Delivering the Impossible

A book proposal

by Mark Stringer

Table of Contents

Overview

"Point of view is worth 80 IQ points" - Alan Kaye.

This quote from the software pioneer Alan Kaye, inventor of object-oriented programming and the windows user interface, is the inspiration for this book.

In it I show how seeing things from a different point of view can make anyone much smarter. In the specific case of managing projects for software development, changing point of view can make the chances of delivering projects which initially seem impossible much more possible.

The book looks at dangerous and unhelpful ways of seeing software development projects. These are points of view that can get in the way and make things more difficult than they need to be. Sometimes, like wearing a blindfold leaving the lens cap on a camera, these points of view make, possible, viable projects completely impossible.

The author has been managing software development projects for 25 years. His passion, his fascination, is in managing the development of software well and helping other people to do it effectively.

This is lucky because it's really important. The people and organizations who are good at developing software are ruling the world. Companies that can develop and deliver the software they and their customers actually need are thriving. Companies that can't are disappearing or being swallowed up by the ones that can.

Book Specifications

The book will be approximately 70,000 words. I currently have a version which is 60,000 words. There will be very few diagrams. At most there will be one diagram per chapter.

Voice

The voice of the book is clear and frank. The voice of someone who's been a project manager for over fifteen years and wants other people to know what he knows.

The aim is to make the style clear and readable and as much as possible, to avoid business language or specialized terminology.

Structure

The Structure of the book is relatively simple.

Introduction: why does this subject need to be covered? Why am I a good person to cover it? Why now? Who should be interested?

Part 1 - useful and helpful ways of seeing projects.

Part 2 - unhelpful and potentially damaging ways of seeing projects and how to avoid them.

Part 3 - practical actions that projects managers and senior stakeholders can take to make their projects likely to succeed.

Research methodology

The content of this book is based on my experience as a project manager over the last fifteen years (although I managed my first project in 1998). It's also based on my temptation to "read the wrong

thing" meaning that I tend to read much wider than books about project management. Rather I tend to read books about psychology of manipulation and motivation; improvised theatre; why life is more like Poker than chess; Japanese car and washing machine manufacture; and how to negotiate with kidnappers.

Content Length

The current, alpha version of the book is 70,000 words. I expect the final book to be around the same length.

Time needed to complete

I would be able to complete the book in 6 months after a contract was signed.

Author Bio

Mark Stringer is a speaker, writer, and commentator on Agile project management. But he's also an amateur comedian and improviser.

He still works as a projects manager delivering software projects.

He has been working in the software industry for thirty years. But he's not really a technical person. His first degree is in Philosophy.

His first experience of managing software development was working with a team of two developers who had started their programming careers in the former East Germany on Commodore 64s smuggled over the Berlin wall.

Since becoming a full-time project manager he's delivered projects for large multi-national companies, small independent companies, and big, confused government departments.

In the summer of 2023, he took his one-man show "Delivering the Impossible: A short talk about how to rule the world," to the Edinburgh Festival Fringe and received extremely positive reviews.

He lives in London and Thessaloniki, Greece.

Author Platform

At the Edinburgh Fringe Festival in 2023, I arrived in the middle of the August Fringe month without a venue or any publicity to deliver an unplanned unlisted "Fringe Show."

I designed flyers, got them printed and distributed them to potential audience members in Bristow square alongside others who were publicizing their shows.

I performed shortened versions of the "show" in the open air on the Meadows in Edinburgh, mostly to a small audience of friends. I put videos of these "shows" online.

For one night only, I took over the slot of a friend who was sick and performed to a house of 18 people who were expecting a stand-up comedy show about topical issues.

I made it clear to the audience what my show was about and that they were perfectly entitled to leave if they didn't want to listen to a show about project management.

Nobody left.

At the end of the show, I took an exit video of opinions of the show - this is available on youtube.

As a result of this activity, I sold 4 electronic copies of the current version of my book on leanpub.com.

Through the remaining course of 2023 and into 2024 I will perform in the fringe circuit, including the Brighton Fringe and the Camden Fringe, also possibly the Leicester Comedy Festival.

I also intend to run work-in-progress versions of the show at small theatres in London in Autumn 2023 and Spring 2024.

I will take a listed one-man 50-minute Fringe show to Edinburgh in August 2024 for two weeks.

Obviously all of these shows and the publicity for these shows are opportunities to market the book.

Since the Edinburgh Fringe I've been working every day to increase my LinkedIn and Facebook page presence. I am currently live reading a version of my book every weekday and posting the videos to LinkedIn and Facebook.

Audience

As I say in the introduction to the book, I think there are three audiences for this book.

The first and most straight-forward audience is people who have been tasked with delivering software development projects, especially project managers, but also members of development teams; product owners and managers; and senior stakeholders.

The second audience is other people who are in organisations that are dependent on the successful delivery of software for the success of their organisation – this is a much larger group of people.

The third, most difficult to justify group is "everyone else." Maybe not *everyone* else, but certainly anybody who has a project of any kind in mind that they might need to manage.

Competition

There are excellent books that outline an Agile approach to managing software development.

Scrum and XP from the trenches - Henrik Kniberg, Lulu Press, 2007

This is a great, short book which has now gone through several editions. It walks beginners through the practicalities of how to apply both Scrum and Extreme programming methodologies (although neither methodology admits it, Agile is most effective when both methods are applied together).

Extreme programming explained - Kent Beck, Addison-Wesley Professional, 2nd ed., 2004

This is a wonderfully clear explanation of the Agile approach to software development by one of the signatories of the Agile manifesto.

There are books that acknowledge the complex nature of project management without specifically focusing on Agile.

Making Things Happen - Scott Berkun, O'Reilly Media, 2008

This is one of the best-selling books on project management, written by a project manager who worked for many years at Microsoft. It's filled with real-world examples from his time at Microsoft

and is valuable because it acknowledges the messy real-world complexity of project management in a way that many other books don't.

The Complex Project Toolkit - Kieran Duck, Major Street Publishing, 2021

This is the book that is perhaps spiritually closest to "Delivering the Impossible." It tackles the idea that projects are inherently complex head on. And then goes on to offer a set of tools for dealing with complexity in projects.

Finally, there are books that explicitly talk about the importance of adopting a point of view, "or stance" as a way of tackling the complexity of coaching teams.

Coaching Agile Teams - Lyssa Adkins, Addison-Wesley Professional, 2010

This has become a recommended text for anyone seeking to become and Agile coach. In Adkin's view, the team have most of the answers to their problems, what they need is a facilitator to give them the time and the space to find the answers. The coach is a "servant leader" that helps the team through their problems. Part of the way that the coach does this is by adopting various "stances"

My book sits squarely in a space between Adkins' book and Duck's. It deals directly with the difficulties of successfully managing Agile software development projects because of their inherent complexity. It seeks to do this through different "Ways of seeing" software development project management. These could be interpreted as a different, alternative or related series of coaching "stances" as discussed by Adkins.

The way in which my book is different is that it is focused on the successful delivery of software. It advocates a series of effective ways of seeing projects that can make the difference between a project being a deliverable success or an undeliverable disaster.

Book - Table of Contents

Detailed Outline

Detailed Outline

Introduction

- Looking at problems in the right way makes them easier to solve.
    - In this book I'll show you
        - Some helpful ways of looking at project management that makes the problem much easier.
        - Some unhelpful ways of looking at projects that should be avoided.
        - Practical steps that you can take to give your projects the best chance of success.
- Why now?
    - We've always needed this since the start of software
    - Now software is eating everything
        - Companies that can get the software they need rule the world
- Why me?
    - Boring answer
        - Involved in digital 30 years
        - Managing projects 25 years
        - Delivering using Agile since 2010
    - Non-Boring answer
        - I understand it isn't just about following the right methodology
        - Understand project management is irrational, emotional, stressful
        - Want to make projects, less emotional, less stressful and more effective?
            - I have the answer
- Why you?
    - Do you work on software development projects?
    - Do you need software for your organisation?
    - Are you human?

Part 1 – Helpful ways of looking at projects

Chapter 1 – Projects happen between two value streams

- Existing products have a mature complete value stream
- Non-existing products have two value streams
    - The idea (dream) value stream
    - The real (unfinished) value stream
- The contradiction
    - Starting to build the real value stream undermines / attacks the "dream" value stream
    - There are three solutions to this contradiction
        - Side with the dream
        - Side with reality
        - Negotiate

Chapter 2 – Projects are a complex space like a swamp

- The space where we're trying to build a new value stream is complex, like a swamp.
    - Existing structures

        - Natural resources
        - Scarcity of resources
        - Residents
        - Potential residents
        - Owners
        - Property developers
        - Governors and regulators
        - Environmental campaigners
        - Law, religion, superstition
- It's a really good idea to learn about the swamp
    - User research
    - "Probe" the swamp with working software
    - Develop the best relationships you can with stakeholders

Chapter 3 – Software wants to escape

- The ultimate goal of the project is to connect with its users
- In many projects there are barriers to this
- Releasing software to live undermines the value of the idea
- But you need to do it anyway
    - Just getting software nearer live is informative
    - But there's another reason – pull
        - Connecting with users changes the dynamic of the project
- How to plan an escape

Chapter 4 – Never avoid the pirate ship

- In improvisation there is an idea called "agreed activity"
    - Makes improvisation boring
    - Makes projects unlikely to succeed
- We need to deal with whatever is on the horizon
    - Dealing with the obvious ship on the horizon moves the story forward in improvisation
    - Dealing with the obvious ship on the horizon moves the project forward in project management
- But there's a complication
    - Few people want to hear bad news
    - Pointing out the pirate ship can get you silenced, or fired
- Conclusion / sophistication
    - What you see is all there is
        - You need to change people's thinking by changing what they see

Chapter 5 – Empirical process is like driving a car

- In this book we advocate empirical process
- There are three pillars to an empirical process
    - Transparency

    - Inspection
    - Adaptation
- Think of it like driving a car
    - You need to be able to see out of the wind screen (transparency)
    - You need to look through the wind screen (inspection)
    - You need to change what you do because of what you see (adaptation)
- Everything you to manage a project should be
    - Increasing transparency
    - Facilitating inspection
    - Changing (adaptation) what you do in light of what you see

Chapter 6 – Everything is a bet

- All projects are risky
- Orgs take the risks because they want the reward
- But most people don't want to look like gamblers
- So "project management theatre"
    - Carefully follow a methodology
    - Plans showing the project can be delivered
    - Optimistic outlook and "groupthink"
- But if you're the project manager
    - Understand the project is a bet
    - Do what you can to de-risk the project
    - Take it easy on yourself when things go wrong

Part 2 – Unhelpful ways of looking at projects

Chapter 7 – Keeping a promise

- Why are we tempted to see projects are promises?
- Commitment and consistency is a powerful driver of human beings
- But everybody knows this!
- Typical project management agreement
        - Set of high-level project requirements
        - For a particular time
        - For a particular cost
- Looks like a commitment but…
    - All projects are bets
    - Everybody secretly knows this
- Seeing things in terms of commitment interferes with healthy working of the project
    - Heroics
    - Rushed work and bad work
    - Big, nasty surprises
- If we understand how commitment and consistency works we can
    - Remain comfortable in a state of tension
    - Discover value
    - Connect to pull

- Get the right kind of commitment from senior stakeholders

Chapter 8 – Just a simple matter of…

- Why are people tempted to see complex, risky projects as being simple?
- What's the problem with treating work as simple
- What can you do about it?
    - Call out the problems when they arise
    - Publicise the trade-offs and contradictions
    - "Televise" the complexity

Chapter 9 – Faster

- Why are people tempted to want to go faster?
- Why simply going faster isn't necessarily a great idea
- What can you do about it?
    - Show *actual* progress
    - Discourage
        - Corner cutting
        - Heroics
        - "Burning man"
    - Understand what the team really are capable of

Chapter 10 – All

- People are tempted to want everything
- Why are people tempted to want everything
- What's wrong with wanting everything
- What can you do about it?
    - Welcome deadlines
    - Wherever you can connect with users to understand priorities
    - Deliver working software

Part 3 – Practical Suggestions

Chapter 11 – Fried Egg Agile

- This is the way I fry an egg
    - You might do it differently
    - That's OK
    - Obsessing over detail of practice is an "agreed activity"
- My fried egg
    - Meetings
    - Settings / parameters
    - Original idea of Agile has been shown to be a success
        - Lots of variations
        - Variations are different settings for the parameters
        - Remember, it's a lightweight methodology

Chapter 12 – Tips from Negotiation

- Create artefacts
- Understand the basics
    - Lead time
    - Current constraint
- Side by side rather than face to face
    - What should we do about these things
- Explore the value landscape

Chapter 13 – Advice for senior stakeholders

- The streams
- Escape
- The swamp
- The Pirate Ship
- The Bet
- Commitment
- Fast
- All
- Rule the world

Chapter 14 – Conclusion

- Tension is built into the way that projects originate
- Projects are complex
    - Pretending they aren't is a really bad idea
- When we're in a project we can't relieve that tension completely
    - Looking at things in the right way helps manage the tension
    - Avoiding seeing things in the wrong way helps avoid disaster
- Hope
    - There are other ways of looking at projects
    - If you look at projects in these ways you can get more success
- Joy
    - If you and your team get it right, sometimes it sings
    - You deliver something of genuine value for you and your organisation
    - Arthur C Clarke "Sufficiently advanced technology is indistinguishable from magic."
        - When things go right
            - You've delivered some magic

Sample Writing

Introduction

Many projects appear impossible. Most of them aren't. There are almost always things you can do to make them run more smoothly, deliver more value to their owners and users, and make the people who work on them happier and more effective. You just need to look at them in the right way. This book offers some different ways to do just that.

The title of this book is "Delivering the impossible". But of course, you, or anyone, can only deliver possible things. So why did you pick up this book that has a contradiction right there on the front cover?

Perhaps this sounds familiar?

You're working on a project. You're the project manager, and you feel stuck. It seems like the project you've been asked to deliver is impossible.

You're not quite sure how this happened. But at some point, somebody promised somebody that the project would be finished by a certain date (probably Christmas). Maybe you have lots of evidence or maybe it's just instinct, but somehow you know that your team can't deliver it.

Maybe it's worse than that; maybe you still don't even know what needs to be delivered. People are telling you that the project has to be finished by a certain date. But when you or your team try to get a definitive answer as to what "finished" means, you don't get the information you need.

If you're that project manager, then this book is for you.

How do you deliver a project that seems impossible? The answer, which I will explore in depth in this book, is that you look at it from different angles. You find a way of seeing – or multiple ways of seeing – that make it much more possible. I'm going to present several different ways of looking at projects and show how switching between them can give projects that appear impossible a substantial chance of succeeding.

The software pioneer Alan Kay said:

"Point of view is worth 80 IQ points."

My take on this statement is that if you can just figure out the right way to look at a problem, solving it becomes easier. An additional 80 IQ points would make pretty much anyone a genius. And that's the crux of my message to you. I'm going to show you different ways of looking at the problem of project management that can make us much smarter. The idea is that if we can just find the right perspective, we can be much cleverer and more effective in what we do.

But of course, there's also a downside to Alan Kay's nugget of wisdom. Looking at something in the wrong way could take away 80 IQ points. Some ways of looking at things can make us all stupid – and some of these unhelpful ways of looking at a project are extremely common. So I'll talk about these as well, and I'll try to explain why looking at things in certain ways is so unhelpful and show some ways of escaping from their grip.

Why now?

You might think: “OK, I get it, seeing things in a different way is useful. But my project has some specific problems. I just want to fix them. I don’t have time to kick everything up in the air and philosophise. Maybe we could do this another time.”

Part of my message throughout this book is that if you want to fix your project’s immediate problems, now is absolutely the time. Well, for most projects, now is much later than would have been ideal. But now is better than never.

We’ve needed better ways of thinking about software projects since the 1950s. Humans have been managing the writing of software for at least 75 years, and we’re still not very good at it.

And being good at writing software has never been a more important competitive advantage. Being good at developing software that customers want, that is genuinely valuable to them, when they need it, has never been more needed. Being good at software development has become a way of – quite literally – conquering the world.

Am I overstating things? I don’t think so. Marc Andreessen wrote the first widely used web browser – Mosaic. Now he runs a hugely influential venture capital firm in Silicon Valley. As He declared more than ten years ago: “Software is eating everything.” And since he wrote that article, software has increased its appetite. It’s taken huge bites out of sex and dating, politics, social life, music and television. Now, with the arrival of AI, it looks like it’s also taking a massive bite out of lots of other, completely different kinds of work. That includes the writing of software, although this long-predicted trend has yet to truly materialise. Some companies are good at developing software, like Amazon, Facebook and Google. And some of those companies are now more powerful than governments.

Ironically, the craft of managing software delivery doesn’t seem as susceptible to being eaten by software. For now, it remains a resolutely human pursuit.

Why me?

Let’s say that you believe me. Let’s say you agree that now is a good time to get very good at software development. Why should you listen to me in particular?

Well, there are some obvious, boring reasons. I’ve been working in the software development industry since 1994. I started out as a developer, writing software for oil companies, the military and then a newfangled thing called the internet. Then I worked as a researcher, first for Xerox, then for Cambridge University.

You should listen to me because I know how to deliver software projects. I’ve been doing it now for more than ten years. Mostly I’ve been successful.

Some of these projects were successful and made millions of pounds for their owners. Some of them weren’t about money at all. They were literally about making the world a better place. They helped their users do scientific research, fight Covid-19 and cure cancer, or they helped the course of justice run more smoothly.

I really believe that some of those projects wouldn’t have done as well if I hadn’t been involved. And that’s about as much as you can ever say if you’re a project manager.

But there’s a slightly less boring, less obvious reason. Most people who claim to be experts and talk or write about project management will try to give you the impression that it is a rational endeavour.

They will suggest that if you can simply follow the steps outlined in their methodology, your project will be guaranteed to succeed. I am under no such illusion. I know that project management is an irrational and emotional business.

As we'll see, that irrationality and emotionality – far from being accidental and only cropping up when projects are managed badly – is built into the way projects happen. There will be substantial stretches of time in any project where, if you are involved with its management, things will be tense and you'll feel bad. What I try to explain in this book is something that I haven't seen explained in many other places. Certainly, I haven't seen these ideas collected in one place. I want to explain why you're feeling bad, why it's OK to feel bad and what you can best do to deal with these negative feelings. And I want to show you that the way to do this is by looking at project management from a different angle.

Why you?

I think there are three groups of people who should read this book.

The first group is the most obvious. Are you directly involved with the development of software or with the project management behind the development of software – then you should read this book.

But there's also a second group. Software is eating everything, as we've just established. Do you work in a business that's getting eaten by software? Do you work for a company that needs to be able to deliver software in order to stay relevant and competitive? If you do, you should also probably be interested in what makes ways of looking at project management either smart or dumb. You should be interested in getting software developed and delivered, and so, you should also be interested in this book.

Then, finally, there's a third group of people. And that's everybody else. If you're involved in any sizeable project that needs any kind of management – not just software but all kinds of projects – if you might ever be involved in such a project, or if anyone you know might be involved in such a project, then you might also benefit from reading this book.

Chapter STREAMS – Two different value streams – and an awkward phase in the middle.

A value stream is a set of processes that add value to a product and then deliver the product to people who want it. When you manufacture existing products, there is an existing value stream. But when you are developing new products, there are two value streams: a weird one, and one that doesn't exist.

"You can't step into the same river twice, it isn't the same river, it isn't the same you."- Heraclitus

This might be the trickiest point that I make in the whole book, and it's right here, front and centre at the start of chapter one. That's because in this chapter, I need to talk about "value streams", and I know that this will be an unfamiliar concept to many readers.

Value streams are often talked about when discussing the manufacture of cars, especially in Japan. In

the mid-1970s there was an energy crisis, and oil prices were high. American car manufacturers started to lose out to the Japanese. Japanese cars were more reliable, cheaper and far more fuel-efficient than American cars. In many ways, Japanese cars were just better.

Researchers at US business schools began to write about why the Japanese made better cars than the Americans. A key idea in explaining why Japanese industry, and the Toyota car company in particular, had become so successful was the idea of the "value stream". And together with the idea of a value stream came a very specific notion of waste.

A value stream is a series of actions that culminate in the production of a product that is valuable to its customers. So as the chassis of a car moves down a production line, value gets added to it. What sorts of things add value? Well, in the case of a car, an engine. An engine adds value. Paint, wing mirrors, seats. You get the idea. But note, none of this value gets realised until the car is finally being driven by the people who want it, the end-users.

American observers also noted the Japanese understanding of what *doesn't* add value. Activities that go on in a factory without contributing to the value stream are considered waste. There's an example in Taiichi Ohno's book The Toyota Production System of a machine that is part of the production line and must be changed whenever a different model of car is made. All the time that it takes to swap the machine, is, from the point of view of a value stream, waste. That seems easy to understand. But Taiichi Ohno also identified other kinds of waste at Toyota which perhaps aren't so intuitive. Once they have rolled off the production line, any completed cars waiting around in a yard aren't delivering any value. That's also waste. If parts are stacked high in warehouses and not being added to a car for long periods of time, that's waste too.

This is a crucial aspect of thinking about value streams. The value is only truly realised when the product that has been through the production line is sold and delivered into the hands of a customer.

This study of how the Japanese made cars resulted in at least three interesting ideas:

1. The manufacture of a product can be arranged into a series of value-adding steps.
2. Any waiting around between these value-adding steps is waste.
3. Any waiting around before the final step of getting the product to the people who use it is also waste.

These ideas were interesting to a lot of people. They offered an insight into how any organisation might improve its process and make it more efficient. They did, however, involve quite a leap: as you may have noticed, not all organisations make cars.

Above all, one core idea came out of these discussions: that it is possible to map a value stream. It's possible to see what all the steps in a process are. By extension, it's possible to see where there are delays and where there is waste. Once you've found where the waste is, you can work to reduce it. From this way of thinking came a whole new way of organising many industries and the notion of just-in-time manufacturing.

You might be able to see the appeal of looking at work in this way. If we can map the value stream, we can reduce waste between processes. We can look in particular at reducing waiting times. Then we can, at least in theory, speed up any production process and make it more efficient.

So, there it is, that's the idea of a value stream: a series of steps that add value to a product. And various people have made good solid efforts to apply these ideas to software development. Some

ideas that have emerged from these attempts are useful, like the idea of limiting the amount of work in progress. But overall I don't think these attempts have been as successful as they could be. And that's for one very obvious reason: manufacturing existing products is very different from creating new ones.

This is because when you are developing software for a new product, there isn't just one value stream. There are two value streams. And at the beginning of a project, one of these value streams doesn't exist and the other is weird. Oh, and there's a final complication: the two value streams in software product development interfere with each other. And when I say "interfere", I mean that they try to kill each other.

So, what are these two value streams? The first is what I'm going to call the "virtual" value stream. This is a stream that delivers value by looking good, sounding good or coming up with an attractive new idea. The payoff for this value stream is in the funding and institutional support that arrives to make the idea happen. It's also in an increase in reputation and standing of the people who came up with the idea or who manage to take credit for it.

Is this really a value stream? In some sense, it must be. Ideas don't just instantly appear fully formed, and projects don't just instantly get funded. So, there must be a value stream here. But quite how it works in most organisations is opaque, which is why I say that it's weird. We will talk more about this stream a bit later. For now, I want to move on to the other stream in software development projects: the one that, at the start at least, doesn't exist.

This second stream is the one I'm going to call the "virtuous" value stream. This is the stream that delivers value, not by looking good, or sounding like a good idea, but by actually *doing* good. "Doing good" means delivering something that is valuable to the people who use it.

Because "virtual" and "virtuous" sound a bit alike, I'm also going to borrow an idea from Taoist thinking. This is the distinction between flowers and fruit. The virtual value stream is like flowers. The virtuous value stream is like fruit.

Don't get me wrong. It might seem as we go through this chapter that I'm a little bit down on flowers. I'm not down on flowers. Flowers are great. In project management, the part of flowers is played by ideas and dreams. Without ideas there wouldn't be any projects.

People have ideas, those ideas get money and support, and that's how projects are born. Something that some ideas and most flowers have in common is that people like them straight away. They don't have to think about it. They have some attributes that are instantly attractive. It doesn't take any thought to like a rose. Of course, some people might not like roses. But the people who do? They don't have to take time to work it out.

Something similar seems to happen with ideas. The kind of ideas that get money to turn into projects seem to have a structure that makes people like them without too much thought. Just like flowers, those who are exposed to them tend to like them without doing much or any thinking.

Often ideas are talked about in these terms: easy, fast or cheap. Other times, the idea's appeal is that some things won't change:

- "It will be fully compatible"
- "It will have the same functionality"

Sometimes the idea's appeal is difference:

- "A fresh look"
- "A different take"

Sometimes all these things get bundled together:

- "A radically different take, which does exactly what the old system did, but faster, quicker, more easily and for less money"

And of course, all of this is almost always going to be delivered using the latest technology:

- "Does everything that the old system does, but quicker and cheaper and using AI"
- "Deals with all customer enquiries, using chatbots without the need for human involvement"
- "A social network for dogs, using blockchain"

This "instant appeal" aspect of ideas plays a substantial part in why I say that the virtual value stream is weird. Yes, there might be an application process for getting ideas funded, and as I mentioned there must be some steps to the process. There might be a lot of "theatre" surrounding the process of approving funding, but equally there might not be. It could just be the idea of someone senior, or someone who has the senior person's ear. Some projects get funded on a whim. Maybe no one in the organisation has *any* idea why they're doing this. Sometimes all it takes is the fact that another organisation is doing it and there's a strong inclination to do whatever their competitors do. Even if there is a more involved process, at some point the central proposal of the project is going to be evaluated. And then the mechanism of how an idea appeals to people is going to be vague.

After all that, The appeal of certain ideas might not have any obvious logic to it. But from the examples I've given, we can identify some common themes in the ideas that get funded. The chosen ideas often have these concepts lurking within them: "all" and "same", but also "new, " "fast" and "cheap".

Where ideas come from is a fascinating subject, but I'm going to sidestep that for now. That's because the whole point of this chapter is that, for a project to deliver, it needs to stop being fascinated by the idea. Rather, it needs to start being fascinated by the reality. We need to stop smelling the flowers and think about how we're going to grow the fruit.

And there's another important difference between flowers and fruit. Users eat fruit. They only look at flowers. Nobody ever died from looking at a flower (probably). A sour apple can give you bad indigestion, and some fruit is poisonous. A *lot* of fruit is either poisonous or unpalatable until it's been properly processed.

Similarly, delivered projects need to give value to their users. This involves interaction with users, which is different from looking at, or smelling, flowers. Users need to get some good, some value out of using a software product. The nature of the "good" or value is different for different kinds of products. For example, it's very different for a social media product than for a government form. But like fruit, if it doesn't taste right, users will spit it out. Also like fruit, if software isn't put together (or grown) properly, it could do them harm.

So, what about fruit? What about this other value stream – the virtuous value stream? The one that gives people what they want? In the Toyota product system I talked about above, this second value stream is the focus of all the attention. In fact, it's talked about as if it's the only value stream. Taiichi Ohno is always asking: "How can we improve flow through the system?" and "How can we get parts

to the right place in the process just in time?"

There is an aspect to this that often gets missed when people try to transfer these ideas to the development of software. Most software development projects are creating an entirely new product. The problem that they're trying to solve isn't one of trying to make a better car. Toyota didn't invent cars. Our problem is like it would be if *nobody* had ever made a car. Our product is a new product; it's a one-off. Nobody has ever made one exactly like this before.

Am I exaggerating? Maybe a little. It's more like you and your team are trying to build a new kind of vehicle. If you're building a video game, well, there have been video games before. But you're still going to need to take aspects of the idea and find out which bits users respond to. And then you're going to need to connect them together in a game that they want to buy and spend hours playing. If you're building a form-filling application, well, there have been those too. But you're still going to need to understand what information you have to gather. You need to understand what the best way is to ask for the information. Then, when you've got the information, you need to know how it's going to be stored and who's going to want to look at it. The important thing to understand and admit is that you're in a *prototyping* process, not a manufacturing process. You're assembling (irrigating?) a value stream, not operating or improving an existing one.

So how do you do this? How do put together the value-adding steps that will eventually deliver something that realises value for the customer?

Well, you have to do two things.

Firstly, discover value: explore and find out what the potential users of this product might want.

Secondly, construct a value stream: try to put together some software in an application that can deliver these values.

In the next chapter [Chapter SWAMP], we'll talk about why, how and with whom you should work to discover value. Then in [Chapter ESCAPE] we'll consider how to put these bits of value together and start a value stream.

Simple, right? Well, it would be, if it weren't for a third thing that you have to do as a result of these same two steps. You have to deal with all the flak and fallout that comes from discovering value and constructing a value stream.

This is the central paradox of project management. Starting to do the project results in the alienation of the people who got the project funded. Why? Because when you do things, you uncover problems. Doing things also inherently involves the risk of making mistakes. By trying to implement the idea, you undermine the idea.

This is especially true of using an Agile approach to software development. Most Agile approaches to project management have a short meeting that happens every day. This meeting is the team's opportunity to report any problems they're encountering in their attempt to realise the dream, i.e. the idea that has been funded. Add into this mix user researchers who are talking to the users about their needs – and we definitely should add them. Then, pretty soon after the project starts, you begin to get a picture of what users think of the idea. Maybe they love it, maybe they hate it, maybe they think about it in a way that is totally unreasonable. Maybe they point out really good reasons why it won't work. They think about it in a way that you really didn't anticipate.

That's the thing. What precisely users think about it doesn't really matter. What matters is that the

way they think about it is pretty much guaranteed not to precisely mirror what the people who had the idea think. And to them – the "keepers" of the idea, the people who have attached their reputations to getting the idea funded – unless they have deep and strong experience in product management and development, this is going to appear as a threat.

This is a sticky, difficult, delicate business. And we're going look at it from a series of different angles in the hope of reaching a smarter way of viewing it [Chapter CHAOS, Chapter CONVERSATION]. But it's always going to be tricky. On the one side we have the dreams; on the other side, where we're headed, where we need to be headed, is a value stream. But it's a value stream that, if it delivers at all, will deliver something that is slightly, or hugely, different to the dream. The reality, if it's to be a success, will be intimately involved with users. It will engage with their enthusiasms, capabilities and limitations. If it's going to have a chance of success, it should also address the interests and concerns of lots of other stakeholders. Regulators, spectators and potentially investors will also be interested, as well as the people who came up with the idea.

Note that this is a vision of a project as a balancing act between dreams and reality. On the one hand are the dreams of the people who had the idea and funded it. On the other are the wants and needs of the people who experience the reality. And between these two is a hard place – a very hard place indeed. This is where support from the people who had the idea is running out and support from the people who will benefit from the reality hasn't started. This isn't exactly a "standard" view of project management.

But it is a standard account of something else. Stories and adventures.

In a lot of adventures, the hero has a dream, a vision. Or there's always been a dream hanging around, maybe a prophecy. The hero has no real idea how they're going to realise that vision. But, after some reluctance, they set off to try anyway. On the way, they find things (a sword, a statue, some writing on a wall). People tell them things (an old man in the forest points in a certain direction). And then they start to get a better picture of what their quest might be about. They have lots of struggles and setbacks. They think about giving up. But they don't. In a final scene, they put all these things together, save the day and win the prize. Then they go home for whatever is the nicest meal possible, in that time period, on that planet or in that culture.

If you're trying to deliver a project, you're on that kind of adventure. You might not be comforted to hear this. And, weirdly, neither are the people who had the original idea for the project. The people who funding the project – they probably won't be that pleased either. But that's where you are. The idea was just the start. Someone has to head out on that journey. You have to find what's valuable. You have to find the jewel, the chalice and the sword. Then you have to throw those things that you found into the fire in the temple in the right order. Then and only then can you save the world and go home for tea.

How does this help? How does seeing a project in these terms make "delivering the impossible" any easier. Remember the quote from Alan Kay? "Point of view is worth 80 IQ points."

I believe that thinking of a project in terms of value streams is an interesting and different point of view. One value stream is a virtual one (the flowers). It deals in dreams. The other is a virtuous one (the fruit). It needs to be built through the course of the project. I think this point of view makes it easier to understand what's going on in a project. For example, viewing a project in this way explains the team's reluctance to tackle the pirate ship that I talk about in [Chapter SHIP]. Why don't the team want to tackle the pirate ship? They don't want to go on the adventure! And why would they?

Adventures are dangerous and uncertain. That sense is compounded if the captain has said that this shouldn't be an adventure, that he doesn't want to hear about anything going wrong, they won't want to go. The leader might not have said anything. Maybe they're just relying on intuition. The leader's feeling about taking risks was shown by the way he reacted to even the tiniest thing going wrong.

To embark an adventure, a team need encouragement and support and someone who knows the kinds of things they should being doing. They need someone who is prepared to go with them and provide encouragement. They want someone to acknowledge their struggles and try to give them the best chance of success – and that someone is you!

Taking this point of view, seeing a project as constructing a path through a wild and unknown territory, helps. It helps with one of the major things that makes any project manager feel bad – the idea of commitment and consistency.

The thing that gets a project funded is an idea. The aspects of an idea that get ideas funded aren't very closely, if at all, connected with their deliverability. Rather they're very closely connected with a certain kind of "shiny" simplification. To some people, words and phrases like "new", "fast", "just like the old system, but better" and "using AI and blockchain" have a flower-like appeal. They don't need to think about why they like them. When you understand this, you start to realise that a funded project is quite simply a signpost pointing in a certain direction. Nobody knows *exactly* what you're going to find there. Nobody knows if you're going to be able to put together what you find there and get it to deliver value. If you're the project manager of this "gesture in a certain direction" that's got funding and resources, what can you possibly promise? What can you possibly commit to? How can you be consistent?

This isn't a rhetorical question. There are a bunch of things you can commit to. You can commit to going on the journey and taking the team with you. You can commit to doing all you can, as a team, to explore the area that the project points to. This is the "swamp" that we'll talk about in the next chapter [Chapter SWAMP]. And you can commit to finding what's useful, valuable and attractive there. You can commit to putting together a prototype value stream that connects these values together to deliver that value as soon as you possibly can. That's what working software is, and we'll be talking about it in [Chapter ESCAPE].

When you start to look at projects in this way, there is both good news and bad news. The bad news is that for almost all projects you will have to go on the adventure. There is no way of taking the idea, the thing that got funded through the "virtual value stream", and just implementing it. There is almost never enough detail in the idea. In fact, lack of detail is probably part of the idea's appeal. And then there's even more bad news. The only way to realise the idea is to extensively explore the area that it points at. But then, having explored, you need to get decisions from interested parties, users, stakeholders, on which of the things you find are valuable. Then, out of all the valuable things you've found, you, your team and some representation of all the other interested stakeholders need to decide which of those you're going to put together to form a value stream. Finally, you have to build that value stream. That's a lot of work. And it's work that has to be done while somehow not upsetting the people who had the idea.

So, lots of bad news. Is there any good news? Well yes, there is. If you follow this approach, you will have given yourself and the people you work for the best odds of delivering that they can have. You will have given yourself a chance of success, even on the most impossible-sounding projects. Looking at projects in this way might just give you the bump of 80 IQ points that Alan Kay talked about. It helps you know what you're doing. Why are you doing user research? Because that's how we

discover values. Why are you putting together working software and putting it in the hands of users? Because that's how we take the values that we've uncovered and start to put them together into a value stream. And that can deliver for almost any project, no matter how crazy the idea sounds.

Finally, there's one more bit of news that you could get. It doesn't come to every project, but when it does, it's really good news.

This is something that comes from the infamous veteran sales copywriter Gary Halbert. Halbert wrote one book – and he wrote it from jail where he was serving time connected with something he'd put in some sales copy. Gary Halbert tells a story of a question that he asks people who come along to his copywriting courses. He sets the scene. These aren't the exact words. But this is the drift: "You're going to have a hamburger stand, and I'm going to have a hamburger stand. You can have anything you want to make your hamburgers sell and I'm going to have just one thing, and I'm going to beat you every time."

Of course, his students now play what we'd recognise as the "hipster" burger game. "Kobe beef", "sourdough bread" and "ethical lab-grown meat". Lots of fancy features on the product. Or they go the "super-value" way: "buy one, get one free".

Then Gary Halbert tells his students the thing that he's going to have that's going to mean his burger stand is more successful than theirs – a hungry crowd.

A hungry crowd.

This applies to burgers, and it applies to software product development too. A hungry crowd is the best bit of news you can possibly get. If the organisation that is paying you to deliver this project really, really needs this project to be delivered, that is a huge benefit. Even better is if the users of the product really need it to be delivered. In my experience, when an organisation is desperate for a piece of software, they will tolerate the kind of approaches that we've talked about. They will allow user research; they will allow early release of working software. Notice I say "tolerate" and "allow". That's very different from "enthusiastically support".

So, there's a simpler idea, beyond all the talk of fruit and flowers, virtual and virtuous value streams. A simpler way of seeing projects and seeing what you're doing when you're managing them. And this is in terms of push and pull. Ideas are push. A hungry crowd is pull. Ideas push you out into this "land of adventure". You find all sorts of things that might be valuable; the challenge is to decide which of those to connect together.

Chapter SWAMP – The Swamp

Any project that has the potential to be of value has an element of the unknown about it. It's like trying to build on a swamp. It's a complex space, with complex geology, history and ownership. The only way to progress is to acknowledge its complexity and explore.

"In wildness is the preservation of the world" – Henry David Thoreau

Imagine that a property developer has given you money to develop houses in a swamp. How would you go about doing that? Would you just start work? Or would you feel the need to do a little bit of

exploration first? Wouldn't it be a good idea to know what's in the swamp?

What if I told you that there were people who already live in the swamp? And that there's no way that you're going to be able to build without upsetting them? They don't want you to build there; they're quite happy with things just the way they are.

But then again, there are also people who are keen to move into the swamp when you have built something. They've already given the property developer money. They want you to build in the swamp fast, so they can move in.

People who live there and people who want to live there both have interests in the swamp. But they aren't the only ones. There are people who care generally about bogs and marshes. There are people who are interested in building regulations for any kind of engineering or construction. There is government. Local government will probably want to levy some kind of tax. You will probably need planning permission. National government will have building regulations. There are people you need to talk to about getting a road to come right up to the swamp. Because whatever you're doing in the swamp, the likelihood is that you'll need to connect to existing infrastructure.

And beyond government there might even be global bodies that care about this swamp. There are people who don't care about you, don't care about the property developers, all they care about is the swamp. These people might be eco-activists. They might be political activists. The swamp might be on a very old burial ground. It might be the site of a buried temple.

And that's just the people who are interested in the swamp. What about the plants and animals that are already in the swamp? Maybe there are new species there of interest to science. Maybe there are plants, unknown to western medicine, but the locals use them as a miracle cure. What about the geology? Maybe there's oil. Maybe there's gold.

Why am I babbling on about swamps? What does this have to do with software product development? What's the point I'm trying to make?

My point is that there are a lot of people who have very varied interests in what you're doing. Some are directly associated with the project; some of those will directly benefit from it. But there are also a lot of other people who are only distantly associated with your project. When you start, you might not have heard of them and they might not have heard of you, but you're still important to each other. If the project is going to succeed, it would be good to keep those people happy. Maybe there is something that you can do for those people that would make them happy, that would change their world for the better, or make the organisation you're working for a fortune, or both.

Equally, it might be a bad idea to annoy them. Maybe you can't avoid it, but it would still be good to know that it's going to happen.

Here we're picking up on the way of seeing projects that I talked about in [Chapter STREAMS]. Projects are things that are done in a complex environment. There are lots of people who are interested in that environment. And the interests that they have are extremely varied.

I didn't really fully understand this until I worked on one particular project. This involved working with a woman who did research with users. I'd worked on other projects that had user researchers. But they hadn't done research like she did it. She *really* did research with users. As soon as the project started, she contacted potential users of the product and arranged meetings. She was running workshops with them, getting to know them and developing a relationship. First, she got to understand their "pain points". Then she got them to start to sketch, with pen and paper, graphical

user interfaces that might start to address these points.

Then the designer on the team designed interfaces informed by this research. They looked real, but at this stage they were still only models. These were again tested with users. If there were serious problems, they were modified and then tested again.

This kept going. She hired another researcher. They both did research on users. They did research with internal users in the organisation we were working for and external users, who were the organisation's customers. I worked on this project for over three years. And the research never stopped. It didn't stop when we had working software that we could test with users. It didn't stop when we had a version of the software that had a copy of the users' sensitive data running on it. This gave them a chance to see what it would be like using the live product. It certainly didn't stop when we went live.

As I said, I'd been involved in projects that had user researchers before. But this was the first time that I'd seen a project that was led by user research right from the start. And the project continued to be informed by user research all of the way through development.

I learned a lot from watching this happen. And there were a couple of things that came as a surprise to me. The first was that if you do this much research on your users, you won't just find out about your users. You'll get to know a lot about other people who aren't exactly users. But, to return to our metaphor, these are still people who are interested in the swamp. We found other regulatory groups that our users needed to please and that our product had to support. These were people who, even if they weren't users, were stakeholders. And they went on our stakeholder map.

This was another important idea that I hadn't come across before.

Do enough research on your users and you end up with a "stakeholder map". Do even more research on your users and you get a "stakeholder ecology". You start to understand not only who all the people are who have some interest in the swamp, but also how these stakeholders interact. You start to see what's really happening in and around the swamp. You start to see opportunities to do good and to succeed. Many of these would not be at all obvious if it weren't for these investigations. And you also start to see where there are dangers and problems and potentially unhappy people or tricky regulations that could cause trouble for you and the project.

This project was the first time that I really understood the value of user research. But I really should have understood it about 15 years earlier. After all, I had previously worked on a user-centred design project as a researcher. Maybe the thing was that it wasn't exactly a software development setting. Maybe I'm just a bit slow. But I hadn't quite joined the dots.

"What can a ten-year-old teach me?" I was sitting in an office in Athens, arguing with a Polish software developer in a room full of technical guys from all over Europe. None of them seemed to think that we needed to do any user research on a project that had the phrase "user-centred design" in its title.

It was an EU-funded project. The first aim of the project was to design an application involving tangible user interfaces. These are interfaces that you can get hold of, pick up and move around. The second aim of the project was to use these interfaces to help schoolchildren construct discursive arguments. My organisation was doing the user research for the project. As I said, "iterative, user-centred design" was in the title of the funding bid. Still, most of the software developers and search engine experts on the team were sceptical. They didn't see the point of talking to users, especially

since in this case our users were schoolchildren. But still, we did it.

And what we found was interesting and – as I realise now is almost always the case when you do user research – unexpected. It pointed out a potential route towards making a successful product. And this was a route that, if we hadn't talked to users, we would never have imagined.

One thing that we found straight away from talking to users (schoolchildren aged 10–11) was that they didn't really need help with the logic of an argument. For example, when discussing the subject of graffiti, they understood that there was a contradiction between statements like:

People should be able to express themselves freely

and

It's against the law to paint messages on public buildings

What the kids did seem to need help with was putting the bits of evidence that they found into a structure that worked as a persuasive argument.

That was the great thing about doing research in real schools. But when we started to do that research, we found out about another concerned party who didn't live in the swamp (which in this case was a school – a very nice school I should say,in Leamington Spa). The concerned party was a long way away from Leamington Spa. Even so, they really cared about what happened there. Who were they? They were the government. At the time when we did this research, the government insisted that all schools in England follow a National Curriculum. This meant that pretty much every lesson the kids did had to show that it satisfied a specific learning requirement.

We realised this after these early research sessions. We needed to design our future research sessions so that, at the same time as giving us what we needed for research, they directly addressed something mentioned in the curriculum. And we needed to make it obvious to the teachers that that was what we were doing. Why? Because the teachers were also swamp residents, of course! If we did this, we made it much easier for the teachers to support us. Otherwise, the number of lessons that we'd be able to have with the children would have been severely limited.

Fortunately, we had an ex-teacher on our team. This allowed us to design our research sessions so that they also made sense as National Curriculum-focused lessons.

A long time later, I found myself working on a software development project. This company was successful and highly regarded in one area of business: mobile phone service provision. In the world of mobile phone service provision, it had a pretty cool image. So, some of the senior people there wondered if they could succeed in another area: personal loans. Yes, that was the project. Making personal loans cool. Again, this team had done a lot of work developing a personal loan website and they'd done lots of research with users. Then a potential business partner mentioned a possible problem. They pointed out that any company that lends money needs to be regulated by the financial authorities.

You might raise your eyebrows slightly at a company that didn't talk to the regulators before starting up in the money-lending business. But I would claim that a substantial part of the reason that this got missed was that they hadn't done enough research. On this occasion, that might not strictly be user

research. But one way or another they hadn't sufficiently explored the swamp.

My bet is that research with users and mapping of stakeholders would have made something as important as "If you want to be a money lender you have to fill in a big application form" very clear, very quickly. This project would also have seriously benefited from user research; it later transpired out (another big surprise) that most of the people who want to borrow money are either expensive or impossible to lend to because they are bad credit risks.

But the point I'm making here is that it isn't enough to do research with users. You also have to find all these other people: regulators, investors and commentators on the ethics of your industry (the interest rates needed to make lending to people who are a poor credit risk viable come with a set of commentators and critics). To return to the swamp metaphor, these are people who don't live in the swamp. These are people who live miles from the swamp and will never visit it. But they still care what goes on in there. And if these people aren't happy, they can sometimes kill a project stone dead.

What am I really saying when I advocate for user research, when I talk about the environment of a project being a swamp? What I'm saying is that looking outside of a project, at what is surrounding it, what is influencing it and then changing what you do in light of what you find, is one of the main ways of increasing the chances of success. This is something we'll talk about again from two other points of view, in [Chapter SHIP] and [Chapter SWAMP].

But lots of people will say that user research is a waste of time. Actually, more often than not, they won't say that. They will agree with you. Yes, yes, user research is very important, but it just isn't needed on this project. Or they won't bother making any argument, they just won't put any budget for it in their projects. They won't hire people to do it. Lots of people will see a software development project as merely that – a matter of software development.

Do you know what you should do if you're managing a team in this kind of situation? You should do "user research" anyway. If you don't have somebody on the team who's a dedicated user researcher, you should still do some user research. You should do "user research" with the people in the team that you have – on the basis that they're sentient humans and their responses to user interfaces will still be interesting and useful. And you should still push to talk to users, or where possible include a user as a product owner or business analyst. OK, it won't be anywhere near as good as a full programme of user research carried out by professionals. But one of the main aims of doing this research should be to find out surprising stuff (you will). Then maybe you can take what you find back to whoever is paying for the project. Maybe you can show them what you have found and use it as a case for more user researchers and stakeholder research.

Another criticism of my insistence on user research is what might be called the "faster horse" objection. Henry Ford is said to have declared something like "If I'd given the people what they wanted, I'd have given them a faster horse." When I talk about the swamp, I'm not saying that you should give any of those people associated with it exactly what they want. What I am saying is that whatever you do has a much greater chance of success if it's informed by what they want, hate, fear, hope for and wish for.

I've worked on lots of projects where we either haven't done any user research or haven't done anywhere near enough. And, if I'm being totally honest, on most of those projects, it didn't bother me

that much at the time. It's easy to get wrapped up in the problem as it's presented and not go "looking for trouble" by exploring the swamp. But all of those projects found trouble anyway or found no audience for what got built because of a lack of understanding of the environment in which they were working. And that's exactly why you should be getting user researchers involved in projects right at the beginning.

I described a project earlier: the one where we did research with children in the classroom. That project was described as "iterative, user-centred design", and that is pretty much what we did. I should have learned from that project how important user research is. But I still didn't really put it together. I still didn't quite realise how important for the success of a project it is that you do iterative design and development, and that it should be user-centred. The only way to do that is to keep talking to users and keep putting working software in their hands.

Now that I've seen it really work, I'm convinced that it's one of the best ways of reducing the risk on a project. But it's still going to be hard to persuade the people who are paying for that project that they need user researchers. And not just one, and not just at the beginning. They need people talking to users and potential users all the way through.

There's something I feel I should confess here. I'm convinced of the value of continuing user research throughout a software product development project. But I don't know how to combine user research and the early development of working software without creating disagreements and arguments.

Research on users and stakeholders needs to keep going all the way through a project. And what the researchers find out needs to inform software development. Then that software needs to be put into the hands of users. This is what we'll talk about more in [CHAPTER ESCAPE]. Then the team needs to listen to the feedback that comes back from users. This feedback needs to be considered and incorporated into future versions of the software.

Making sure that the team are well-informed is important. The best chance of a project succeeding is when the team are aware of what users and stakeholders want. The chances get even better if the team can put working software into users' hands. And even better still if feedback from users starts to guide future iterations. But it isn't an easy ride.

When the interests of some or all of the interested parties in the swamp are discovered, it's very unlikely that they'll all be compatible. And if they aren't, there are often going to be a lot of different possible solutions that fully or partially satisfy one or other of the groups' needs.

In Agile methods, the final decision about what goes in the product is supposed to rest with the product owner. The idea is that they listen to the feedback coming in from all directions and make good decisions about which bits to listen to and which bits to ignore. But of course, the product owner is just another human being. It's very likely that the decisions that they make won't be perfect.

In a way, it's your job as a project manager to make sure that these conflicts happen. But at the same time, you have another important job as a project manager: you need to look after the team's psychological safety . Reassure your team that conflicts that emerge from doing research are good, healthy conflicts. You may not get things right initially. But if you're setting up an iterative process, you don't have to. At every stage, even if you get something very wrong, further user research will pick it up and allow you to correct it.

What if your project has no budget at all for user research? What if it has all the resources it needs to do good research? There is still something else that's good to do as an additional way of exploring the

swamp.

Talk to your stakeholders about risks.

Talking to senior stakeholders is, in my experience, an extremely good way of finding out about what's going on in the organisation. I had worked on many projects that had a "risk register". My experience was that this was a document that got written at the very beginning of a project and then ignored.

As a result, my opinion of risk registers was that they weren't very useful. Then, at the very beginning of a project, the product owner informed me that the organisation mandated that we have a risk register. I told him that I didn't think that they were a good idea. But he gently insisted (the power of gentle insistence) and came back with a suggestion he had found online for an Agile risk register. This was a Kanban-style board with three columns. At the outset, a group of us got together: me (the project manager), the product owner, the product owner's boss and a few additional senior stakeholders. In a one-off workshop, we came up with a list of risks. We added them to the Kanban board and then scored them.

The idea of the score was that it was a combination of the likelihood of a risk transpiring and the "cost" to the project of fixing the fallout if it did. We expressed that in days of work for the team. For example, a full security breach might take the whole team a month to fix. So, we would give that risk a cost of 20 (working days for the team). Then we gave each risk a probability, expressed as a percentage. Collectively, we thought that there was a small but not insignificant chance of this happening, so we gave it a probability score of 5. The overall score for the risk was therefore 100 (20 x 5).

Once we'd done this for all the risks that we'd identified, we ended up with a three-column board. One column had high risks (>1,000 points). Another had medium risks (500–1,000 points), and the third had low risks. And we also had a total – the sum of scores on all the tickets.

Every two weeks, this same group of people took another look at the risk board. We talked through the risks from highest to lowest and amended the scores. We also added any new risks that anyone in the group identified.

In one sense, none of this discussion about specific risk scores mattered. But in another sense, it was a lifesaver for the project. Because what became obvious was that when we talked about the risks and changes to the risks' scores, pertinent news about what was going on in the broader organisation got discussed. Stuff that otherwise we might not have been officially "told", but that frequently was really pertinent to the project. Often these were things to do with budgets and deadlines. Sometimes we were being asked to make text changes that appeared strange to us, but that it turned out were being made to pacify distant but influential pressure groups.

Do a "show and tell" of working software.

Scrum is the most popular Agile framework. And "show and tell" is one of its names for a meeting where the team demonstrate working software to stakeholders. If you're doing user research, it's also an opportunity to talk about the findings that you got from that research. Invite people to the meeting from as broadly as you can across the organisation, and even include the public if that's possible. Demonstrate to yourselves and to anyone who will watch and listen. Talk through the findings of your user research. Walk through the designs that have been created as a result of that user research.

Demonstrate the working software that your team has been working over the last (typically) two weeks. The development team are often reluctant to do this, especially if what they're working on doesn't have an obvious UI. It's still important that they do it.

Just one comment might let you know something that's vital for your project. Yes, it could derail your project, but when would you rather know? Sooner, or later?

Embrace your deadlines.

Yeah, I said it. Embrace deadlines as opportunities to negotiate scope and set software free. But also, you'll find out more about the swamp. Some bosses see deadlines as opportunities to mess with you. They want to make you and your team work longer hours while giving them time off from thinking about what a product really needs to do. Some bosses just see the promise of delivering for a deadline as a way of getting their own boss off their back. But you don't have to see them like that. You can see them as powerful, if slightly clumsy, ways of exploring the stakeholder map. The map of the swamp. If you do try to release something, who jumps out and tells you that you can't? Who suddenly appears and tries to take credit? Who objects to you doing anything on the grounds of security, accessibility, secrecy or organisational embarrassment? Which non-functional requirement really is a deal-breaker that stops the software from going live? Software wants to be free. You should want it to be free. You should do your best to help it. I'll talk more about this in [Chapter ESCAPE]

Exploit the crisis.

What if your team releases software that nobody wants to use? What if your team releases software that nobody is allowed to use because it doesn't meet regulations? One way to think about this is that sooner or later all successful software becomes iterative and incremental. All successful software eventually incorporates feedback from its users. Or it dies.

This is the point of comparing a software development project to a development in a swamp. A software development project is, to some degree, building on unchartered ground – even though the people funding it will rarely talk about it in such terms. It's not clear what's hidden in the waters of the swamp. It might be untold riches. It might be a virulent disease. But whatever is found there, lots of people are going have an opinion about it. The people who already live there, the people who want to live there they have opinions. They have things that they value. The same with the people who will never live there but still have opinions and interests. This might be because they are in power, or because they want to preserve some aspect of the environment, or for some other reason entirely. Sometimes that reason would be difficult to really understand without research.

The message of this chapter is that the place pointed to by the dream of a project that we talked about in [Chapter STREAM] is a mysterious and complex one. And the way to give your project the best chance of thriving in this mysterious place is through research, investigation and frequent releases of working software. Talk to as many of the interested and potentially interested parties as you can. Understand how their interests and values mesh – or conflict. Find out as much as you can about the lie of the land. And one way to do this most effectively is to bring into the team "professional surveyors", i.e. user researchers who can talk to these parties. But having conversations with interested parties isn't just the responsibility of user researchers. It's also your responsibility as project manager. One way of thinking about project management is that you facilitate conversations. And one

way of improving the chances of a project succeeding is by enhancing the quality of the conversations that you have with the people who have interests and stakes in the project. The people who own, live in and care about the swamp.

In this chapter, I've mostly talked about the importance of exploring the swamp through user and stakeholder research. I've also touched on a complimentary way of exploring the swamp, and that's by building something there. Letting interested parties see, touch and even feel what it's like to use this new thing. And that's what we're going to consider next in [Chapter ESCAPE].

Chapter ESCAPE – Working software

Working software is magical. Users interact with working software in ways that could never be imagined from mocked-up designs or a list of specifications. Working software helps you discover value and create "pull" from users. Getting working software into the hands of users will give your project the best chance of success. But for many reasons, it's often a struggle.

"Sufficiently advanced technology is indistinguishable from magic"- Arthur C. Clarke

At the turn of the millennium, a bunch of men (they were all men) got together in a ski lodge in Utah. There they wrote the Agile manifesto. They had all been struggling with a difficult question: "How can we make project management of software development simpler and more effective?" They wanted something that was simple and lightweight that increased the chances of success. That's maybe why the manifesto itself isn't so big. It's only 68 words long. The Agile manifesto contains four key values. And working software is the second one.

"Working software over comprehensive documentation"

Right after:

"Individuals and interactions over processes and tools"

The primary message of the Agile manifesto is "people first". Good software comes out of conversations between people. Conversations between people? Well, that's certainly one description of the user and stakeholder research that I talk about in [Chapter SWAMP]. But right after that, the founding fathers of Agile talk about working software.

Why? Why do we think that might be? Those guys who got together to talk about "lightweight" software methodologies, what was their experience? What in their experience caused them to specify "working software" as one of their four values?

My guess is that they had all had experience of projects where the delivery of working software had been delayed or had never happened. They had worked on projects where months or years could go by before a specification was agreed and software development could start. Then they'd waited more months or years before any of the people who were supposed to use the software got to touch it.

For the first 40 or 50 years of software development, that's how people thought it should be done.

Software development was called software engineering, and it was thought to be an offshoot of other kinds of engineering. In other kinds of engineering, at least according to this theory, nothing is built before detailed plans have been finalised.

This is what makes the second principle of the Agile manifesto so revolutionary. It is saying the engineering equivalent of "have a go at building a bridge and see how you get on". And "as soon as you've got a prototype bridge, let people have a go at driving over it". What these guys are saying in this tiny statement is that software works differently; it isn't like building bridges. It has a different logic. It has different needs, different safety levels, a different potential path to success. Software is different.

The people who wrote the manifesto had experience of what happens if you don't push for working software. When the software is finally delivered using a traditional "Big Design Up Front" approach, there are typically nasty surprises. It isn't what the users want, or maybe it was what the users wanted several years ago when the specifications were written. But in the time since you started the project, the world has changed.

So, what's so important about working software?

One thing about working software is that it's magic. Like the quote that starts this chapter says. If there's anything that counts as "sufficiently advanced technology", working software is it. People respond to it as if it is magic. They respond to it instantly. They respond automatically and with emotion. When they press a button or click on a picture, they have an expectation of what will happen. Working software creates needs and expectations, and it evokes emotions: delight, joy, frustration and anger. This is very different from the way people react to a checklist of functions in a requirements document. That means that pretty much the only way to find out how users are going to respond is to put actual working software into their hands.

And when you do put software in their hands, users will find ways of using it that you didn't imagine and will value aspects of the software that you didn't expect. Remember the swamp? When you build things in the swamp, you will start to discover value; people will use what you've built in unexpected ways. You thought these were ornamental ponds. People are swimming in them. You thought these buildings you put up were just fishing huts. People are living in them all year round. When you dig a hole or a build a structure in a place that isn't well understood, you will find things that you didn't expect. Similarly, when you give people working software, people will react in ways that you didn't imagine, and they'll want it to do more than it currently does in directions that you didn't foresee. When people like something, they want more of it. And this leads to something called "pull".

Pull is the most powerful thing about putting working software in the hands of users. Right up to the point where users find something in your software that they like, that helps them achieve what they want or need to do, your project has been in "push" mode. The sole driver for the functionality in your software was the stakeholders who originated the project.

Now you have users, and they have things that they want. You are in "pull mode". This doesn't necessarily mean that you should give users everything that they want, but it does mean that the dynamic of the project has changed.

It means that your project has gone from being just someone's idea to something that people want, something that has value. When we make working software available early, we're giving it an opportunity to connect with its audience. We're trying to create "pull". We're being guided by the people who use it or who want to use it in the direction of delivering more and more value.

This is an extremely powerful reason for delivering working software early. But there are also others.

> *"You will never be alone if you always carry the ingredients and equipment to make a martini. Even if you think you are completely stranded on a desert island. Because the minute that you start to make the martini, someone will jump out from behind a tree and say: 'That's not how you make a martini.'" – old joke.*

Something like this is also true of developing working software. Ideally, you want your software to be in the environments where it will ultimately be used. That means running on the computers that it's going to be running on when it's live. You want it to be accessible to real users. You want them to be able to use their real data, on the real software. But the truth is that the nearer you get to doing that, the more likely it is that someone that you didn't even know existed will jump out from behind a tree. These people will tell you that "you're doing it wrong" and try to stop you getting working software into the hands of users.

This is why this chapter is called "Escape". Because in my head, I imagine the job of getting working software out into the world, on a working environment, using live data, to be like trying to escape from a prison camp. There's only one way to find out what all the traps are out there in no-man's land, and that's by trying to set them off.

Cocktails? Prison camps? Ok, I'm mixing my metaphors. Let's stick with the man behind the tree for a moment. Most of those things that the man who jumps out from behind a tree will bring up are called "non-functional requirements". These are requirements that the software needs to meet but that aren't strictly about the thing that it does. The software needs to be accessible to users with visual impairments. The software needs to be secure. The software needs to comply with a bunch of regulations specific to a particular industry.

One way to tackle all the rules and regulations that your software might contravene is to try to take care of them in the specification. But in my experience, it's very hard to find out exactly what you can and can't do without trying to do something. We can guess that the framers of the Agile manifesto had had similar experiences.

When you're trying to list requirements, especially non-functional requirements, without a piece of working software, you're only dealing with "known knowns" and also maybe "known unknowns". For example, we know that all the buttons on a web form should have what's called "alt text" so that buttons are readable by people with visual impairments. We know it shouldn't be possible to gain control of the website by dropping sneaky code into one of the text boxes. We don't know how many people might decide to submit this form all at the same time. But we know we don't know it – we know we have to guess.

So, by sitting down and trying to think of all the things that might stop us getting working software out into the wild, we might come up with a pretty good list of "known knowns" and "known unknowns". But to return to my escape analogy, when you try to get some working software as far as you can through the barbed wire to the outside world, you soon start to be surprised by "unknown unknowns".

Unknown unknowns? Like what?

- Most of the users of this system don't speak English as a first language. So, they use an agent to complete the form. The site isn't set up to deal with a third party filling in the form.

- All applications in your field are regulated by a body that you've just heard of for the first time.
- The senior stakeholders knew that this site would be regulated by a particular governing body but were trying to sneak it out without them knowing – which would never have happened.
- One part of the system gets the users so excited that they won't talk to you about anything else. They just want this new function to do a whole bunch of extra things.

Of course, it's hard to say what these things will be, because by nature they are a surprise. You knew there would be other tripwires out there in no-man's land, but you didn't know what they were. And for some of those things, there really was only one way to find out. Trying to put out working software is an important way of understanding the environment that the project lives in.

This is the counterintuitive thing I'm trying to tell you, which is so important that the founding fathers of Agile put it number two in their manifesto. You should try to get working software as near as you can to live. Why? Because if you do a man will jump out from behind a tree and tell you why you can't.

By actually getting your software into the hands of users, you will certainly hear from those who try to stop you. But you will also get the answers to a whole bunch of other questions:

- Are our team capable of writing this software?
- Do our team have access to the tools and resources that they need to write this software?
- Does the technology that we've decided to use work?
- Is the organisation that we're working with capable and willing to pay for the servers, people and set-up that we need to deploy this software to live?
- Does the organisation that we're working with have a workable, reasonable definition of "secure" that will actually allow them to release software to live?"
- And, perhaps most importantly, what is the value of this software? What does it mean to its users?

I worked on a project for an organisation that was still doing pretty much all of its business using paper documents. This was a huge organisation, and it processed a lot of paper. The project that I was working on looked at just one of these paper processes. The aim was to take it over and improve it using an electronic document storage system.

The project had a couple of good things going for it. Firstly, it was using an Agile approach. Secondly, the product owner, the person responsible for making decisions about what the product did, was a former clerk of the company. She knew all the other clerks and she knew their business very well.

To start with, the project had a tough time. We couldn't get servers; we had nowhere to deploy the software. The open-source document management system that we'd chosen wasn't as mature as we thought. We pushed on through some early design iterations. We dealt with a load of technical problems. Finally, we got to one "show and tell" where the team had "working software" that they could show.

It was an odd turning point in the project, because that first demo was so terrible. We'd put together a

demo: a view of a collection of documents, then a display of the document when its title was clicked. In that first demo, when we clicked on the document link, a window came up that said, “Do you want to veiw [sic] your document?” Yes, with that spelling mistake. And then when the user clicked “OK” an error message appeared.

The look on the product owner’s face! At this point we were about half a million pounds into a two million pounds project. And all she had to show was a misspelled dialog box that led to an error message. It was a hard time for her. It was extremely embarrassing for the team.

But two weeks later it was a slightly different story. Now there was a list of documents. Now when the document was clicked, the prompt to open the document had a button that was spelled correctly. And when it was clicked, the document was displayed!

The product owner seemed a little bit more relaxed.

Not too many “show and tells” after that, the product owner had a question: “Can I get this on a laptop so I can show it to the clerks?”

The short answer to that question was “no”, because all of this nearly half a million pounds worth of software was deployed only on developer laptops. We were in dispute with the client over who should pay for server space. It hadn’t been included in the original bid, so the client was arguing that this was implied, and that we as suppliers should pay.

But the product owner’s request to have a version that she could take around the country and show off was a powerful step in the right direction. If we could do it, it provided a good extra reason to negotiate with the people who were supposed to be giving us server space.

Once the software was on her laptop, the product owner went on the road to show it to her colleagues. The demo still wasn’t up to much. It still had bugs. We still had some spelling problems! But the product owner’s demonstration to her own former workmates went very well. She could shrug off any user interface issues or problems. She could show the other clerks first sight of something that could make their lives a lot easier.

She came back with a list of problems she’d found while using the demo. She also had a list of suggestions for features that had come from the clerks. But she also came back with one important question: “When will it be ready?”

From that point on, the project changed. It wasn’t about delivering on a list of functions. It was about rolling out to the clerks all across the country the tiny bit of working software that the product owner had shown them. Suddenly the path of what we needed to do rolled out before us, with the people we were delivering it to cheering us on.

There was another very interesting thing about getting working software in front of the people who might use it. By doing this we both asked and then started to answer the two questions that I’ve already talked about: “Can we do this?” and “Who is going to jump out and stop us if we do?”

To the first question, the answer at first was “no”. We didn’t have any server space where we could deploy a live service. This was blocked because of the dispute.

But suddenly the people who from our point of view were trying to negotiate for free stuff were in the way of their own colleagues. Now they weren’t helping the company by being tough on costs. Now they were stopping people who worked for their own organisation from getting something that could make their lives easier. The servers appeared, and then money for people to support the servers

appeared. A real rollout of the software started to happen.

The answer to the second question, “If we try to do this, is anybody going to jump out from behind a tree and stop me?” was “yes”. In fact, two people jumped out: an accessibility guy and a security guy. The accessibility guy claimed that there was no way the software could be released until it met an extra set of requirements. Then true to form, the client claimed that we should have known about these requirements right from the start and so we should pay for them.

And by the way, the client was (partly) right. We should have built in accessibility right from the start. It takes (almost) no extra effort in coding to make sure that a website is accessible. And it actually makes the site much easier for all kinds of people to use. People who you might not think of as disabled. Do it. It makes sense. If in doubt, pay a blind guy to look at your site. Does that sound insensitive? I really don’t think it is, but maybe that’s because I know several blind guys who dearly wish that people would ask a blind guy to look at their site.

Then out from behind another tree jumped the security guy. He said that the project should never be allowed to go live until we could prove that it was secure. Just to leave us in no doubt about his effectiveness as a blocker, he also refused to tell us what it was we needed to change so that it would be secure. And of course, the cost of any changes we made would have to be borne by us rather than the client.

As people who jump out from behind trees trying to block progress go, these two looked pretty effective. Both were telling us we couldn’t release until we did what they said; both were telling us that we had to pay to do what they said. The security guy was being even more effective at blocking us because he was also not telling us what it was that we had to do.

But neither of these guys was a match for the clerks. There were a lot of clerks. They’d seen that this software would make their lives much easier. We kept gradually improving the accessibility, but we went live with what we had. We submitted the software to outside security testing. We addressed the most serious issues that arose, but we went live with some other, less serious, issues still being looked at.

Putting software that just about worked in front of real users completely changed the project. Why? Because it started to build the product value stream that I talked about in [chapter STREAMS]. Putting real working software in front of real users created pull. And that completely changed the dynamic (and control) of the project.

And this is how you deliver the impossible. This is how you make a project that seems impossible, possible. You build something, you put it in the hands of users until at some point you start to get pull. From that point on, the dynamic of the project has completely changed. Your project is starting to shift from being a dream to being a reality.

How do you do this on your project? Every project is a little bit different. I’ll give you a month to do some set-up. If after not much more than a month, you and your team don’t have *something* that works, you’ve waited too long. By the end of that time, you need a tiny piece of working software. A small bit of software that starts to do the thing that it’s supposed to be doing. And you need to be able to show this software to the people who will ultimately use it, maybe even let them use it themselves.

And once you’ve got this tiny little thing, you should be looking to promote its status in some way. By that, I mean get the software being used by more users. Or set the software up so that it’s using live, or like-live data. Or make its deployment more automatic so it can run on more than one machine and

in more than one environment.

What's important with working software, at least at the beginning, is not so much the content of what you've got, but that you've got something and it's moving in all the right directions. Sure, it needs to be progressing in terms of what it can do. But it also needs to be moving from developer environments, to test environments and onto the final live environments where it will run. And it needs to be moving from being tested by the team, to being tested by "friendly" users, to being used by complete strangers. If it's a business application, it needs to move from using dummy data, to using data that looks like live data, to using real data.

I know nothing about rock climbing. But this seems to me to be a little bit like if you're climbing a huge rock face. You climb a bit, then you put in one of those things that holds the rope to the rock. Then you tug on it to make sure it's firm and would hold you if you fell, and then you can climb a bit more.

Yes, delivering software is a bit like that. It's a bit like climbing a rock face (and it's like escaping a prison camp, and there's a guy jumping out from behind a tree with a martini; I know, I know, too many metaphors). The odd thing is how many people think that you can get to the top of the cliff without a rope. Even odder are the people who somehow imagine that you can get to the top of the cliff in one single bound.

Gradually developing working software in all of those directions – environment, security, audience, reality of data – is the careful and effective way to deliver something. Especially if that something seems impossible. And if the thing that you're doing is in any way useful or interesting to the people who will use it, at some point you will start to get pull. Obviously, you'd like this sooner rather than later. But at some point, you will start to get demand for the software. And then you will start to get demand for more functionality from the software. Not from the people who sold the idea, or the people who funded it – that's push – but from the people who are likely to use it – magical pull.

And of course, gradually developing software in these directions will also result in people jumping out from behind trees. They will ask you about extra rules and regulations that you need to follow. Sometimes they'll tell you that you can't proceed until you're certified. How do you get certified? That, they might not know.

One way of summarising these ideas around the importance of working software is the "six Ps: six ideas beginning with the letter P (OK, one of them doesn't, but we'll get to that in a bit.

Pull

The first one is pull. By releasing working software, by putting working software in the hands of users, you create pull from those users.

Pivot

Moving from being pushed into doing things by internal stakeholders who value the idea to being pulled into doing things by users who value the interactions of the software is a fundamental shift. It's a pivot.

It's a power shift. As I said, the dynamic completely changes.

Prison camp

Most software that gets developed is in prison. It's inside an organisation and it can't get out. And it

needs to get out. It needs to be with its users. That's the only way that it can be valuable. So how can you set it free?

Your software is in prison; it's your job to find the fences, mines and tripwires and get it out.

Promotion

A lot of the value from trying to break software out of prison comes when we promote it. And it comes not only from succeeding, but also from trying and finding the things that are stopping you. When we talk about putting software into the hands of users, we're not talking about necessarily doing that in one single step.

We're talking about trying to push software as far as we can towards the outside world because we know there's value in what we'll find out when we do, and mounting risk if we don't.

Surprise

This is what we're looking for. And yes, I know, it doesn't begin with a P. When you try to push software as far as you can out into the real world, you will be surprised – and that is because software is magic. This goes back to the idea that "sufficiently advanced technology is indistinguishable from magic." And that's absolutely the case with software. People will react to your software in ways that you never imagined. They will do things with it that you didn't expect them to do.

They will react to software in a way that is completely different from the way they would react if they were reading a requirements specification.

And when you do this, when you put software in the hands of users, they will do amazing things.

You will find yourself thinking: What was that that she just did? Why did she do that? They'll put things together in ways that you didn't expect. Users will totally ignore or be unimpressed by parts of the software that your team struggled with and worked hard to get right. But at the same time, they'll get enthusiastic about parts of the software that you didn't think were a big deal.

Part of the point of pushing software as far as you can out into a live environment is to be surprised.

But of course, some of those surprises won't necessarily be pleasant. It's like the prison camp: as you try to get the software out past the fences, you'll discover other obstacles. Often these obstacles will take the form of people who suddenly appear and tell you that you need to comply with a certain regulation.

Other people will tell you that you must pass an assessment of some sort, or that you must answer a questionnaire.

All those kinds of things have to happen, and that is part of why you're trying to get the software as near live as you possibly can. By trying to promote software, you're aiming to be surprised. But that doesn't mean that the experience won't be unnerving. And it doesn't mean that a lot of people connected with the project won't get upset when the surprises arrive.

And this brings us to our next P.

P**d off** (or "perturbed" for the faint-hearted?)

Trying to promote software, trying to get it out into the real world will annoy people.

And this is an important thing to understand about getting working software into the world. Very few people will like you if you do this, and it is something that you must do.

Who will get annoyed? Here's just a short list. It is by no means exhaustive.

Let's start with the people who had the original idea. And the people who funded the original idea. They'll feel very uncomfortable, to put it mildly. Because it might start to seem obvious that this idea is very expensive to realise. Why might that happen? Because all software is expensive to realise. Maybe as you start to push working software nearer to real users or to push the working software nearer to using live data it becomes obvious that there are some superficial problems with it (like spelling mistakes or logos that unwittingly turn out to look stupid or rude). Or maybe there are some stark fundamental problems. Once you start to get this software near a live environment, maybe it becomes obvious that this project is a dumb idea, or that the technology will never perform in the way imagined.

Maybe all these problems can be overcome. The idea is still a great idea. But the possibility – or uncertainty – of it being revealed as a dumb idea will still be threatening. Maybe the senior stakeholders suddenly realise that having this software go live will mean work for them, or work for a team that they haven't yet hired.

Senior stakeholders have a lot invested in the project: they are linked into the hierarchy of the organisation or connected to a group of investors in ways that means they'll be exposed when the software finally goes live. So even though the only way to be successful is to release the software, trying to release the software by going through various steps of promotion will still upset a lot of the people who claim they want the software.

Trying to release software will also upset people on your team.

Who on your team? Well, it could be just about everybody. User researchers: they'll want lots more user research before anything is released. Designers: they'll probably want an end-to-end flow and a coherent look and feel for the whole thing before anything is released.

Developers will tell you that there's no point just releasing a bit of this thing. You might as well wait until the whole thing is ready. Testers will tell you that they haven't tested it enough. All those keepers of risk, regulations, accessibility, compliance and security. All those people, they'll be annoyed if you try to release things.

So, this is an important thing to understand about working software. The business of trying to get it into live, which is absolutely the thing that you should be doing, is going to annoy a lot of people. In fact, the only people who might not be annoyed with you trying to push working software into the hands of users are users themselves.

If you push some new software that makes their lives easier, if you push some new software that entertains them, if you push some new software that makes it easier for them to find a restaurant, find love, find the perfect gift for their family or find a job, maybe you'll make them happy.

But what if you don't make them happy, what if they're highly annoyed? Or what if they're utterly indifferent to the software that you put in front of them? What if they laugh, snigger, shout, scream, cry, moan or yawn?

There's still an important reason why it's worth pushing software as far as you can, as near the outer fence of reality as possible.

The reason that it's worth doing that is because… Well, the reason is a rhetorical question. "When do you want to find out that nobody is interested in this piece of software? When do you want to find out

that they hate it? Sooner? Or later?"

If you know the reaction of the users when you are 10 percent of the way through the project, you've still got a chance to move and change and try something else. If you only find this out when you are 110 percent of the way through, when you've spent all the money, it's much harder to do anything about the things that you've discovered.

So, even though you're going to annoy a lot of people by pushing working software, the only chance that you have of pleasing the users is by putting software in their hands.

So, that's the six peas.

Pull, pivot, prison camp, promotion, surprise and, erm, perturbing everyone.

And just one final thing. Actually, it's two final things. Two common objections to the idea of promoting software as far as we possibly can.

"Reputational damage". Somebody is always going to try to stop you promoting software by claiming reputational damage. But remember what we talked about with promotion. Remember that's different from simple publication. There are different levels of promotion.

By recommending releasing working software, I'm not saying "show this thing to everybody all at once". Certainly not as a first move. Rather, I'm saying promote your software. Make it available to a wider group.

Do you have a group of friendly users that you could show this to? Is there a group of people you trust? How do you frame this? Can you say this is an early alpha, we're just getting feedback for this? How can user researchers use this working software in their research?

Then there's the second thing: legal requirements. I don't want you to tell me that something is a legal requirement as a way of stopping software getting promoted and then actually escaping unless you're surrounded by a bench of judges.

I absolutely don't want you to tell me that something is a legal requirement if you haven't talked to anybody who knows about the law, or you're not going to talk to anybody who knows about the law and you're just saying it to stop the software from being released.

And again, regarding promotion: yes, some projects can't just put their software out on the doorstep and let anybody pick it up and use it. There are all sorts of security concerns. There are all sorts of regulatory concerns. But how far can you push things? So you can be surprised? So you can connect with pull? So you can pivot from pushing things out into the world to having them pulled into the world by the genuine needs of your users.

Over the last three chapters I've explained that projects exist in the difficult space between the idea value stream and the still-to-be-built product value stream. I've explained the value of seeing this awkward space as a swamp that needs to be explored. A complex space with many geographical features and patterns of interest and ownership that can be explored using user research and other techniques. Now, in this chapter, I've talked about another way of exploring this complex space: by trying to help working software escape it.

Of course, all this exploration of the swamp with user research and working software assumes that you as the project manager will do something useful with what you find.

In the next chapter, I'll discuss what happens if you don't do that.