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

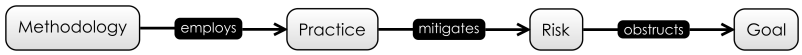

Figure 1: Methdologies, Risks, Practices

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:

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

Software Dependency Risk

In this section, we're going to look specifically at *Software* dependencies, although many of the concerns we'll raise here apply equally to all the other types of dependency we outlined in Dependency Risk.

1.1 Kolmogorov Complexity: Cheating

In the earlier section on Complexity Risk we tackled Kolmogorov Complexity, and the idea that your codebase had some kind of minimal level of complexity based on the output it was trying to create. This is a neat idea, but in a way, we cheated. Let's look at how.

We were trying to figure out the shortest (Javascript) program to generate this output:

```
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

And we came up with this:

```
const ABCD="ABCD";

function out() {
    return ABCD.repeat(10)
}
```

```
(11 symbo
(7 symbol
(7 symbol
(1 symbol
```

Which had **26** symbols in it.

Now, here's the cheat: The `repeat()` function was built into Javascript in 2015 in ECMAScript 6.0[1]. If we'd had to program it ourselves, we might have added this:

```
function repeat(s,n) {                  (10 symbols)
    var a=[];                           (7 symbols)
    while(a.length<n){                  (9 symbols)
        a.push(s)                       (6 symbols)
    }                                   (1 symbol)
    return a.join('');                  (10 symbols)
}                                       (1 symbol)
```

... which would be an extra **44** symbols (in total **70**), and push us completely over the original string encoding of **53** symbols. So, *encoding language is important*.

Conversely, if ECMAScript 6.0 had introduced a function called `abcdRepeater(n)` we'd have been able to do this:

```
function out() {                        (7 symbols)
    return abcdRepeater(10)             (6 symbols)
}                                       (1 symbol)
```

.. and re-encode to **14** symbols. Now, clearly there are some problems with all this:

1. Clearly, *language matters*: the Kolmogorov complexity is dependent on the language, and the features the language has built in.
2. The exact Kolmogorov complexity is uncomputable anyway (it's the *theoretical* minimum program length). It's just a fairly abstract idea, so we shouldn't get too hung up on this. There is no function to be able to say, "what's the Kolmogorov complexity of string X"

[1] http://www.ecma-international.org/ecma-262/6.0/

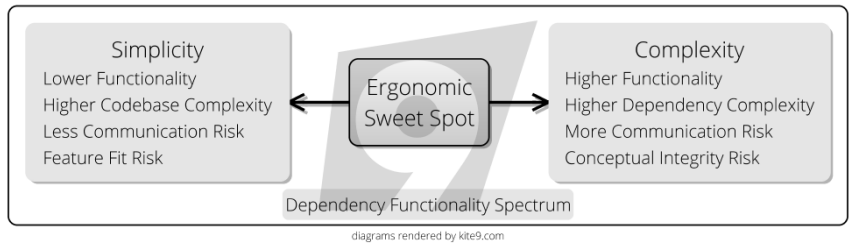

Figure 1.1: Software Dependency Ergonomics: finding the sweet spot between too many features and too few

3. What is this new library function we've created? Is `abcdRepeater` going to be part of *every* Javascript? If so, then we've shifted Codebase Risk away from ourselves, but we've pushed Communication Risk and Dependency Risk onto every *other* user of Javascript. (Why these? Because `abcdRepeater` will be clogging up the documentation and other people will rely on it to function correctly.)
4. Are there equivalent functions for every single other string? If so, then compilation is no longer a tractable problem: is `return abcdRepeater(10)` correct code? Well, now we have a massive library of different `XXXRepeater` functions to compile against to see if it is... So, what we *lose* in Kolmogorov Complexity we gain in Runtime Complexity.
5. Language design, then, is about *ergonomics*. After you have passed the relatively low bar of providing Turing Completeness[2], the key is to provide *useful* features that enable problems to be solved, without over-burdening the user with features they *don't* need. And in fact, all software is about this.

1.2 Ergonomics Examined

Have a look at some physical tools, like a hammer, or spanner. To look at them, they are probably *simple* objects, obvious, strong and dependable. Their entire behaviour is encapsulated in their form. Now, if you have a drill or sander to hand, look at the design of this

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

too. If it's well-designed, then from the outside it is simple, perhaps with only one or two controls. Inside, it is complex and contains a motor, perhaps a transformer, and is maybe made of a hundred different components.

But outside, the form is simple, and designed for humans to use. This is *ergonomics*[3]:

> "Human factors and ergonomics (commonly referred to as Human Factors), is the application of psychological and physiological principles to the (engineering and) design of products, processes, and systems. The goal of human factors is to reduce human error, increase productivity, and enhance safety and comfort with a specific focus on the interaction between the human and the thing of interest."
> - Human Factors and Ergonomics, *Wikipedia*

Interfaces

The interface of a tool is the part we touch and interact with. By striving for simplicity, the interface reduces Communication Risk.

The interface of a system expands when you ask it to do a wide variety of things. An easy-to-use drill does one thing well: it turns drill-bits at useful levels of torque for drilling holes and sinking screws. But if you wanted it to also operate as a lathe, a sander or a strimmer (all basically mechanical things going round) you would have to sacrifice the ergonomic simplicity for a more complex interface, probably including adapters, extensions, handles and so on.

So, we now have split complexity into two: - The inner complexity of the tool (how it works internally, it's own Kolmogorov Complexity). - The complexity of the instructions that we need to write to make the tool work (the interface Kolmogorov Complexity).

Software Tools

In the same way as with a hand-tool, the bulk of the complexity of a software tool is hidden behind it's interface. But, the more complex the *purpose* of the tool, the more complex the interface will be.

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

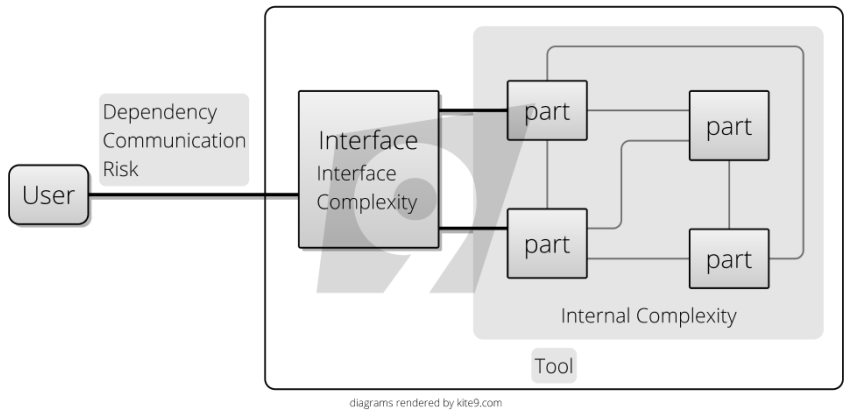

Figure 1.2: Types of Complexity For a Software Dependency

Software is not constrained by *physical* ergonomics in the same way as a tool is. But ideally, it should have conceptual ergonomics: ideally, complexity is hidden away from the user behind the Application Programming Interface (API)[4]. This is the familiar concept of Abstraction we've already looked at.

That is, the tool should be as simple to use and understand as possible. This is the Principal Of Least Astonishment[5]:

- **The abstractions should map easily to how the user expects the tool to work.** For example, I *expect* the trigger on a drill to start the drill turning.
- **The abstractions should leverage existing idioms and knowledge.** In a new car, I *expect* to know what the symbols on the dashboard mean, because I've driven other cars.
- **The abstractions provide me with only the functions I need.** Because everything else is confusing and gets in the way.

The way to win, then, is to allow a language to be extensible as-needed with features written by third parties. By supplying mechanisms for extension a language can provide insurances against the Boundary Risk of adopting it.

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

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

1.3 Types Of Software Dependencies

There are lots of ways you can depend on software. Here though, we're going to focus on just three main types: 1. **Code Your Own**: write some code ourselves to meet the dependency. 2. **Software Libraries**: importing code from the Internet, and using it in our project. Often, libraries are Open Source (this is what we'll consider here). 3. **Software as a Service**: calling a service on the Internet, (probably via `http`) This is often known as SaaS, or Software as a Service[6].

All 3 approaches involve a different risk-profile. Let's look at each in turn, from the perspective of which risks get mitigated, and which risks are accentuated.

1. Code Your Own

Initially, writing our own code was the only game in town: when I started programming, you had a user guide, BASIC and that was pretty much it. Tool support was very thin-on-the-ground. Programs and libraries could be distributed as code snippets *in magazines* which could be transcribed and run, and added to your program. This spirit lives on somewhat in StackOverflow and JSFiddle, where you are expected to "adopt" others' code into your own project.

One of the hidden risks of embarking on a code-your-own approach is that the features you need are *not* apparent from the outset. What might appear to be a trivial implementation of some piece of functionality can often turn into it's own industry as more and more hidden Feature Risk is uncovered.

For example, as we discussed in our earlier treatment of Dead-End Risk, building log-in screens *seemed like a good idea*. However, this gets out-of-hand fast when you need: - A password reset screen - To email the reset links to the user - An email verification screen - A lost account screen - Reminders to complete the sign up process - . . . and so on.

Unwritten Software

Sometimes, you will pick up a dependency on *unwritten software*. This commonly happens when work is divided amongst team members, or

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

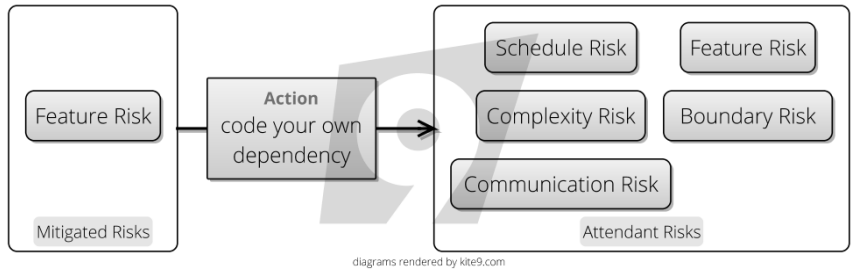

Figure 1.3: Code-Your-Own mitigates immediate feature risk, but at the expense of schedule risk, complexity risk and communication risk. There is also a hidden risk of features you don't yet know you need.

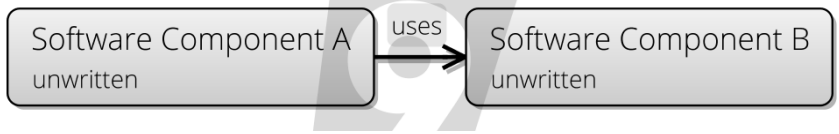

Figure 1.4: Sometimes, a module you're writing will depend on unwritten code

teams.

If a component **A** of our project *depends* on **B** for some kind of processing, you might not be able to complete **A** before writing **B**. This makes *scheduling* the project harder, and if component **A** is a risky part of the project, then the chances are you'll want to mitigate risk there first.

But it also hugely increases Communication Risk because now you're being asked to communicate with a dependency that doesn't really exist yet, *let alone* have any documentation.

There are a couple of ways to do this:

- **Standards**: If component **B** is a database, a queue, mail gateway or something else with a standard interface, then you're in luck. Write **A** to those standards, and find a cheap, simple implementation to test with. This gives you time to sort out exactly what implementation of **B** you're going for. This is not a great long-term solution, because obviously, you're not using the *real* dependency- you might get surprised when the behaviour of the

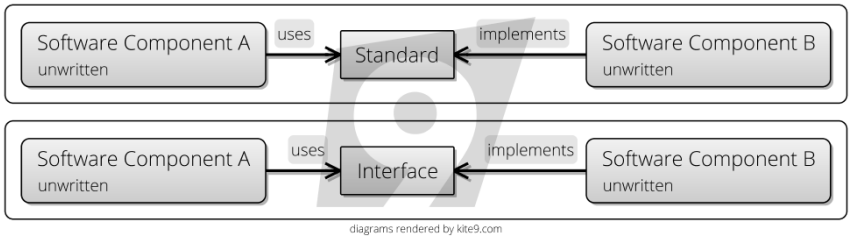

Figure 1.5: Coding to a standard on an interface breaks the dependency on unwritten software

real component is subtly different. But it can reduce Schedule Risk in the short-term.

- **Coding To Interfaces**: If standards aren't an option, but the surface area of **B** that **A** uses is quite small and obvious, you can write a small interface for it, and work behind that, using a Mock[7] for **B** while you're waiting for finished component. Write the interface to cover only what **A** *needs*, rather than everything that **B** *does* in order to minimize the risk of Leaky Abstractions[8].

Conway's Law

If the dependency is being written by another person, another team or in another country, communication risks pile up. When this happens, you will want to minimize *as much as possible* the interface complexity, since the more complex the interface, the worse the Communication Risk will be. The tendency then is to make the interfaces between teams or people *as simple as possible*, modularizing along these organisational boundaries.

In essence, this is Conway's Law[9]:

> "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." —M. Conway, *Conway's Law*

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

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

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

2. Software Libraries

By choosing a particular software library, we are making a move on the Risk Landscape in the hope of moving to place with more favourable risks. Typically, using library code offers a Schedule Risk and Complexity Risk Silver Bullet. But, in return we expect to pick up: - Communication Risk: because we now have to learn how to communicate with this new dependency. - Boundary Risk - because now are limited to using the functionality provided by this dependency. We have chosen it over alternatives and changing to something else would be more work and therefore costly.

But, it's quite possible that we could wind up in a worse place than we started out, by using a library that's out-of-date, riddled with bugs or badly supported. i.e. Full of new, hidden Feature Risk.

It's *really easy* to make bad decisions about which tools to use because the tools don't (generally) advertise their deficiencies. After all, they don't generally know how *you* will want to use them.

Software Libraries - Hidden Risks

Currently, choosing software dependencies looks like a "bounded rationality"-type process:

> "Bounded rationality is the idea that when individuals make decisions, their rationality is limited by the tractability of the decision problem, the cognitive limitations of their minds, and the time available to make the decision."
> - Bounded Rationality, *Wikipedia*[10]

Unfortunately, we know that most decisions *don't* really get made this way. We have things like Confirmation Bias[11] (looking for evidence to support a decision you've already made) and Cognitive Inertia[12] (ignoring evidence that would require you to change your mind) to contend with.

[10] https://en.wikipedia.org/wiki/Bounded_rationality

[11] https://en.wikipedia.org/wiki/Confirmation_bias

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

But, leaving that aside, let's try to build a model of what this decision making process *should* involve. Luckily, other authors have already considered the problem of choosing good software libraries, so let's start there.

In the table below, I am summarizing three different sources, which give descriptions of which factors to look for when choosing open-source libraries.

sd1 - Defending your code against dependency problems[13] sd2 - How to choose an open source library[14] sd3 - Open Source - To use or not to use[15]

Some take-aways:

- Feature Risk is a big concern. How can you be sure that the project will do what you want it to do ahead of schedule? Will it contain bugs or missing features? By looking at factors like *release frequency* and *size of the community* you get a good feel for this which is difficult to fake.
- Boundary Risk is also very important. You are going to have to *live* with your choices for the duration of the project, so it's worth spending the effort to either ensure that you're not going to regret the decision, or that you can change direction later.
- Third is Communication Risk: how well does the project deal with it's users? If a project is "famous", then it has communicated its usefulness to a wide, appreciative audience. Avoiding Communication Risk is also a good reason to pick *tools you are already familiar with*.

Complexity Risk?

One thing that none of the sources consider (at least from the outset) is the Complexity Risk of using a solution: - Does it drag in lots of extra dependencies that seem unnecessary for the job in hand? If so, you could end up in Dependency Hell[16], with multiple, conflicting

[13] https://www.software.ac.uk/resources/guides/defending-your-code-against-dependency-problems

[14] https://stackoverflow.com/questions/2960371/how-to-choose-an-open-source-library

[15] https://www.forbes.com/sites/forbestechcouncil/2017/07/20/open-source-to-use-or-not-to-use-and-how-to-choose/2/#39e67e445a8c

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

	Co Coordination Risk	B Boundary Risk	F Feature Risk	Co Communication Risk	Sources
Is the project "famous"?		☒	☒	☒	[sd2] [sd3]
Is there evidence of a large user community on user forums or e-mail list archives?		☒	☒	☒	[sd1]
Who is developing and maintaining the project? (Track Record)		☒	☒	☒	[sd3]
What are the mechanisms for supporting the software (community support, direct email, dedicated support team), and how long will the support be available? The more support, the better			☒	☒	[sd3]
Is the API to your liking?				☒	[sd2]
Are there examples of using the software successfully in the manner you want to use it?			☒		[sd1]
Are all the features you need, and think you will need, included?			☒		[sd1]
How mature is the project?			☒		[sd2]
In respect to the software licence, do you have the right to use the software in its intended production environment, or the right to distribute it along with your software?			☒		[sd1]
What is its deprecation or versioning policy? Does it have one? If not then it may be more unstable and features may disappear without warning between versioning, especially if releases are frequent.		☒	R Regression Risk	☒	[sd1]
What does the codebase look like?			Im Implementation Risk		[sd1]
How frequent are its releases?			☒		[sd1] [sd2] [sd3]
How well documented is the project?				☒	[sd2]
Does the software have evidence of a sustainable future (e.g. is there a roadmap)?				☒	[sd2]
Does the software support open standards? If it does, it will be easier to replace the software should it come to the end of its lifetime		☒			[sd1]
Does the version you intend to use come from a forked open-source project, or is it from the original source project? If so, which source is more appropriate?		☒			[sd1]
Are there any alternatives to the software?		☒			[sd1]
Has your community converged on using a particular software package?	☒				[sd1]
Totals	**1**	**9**	**15**	**8**	

Figure 1.6: Software Dependencies

versions of libraries in the project. - Do you already have a dependency providing this functionality? So many times, I've worked on projects that import a *new* dependency when some existing (perhaps transitive) dependency has *already brought in the functionality*. For example, there are plenty of libraries for JSON[17] marshalling, but if I'm also using a web framework the chances are it already has a dependency on one already. - Does it contain lots of functionality that isn't relevant to the task you want it to accomplish? e.g. Using Java when a shell script would do (on a non-Java project)

To give an extreme example of this, I once worked on an application which used Hazlecast[18] to cache log-in session tokens for a 3rd party datasource. But, the app is only used once every month, and session IDs can be obtained in milliseconds. So... why cache them? Although Hazlecast is an excellent choice for in-memory caching across multiple JVMs, it is a complex piece of software (after all, it does lots of stuff). By doing this, you have introduced extra dependency risk, cache invalidation risks, networking risks, synchronisation risks and so on, for actually no benefit at all... Unless, it's about CV Building.

Sometimes, the amount of complexity *goes up* when you use a dependency for *good reason*. For example, in Java, you can use Java Database Connectivity (JDBC)[19] to interface with various types of database. Spring Framework[20] (a popular Java library) provides a thing called a `JDBCTemplate`. This actually makes your code *more* complex, and can prove very difficult to debug. However, it prevents some security issues, handles resource disposal and makes database access more efficient. None of those are essential to interfacing with the database, but not using them is Technical Debt that can bite you later on.

1.4 3. Software as a Service

Businesses opt for Software as a Service (SaaS) because: - It vastly reduces the Complexity Risk they face in their organisations. e.g. managing the software or making changes to it. - Payment is usually based on *usage*, mitigating Schedule Risk. e.g. Instead of having to pay for

[17] https://en.wikipedia.org/wiki/JSON

[18] https://en.wikipedia.org/wiki/Hazelcast

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

[20] https://en.wikipedia.org/wiki/Spring_Framework

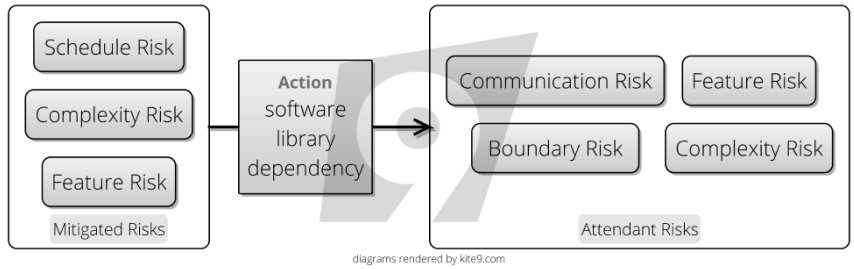

Figure 1.7: Software Libraries Risk Tradeoff

in-house software administrators, they can leave this function to the experts. - Potentially, you outsource the Operational Risk to a third party. e.g. ensuring availability, making sure data is secure and so on.

SaaS is now a very convenient way to provide *commercial* software. Popular examples of SaaS might be SalesForce[21], or GMail[22]. Both of which follow the commonly-used Freemium[23] model, where the basic service is provided free, but upgrading to a paid account gives extra benefits.

By providing the software on their own servers, the commercial organisation has a defence against *piracy*, as well as being able to control the Complexity Risk of the their environment (e.g. not having to support *every* version of the software that's ever been released).

Let's again recap the risks raised in some of the available literature:

sd4 - Saas Checklist - Nine Factors to Consider[24] sd5 - How to Evaluate Saas Vendors[25]

Some take-aways:

- Clearly, Operational Risk is now a big concern. By depending on a third-party organisation you are tying yourself to its success or failure in a much bigger way than just by using a piece of open-source software. What happens to data security, both in the data centre and over the internet?

[21] https://en.wikipedia.org/wiki/Salesforce.com

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

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

[24] https://www.zdnet.com/article/saas-checklist-nine-factors-to-consider-when-selecting-a-vendor/

[25] http://sandhill.com/article/how-to-evaluate-saas-vendors-five-key-considerations/

	Op Operational Risk	B Boundary Risk	F Feature Risk	Co Communication Risk	Sh Schedule Risk	Sources
How does the support process hold up in your trial runs?	⊠					[sd4]
What's the backup plan? (It's vital that you understand how your data are protected, and what redundancies are available should your SaaS provider have an outage.)	⊠	⊡				[sd4] [sd5]
What happens to your data if you sever ties with the vendor?		⊡				[sd4]
Are your current and future user environments supported? (e.g. Browser Compatibility)		⊡	⊠			[sd4]
Can you test in parallel? (i.e. run existing and new SaaS system together)			⊠			[sd4]
How does functionality compare to maturity?			⊠			[sd4]
What's the pricing model? (What might cause a price increase?)		⊡			⊠	[sd4]
What migration and training assistance options are available?				⊠		[sd4]
What integration options are available? (Are there APIs you can use to get at your data?)				⊠		[sd5]
Security (What standards and controls are in place?)	⊠					[sd5]
Service Level Agreements (SLAs) (What are the guarantees? What happens when the service levels are not met?)	⊠					[sd5]
Global Reach. (Is the service usable everywhere you need it?)	⊠					[sd5]
Totals	**5**	**4**	**3**	**2**	**1**	

Figure 1.8: Software As A Service Dependencies

- With Feature Risk you now have to condend with the fact that the software will be upgraded *outside your control*, and you may have limited control over which features get added or changed.
- Boundary Risk is a also a different proposition: you are tied to the software provider by *a contract*. If the service changes in the future, or isn't to your liking, you can't simply fork the code (like you could with an open source project).

1.5 A Matrix of Options

We've looked at just 3 different ways of providing a software dependency: SaaS, Libraries and code-your-own.

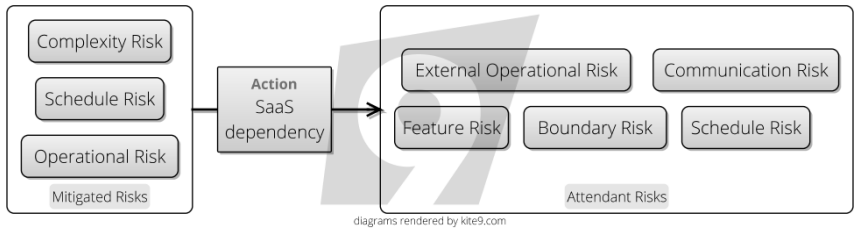

Figure 1.9: Risk Tradeoff From Using _Software as a Service (SaaS)

But these are not the only ways to do it, and there's clearly no one *right* way. Although here we have looked just at "Commercial Saas" and "Free Open Source", in reality, these are just points in a two-dimensional space involving *Pricing* and *Hosting*.

Let's expand this view slightly and look at where different pieces of software sit on these axes:

- Where there is value in the Network Effect[26], it's often a sign that the software will be free, or open source: programming languages and Linux are the obvious examples of this. Bugs are easier to find when there are lots of eyes looking, and learning the skill to use the software has less Boundary Risk if you know you'll be able to use it at any point in the future.
- At the other end of the spectrum, clients will happily pay for software if it clearly **reduces complexity**. Take Amazon Web Services (AWS)[27]. The essential trade here is that you substitute the complexity of hosting and maintaining various pieces of software, in exchange for monthly payments (Funding Risk for you). Since the AWS *interfaces* are specific to Amazon, there is significant Boundary Risk in choosing this option.
- In the middle there are lots of **substitute options** and therefore high competition. Because of this, prices are pushed towards zero, and and therefore often advertising is used to monetarize the product. Angry Birds[28] is a classic example: initially, it had demo and paid versions, however Rovio[29] discovered there was

[26] https://en.wikipedia.org/wiki/Network_effect

[27] https://en.wikipedia.org/wiki/Amazon_Web_Services

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

[29] https://en.wikipedia.org/wiki/Rovio_Entertainment

Pricing	On Premises 3rd Party	In Cloud / Browser 3rd Party	Risk Profile
Free	OSS Libraries • Tools • Java • Firefox • Linux • Programming Languages	Freemium • Splunk • Spotify • GitHub	• Low Boundary Risk Drives Adoption • Value In Network Effect
Advertising Supported	Commercial Software • Phone Apps • e.g. Angry Birds	Commercial SaaS • Google Search • GMail • Twitter	• Low Boundary Risk • High Availability Of Substitutes
Monthly / Metered Subscription	Commercial Software • Oracle Database • Windows • Office	Commercial SaaS • Office 365 • Amazon Web Services • SalesForce	Easy arguments for reduced: • Complexity Risk • Communication Risk • Coordination Risk Higher Boundary Risk

Figure 1.10: Software Dependencies, Pricing, Delivery Matrix Risk Profiles

much more money to be made through advertising than from the paid-for app[30].

Managing Risks

So far, we've considered only how the different approaches to Software Dependencies change the landscape of risks we face to mitigate some Feature Risk or other.

But with Software Dependencies we can construct dependency networks to give us all kinds of features and mitigate all kinds of risk. That is, *the features we are looking for are to mitigate some kind of risk*.

For example, I might start using WhatsApp[31] for example, because I want to be able to send my friends photos and text messages. However, it's likely that those same features are going to allow us to mitigate Communication Risk and Coordination Risk when we're next trying to meet up.

Let's look at some:

Risk	Examples of Software Mitigating That Risk
Coordination Risk	Calendar tools, Bug Tracking, Distributed Databases
Map-And-Territory-Risk	The Internet, generally. Excel, Google, "Big Data", Reporting tools
Schedule-Risk	Planning Software, Project Mangement Software
Communication-Risk	Email, Chat tools, CRM tools like SalesForce, Forums, Twitter, Protocols
Process-Risk	Reporting tools, online forms, process tracking tools
Agency-Risk	Auditing tools, transaction logs, Timesheet software, HR Software
Operational-Risk	Support tools like ZenDesk, Grafana, InfluxDB, Geneos
Feature-Risk	Every piece of software you use!

[30] https://www.deconstructoroffun.com/blog/2017/6/11/how-angry-birds-2-multiplied-quadrupled-revenue-in-a-year

[31] https://en.wikipedia.org/wiki/WhatsApp

1.6 Back To Ergonomics

What's clear from this analysis is that software dependencies don't *conquer* any risk - the moves they make on the Risk Landscape are *subtle*. Whether or not you end up in a more favourable position risk-wise is going to depend heavily on the quality of the execution and the skill of the implementor.

In particular, *choosing* dependencies can be extremely difficult. As we discussed above, the usefulness of any tool depends on its fit for purpose, it's *ergonomics within a given context*. It's all too easy to pick a good tool for the wrong job:

> "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." - Abraham Maslow, *Toward a Psychology of Being*[32]

With software dependencies, we often have to live with the decisions we make for a long time. In my experience, given the Boundary Risks associated with getting this wrong, not enough time is spent really thinking about this in advance.

Let's take a closer look at this problem in the next section, Boundary Risk.

[32] https://en.wiktionary.org/wiki/if_all_you_have_is_a_hammer,_everything_looks_like_a_nail

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