

RISK-FIRST
SOFTWARE DEVELOPMENT
DE-RISKED

Volume 1: The Menagerie



ROB MOFFAT

Risk First: The Menagerie

By Rob Moffat

Copyright © 2018 Kite9 Ltd.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed "Attention: Permissions Coordinator," at the address below.

ISBN: 9781717491855

Credits

tbd

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

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

Thanks to:

Books In The Series

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

Online

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

Published By

Kite9 Ltd.
14 Manor Close

Colchester
C06 4AR

Contents

Contents	iii
Preface	v
Executive Summary	xi
I Introduction	1
1 A Simple Scenario	3
II Risk	7
2 Complexity Risk	9
III Application	27
3 Coming Next	29
4 Estimates	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.

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, 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 by stating:

"It's all about risk" > Kent Beck

This is a promising start. From there, he introduces his methodology, 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 *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.com¹, and at the risk-first.org² 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 Wikipedia³ 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, 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 of knowledge not being widely-enough shared.
- If we write Unit Tests, we're also mitigating the risk of bugs going to production, but we're also mitigating against future changes breaking our existing functionality.
- If we enter into a contract with a supplier, we are mitigating the risk of the supplier vanishing and leaving us exposed. With the contract in place, we have legal recourse against this risk.

Different tools are appropriate for mitigating different types of risks.

5. Different Methodologies for Different Risk Profiles

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

For example:

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

Although many developers have a methodology-of-choice, the argument here is that there are tradeoffs with all of these choices. Methodologies are like *bicycles*, rather than *religions*. Rather than simply *believing*, we can take them apart and see how they work.

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

6. Driving Development With a Risk-First Perspective

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

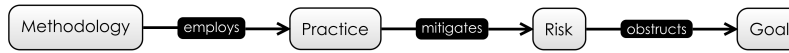


Figure 1: Methodologies, Risks, Practices

How do we take this further?

The first idea we explore is that of 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

A Simple Scenario

First up, I'm going to introduce a simple model for thinking about risk.

1.1 A Simple Scenario

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

Goal In Mind

Now, in this endeavour, we want to be successful. That is to say, we have a Goal In Mind we *probably* have to do some tasks.

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

Attendant Risk

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

When we go about preparing this wonderful evening, we can with these risks and try to mitigate them: shop for the ingredients in advance, prepare parts

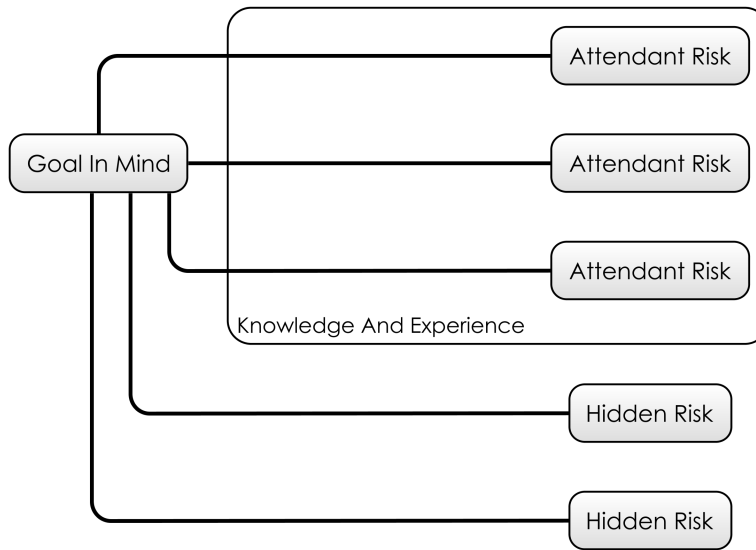


Figure 1.1: Goal In Mind, with the risks you know about

of the meal, maybe practice the cooking in advance. Or, we can wing it, and sometimes we'll get lucky.

How much effort we expend on mitigating Attendant Risks.

Hidden Risks

There are also hidden Attendant Risks¹ that you might not know about: if you're poaching eggs for dinner, you might know that fresh eggs poach best. These are the "Unknown Unknowns" of Rumsfeld's model².

Different people will evaluate the risks differently. (That is, worry about them more or less.) They'll also *know* about different risks. They might have cooked the recipe before, or organised lots more dinner parties than you.

How we evaluate the risks, and which ones we know about depends on our **knowledge** and **experience**, then. And that varies from person to person (or team to team). Lets call this our Internal Model.

¹Glossary#attendant-risk

²https://en.wikipedia.org/wiki/There_are_known_knowns

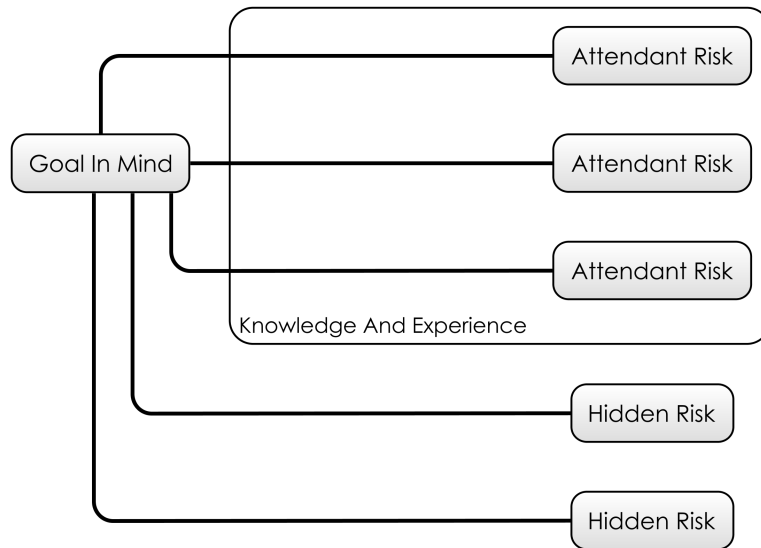


Figure 1.2: Goal In Mind, the risks you know about and the ones you don't

Model Meets Reality

As the dinner party gets closer, we make our preparations, and the inadequacies of the Internal Model reveal themselves; things we were worried about may not materialise, things we thought would be minor risks turn out to be greater.

Our model is forced into contact with reality, and the model changes.

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

1.2 On To Software

In this website, we're going to look at the risks in the software process and how these are mitigated by the various methodologies you can choose from.

Let's examine the scenario of a new software project, and expand on the simple model being outlined above: instead of a single person, we are likely

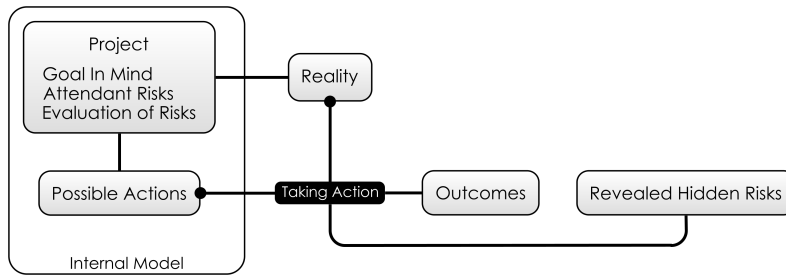


Figure 1.3: How taking action affects Reality, and also changes your internal model

to have a team, and our model will not just exist in our heads, but in the code we write.

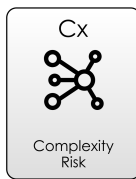
On to Development Process

Part II

Risk

Complexity Risk

Complexity Risk. However, in this chapter, we're going to be specifically focusing on *code you write*: the size of your code-base, the number of modules, the interconnectedness of the modules and how well-factored the code is.



- Risks caused by the weight of complexity in the systems we create, and their resistance to change and comprehension.

Figure 2.1: Complexity Risks

You could think of this chapter, then, as **Codebase Risk**: We'll look at three separate measures of codebase complexity and talk about Technical Debt, and look at places in which **Codebase Risk** is at it's greatest.

2.1 Kolmogorov Complexity

The standard Computer-Science definition of complexity, is Kolmogorov Complexity¹. This is:

“...is the length of the shortest computer program (in a pre-determined programming language) that produces the object as output.” - Kolmogorov Complexity, Wikipedia

¹https://en.wikipedia.org/wiki/Kolmogorov_complexity

This is a fairly handy definition for us, as it means that to in writing software to solve a problem, there is a lower bound on the size of the software we write. In practice, this is pretty much impossible to quantify. But that doesn't really matter: the techniques for *moving in that direction* are all that we are interested in, and this basically amounts to compression.

Let's say we wanted to write a javascript program to output this string:

abcdabcdabcdabcdabcdabcdabcdabcdabcd

We might choose this representation:

```
function out() { (7 )
  return "abcdabcdabcdabcdabcdabcdabcdabcd" (45)
} (1 )
```

The numbers in brackets indicate how many symbols each line contains, so in total, this code block contains **53 symbols**, if you count `function`, `out` and `return` as one symbol each.

But, if we write it like this:

```
const ABCD="ABCD"; (11)

function out() { (7 )
  return ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ABCD+ (16)
         ABCD+ABCD+ABCD; (6 )
} (1 )
```

With this version, we now have **41 symbols** (**ABCD** is a single symbol, because we could have called it anything). And with this version:

```
const ABCD="ABCD"; (11)

function out() { (7 )
  return ABCD.repeat(10) (7 )
} (1 )
```

... we have **26 symbols**.

Abstraction

What's happening here is that we're *exploiting a pattern*: we noticed that ABCD occurs several times, so we defined it a single time and then used it over and over, like a stamp. Separating the *definition* of something from the *use* of something as we've done here is called "abstraction". We're going to come across it over and over again in this part of the book, and not just in terms of computer programs.

By applying techniques such as Abstraction, we can improve in the direction of the Kolmogorov limit. And, by allowing ourselves to say that *symbols* (like out and ABCD) are worth one complexity point, we've allowed that we can be descriptive in our function name and const. Naming things is an important part of abstraction, because to use something, you have to be able to refer to it.

Trade-Off

But we could go further down into Code Golf² territory. This javascript program plays FizzBuzz³ up to 100, but is less readable than you might hope:

```
for(i=0;i<100;)document.write((((++i%3?'':'Fizz')+  
(i%5?'':'Buzz')||i)+"<br>") (62)
```

So there is at some point a trade-off to be made between Complexity Risk is about *misunderstanding*: The more complex a piece of software is, the more difficulty users will have understanding it, and the more difficulty developers will have changing it.

2.2 Connectivity

A second, useful measure of complexity comes from graph theory, and that is the connectivity of a graph:

“... the minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other”

—Connectivity, *Wikipedia*⁴

²https://en.wikipedia.org/wiki/Code_golf

³https://en.wikipedia.org/wiki/Fizz_buzz

⁴[https://en.wikipedia.org/wiki/Connectivity_\(graph_theory\)](https://en.wikipedia.org/wiki/Connectivity_(graph_theory))

To see this in action, have a look at the below graph:

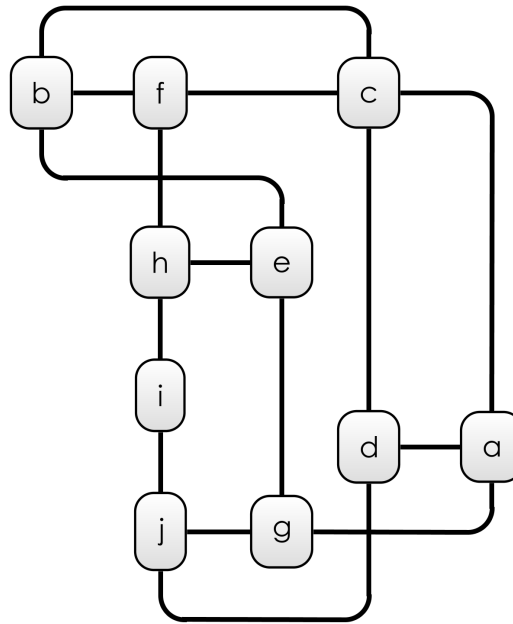


Figure 2.2: Graph 1

It has 10 vertices, labelled **a** to **j**, and it has 15 edges (or links) connecting the vertices together. If any single edge were removed from this diagram, the 10 vertices would still be linked together. Because of this, we can say that the graph is *2-connected*. That is, to disconnect any single vertex, you'd have to remove *at least* two edges.

As a slight aside, let's consider the **Kolmogorov Complexity** of this graph, by inventing a mini-language to describe graphs. It could look something like this:

```
<item> : [<item>,* <item>    # Indicates that the item
                                # before the colon
                                # has a connection to all
                                # the items after the colon
```

So our graph could be defined like this:

```

a: b,c,d
b: c,f,e
c: f,d
d: j
e: h,j
f: h
g: j
h: i
i: j

```

(39)

Let's remove some of those extra links:

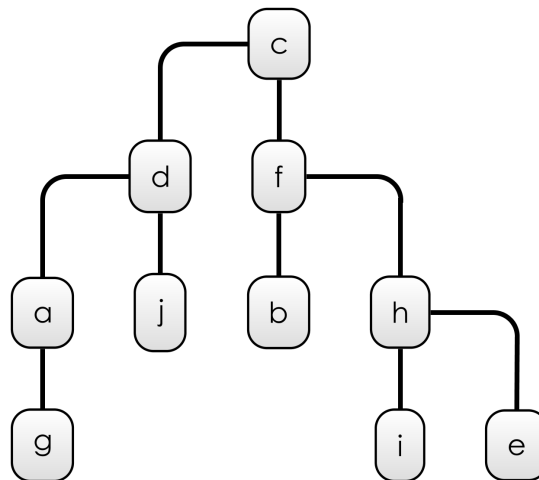


Figure 2.3: Graph 2

In this graph, I've removed 6 of the edges. Now, we're in a situation where if any single edge is removed, the graph becomes *unconnected*. That is, it's broken into distinct chunks. So, it's *1-connected*.

The second graph is clearly simpler than the first. And, we can show this by looking at the **Kolmogorov Complexity** in our little language:

```

a: d,g
b: f

```

```
c: d,f
d: j
f: h
e: h
h: i
```

(25)

Connectivity is also **Complexity**. Heavily connected programs/graphs are much harder to work with than less-connected ones. Even *laying out* the first graph sensibly is a harder task than the second (the second is a doddle). But the reason programs with greater connectivity are harder to work with is that changing one module potentially impacts many others.

2.3 Hierarchies and Modularization

In the second, simplified graph, I've arranged it as a hierarchy, which I can do now that it's only 1-connected. For 10 vertices, we need 9 edges to connect everything up. It's always:

$$\text{edges} = \text{vertices} - 1$$

Note that I could pick any hierarchy here: I don't have to start at **c** (although it has the nice property that it has two roughly even sub-trees attached to it).

How does this help us? Imagine if **a - j** were modules of a software system, and the edges of the graph showed communications between the different sub-systems. In the first graph, we're in a worse position: who's in charge? What deals with what? Can I isolate a component and change it safely? What happens if one component disappears? But, in the second graph, it's easier to reason about, because of the reduced number of connections and the new heirarchy of organisation.

On the downside, perhaps our messages have farther to go now: in the original **i** could send a message straight to **j**, but now we have to go all the way via **c**. But this is the basis of Modularization⁵ and Hierarchy⁶.

As a tool to battle complexity, we don't just see this in software, but everywhere in our lives. Society, business, nature and even our bodies:

⁵https://en.wikipedia.org/wiki/Modular_programming

⁶<https://en.wikipedia.org/wiki/Hierarchy>

- **Organelles** - such as Mitochondria⁷.
- **Cells** - such as blood cells, nerve cells, skin cells in the Human Body⁸.
- **Organs** - like hearts livers, brains etc.
- **Organisms** - like you and me.

The great complexity-reducing mechanism of modularization is that *you only have to consider your local environment*. Elements of the program that are “far away” in the hierarchy can be relied on not to affect you. This is somewhat akin to the **Principal Of Locality**:

“Spatial locality refers to the use of data elements within relatively close storage locations.”

—Locality Of Reference, *Wikipedia*⁹

2.4 Cyclomatic Complexity

A variation on this graph connectivity metric is our third measure of complexity, Cyclomatic Complexity¹⁰. This is:

$\text{Cyclomatic Complexity} = \text{edges} - \text{vertices} + 2P,$

Where **P** is the number of **Connected Components** (i.e. distinct parts of the graph that aren’t connected to one another by any edges).

So, our first graph had a **Cyclomatic Complexity** of 7. ($15 - 10 + 2$), while our second was 1. ($9 - 10 + 2$).

Cyclomatic complexity is all about the number of different routes through the program. The more branches a program has, the greater it’s cyclomatic complexity. Hence, this is a useful metric in Testing: the more branches you have, the more tests you’ll need to exercise them all.

2.5 More Abstraction

Although we ended up with our second graph having a **Cyclomatic Complexity** of 1 (the minimum), we can go further through abstraction, because this representation isn’t minimal from a **Kolmogorov Complexity** point-of-view. For example, we might observe that there are further similarities in the graph that we can “draw out”:

⁷<https://en.wikipedia.org/wiki/Mitochondrion>

⁸https://en.wikipedia.org/wiki/List_of_distinct_cell_types_in_the_adult_human_body

⁹https://en.wikipedia.org/wiki/Locality_of_reference

¹⁰https://en.wikipedia.org/wiki/Cyclomatic_complexity

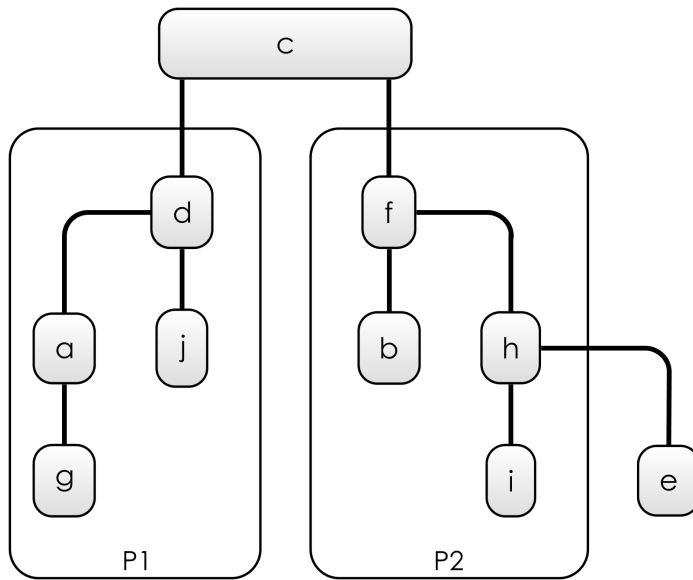


Figure 2.4: Complexity 3

Here, we’ve spotted that the structure of subgraphs **P1** and **P2** are the same: we can have the same functions there to assemble those. Noticing and exploiting patterns of repetition is one of the fundamental tools we have in the fight against Complexity Risk.

So, we’ve looked at some measures of software structure complexity, in order that we can say “this is more complex than this”. However, we’ve not really said why complexity entails Risk.

2.6 Complexity As Mass

The first way to look at complexity is as **Mass** or **Inertia** : a software project with more complexity has greater **Inertia** or **Mass** than one with less complexity.

Newton’s Second Law states:

“ $F = ma$, (Force = Mass x Acceleration)”

—Newton’s Laws Of Motion, *Wikipedia*¹¹

That is, in order to move your project *somewhere new*, and make it do new things, you need to give it a push, and the more **Mass** it has, the more **Force** you’ll need to move (accelerate) it.

Inertia and **Mass** are equivalent concepts in physics:

“mass is the quantitative or numerical measure of a body’s inertia, that is of its resistance to being accelerated”.

—Inertia, *Wikipedia*¹²

You could stop here and say that the more lines of code a project contains, the higher it’s mass. And, that makes sense, because in order to get it to do something new, you’re likely to need to change more lines.

But there is actually some underlying sense in which *this is real*, as discussed in this Veritasium¹³ video. To paraphrase:

“Most of your mass you owe due to $E=mc^2$, you owe to the fact that your mass is packed with energy, because of the **interactions** between these quarks and gluon fluctuations in the gluon field. . . what we think of as ordinarily empty space. . . that turns out to be the thing that gives us most of our mass.”

—Your Mass is NOT From the Higgs Boson, *Veritasium*¹⁴

I’m not an expert in physics, *at all*, and so there is every chance that I am pushing this analogy too hard. But, substituting quarks and gluons for pieces of software we can (in a very handwaving-y way) say that more complex software has more **interactions** going on, and therefore has more mass than simple software.

The reason I am labouring this analogy is to try and make the point that Complexity Risk is really fundamental:

- Feature Risk: like **money**.

¹¹https://en.wikipedia.org/wiki/Newtons_laws_of_motion

¹²https://en.wikipedia.org/wiki/Inertia#Mass_and_inertia

¹³<https://www.youtube.com/user/1veritasium>

¹⁴https://www.youtube.com/watch?annotation_id=annotation_3771848421&feature=iv&src_vid=Xo232kyTs00&v=Ztc6QPNUqls

- Schedule Risk: like **time**.
- Complexity Risk: like **mass**.

At a basic level, Complexity Risk: more complexity means you need more force to get things done, which takes longer.

2.7 Technical Debt

The most common way we talk about unnecessary complexity in software is as Technical Debt:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a standstill under the debt load of an unconsolidated implementation, object-oriented or otherwise.”

Ward Cunningham, 1992¹⁵

Building a perfect first-time solution is a waste, because perfection takes a long time. You’re taking on more attendant Schedule Risk more slowly than you could.

A quick-and-dirty, over-complex implementation mitigates the same Feature Risk.

But, having mitigated the Feature Risk the software to reduce this risk again.

2.8 Kitchen Analogy

It’s often hard to make the case for minimizing Technical Debt

One helpful analogy I have found is to imagine your code-base is a kitchen. After preparing a meal (i.e. delivering the first implementation), *you need to tidy up the kitchen*. This is just something everyone does as a matter of *basic sanitation*.

Now of course, you could carry on with the messy kitchen. When tomorrow comes and you need to make another meal, you find yourself needing to

¹⁵https://en.wikipedia.org/wiki/Technical_debt

wash up saucepans as you go, or working around the mess by using different surfaces to chop on.

It's not long before someone comes down with food poisoning.

We wouldn't tolerate this behaviour in a restaurant kitchen, so why put up with it in a software project?

2.9 Feature Creep

In Brooks' essay "No Silver Bullet - Essence and Accident in Software Engineering", a distinction is made between:

- **Essence:** *the difficulties inherent in the nature of the software.*
- **Accident:** *those difficulties that attend its production but are not inherent.*
- Fred Brooks, *No Silver Bullet*¹⁶

The problem with this definition is that we are accepting features of our software as *essential*.

The **Risk-First** approach is that if you want to mitigate some Feature Risk as a result. But, that's a *choice you get to make*.

Therefore, Feature Creep¹⁷ (or Gold Plating¹⁸) is a failure to observe this basic equation: instead of considering this trade off, you're building every feature possible. This has an impact on Complexity Risk¹⁹, which in turn impacts Communication Risk²⁰ and also Schedule Risk²¹.

Sometimes, feature-creep happens because either managers feel they need to keep their staff busy, or the staff decide on their own that they need to keep themselves busy. But now, we can see that basically this boils down to bad risk management.

"Perfection is Achieved Not When There Is Nothing More to Add, But When There Is Nothing Left to Take Away"

Antoine de Saint-Exupery²²

¹⁶https://en.wikipedia.org/wiki/No_Silver_Bullet

¹⁷https://en.wikipedia.org/wiki/Feature_creep

¹⁸[https://en.wikipedia.org/wiki/Gold_plating_\(software_engineering\)](https://en.wikipedia.org/wiki/Gold_plating_(software_engineering))

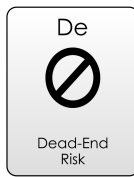
¹⁹Complexity-Risk

²⁰Communication-Risk

²¹Schedule-Risk

²²

2.10 Dead-End 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 2.5: Dead-End Risk

Dead-End Risk is where you build functionality that you *think* is useful, only to find out later that actually, it was a dead-end, and is superceded by something else.

For example, let's say that the Accounting sub-system needed password protection (so you built this). Then the team realised that you needed a way to *change the password* (so you built that). Then, that you needed to have more than one user of the Accounting system so they would all need passwords (ok, fine).

Finally, the team realises that actually logging-in would be something that all the sub-systems would need, and that it had already been implemented more thoroughly by the Approvals sub-system.

At this point, you realise you're in a **Dead End**:

- **Option 1:** You carry on making minor incremental improvements to the accounting password system (carrying the extra Complexity Risk.
- **Option 2:** You rip out the accounting password system, and merge in the Approvals system, surfacing new, hidden Complexity Risk in the process, due to the difficulty in migrating users from the old to new way of working.
- **Option 3:** You start again, trying to take into account both sets of requirements at the same time, again, possibly surfacing new hidden Complexity Risk due to the combined approach.

Sometimes, the path from your starting point to your goal on the Risk Landscape will take you to dead ends: places where the only way towards your destination is to lose something, and do it again another way.

This is because you surface new Hidden Risk in the solutions you choose. This happens a lot.

Source Control

Version Control Systems²³ like Git²⁴ are a useful mitigation of Dead-End Risk²⁵, because it means you can *go back* to the point where you made the bad decision and go a different way. Additionally, they provide you with backups against the often inadvertent Dead-End Risk²⁶ of someone wiping the hard-disk.

The Re-Write

Option 3, Rewriting code or a whole project can seem like a way to mitigate Complexity Risk, but it usually doesn't work out too well. As Joel Spolsky says:

There's a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is the interesting observation: they are probably wrong. The reason that they think the old code is a mess is because of a cardinal, fundamental law of programming: *It's harder to read code than to write it.* - Things You Should Never Do, Part 1, Joel Spolsky²⁷

The problem that Joel is outlining here is that the developer mistakes hard-to-understand code for unnecessary Complexity Risk.

2.11 Where Complexity Hides

Complexity isn't spread evenly within a software project. Some problems, some areas, have more than their fair share of issues. We're going to cover a few of these now, but be warned, this is not a complete list by any means:

- Memory Management
- Protocols / Types
- Algorithmic (Space and Time) Complexity
- Concurrency / Mutability
- Networks / Security

²³https://en.wikipedia.org/wiki/Version_control

²⁴<https://en.wikipedia.org/wiki/Git>

²⁵[Complexity-Risk#dead-end-risk](#)

²⁶[Complexity-Risk#dead-end-risk](#)

²⁷<https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>

Memory Management

Memory Management is another place where Complexity Risk hides:

“Memory leaks are a common error in programming, especially when using languages that have no built in automatic garbage collection, such as C and C++.”

—Memory Leak, *Wikipedia*²⁸

Garbage Collectors²⁹ (as found in Javascript or Java) offer you the deal that they will mitigate the Complexity Risk³⁰ of you having to manage your own memory, but in return perhaps give you fewer guarantees about the *performance* of your software. Again, there are times when you can’t accommodate this Operational Risk³¹, but these are rare and usually only affect a small portion of an entire software-system.

Protocols And Types

Whenever two components of a software system need to interact, they have to establish a protocol for doing so. There are lots of different ways this can work, but the simplest example I can think of is where some component **a** calls some function **b**. e.g:

```
function b(a, b, c) {  
    return "whatever" // do something here.  
}  
  
function a() {  
    var bOut = b("one", "two", "three");  
    return "something "+bOut;  
}
```

If component **b** then changes in some backwards-incompatible way, say:

```
function b(a, b, c, d /* new parameter */) {  
    return "whatever" // do something here.  
}
```

²⁸https://en.wikipedia.org/wiki/Memory_leak

²⁹[https://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

³⁰Complexity-Risk

³¹Operational-Risk#performance-risk

Then, we can say that the protocol has changed. This problem is so common, so endemic to computing that we've had compilers that check function arguments since the 1960's³². The point being is that it's totally possible for the compiler to warn you about when a protocol within the program has changed.

The same is basically true of Data Types³³: whenever we change the **Data Type**, we need to correct the usages of that type. Note above, I've given the javascript example, but I'm going to switch to typescript now:

```
interface BInput {
  a: string,
  b: string,
  c: string,
  d: string
}

function b(in: BInput): string {
  return "whatever" // do something here.
}
```

Now, of course, there is a tradeoff: we *mitigate* Complexity Risk.

Nevertheless, compilers and type-checking are so prevalent in software that clearly, you have to accept that in most cases, the trade-off has been worth it: Even languages like Clojure³⁴ have been retro-fitted with type checkers³⁵.

We're going to head into much more detail on this in the chapter on Protocol Risk.

Space and Time Complexity

So far, we've looked at a couple of definitions of complexity in terms of the codebase itself. However, in Computer Science there is a whole branch of complexity theory devoted to how the software *runs*, namely Big O Complexity³⁶.

Once running, an algorithm or data structure will consume space or run-time dependent on it's characteristics. As with Garbage Collectors³⁷, these

³²<https://en.wikipedia.org/wiki/Compiler>

³³https://en.wikipedia.org/wiki/Data_type

³⁴<https://clojure.org>

³⁵<https://github.com/clojure/core.typed/wiki/User-Guide>

³⁶https://en.wikipedia.org/wiki/Big_O_notation

³⁷[https://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

characteristics can introduce Performance Risk³⁸ which can easily catch out the unwary. By and large, using off-the-shelf data structures and algorithms helps, but you still need to know their performance characteristics.

The Big O Cheatsheet³⁹ is a wonderful resource to investigate this further.

Concurrency / Mutability

Although modern languages include plenty of concurrency primitives, (such as the `java.util.concurrent`⁴⁰ libraries), concurrency is *still* hard to get right.

Race conditions⁴¹ and Deadlocks⁴² *thrive* in over-complicated concurrency designs: complexity issues are magnified by concurrency concerns, and are also hard to test and debug.

Recently, languages such as Clojure⁴³ have introduced persistent collections⁴⁴ to alleviate concurrency issues. The basic premise is that any time you want to *change* the contents of a collection, you get given back a *new collection*. So, any collection instance is immutable once created. The tradeoff is again attendant Performance Risk⁴⁵ to mitigate Complexity Risk⁴⁶.

An important lesson here is that choice of language can reduce complexity: and we'll come back to this in Software Dependency Risk.

Networking / Security

The last area I want to touch on here is networking. There are plenty of Complexity Risk.

In the case of security considerations, exploits *thrive* on the complexity of your code, and the weaknesses that occur because of it. In particular, Schneier's Law says, never implement your own crypto scheme:

“Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's

³⁸Operational-Risk#performance-risk

³⁹<http://bigochaetsheet.com>

⁴⁰<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html>

⁴¹https://en.wikipedia.org/wiki/Race_condition

⁴²<https://en.wikipedia.org/wiki/Deadlock>

⁴³<https://clojure.org>

⁴⁴https://en.wikipedia.org/wiki/Persistent_data_structure

⁴⁵Operational-Risk#performance-Risk

⁴⁶Complexity-Risk

not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis.”

Bruce Schneier, 1998⁴⁷

Luckily, most good languages include crypto libraries that you can include to mitigate these Complexity Risks from your own code-base.

This is a strong argument for the use of libraries. But, when should you use a library and when should you implement yourself? This is again covered in the chapter on Software Dependency Risk.

tbd - next chapter.

costs associated with complexity risk

CHANGE is also more risky why?

⁴⁷https://en.wikipedia.org/wiki/Bruce_Schneier#Cryptography

Part III

Application

Coming Next

- preview of what's to come in part 3.

Bad to leave on the failure notes, let's talk about some successes.

3. What's To Come

- risk based debugging.
- risk based coding.

Estimates

1. Purpose of Estimating
2. (coordination risk)
3. trying to work out *the value* of doing some work
4. How Estimates Fail
5. Hidden Risk
6. Visible Risk
- 7.

4.1 How Estimates Fail

The problem with a developer answering a question such as:

How long will it take to deliver X

Is the following:

- The developer likely doesn't know what X is, and any description of it is inadequate anyway (Invisibility Risk).
- The developer has a less-than-complete understanding of the environment he will be delivering X in (Complexity Risk).
- The developer has some vague ideas about how to do X, but he'll need to try out various approaches until he finds exactly the right one (Boundary Risk).

- The developer has no idea what Hidden Risk will surface when he starts work on it.
- What will happen if he takes too long and misses the date by a day / week / - month / year. (Schedule Risk

... and so on. So, his estimate is both wrong, and contingent on what exact risks appear.

4.2 An Aside

tbd quality, time, functionality pick two of three.

4.3 Agile

One alternative approach, must espoused in DevOps / Agile is to pick a short-enough period of time (say, two days or two weeks), and figure out what the most meaningful step towards achieving an objective would be in that time. By fixing the time period, we remove Schedule Risk from the equation.

The problems with this are: - First, this is just Gradient Descent. - Second, what happens after the two days? If you *abandon the work*, then maybe you are giving up too early: a day or two more might have cracked it. If you undergo the planning process again, you might decide that another objective is more important. Constantly switching ideas all the time *might not get you anywhere*. - Third, how to choose the time period? In Scrum it's between tbd. But, lots of people are saying go smaller.

The choice of using gradient descent means that you have given up on [Grand Objectives].

But essentially, we have here the difference between "Walking towards a destination" and "Walking downhill".

Or, a planned economy and a market economy. But, we don't live in *either*: everyone lives in some mixture of the two.

So, is there a synthesis of these two approaches that makes sense?

4.4 Risk-First Estimating

One approach

Devops approach.