PEELING THE ONION

Hi,

My name is Rob Moffat. I am a Java and Javascript software developer working in London. I am building Deutsche Bank's Symphony Practice.

If you don't know, Symphony is a messaging platform that a lot of the banks use for secure, encrypted chat. The chances are that some of you are probably familiar with that in your jobs.

That's my day job. But I'm not actually here to talk about that, although we might come back to it later in the examples. Instead I want to talk to you about risk in software development.

"THE BASIC PROBLEM OF SOFTWARE DEVELOPMENT IS RISK"

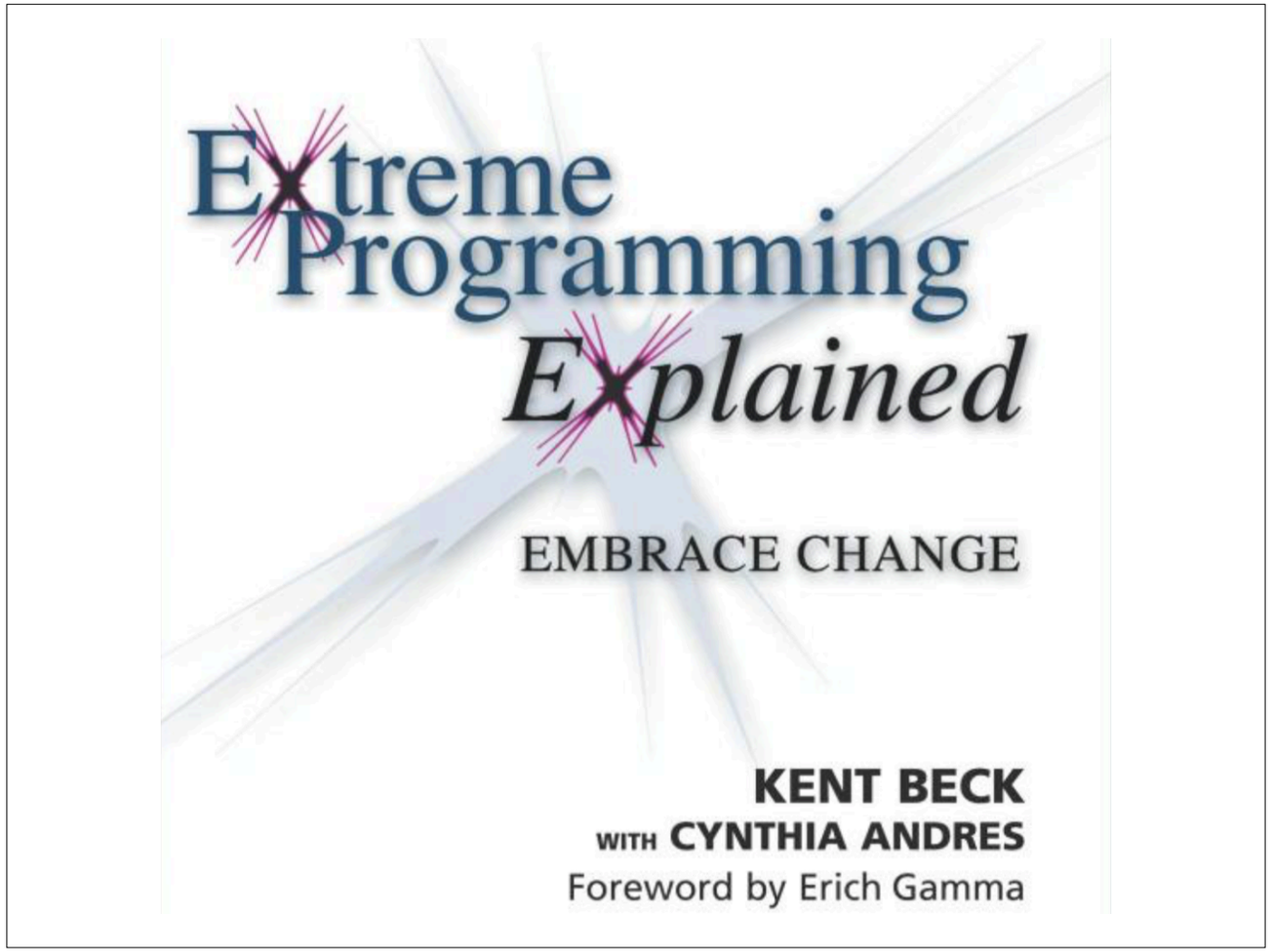
I know a lot of you work in banking, and are probably already about to leave.

I'm not going to mention anything to do with banking risk _at all_. We're going to be looking today at things like debugging, coding, scoping work, pleasing our stakeholders and keeping our jobs.

Proper developer stuff.

The quote above is actually the first sentence from Chapter One of a fairly famous book on Software Development.

Does anyone know which one?



Ok, who's heard of this book? Who's heard of Kent Beck?

Anyone read it?

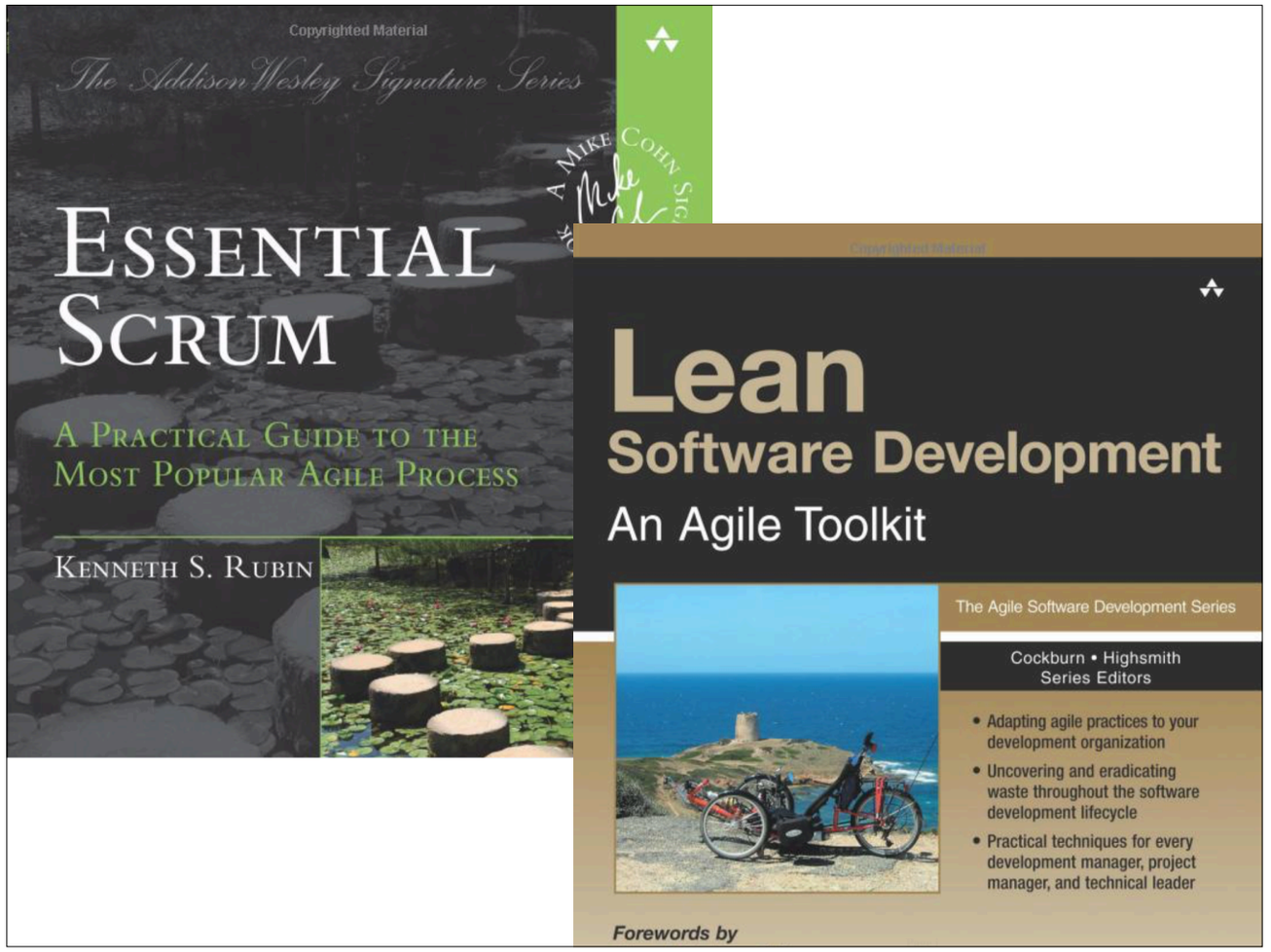
Does anyone actually practice XP in their jobs?

This is a picture of some guys Pair Programming.

A lot of developers grew to hate XP because of this.

Some people love it.

It was my first introduction to Agile techniques.



These approaches are more popular nowadays.

Who uses Scrum at work? Anyone use Kanban?

Does anyone still use Waterfall?

I’m not here to rag on Waterfall.

Sometimes, Waterfall is the right way to go!

We'll dive into that in more detail in a minute, but it feels like collectively, as an industry, or a practice, software development has been on a path of discovery, going something like this:

> **"WE ARE UNCOVERING BETTER WAYS OF DEVELOPING SOFTWARE BY DOING IT AND HELPING OTHERS DO IT."**
>
> **- Agile Manifesto**

Waterfall leads to XP and Scrum and Agile and Kanban, lately DevOps and so on.

Exactly as described in the Agile Manifesto.

Let's get back to Kent Beck.

"THE BASIC PROBLEM OF SOFTWARE DEVELOPMENT IS RISK"

It seems weird that Kent started his book with this statement, but what people really remember about XP is just Pair Programming, Unit Tests and eating pizza.

I think he was actually on to something really powerful right then and he didn't develop it further.

The reason is, he had these other fish to fry: he'd developed XP which, to him and many other people, was demonstrably better than the way we had been doing software development.

And so his book was just a catalog of all these "better" techniques, and how they could be used together.

Some better than others: He didn't *invent* unit testing as a practice, but he did a lot of promote it and was one of the original authors of JUnit.

I actually can't imagine coding now without building tests as I go, and having tools to at least understand my coverage.. It just seems so helpful now to have this.

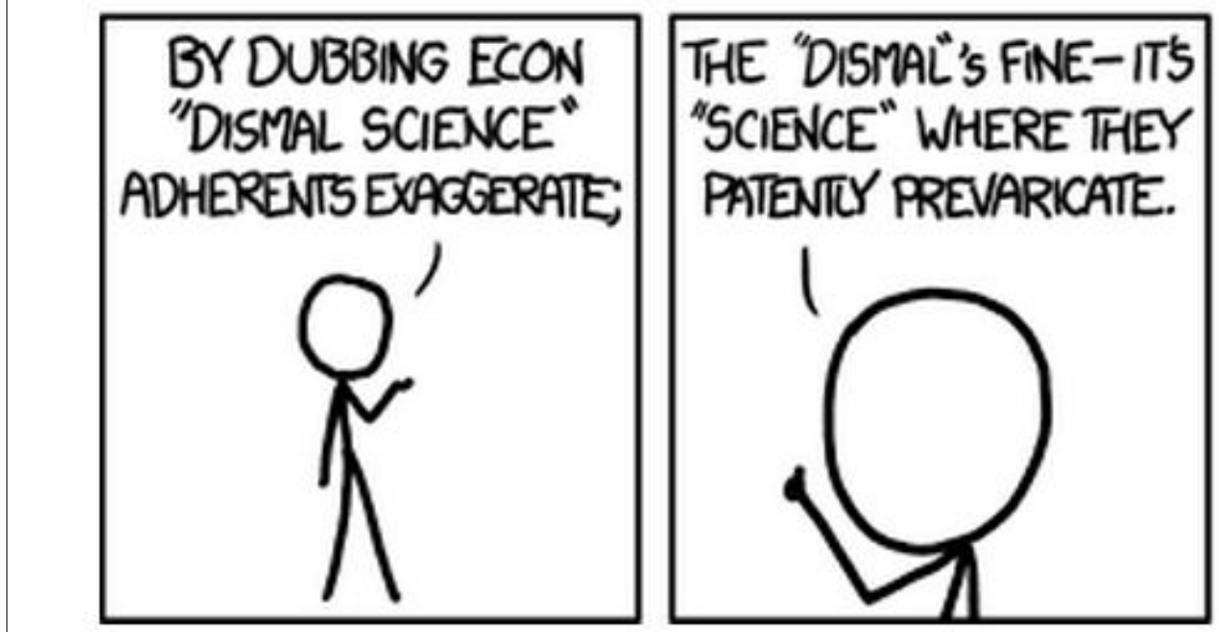
"THE BASIC PROBLEM OF SOFTWARE DEVELOPMENT IS RISK"

When I was at school, I took Economics. We were 17 or 18 years old. And one of the guys just asks:

"What's the point of all this? Surely the markets are just people buying and selling stuff? They don't know about demand curves and price elasticity and all of these things!"

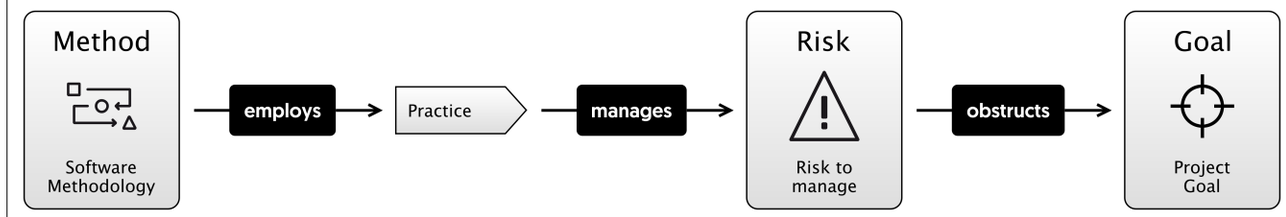
My economics teacher had to take quite a lot of shit actually. We were young and political and pretty annoying teenagers. But he gave a great answer:

"When footballers are playing a match of football, they have an understanding of how the ball works. But they don't need to be good physicists. Nevertheless, physical rules underpin the game they play."



Now, that might be overselling Economics, "the dismal science", as a discipline, comparing it to Physics.

But in this talk I'm going to try and persuade you that something similar is true in software development.



That is, methodologies like Scrum, XP, Waterfall are collections of *practices* meant to manage risk.

Risk is the underlying "physics" that we're dealing with.

So in the rest of this talk, I am going to try and make that case in various ways, and then we can try to apply that principle in a number of different scenarios and see where it gets us.

You could argue, like a footballer might, that understanding that lower level isn't going to make a jot of difference to how you do the higher level stuff. How do you run a good retro? How do you do estimating? How do you code a feature?

Actually though, I think not only that it *does*, but that we're already doing this stuff anyway, we just don't talk about it.

"THE BASIC PROBLEM OF SOFTWARE DEVELOPMENT IS RISK"

So, in order to convince you of this statement, I'm first going to try and convince you of some more specific statements.

Let’s start with..

So instead of looking at Software Development for a moment, I’m just going to talk about the board game, Risk.

My son Charlie and I love to play risk.

This is actually the board from the last time we played. He is Yellow and I am Grey.

If you understand the game risk, you'll know that basically, it's a regressive game.

The person winning has the most land, and they get the most troops.

The only nod at progressive game play (that is, trying to even everyone out) is that attacking is harder than defending.

In this game, I won on the next turn, because I could see Charlie's position was over-extended, and I had a bunch of cards and you get lots of troops for continents - I've got 3. North and South America, and Australasia.

Now, Charlie is 12. He plays a good game of chess, but I can beat him at most other things if I want to. Now, we can get into whether or not, as a parent, you should do that another time.

I like playing games with my kids and I feel it's good for them, so I tend to let them win about half the time.

Apart from Chess, where I have to fight for my life.

Why did I know I was going to win? Basically, I have an *Internal Model* of Risk that's better than his. I could see the danger areas. For example, I knew that by holding Thailand down here, I would prevent him from owning Asia, which gives him 7 extra troops each go. And by defending Kamchatka up the top I can stop him marching

INTERNAL MODEL

So, to win at risk, I was managing a balance of risks. And those risks are the ones I know about, in my "Internal Model".

It's possible that there are things that I'm not seeing, that a better Risk player would. The reason I beat Charlie is I am 44 and my Internal Model is a bit better developed than his.

And, I've played more Risk.

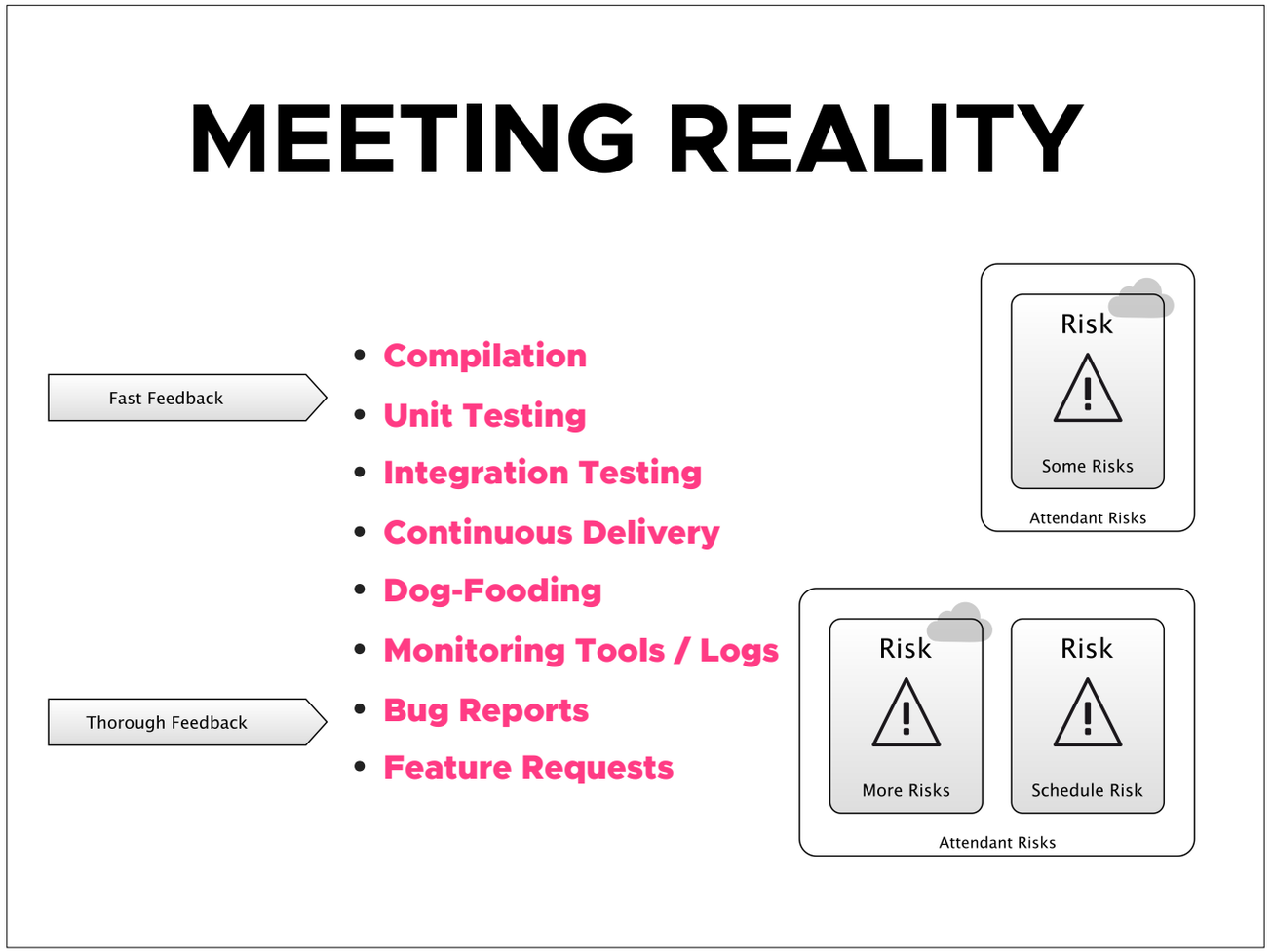
Roulette is another game that's all about risk. Generally, most people have the same internal model of this game: If I put a chip down on number 12, I will get back 36 chips if it comes up.. but there are 37 slots on the wheel.

So, overall, over time, I’m going to lose.

But this guy, Joseph Jagger, made a ton of money in the 1880's out of Roulette because he had a better Internal Model. He sent stooges to casinos to record the results of each spin of the wheel. And he found that some wheels were biased and so certain numbers came up more often than others.

All he had to do was bet on those numbers to win.

And all he had to do to get this "improved" Internal Model, was to go out and experiment in the real world, and record some observations. He had to do what I call "Meeting Reality".



So, we all know feedback loops are important right?

I've listed some here. The "tightest" loops are down at the bottom, the "longest" loops at the top. But these top ones give you "less" than the bottom ones - the Reality you meet isn't the full, gory reality of those ones at the bottom.

Just because your code compiles, doesn't mean people are going to enjoy using it.

When you play the Risk game or learn about the Roulette wheels, you "Meet Reality" - your Internal Model improves. This is the general process whereby experience gives you a better understanding of the risks.

Does that apply to software development?

I would argue it does. Going back to Charlie, we were doing some revision for his Computer Science exam the other day, and he had to understand HTML.

He wrote a webpage that looked like this, to start with.

```
<HTML
 <HEAD>
  <TITLE>My Webpage
 </HEAD>
 <BODY>
  <P>A Paragraph<P/>
  <A href=“http::/google.com>Go To Google</A>
</HTML>
```

Now, for most of us... this just hurts our eyes. We can see all of the problems right off. I have no idea what a browser would do with this. Browsers are very lenient so maybe some of it would work.

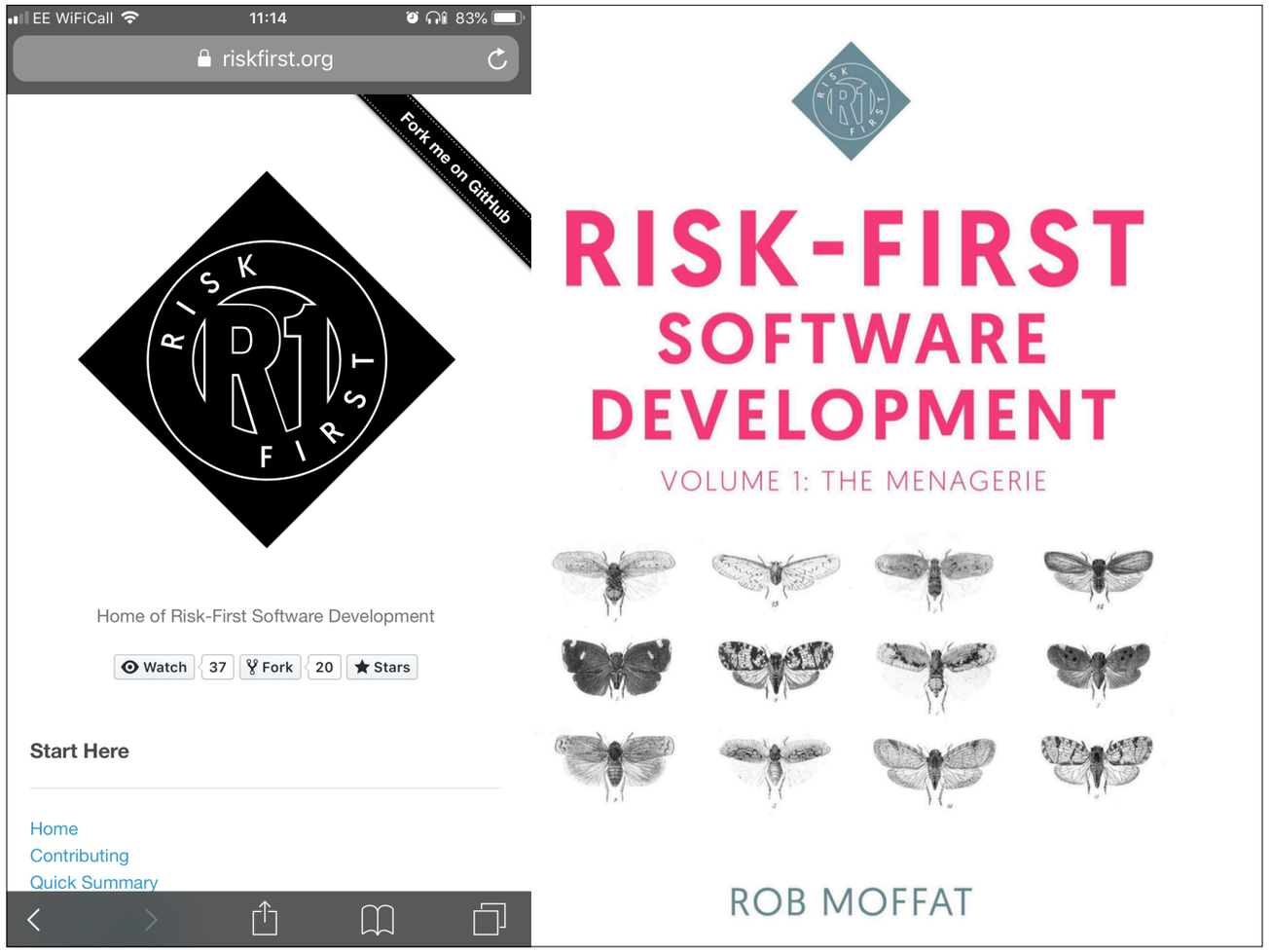
But the reason this sets off alarm bells for us, is that we have really well developed *Internal Models* of HTML and the rest.

Whereas this is Charlie's first webpage.

By loading this up in a browser, he is Meeting Reality. He’s closing a feedback loop, because he’ll see the results of his efforts.

And, his Internal Model will improve.

The next time he writes a webpage, his Internal Model will warn him about the risks of it going wrong, and he’ll make fewer mistakes.



Now people ask me about how I got into all this.

It wasn't actually from re-reading the first sentence of the XP book.

My journey started with a few notes in my journal which went like this:

10 June 2016: Developing To Minimise Risk

"As developers we should be considering what the Risk Landscape will look like after the first release...

“The concept of Minimum Viable Product is basically saying that Lack of Access is a bigger problem than Lack of Functionality ...

"A startup that goes well for the first month, gains huge traction and then has a massive data breach, or storage failure is not going to last. But a Minimum Viable Product might not necessarily need off-site backups and encrypted data. “

So, I was talking here about the concept of "Minimum Viable Product", and the idea is that you want to show something to the world, and do the least amount of work to get there.

But, when you put work in the public domain, and get people using it, you're suddenly opened up to lots of new risks because of that... so maybe what we're really doing is trying to find a spot on the Risk Landscape that is suitable.

It's very easy with Agile to just consider "the next most important thing". And, that's really what I'm alluding to with the above journal entry. You can build your MVP, but you really need to consider what could happen if things go wrong.

What if you're hacked, and lose all your data?

What if the servers go down, and the users can't log in for an afternoon?

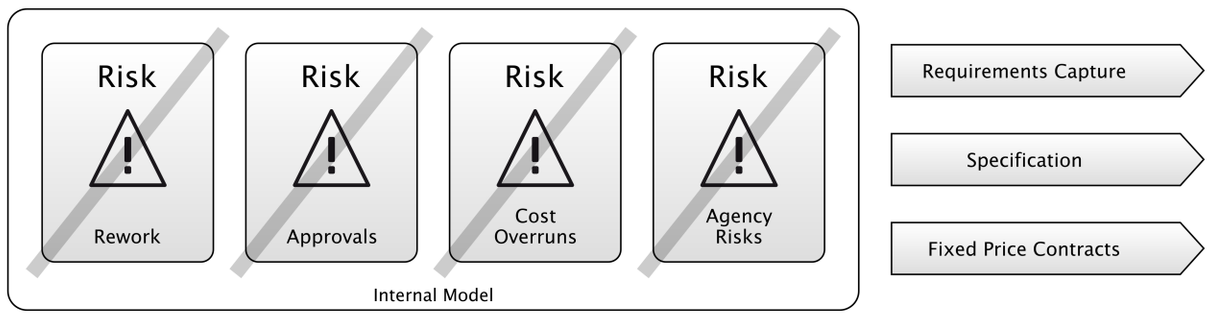
How do these things affect your reputation? You might end up dead in the water if you piss off all those early adopters. Maybe.

It depends.

So the idea of the "Risk Landscape" is, we have an idea of the direction we want to go, and our Internal Models help us figure out where that might be, but we don't exactly know how it'll play out when we get there.

And we want to avoid dragons on the way.

WATERFALL



So let's look again at Waterfall. This used to be a popular approach people used to get software done.

Because Waterfall methodologies are borrowed from the construction industry, they manage the risks that you would care about in a construction project, specifically, minimising the risk of rework, and the risk of costs spiralling during the physical phase of the project.

For example, pouring concrete is significantly easier than digging it out again after it sets.

Construction projects are often done by tender which means that the supplier will bid for the job of completing the project, and deliver it to a fixed price.

This is a risk-management strategy for the client: they are transferring the risk of construction difficulties to the supplier, and avoiding the Agency Risk that the supplier will “pad” the project and take longer to implement it than necessary, charging them more in the process.

In order for this to work, both sides need to have a fairly close understanding of what will be delivered, and this is why a specification is created.

In this diagram, the risks *on the left* are addressed by the actions on the right: requirements capture, specification, fixed price contracts. That’s why they have a bar through them.

Those actions are doing something to address those risks.

WATERFALL

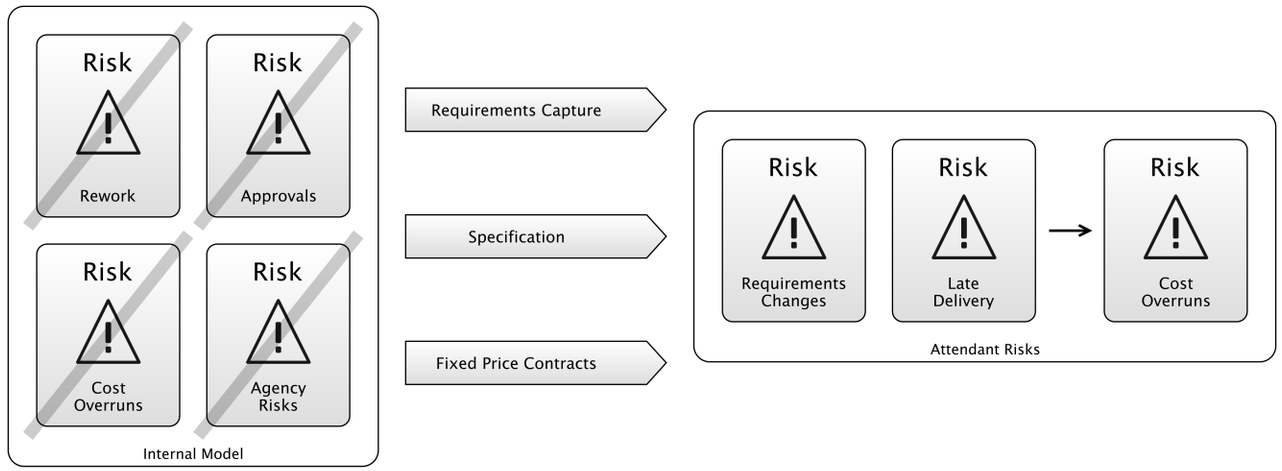
“1. Clients may not know exactly what their requirements are before they see working software and so change their requirements, leading to redesign, redevelopment, and re-testing, and increased costs.”

“2. Designers may not be aware of future difficulties when designing a new software product or feature. “

- Waterfall Model, Wikipedia

But, a lot of the time, waterfall was *addressing the wrong risks*.

WATERFALL



So, here we have the full equation. By taking these actions in the middle, we address the risks on the left.

But… we end up with the risks on the right.

We have the risk that the requirements change, or that the project takes longer than expected.

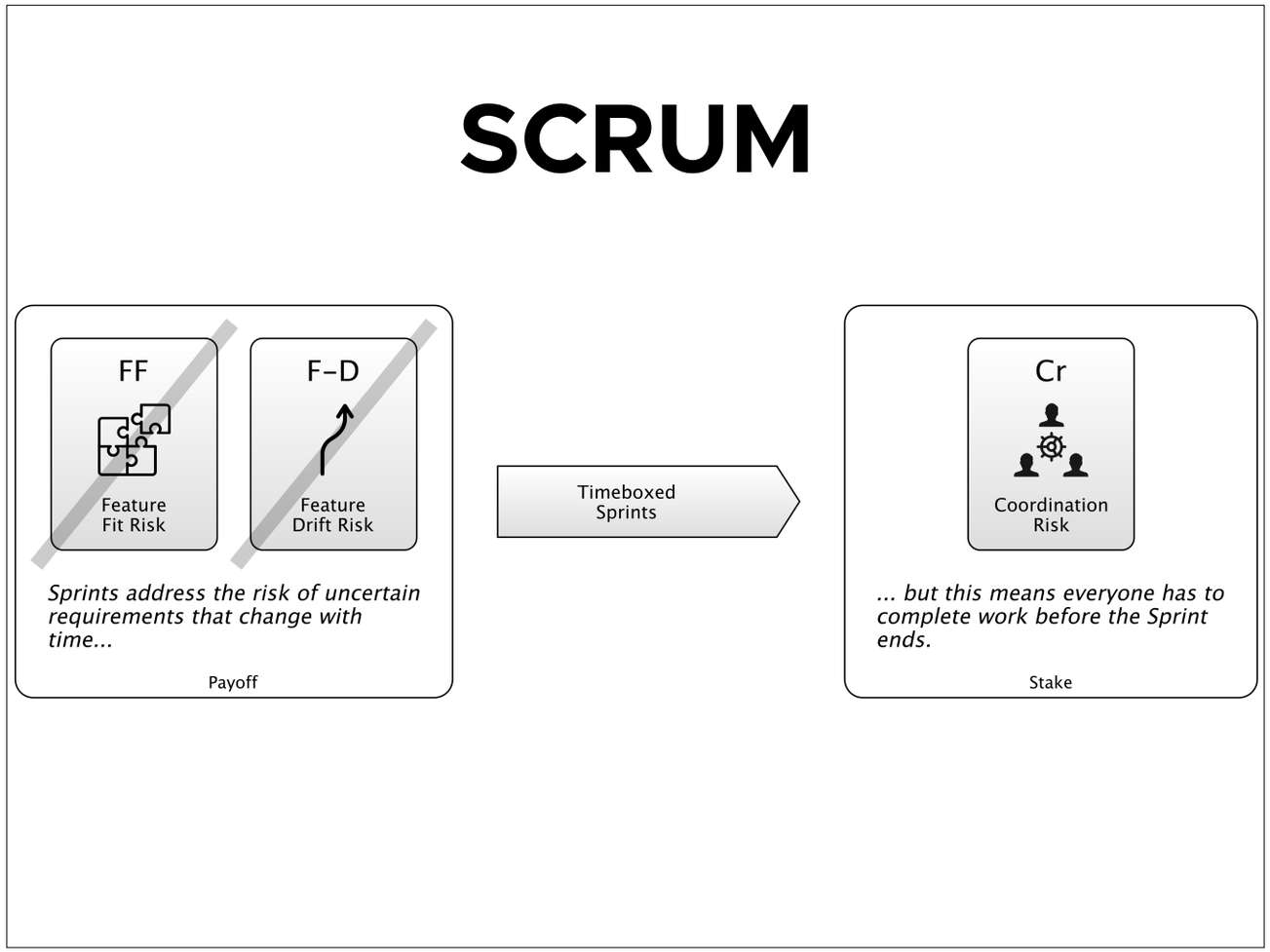
For big government IT projects, done to tender, this usually meant that the government ended up bailing out the supplier, and shouldering the extra expense.

So as you can see, some of the risks on the left are *the same as the ones on the right*: the actions taken to manage them made no difference (or made things worse).

The inability to manage these risks led to the identification of a “Software Crisis”, in the 1970’s.

This is how we end up with Agile.

Scrum is probably the most popular Agile methodology, so let’s have a quick look at how that works for a second.



So one of the key features - maybe *the* key feature - of Scrum is that you work in Sprints.

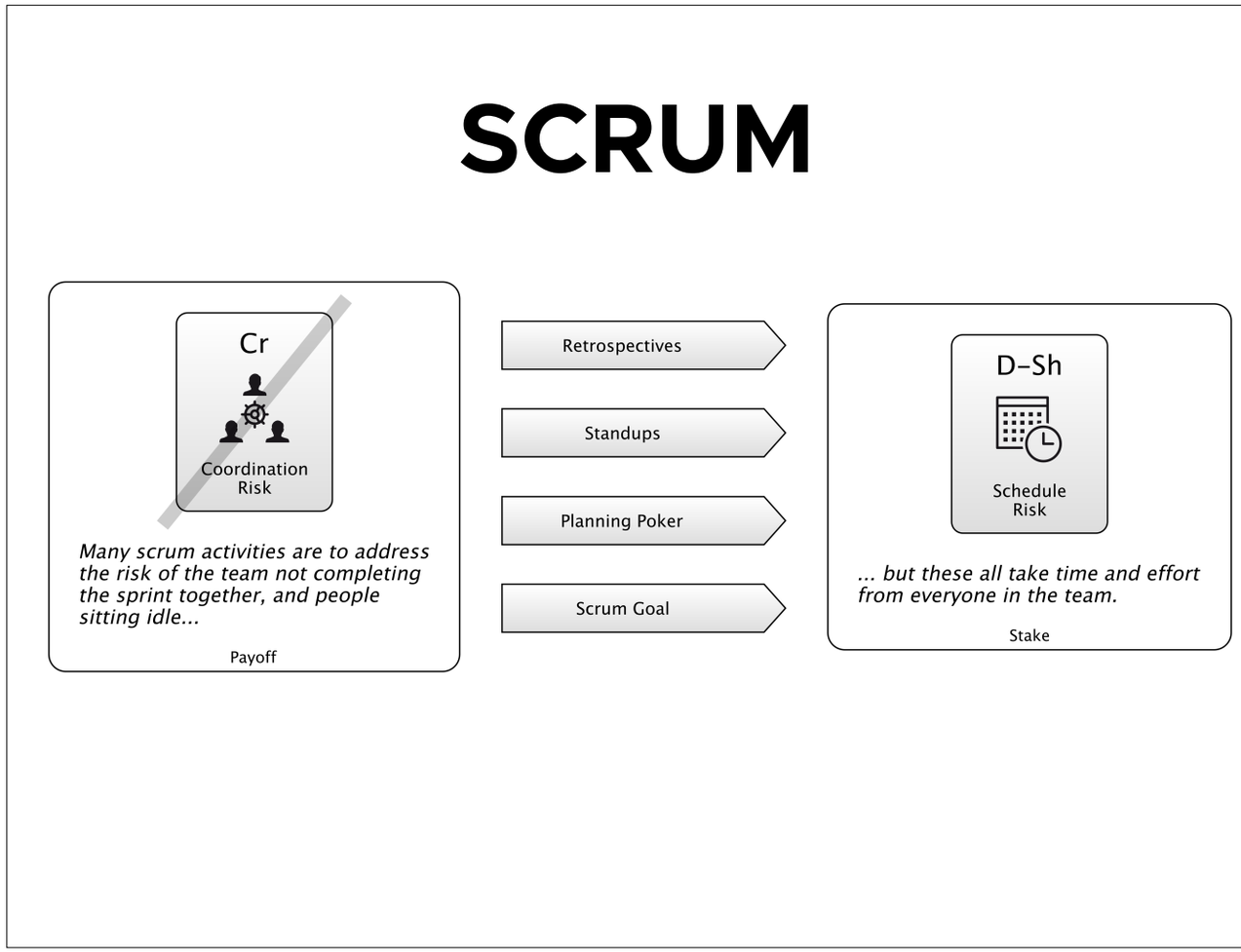
These are like little 2 or 3-week windows of time, where you deliver something to production.

You can see that practice in the middle there. The *reason* Scrum recommends this is to address those risks on the left:

- The Risk that you don't build what the customers want the first time,
- and the risk that what the customers want *changes* while the project is in progress.

But in order to do a Sprint, everyone needs to be given *just enough* work to do to fill that time, otherwise they're going to blow out of the end of the Sprint, or be sitting around with nothing to do.

(Well, there's always something to do, but…)



So, that Coordination Risk is the major problem with doing Sprints.

How do we address that Risk?

Well, Scrum recommends a bunch *more* practices to fix that.

Like Retrospectives, where you try and learn what went wrong in the previous sprint.

Daily standup meetings, where you keep track of where everyone is, and check they're all going to arrive on time.

Planning Poker: this is a kind of estimating game which is done to figure out how much work is going to be put in the Sprint

And a Scrum goal, where everyone commits to achieving some kind of goal with the Sprint, and working towards that.

Having a goal is a great way of Coordinating people so they don't accidentally work antagonistically, with one person doing something that undoes what another is doing.

But obviously, all that stuff takes time and effort, which may be better spent elsewhere.

If you didn't *have* that Coordination Risk problem, you might not have to do some of this other stuff.

DIFFERENT PRACTICES FOR DIFFERENT RISKS

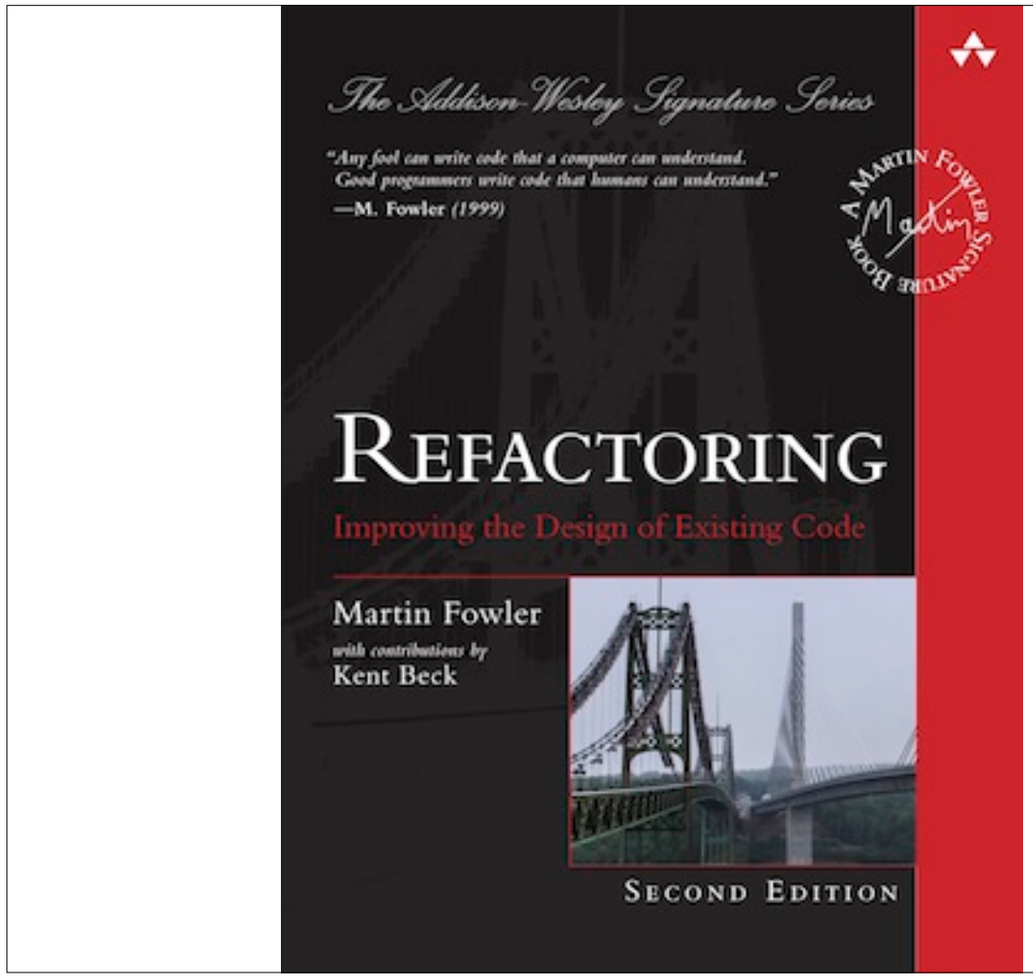
So, both Waterfall and Scrum have different practices, and those practices *address* different risks and also lead to new risks.

When we perform these practices according to a methodology on a project, we expect it to pay-off and lead us somewhere better on that Risk Landscape.

If this doesn't materialise, then we feel the methodology is failing us.

But the practices need to fit the risks.

Let's look at some examples of how this works with *coding.*



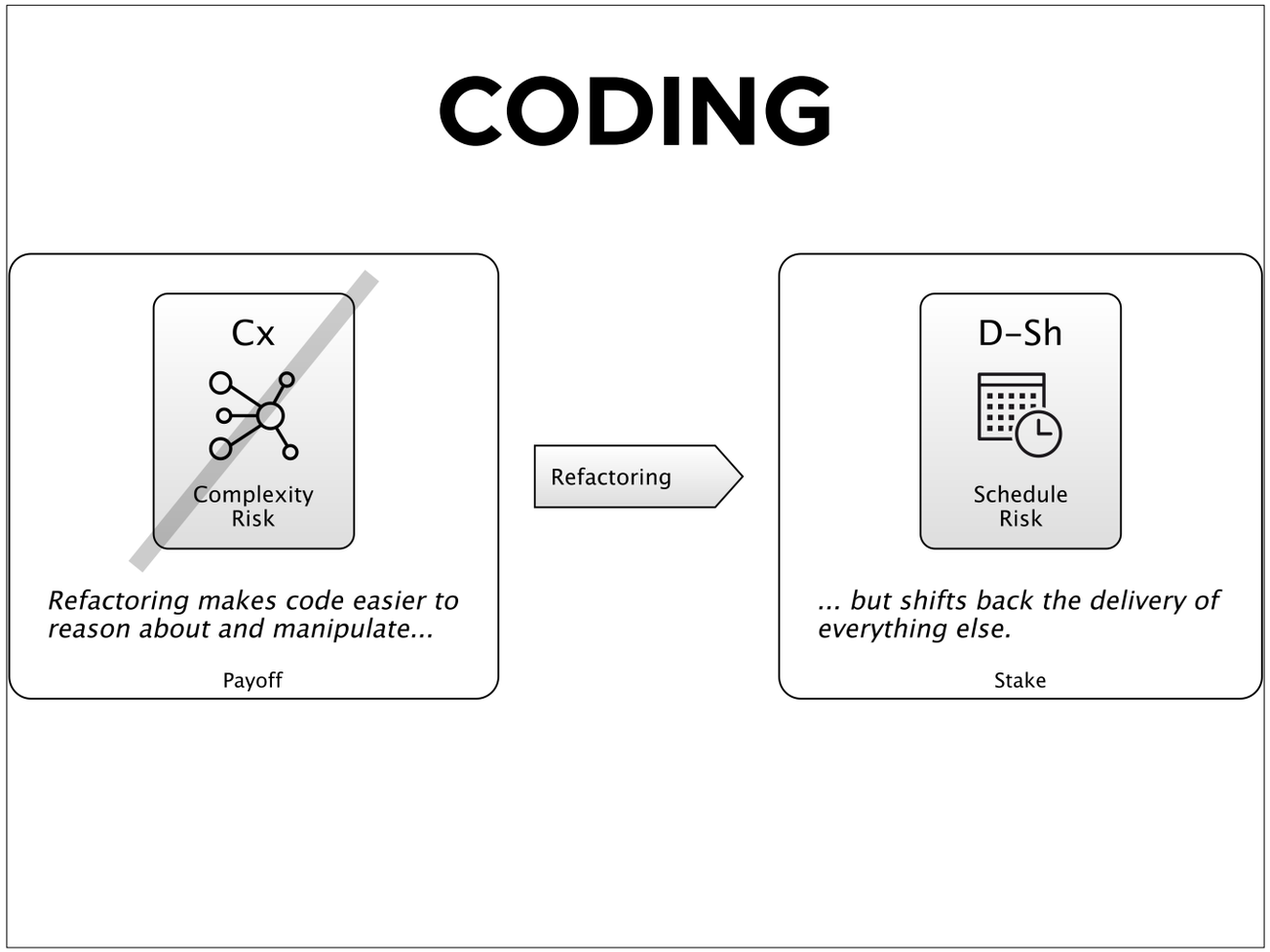
So there are lots of different types of coding.

Like, building a new feature is one type.

Spike Solutions are another type, where you try out a new technology for a particular purpose.

Refactoring is a third type of coding, where you're trying to make your code *simpler.*

This guy, Martin Fowler, did a lot to popularise the idea of Refactoring.



Here's the Risk diagram for Refactoring.

What is the value of it? Refactoring is addressing Complexity Risk in your code-base.

That is, the weight of complexity in the codebase, and its resistance to change and comprehension.

Projects can sometimes drown in their own complexity "big balls of mud". So this is a Risk.

And Technical Debt is an unnecessary excess of this. A lot of the Complexity Risk is risk you've taken on because you want to deliver functionality to people, quickly. But sometimes, it accrues accidentally, or because you take short-cuts.

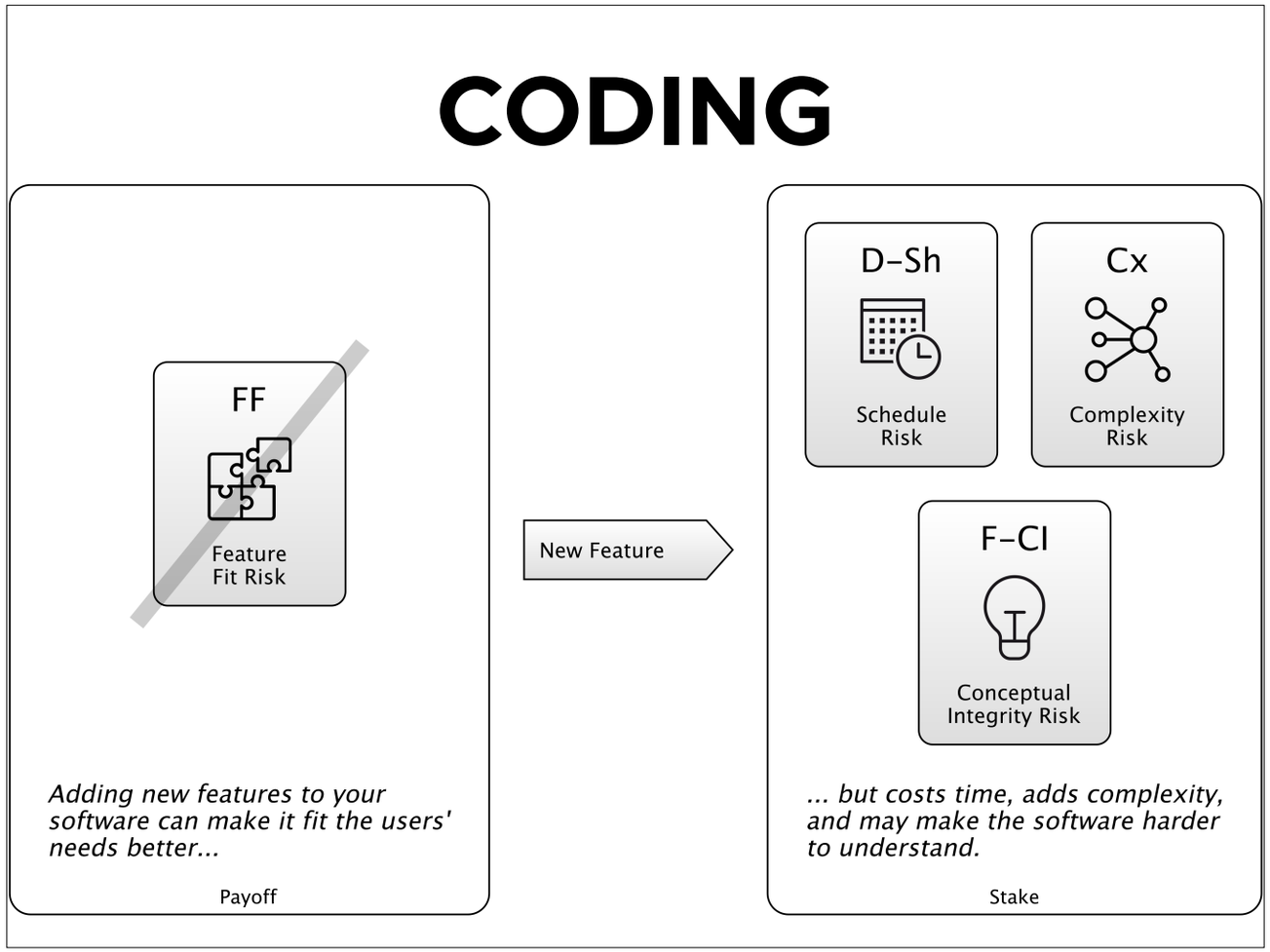
Ideas gets removed, but the code stays complicated. That's Technical Debt.

So, you *refactor.*

But that can take time and sometimes it doesn't work out.

So like a game of Roulette, I've labelled these parts of the diagram as the "Payoff" and the "Stake": it's like a bet.

You're betting that you can move to this new place on the Risk Landscape where there is less Complexity Risk.



Here, we're introducing a new feature.

The payoff of doing that is we address *Feature Fit Risk:* hopefully, our customers find our software a better *fit* for the features -or functionality- they need out of it.

But that can sometimes go wrong too: look at the stakes.

Clearly, adding features takes time out of the schedule.

It also adds to our Complexity Risk: our software is heavier, there's more of it, it's harder to understand, it's harder to move about.

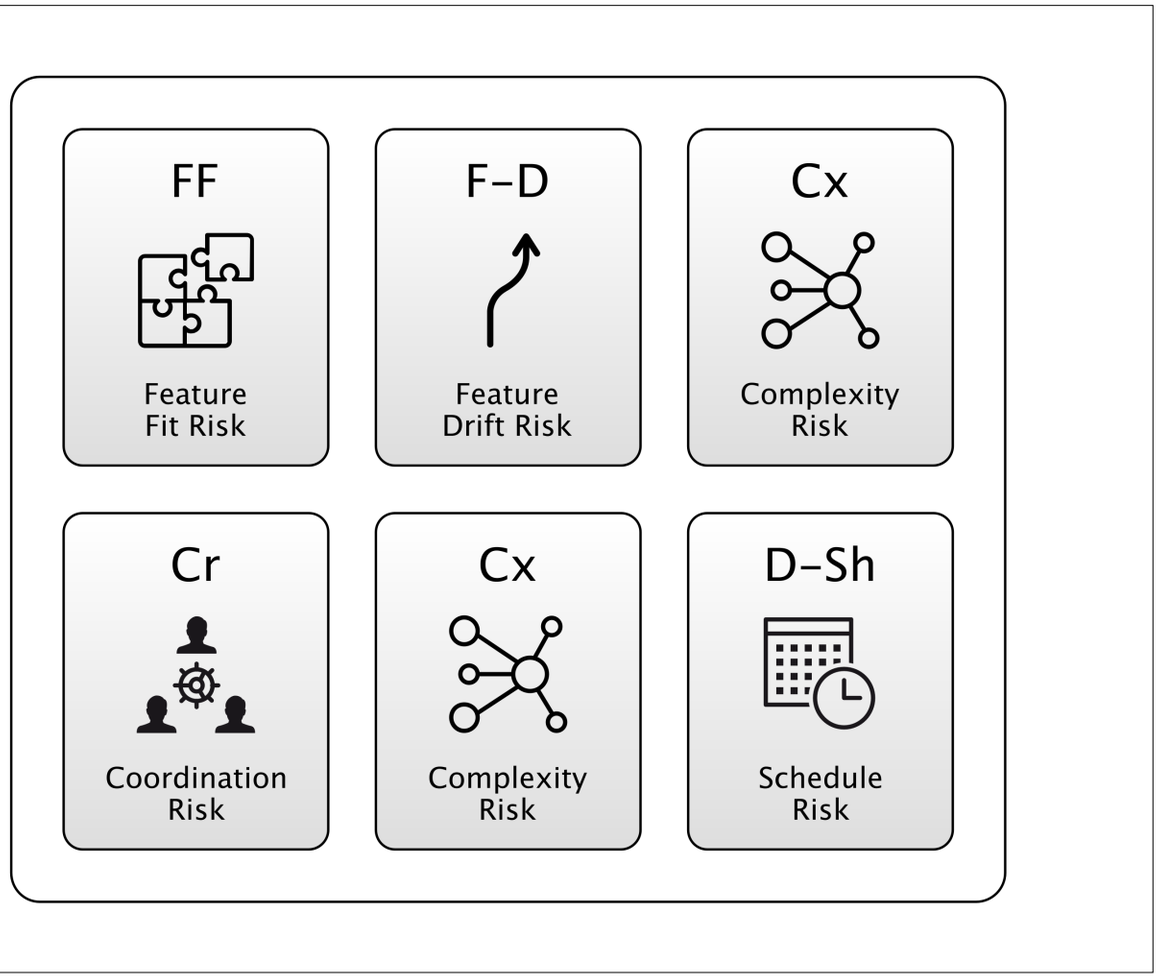
But also, it might be harder for our users to understand: that's Conceptual Integrity Risk.

A good example of that is Feature Phones. Before the iPhone came out, phone manufacturers used to like to cram in every function under the sun.

But they were all lost in complicated menus, and terribly-written and translated manuals.

The iPhone obliterated all of that. It had maybe four features: phone, internet, text message, play music. It had a wonderful, conceptually-simple interface that even a two-year-old could use.

The Feature Phone builders hadn't realised, until the iPhone came out, that their phones were drowning in Conceptual Integrity Risk, which sunk them.



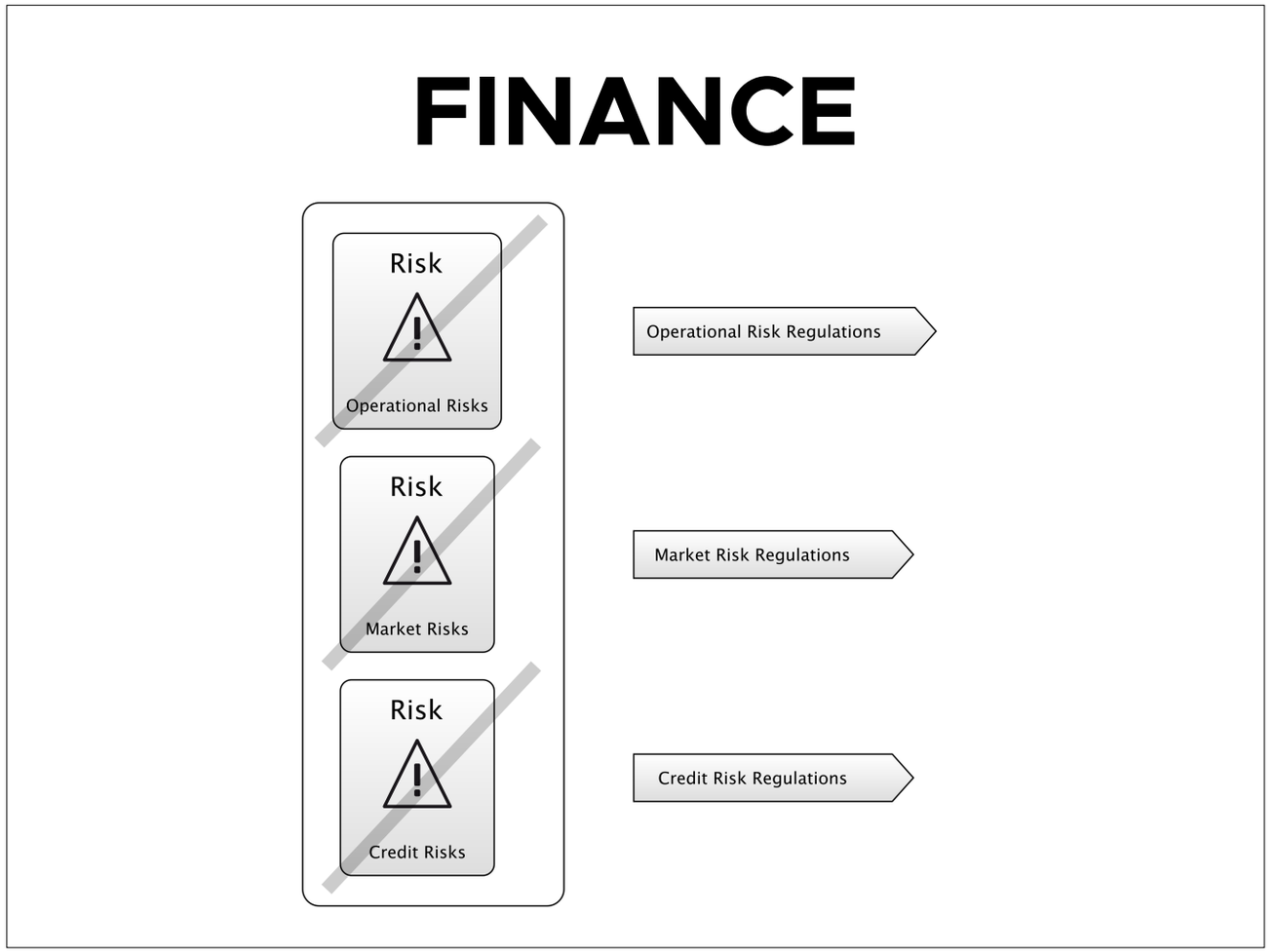
So, we've looked at some of the risks of software development. There are a bunch more of these - dependency risks, agency risks, operational risks and so on.

Is it useful to do this classification of the risks we face?

It feels like this happens a lot in life already.

I can buy Car Insurance, to cover risks related to my car.

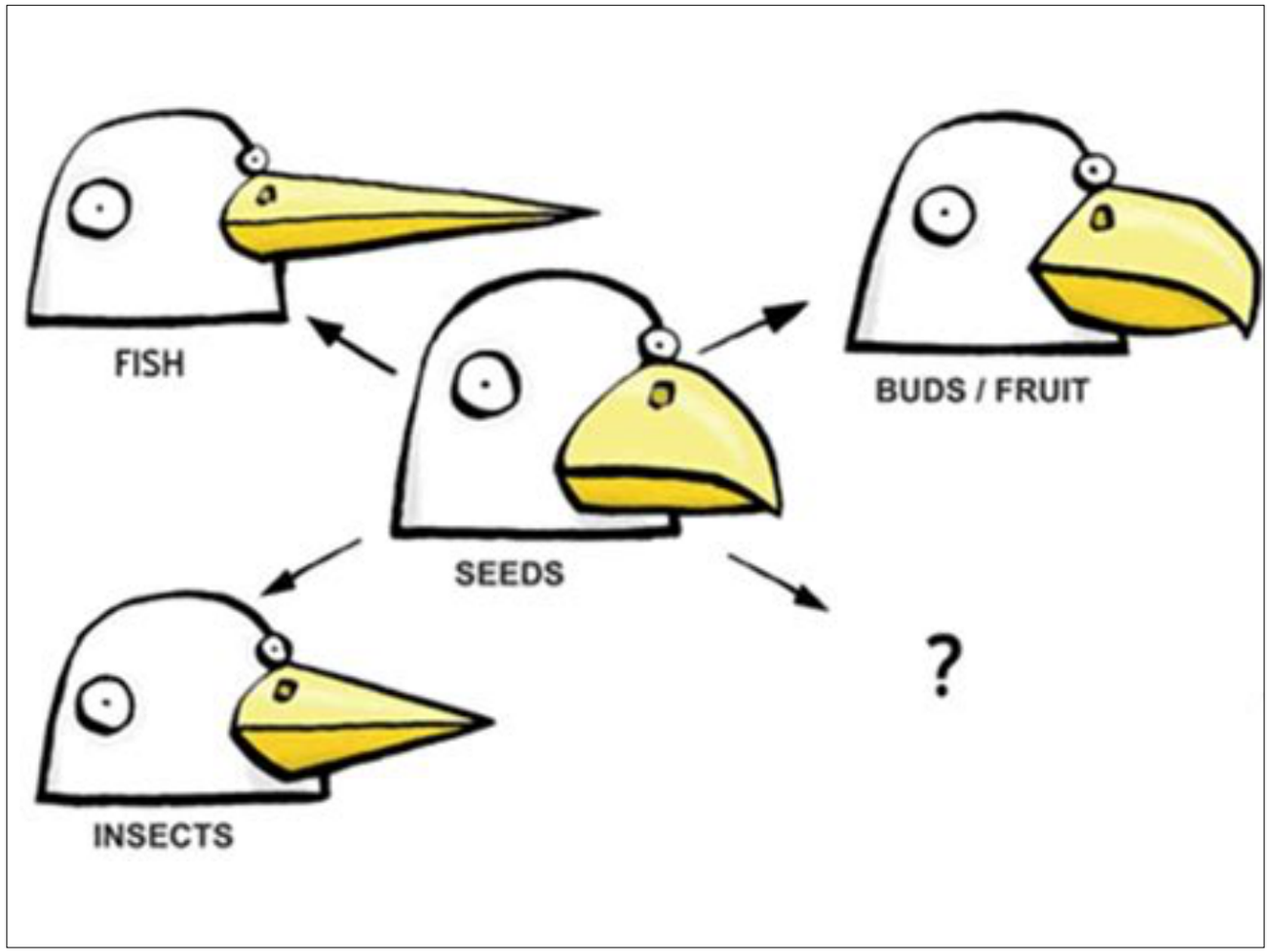
Or Travel Insurance, for risks I face going abroad.



In the finance industry, this has been the general approach for many years.

We have Operational Risks, Market Risks, Credit Risks, Liquidity Risks and so on.

And there are regulations and measures for addressing each of them.



So, this idea has plenty of history, we see it in our lives generally with insurance,
it affects board games, software methodologies, coding and banking.

And in the book, I talk about dinner parties. On the website, I've got articles applying this to testing and debugging.

And so on.

Also, it's how evolution works too, kind of.

Over millions and millions of years, creatures evolve to find positions on the risk landscape that can support life.

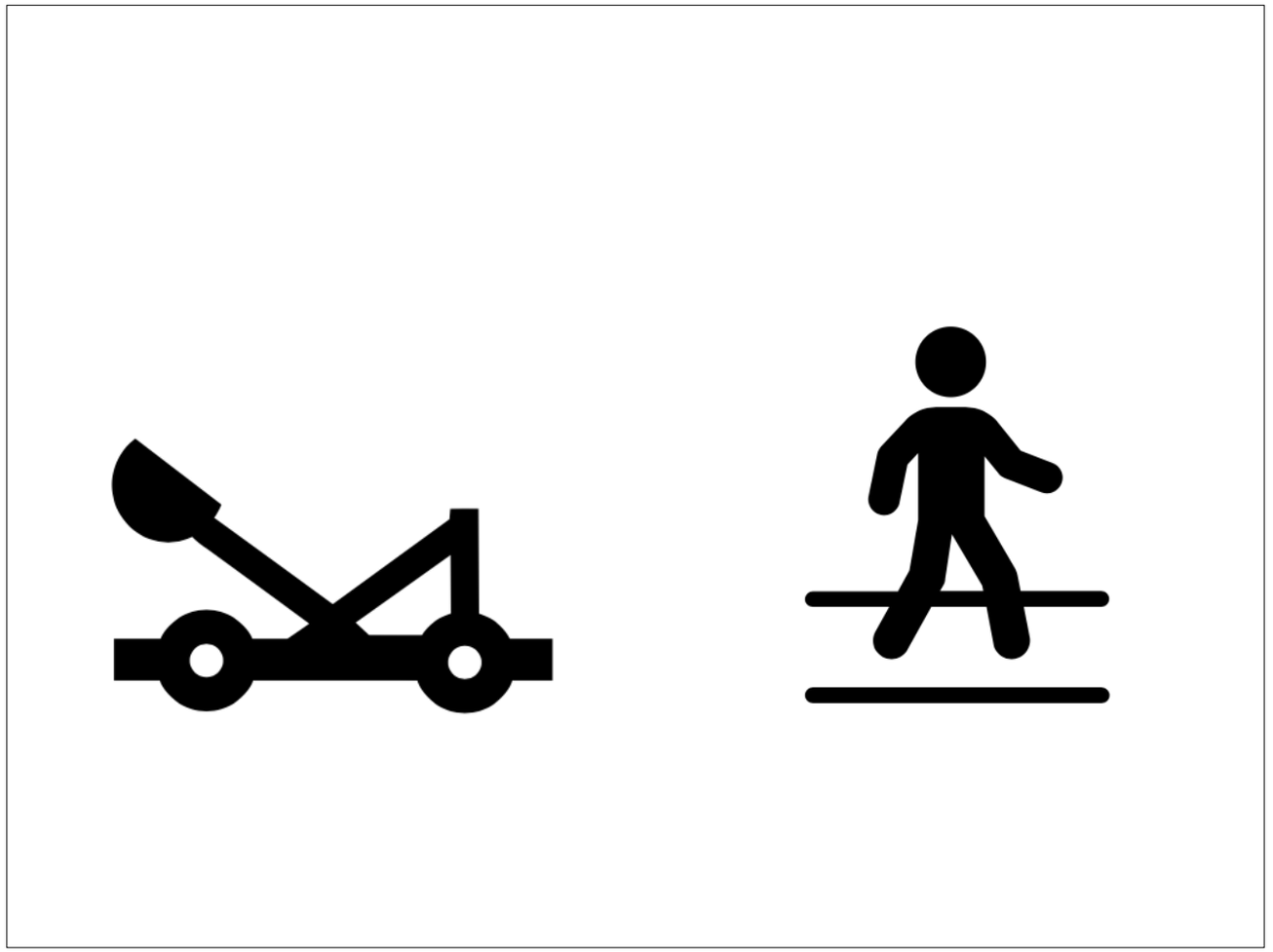
``We've survived 200,000 years as humans. Don't you think there's a reason why we survived? We're good at risk management."

Nassim Nicholas Taleb

That might mean for us, things like minimising Dependency Risk: we need food and water and air and sunlight, and those things are common.

Evolution forces us to address big risks and deal with them, otherwise we leave the gene pool.

This idea of *moving on a risk landscape* and *classifying the risks we see on it* seem, to me to be at the heart of what Software Development is about.



Agile introduced a new way of moving around on that Risk Landscape. Previously, people had been doing lots of planning, and preparation, and calculation. Like shooting a projectile or a rocket.

Agile said: there's maybe another way. We'll figure things out as we go.

True, projectiles go faster than wandering around. But that's a waste if you point them in the wrong direction.

But that was 20 years ago. The Risk Landscape has changed for Software now.

When Kent Beck wrote his book, there were *barely any* open-source projects.

There was no Stack Overflow.

The Wiki had just been invented.

The Risk Landscape really only supported walking.

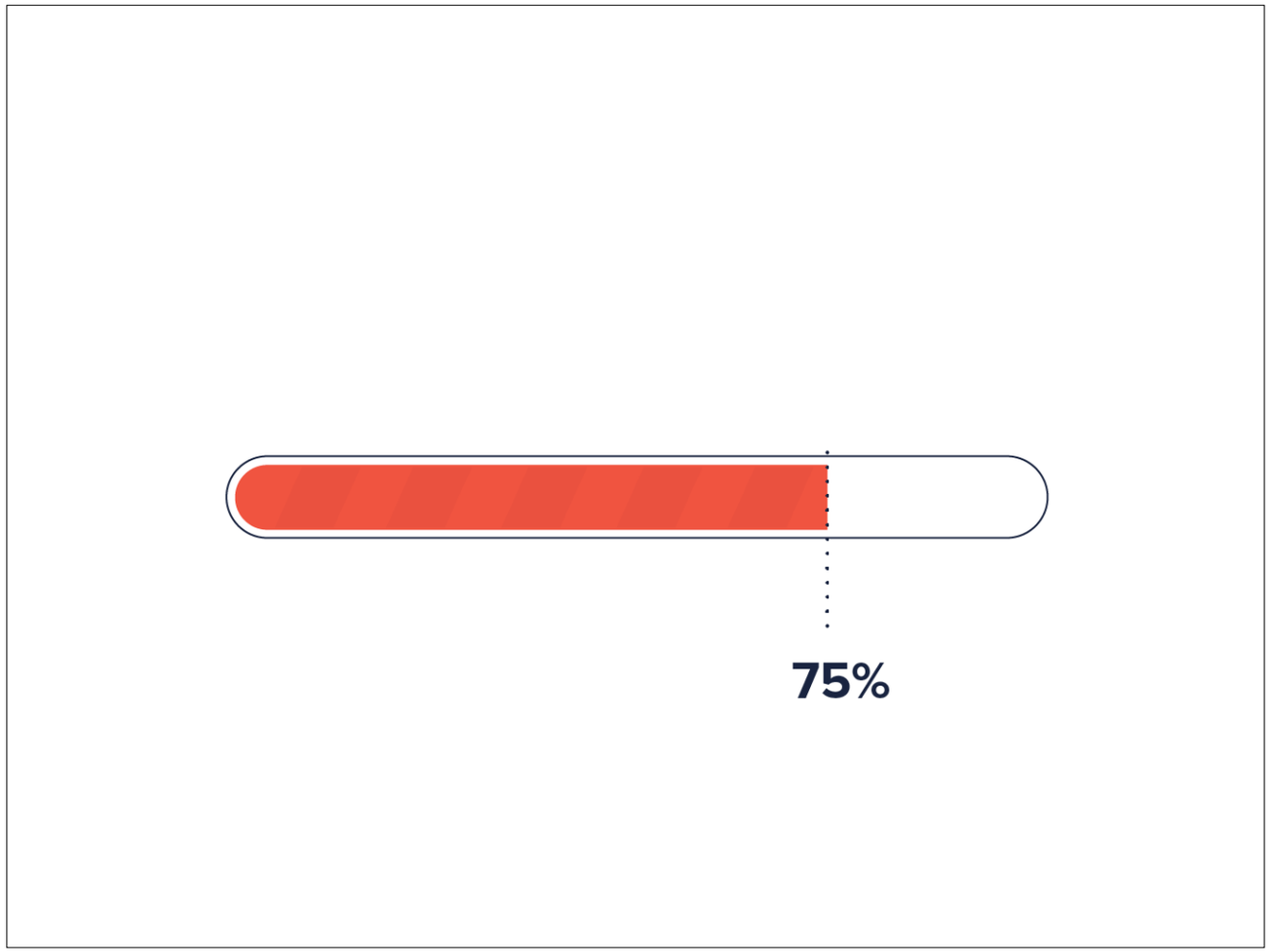
Now, it’s really complicated. The Risk Landscape has been colonised.

A successful IT project is more like bolting lots of reliable components together, which is like taking a journey involving multiple separate stages.

You could take Postgres Route, or the Spring Boot Flyover, or Hadoop Street, or whatever.

They might all promise to get you to the same rough place. But there are different *risks* associated with each, and with combining them together.

What’s the best way?



I am *still* figuring this stuff out.

What I am presenting to you today isn't *done*.

I have a lot of stuff still to work out in this field. If anyone wants to help me do that, maybe apply what we're talking about here to their area of software, that would be really interesting.

Maybe you have a view on how Risk affects Scaled Agile, or Hiring Strategy or something.

That would be a really interesting angle.

riskfirst.org

https://github.com/risk-first