

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

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

[6] https://designers.hubspot.com/blog/the-history-of-web-design-infographic

1.7 Analysis

At this point, it would be easy to stop and say, look, here are a bunch of Feature Risk issues that you could face. But, it turns out that we're going to be relying heavily on Feature Risk as we go on in order to build our understanding of other risks, so it's probably worth spending a bit of time up front to classify what we've found.

The Feature Risks identified here basically exist in a 3-dimensional space:

- **Fit**: How well the features fit for a particular client.
- **Audience**: The range of clients (the *market*) that may be able to use this feature.
- **Evolution**: The way the fit and the audience changes and evolves as time goes by.

Fit

> "Survival Of The Fittest" - Darwin, tbd.

Darwin's conception of fitness was not one of athletic prowess, but how well an organism worked within the landscape.

tbd: definition of biological fitness

Fit Risk, Conceptual Integrity Risk and Implementation Risk all hint at different aspects of this "fitness". We can conceive of the relationships between them in the following way:

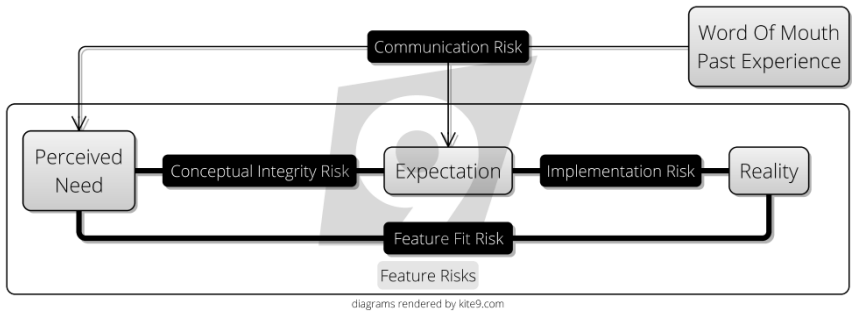


Figure 1.8: Feature Risks Assembled

For further reading, you can check out The Service Quality Model[7], whcih this model is derived from. This model analyses the types of *quality gaps* in services, and how consumer expectations and perceptions of a service arise. In Staging And Classifying, we'll come back and build on this model further.

Fit and Audience

Two risks, Feature Access Risk and Market Risk considers *Fit* for a whole *Audience* of users. They are different: just as it's possible to have a small audience, but a large revenue, it's possible to have a product which has low Feature Access Risk (i.e lots of users can access it without difficulty) but high Market Risk (i.e. the market is highly volatile or capricious in it's demands). *Online services* often suffer from this Market Risk rollercoaster, being one moment highly valued and the next irrelevant.

Fit, Audience and Evolution

Two risks further consider how the **Fit** and **Audience** *change*: Regression Risk and Feature Drift Risk. We call this *evolution* in the sense that:

- Our product's features *evolve* with time, and changes made by the development team.
- Our audience changes and evolves as it is exposed to our product and competing products.
- The world as a whole is an evolving system within which our product exists.

tbd. regression risk and feature drift risk.

1.8 Applying Feature Risk

Consider Feature Risk carefully next time you are grooming the backlog:

[7] http://en.wikipedia.org/SERVQUAL

- Can you judge which tasks mitigate the most Feature Risk?
- Are you delivering features that are valuable to a large audience? How well do you understand your audience? How does the size of the audience for a task impact it's importance in the backlog?
- Does the audience *know* that the features exist? How do you communicate feature availability to them?
- How does writing a specification mitigate Fit Risk? For what other reasons are you writing specifications?

In the next chapter, we are going to unpack this third point further. Somewhere between "what the customer wants" and "what you give them" is a *dialog*. In using a software product, users are engaging in a *dialog* with its features. If the features don't exist, hopefully they will engage in a dialog with the development team to get them added.

These dialogs are prone to risk, and this is the subject of the next chapter, Communication-Risk.

Part III

Preview

book1/Part3.md practices/Estimates.md

Glossary

Abstraction

Feedback Loop

Goal In Mind

Internal Model

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

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

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

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

Meet Reality

Risk

Attendant Risk

Hidden Risk

Mitigated Risk

Take Action