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

Contents ii

Preface iii

Executive Summary ix

I Introduction 1

II Risk 3

1 Boundary Risk 5

III Preview 27

Glossary 31

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

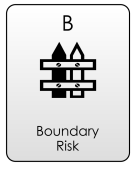
CHAPTER 1

Boundary Risk

In the previous few sections on Dependency Risk we've touched on Boundary Risk several times, but now it's time to tackle it head-on and discuss this important type of risk.

In terms of the Risk Landscape, Boundary Risk is exactly as it says: a *boundary*, *wall* or other kind of obstacle in your way to making a move you want to make. This changes the nature of the Risk Landscape, and introduces a maze-like component to it. It also means that we have to make *decisions* about which way to go, knowing that our future paths are constrained by the decisions we make.

And, as we discussed in Complexity Risk, there is always the chance we end up at a Dead End, and we've done work that we need to throw away. In this case, we'll have to head back and make a different decision.



- Risks due to the choices we make around dependencies, and the limitations they place on our ability to change.

Figure 1.1: Boundary Risk

- The risk that a particular approach to a change will fail. Caused by the fact that at some level, our internal models are not a complete reflection of reality.

Figure 1.2: Dead-End Risk

1.1 Emergence Through Choice

Boundary Risk is an emergent risk, which exists at the intersection of Complexity Risk, Dependency Risk and Communication Risk. Because of that, it's going to take a bit of time to pick it apart and understand it, so we're going to build up to this in stages.

Let's start with an obvious example: Musical Instruments. Let's say you want to learn to play some music. There are a *multitude* of options available to you, and you might choose an *uncommon* instrument like a Balalaika[1] or a Theremin[2], or you might choose a *common* one like a piano or guitar. In any case, once you start learning this instrument, you have picked up the three risks above:

- Dependency Risk You have a *physical* Dependency on it in order to play music, so get to the music shop and buy one.
- Communication Risk: You have to *communicate* with the instrument in order to get it to make the sounds you want. And you have Learning Curve Risk in order to be able to do that.
- Complexity Risk: As *a music playing system*, you now have an extra component (the instrument), with all the attendant complexity of looking after that instrument, tuning it, and so on.

Those risks are true for *any* instrument you choose. However, if you choose the *uncommon* instrument like the Balalaika, you have *worse* Boundary Risk, because the *ecosystem* for the balalaika is smaller. It might be hard to find a tutor, or a band needing a balalaika. You're unlikely to find one in a friend's house (compared to the piano, say).

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

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

Even choosing the Piano has Boundary Risk. By spending your time learning to play the piano, you're mitigating Communication Risk issues, but *mostly*, your skills won't be transferrable to playing the guitar. Your decision to choose one instrument over another cements the Boundary Risk: you're following a path on the Risk Landscape and changing to a different path is *expensive*.

Also, it stands to reason that making *any* choice is better than making *no* choice, because you can't try and learn *all* the instruments. Doing that, you'd make no meaningful progress on any of them.

1.2 Boundary Risk For Software Dependencies

Let's look at a software example now.

As discussed in Software Dependency Risk, if we are going to use a software tool as a dependency, we have to accept the complexity of it's Interface, and learn the protocol of that interface. If you want to work with it, you have to use it's protocol, it won't come to you.

Let's take a look at a hypothetical system structure, in the accompanying diagram. In this design, we have are transforming data from the `input` to the `output`. But how should we do it?

- We could go via `a`, using the Protocols of `a`, and having a dependency on `a`.
- We could go via `b`, using the Protocols of `b`, and having a dependency on `b`.
- We could choose the middle route, and avoid the dependency, but potentially pick up lots more Complexity Risk and Schedule Risk.

This is a basic **Translation** job from `input` to `output`. Since we are talking about **Translation**, we are clearly talking about Communication Risk again: our task in **Integrating** all of these components is *to get them to talk to each other*.

From a Cyclomatic Complexity point of view, this is a very simple structure, with low Complexity. But the choice of approach presents us with Boundary Risk, because we don't know that we'll be able to make them *talk to each other* properly:

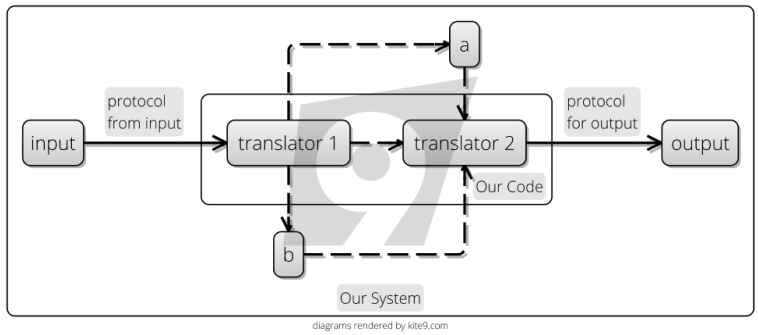


Figure 1.3: Our System receives data from the `input`, translates it and sends it to the `output`. But which dependency should we use for the translation, if any?

- Maybe a outputs dates in a strange calendar format that we won't understand.
- Maybe b works on some streaming API basis, that is incompatible with the input protocol.
- Maybe a runs on Windows, whereas our code runs on Linux.

. . . and so on.

1.3 Boundary Risk Pinned Down

Wherever we integrate dependencies with complex Protocols, we potentially have Boundary Risk. The more complex the systems being integrated, the higher the risk. When we choose software tools or libraries to help us build our systems, we are trading Complexity Risk for Boundary Risk. It is:

- The *sunk cost* of the Learning Curve we've overcome to integrate the dependency, when it fails to live up to expectations.
- The likelihood of, and costs of changing in the future.
- The rarity of alternatives (or, conversely, the risk of Lock In).

As we saw in Software Dependency Risk, Boundary Risk is a big factor in choosing libraries and services. However, it can apply to any kind of dependency:

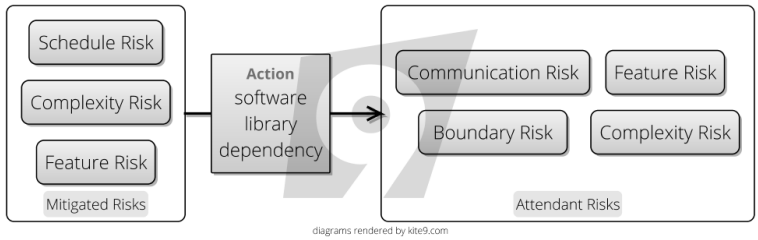


Figure 1.4: The tradeoff for using a library

- If you're depending on a Process or Organisation, they might change their products or quality, making the effort you put into the relationship worthless.
- If you're depending on Staff, they might leave, meaning your efforts on training them don't pay back as well as you hoped.
- If you're depending on an Event occuring at a particular time, you might have a lot of work to reorganise your life if it changes time or place.

1.4 Avoiding Boundary Risk Now...

Because of Boundary Risk's relationship to Learning Curve Risk, we can avoid accreting it by choose the *simplest* and *fewest* dependencies for any job. Let's look at some examples:

- `mkdirp` is an npm[3] module defining a single function. This function takes a single string parameter and recursively creating directories. Because the protocol is so simple, there is almost no Boundary Risk.
- Using a database with a JDBC[4] driver comes with *some* Boundary Risk: but the boundary is specified by a standard. Although the standard doesn't cover every aspect of the behaviour of the database, it does minimize risk, because if you are familiar with one JDBC driver, you'll be familiar with them all, and swapping one for another is relatively easy.

[3] `https://www.npmjs.com`

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

- Using a framework like Spring[5], Redux[6] or Angular[7] comes with higher Boundary Risk: you are expected to yield to the framework's way of behaving throughout your application. You cannot separate the concern easily, and swapping out the framework for another is likely to leave you with a whole new set of assumptions and interfaces to deal with.

1.5 . . . And In The Future

Unless your project *ends*, you can never be completely sure that Boundary Risk *isn't* going to stop you making a move you want. For example:

- `mkdirp` might not work on a new device's Operating System, forcing you to swap it out.
- You might discover that the database you chose satisfied all the features you needed at the start of the project, but came up short when the requirements changed later on.
- The front-end framework you chose might go out-of-fashion, and it might be hard to find developers interested in working on the project because of it.

This third point is perhaps the most interesting aspect of Boundary Risk: how can we ensure that the decisions we make now are future-proof? In order to investigate this further, let's look at three things: Plugins, Ecosystems and Evolution (again).

1.6 Plugins, Ecosystems and Evolution

On the face of it, WordPress[8] and [Drupal](https://en.wikipedia.org/wiki/Drupal *should* be very similar:

- They are both Content Management Systems[9]
- They both use a LAMP (Linux, Apache, MySql, PHP) Stack[10]

[5] `https://spring.io`

[6] `https://redux.js.org`

[7] `https://angularjs.org`

[8] `https://en.wikipedia.org/wiki/WordPress`

[9] `https://en.wikipedia.org/wiki/Content_management_system`

[10] `https://en.wikipedia.org/wiki/LAMP_(software_bundle)`

- They were both started around the same time (2001 for Drupal, 2003 for WordPress)
- They are both Open-Source, and have a wide variety of Plugins[11]. That is, ways for other programmers to extend the functionality in new directions.

In practice, they are very different. This could be put down to different *design goals*: it seems that WordPress was focused much more on usability, and an easy learning curve, whereas Drupal supported plugins for building things with complex data formats. It could also be down to the *design decisions*: although they both support Plugins, they do it in very different ways.

(Side note: I wasn't short of go-to examples for this. I could have picked on Team City[12] and Jenkins[13] here (Continuous Integration[14] tools), or Maven[15] and Gradle[16] (build tools). All of these support plugins), and the *choice* of plugins is dependent on which I've chosen, despite the fact that the platforms are solving pretty much the same problems.)

Ecosystems and Systems

The quality, and choice of plugins for a given platform, along with factors such as community and online documentation is often called its ecosystem[17]:

> "as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them" - Software Ecosystem, *Wikipedia*

You can think of the ecosystem as being like the footprint of a town or a city, consisting of the buildings, transport network and the people that

[11] https://en.wikipedia.org/wiki/Plug-in_(computing)

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

[13] https://en.wikipedia.org/wiki/Jenkins_(software)

[14] https://en.wikipedia.org/wiki/Continuous_integration

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

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

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

live there. Within the city, and because of the transport network and the amenities available, it's easy to make rapid, useful moves on the Risk Landscape. In a software ecosystem it's the same: the ecosystem has gathered together to provide a way to mitigate various different Feature Risks in a common way.

tbd: talk about complexity within the boundary. (increased convenience?)

Ecosystem size is one key determinant of Boundary Risk: a *large* ecosystem has a large boundary circumference. Boundary Risk is lower because your moves on the Risk Landscape are unlikely to collide with it. The boundary *got large* because other developers before you hit the boundary and did the work building the software equivalents of bridges and roads and pushing it back so that the boundary didn't get in their way.

In a small ecosystem, you are much more likely to come into contact with the edges of the boundary. *You* will have to be the developer that pushes back the frontier and builds the roads for the others. This is hard work.

Evolution

In the real world, there is a tendency for *big cities to get bigger*. The more people that live there, the more services they provide, and therefore, the more immigrants they attract. And, it's the same in the software world. In both cases, this is due to the Network Effect[18]:

> "A network effect (also called network externality or demand-side economies of scale) is the positive effect described in economics and business that an additional user of a good or service has on the value of that product to others. When a network effect is present, the value of a product or service increases according to the number of others using it."
> - Network Effect, *Wikipedia*

You can see the same effect in the adoption rates of WordPress and Drupal, shown in the chart below. Note: this is over *all sites on the*

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

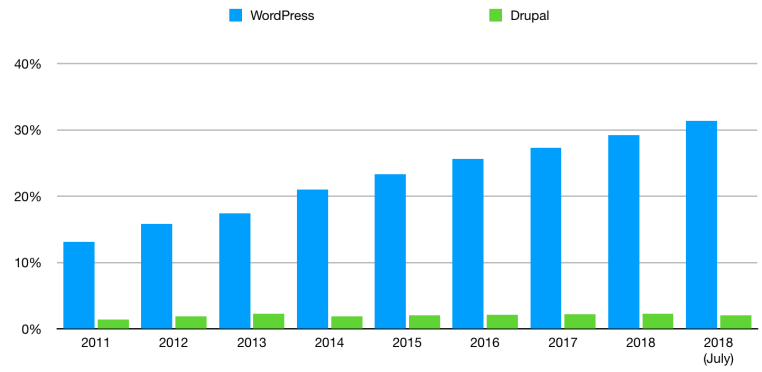


Figure 1.5: Wordpress vs Drupal adoption over 8 years, according to w3techs.com[19]

internet, so Drupal accounts for hundreds of thousands of sites. In 2018, WordPress is approximately 32% of all websites. For Drupal it's 2%.

Did WordPress gain this march because it was better than Drupal? That's arguable. That it's this way round could be *entirely accidental*, and a result of Network Effect.

And maybe, they aren't comparable: Given the same problems, the people in each ecosystem have approached them and solved them in different ways. And, this has impacted the 'shape' of the abstractions, and the protocols you use in each. Complexity *emerges*, and the ecosystem gets more complex and opinionated, much like the way in which the network of a city will evolve over time in an unpredictable way.

But, by now, if they *are* to be compared side-by-side, WordPress *should be better* due to the sheer number of people in this ecosystem who are. . .

- Creating web sites.
- Using those sites.
- Submitting bug requests.
- Fixing bugs.
- Writing documentation.
- Building plugins.
- Creating features.

- Improving the core platform.

But, there are two further factors to consider. . .

1. The Peter Principle

When a tool or platform is popular, it is under pressure to increase in complexity. This is because people are attracted to something useful, and want to extend it to new purposes. This is known as *The Peter Principle*:

> "The Peter principle is a concept in management developed by Laurence J. Peter, which observes that people in a hierarchy tend to rise to their 'level of incompetence'." - The Peter Principle, *Wikipedia*[20]

Although designed for *people*, it can just as easily be applied to any other dependency you can think of. Let's look at Java[21] as an example of this.

Java is a very popular platform. Let's look at how the number of public classes (a good proxy for the boundary) has increased with each release:

Why does this happen?

- More and more people are using Java for more and more things. It's popularity begets more popularity.
- Human needs are *fractal* in complexity. You can always find ways to make a dependency *better* (For some meaning of better).
- There is Feature Drift Risk: our requirements evolve with time. Android Apps[22] weren't even a thing when Java 3 came out, for example, yet they are all written in Java now, and Java has had to keep up.

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

[21] https://en.wikipedia.org/wiki/Java_(software_platform)

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

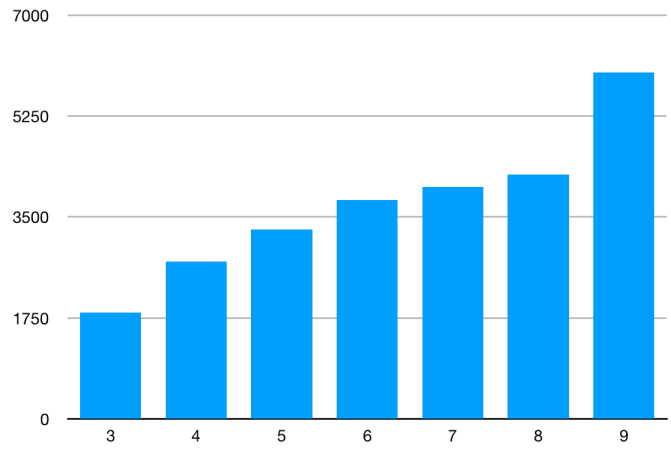


Figure 1.6: Java Public Classes By Version (3-9)

2. Backward Compatibility

As we saw in Software Dependency Risk, The art of good design is to afford the greatest increase in functionality with the smallest increase in complexity possible, and this usually means Refactoring. But, this is at odds with Backward Compatibility.

Each new version has a greater functional scope than the one before (pushing back Boundary Risk), making the platform more attractive to build solutions in. But this increases the Complexity Risk as there is more functionality to deal with.

Focus vs Overreach

You can see in the diagram the Peter Principle at play: as more responsibility is given to a dependency, the more complex it gets, and the greater the learning curve to work with it. Large ecosystems like Java react to Learning Curve Risk by having copious amounts of literature to read or buy to help, but it is still off-putting.

Because Complexity is Mass, large ecosystems can't respond quickly to Feature Drift. This means that when the world changes, *new* systems will come along to plug the gaps.

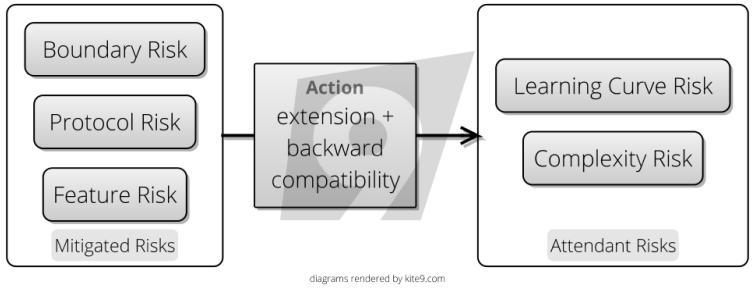


Figure 1.7: The Peter Principle: Backward Compatibility + Extension leads to complexity and learning curve risk

This implies a trade-off: - Sometimes it's better to accept the Boundary Risk innate in a smaller system than try to work within the bigger, more complex system.

example:

In the late 80's and 90's there was a massive push towards *building functionality in the database*. Relational Database Managment Systems (RDBMSs) were all-in-one solutions, expensive platforms that you purchased and built *everything* inside. However, this dream didn't last:

why? (need some research here).

This tbd

tbd. diagram here.

1.7 Beating Boundary Risk With Standards

Sometimes, technology comes along that allows us to cross boundaries, like a *bridge* or a *road*. This has the effect of making it easy to to go from one self-contained ecosystem to another. Going back to WordPress, a simple example might be the Analytics Dashboard which provides Google Analytics[23] functionality inside WordPress.

I find, a lot of code I write is of this nature: trying to write the *glue code* to join together two different *ecosystems*.

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

Standards allow us to achieve the same thing, in one of two ways:

- **Mode 1: Abstract over the ecosystems.** Provide a *standard* protocol (a *lingua franca*) which can be converted down into the protocol of any of a number of competing ecosystems.
- **Mode 2: Force adoption.** All of the ecosystems start using the standard for fear of being left out in the cold. Sometimes, a standards body is involved, but other times a "de facto" standard emerges that everyone adopts.

Let's look at some examples:

- ASCII[24]: fixed the different-character-sets boundary risk by being a standard that others could adopt. Before everyone agreed on ASCII, copying data from one computer system to another was a massive pain, and would involve some kind of translation. Unicode[25] continues this work. (**Mode 1**)

- C[26]: The C programming language provided a way to get the same programs compiled against different CPU instruction sets, therefore providing some *portability* to code. The problem was, each different operating system would still have it's own libraries, and so to support multiple operating systems, you'd have to write code against multiple different libraries. (**Mode 2**)

- Java[27] took what C did and went one step further, providing interoperability at the library level. Java code could run anywhere where Java was installed. (**Mode 2**)

- Internet Protocol[28]: As we saw in Communication Risk, the Internet Protocol (IP) is the *lingua franca* of the modern internet. However, at one period of time, there were many competing standards. and IP was the ecosystem that "won", and was subsequently standardized by the IETF[29]. (**Mode 1**)

[24] https://en.wikipedia.org/wiki/ASCII

[25] https://en.wikipedia.org/wiki/Unicode

[26] https://en.wikipedia.org/wiki/C_(programming_language)

[27] https://en.wikipedia.org/wiki/Java_(programming_language)

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

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

1.8 Complex Boundaries

As shown in the above diagram, mitigating Boundary Risk involves taking on complexity. The more Protocol Complexity there is to bridge the two ecosystems, the more Complex the bridge will necessarily be.

Protocol Risk[a] From A [a]	[Protocol Risk][br2] From B	Resulting Bridge Complexity	Example
Low	Low	Simple	Changing from one date format to another.
High	Low	Moderate	Status Dashboard, tbd
High	High	Complex	Object-Relational Mapping (ORM) Tools, (see below)
High + Evolving	Low	Moderate, Versioned	Simple Phone App, e.g. note-taker or calculator
Evolving	High	Complex	Modern browser (see below)
Evolving	Evolving	Very Complex	Google Search, Scala (see below)

From examining the Protocol Risk[30] at each end of the bridge you are creating, you can get a rough idea of how complex the endeavour will be:

- If it's low-risk at both ends, you're probably going to be able to knock it out easily. Like translating a date, or converting one file format to another.
- Where one of the protocols is *evolving*, you're definitely going to need to keep releasing new versions. The functionality of a `Calculator` app on my phone remains the same, but new versions have to be released as the phone APIs change, screens change resolution and so on.

[30]

- tbd

Where boundaries

tbd Trying to create a complex, fractal surface. User requirements are fractal in nature.

Object-Relational Mapping

For example, Object Relational Mapping (ORM) has long been a problem in software. This is [Boundary-Crossing] software trying to bridge the gap between Relational Databases and Object-Oriented Languages like [Java]. Building a *general* library that does this and is useful tbd said:

> 'ORM is the vietnam of . . . ' -

This is a particularly difficult problem because the two ecosystems are so *rich* and *complex* in the functionality they expose. But what are the alternatives?

- Either back to building functionality within the database again, using stored procedures
- Building Object Oriented Databases. It's interesting that neither of these really worked out.
- Custom-building the bridge between the systems, one database call at-a-time in your own software.

This is tbd hobson's choice, there is strong debate about whether ORM is a worse trade of mitigated Boundary Risk for attendant Complexity Risk or not, and clearly will depend on your circumstances.

Scala

Mapping between complex boundaries is expecially difficult if the Boundaries are evolving and changing as you go. This means in ecosystems that are changing rapidly, you are unlikely to be able to create lasting bridges between them. Given that Java is an old, large

and complex ecosystem, you would imagine that it would have a slow-enough rate of change that abstracting technologies can be built on top of it safely.

Indeed, we see that happening with Clojure and Kotlin, two successful languages built on top of the Java Virtual Machine (JVM) and offering compatibility with it.

Scala is arguably the first mainstream language that tried to do the same thing: it is trying to build a Functional Programming paradigm on top of the Java Virtual Machine (JVM), which traditionally has an Object Oriented paradigm.

The problem faced by Scala is that Java didn't stay still: as soon as they demonstrated some really useful features in Scala (i.e. stream-based processing), Java moved to include this new functionality too. If they hadn't, the developer community would have slowly drifted away and used Scala instead.

So, in a sense, Scala is a *success story*: they were able to force change to Java. But, once Java had changed, Scala was in the difficult position of having two sets of competing features in the platform: the existing Scala streams, and the new Java streams.

Clojure can interop with Java because on one side, the boundary is simple: lisp is a simple language which lends itself to reimplementation within other platforms. Therefore, the complexity of the bridge is *simple*: all that needs to be provided is a way to call methods from Java to clojure.

Scala and Java have a complex relationship because Scala creates it's own complex boundary: it is syntactically and functionally a broad language with lots of features. And so is Java. Mapping from one to the other is therefore

for interop here. Why is one so different from the other?

Browsers

Web browsers are another suprisingly complex boundary. They have to understand the following protocols:

- HTTP for loading resources (as we already reviewed in Complexity Risk
- HTML Pages, for describing the content of web pages Complexity Risk
- Various image formats
- Javascript for web-page *interactivity*
- CSS for web-page styling, animation and so on.
- . . . and several others.

Handling any one of these protocols alone is a massive endeavour, so browsers are built on top of Software Libraries which handle each concern, for example, Networking Libraries, Parsers and so on.

One way of looking at the browser is that it is a *function*, where those elements listed above are the *inputs* to the function, and the output is *what is displayed on the screen*, as shown in the image below.

tbd. browser as a function

There are three specific problems that make this a really complex boundary:

1. All of the standards above are *evolving and improving*. And, although HTML5 (say) is a reasonably well-specified standard, in reality, web pages tend not to adhere exactly to the letter of it. People make mistakes in the HTML they write, and it's up to the browser to try and figure out what they *meant* to write, rather than what they did write. This makes the *input* to the function extremely complex.

2. Similarly, the *output* of the function is not well defined either, and relies a lot on people's *subjective aesthetic judgement*. For example, if you insert a `<table>` into an HTML page, the specification doesn't say anything about exactly how big the table should be, the size of it's borders, the spacing of the content and so on. At least, initially, *none* of this was covered by the HTML Specification. The CSS specification is over time clearing this up, but it's not *exactly nailed down*, which means. . .

3. That because there are various different browsers (Chrome, Safari, Internet Explorer, Microsoft Edge, Firefox etc.) and each

browser has multiple different versions, released over a period of many years, you cannot, as a web-page developer know, *a priori* what your web-page will look like to a user.

As developers trying to build software to be delivered over the internet, this is therefore a source of common Boundary Risk. If you were trying to build software to work in *all browsers* and *all versions*, this problem would be nearly insurmountable. So, in order to tackle this risk, we do the following:

- We pick a small (but commonly used) subset of browsers, and use features from the specifications that we know commonly work in that subset.
- We test across the subset. Again, testing is *harder than it should be*, because of problem 2 above, that the expected output is not exactly defined. This generally means you have to get humans to apply their *subjective aesthetic judgement*, rather than getting machines to do it.
- There is considerable pressure on browser developers to ensure consistency of behaviour across the implementations. If all the browsers work the same, then we don't face the Boundary Risk of having to choose just one to make our software work in. However, it's not always been like this...

1.9 Vendor Lock-In

In the late 1990s, faced with the emergence of the nascent World Wide Web, and the Netscape Navigator browser, Microsoft adoped a strategy known as Embrace and Extend. The idea of this was to subvert the HTML standard to their own ends by *embracing* the standard and creating their own browser (Internet Explorer) and then *extending* it with as much functionality as possible, which would then *not work* in Netscape Navigator. They then embarked on a campaign to try and get everyone to "upgrade" to Internet Explorer. In this way, they hoped to "own" the Internet, or at least, the software of the browser, which they saw as analogous to being the "operating system" of the Internet, and therefore a threat to their own operating system, Windows.

There are two questions we need to ask about this, from the point-of-view of understanding Boundary Risk:

1. Why was this a successful strategy?
2. Why did they stop doing this?

Let's look at the first question then. Yes, it was a successful strategy. In the 1990s, browser functionality was rudimentary. Developers were *desperate* for more features, and for more control over what appeared on their webpages. And, Internet Explorer (IE) was a free download (or, bundled with Windows). By shunning other browsers and coding just for IE, developers pushed Boundary Risk to the consumers of the web pages and in return mitigated Dependency Fit Risk: they were able to get more of the functionality they wanted in the browser.

It's worth pointing out, *this was not a new strategy*:

- Processor Chip manufacturers had done something similar in the tbds: by providing features (instructions) on their processors that other vendors didn't have, they made their processors more attractive to system integrators. However, since the instructions were different on different chips, this created Boundary Risk for the integrators. Intel and Microsoft were able to use this fact to build a big ecosystem around Windows running on Intel chips (so called, Wintel).
- We have two main *mobile* ecosystems: Apple's [iOS] and Google's Android, which are both *very* different and complex ecosystems with large, complex boundaries. They are both innovating as fast as possible to keep users happy with their features. Tools like Xamarin exist which allow you to build
- Currently, Amazon Web Services (AWS) are competing with Microsoft Azure and [Google tbd] over building tools for Platform as a Service (PaaS) (running software in the cloud). They are both racing to build new functionality, but at the same time it's hard to move from one vendor to another as there is no standardization on the tools.
- As we saw above, Database vendors tried to do the same thing with features in the database. Oracle particularly makes money over differentiating itself from competitors by providing features

that other vendors don't have. Tom tbd provides a compelling argument for using these features thus:

tbd.

The next question, is why did Microsoft *stop* pursuing this strategy? It seems that the answer is because they were made to. tbd.

1.10 Everyday Boundary Risks

Boundary Risk occurs all the time. Let's look at some ways:

- **Configuration**: When software has to be deployed onto a server, there has to be configuration (usually on the command line, or via configuration property files) in order to bridge the boundary between the *environment it's running in* and the *software being run*. Often, this is setting up file locations, security keys and passwords, and telling it where to find other files and services.
- **Integration Testing**: Building a unit test is easy. You are generally testing some code you have written, aided with a testing framework. Your code and the framework are both written in the same language, which means low boundary risk. But, to *integration test* you need to step outside this boundary and so it becomes much harder. This is true whether you are integrating with other systems (providing or supplying them with data) or parts of your own system (say testing the client-side and server parts together).
- **User Interface Testing**: If you are supplying a user-interface, then the interface with the user is already a complex, under-specified risky protocol. Although tools exist to automate UI testing (such as Selenium, these rarely satisfactorily mitigate this protocol risk: can you be sure that the screen hasn't got strange glitches, that the mouse moves correctly, that the proportions on the screen are correct on all browsers?
- **Jobs**: When you pick a new technology to learn and add to your CV, it's worth keeping in mind how useful this will be to you in the future. It's career-limiting to be stuck in a dying ecosystem and need to retrain.

- **Teams**: if you're given license to build a new product within an existing team, are you creating Boundary Risk by using tools that the team aren't familiar with?
- **Organisatations**: Getting teams or departments to work with each other often involves breaking down Boundary Risk. Often the departments use different tool-sets or processes, and have different goals making the translation harder. tbd

Boundary Risk and Change

You can't always be sure that a dependency now will always have the same guarantees in the future:

- **Ownership changes** Microsoft buys Github. What will happen to the ecosystem around github now?
- **Licensing changes**. (e.g. Oracle[31] buys **Tangosol** who make Coherence[32] for example). Having done this, they increase the licensing costs of Tangosol to huge levels, milking the Cash Cow of the installed user-base, but ensuring no-one else is likely to use it.
- **Better alternatives become available**: As a real example of this, I began a project in 2016 using Apache Solr. However, in 2018, I would probably use ElasticSearch[33]. In the past, I've built websites using Drupal and then later converted them to use WordPress.

1.11 Patterns In Boundary Risk

In Feature Risk, we saw that the features people need change over time. Let's get more specific about this:

- Human need is Fractal. This means that over time, software products have evolved to more closely map to human needs. Software that would have delighted us ten years ago lacks the sophistication we expect today.

[31] http://oracle.com

[32] https://en.wikipedia.org/wiki/Oracle_Coherence

[33] https://en.wikipedia.org/wiki/Elasticsearch

- Software and hardware are both is improving with time, due to evolution and the ability to support greater and greater levels of complexity.
- Abstractions build too. As we saw in Process Risk, we *encapsulate* earlier abstractions in order to build later ones.

If all this is true, the only thing we can expect in the future is that the lifespan of any ecosystem will follow an arc through creation, adoption, growth, use and finally either be abstracted over or abandoned.

tbd diagram.

Although our discipline is a young one, we should probably expect to see "Software Archaeology" in the same way as we see it for biological organisms. Already we can see the dead-ends in the software evolutionary tree: COBOL and BASIC languages, CASE systems. Languages like FORTH live on in PostScript, SQL is still embedded in everything

Boundary risk is *inside* and *outside*

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