Delivering the impossible

[N] ways of looking at software development project management

Chapter INTRODUCTION – Introduction

Many projects appear impossible. Most of them aren't. If you look at them in the right way and are doing the right things to manage them, most projects can deliver value to their owners and their users. Looking at projects in the right way and managing them in the right way will allow you to see the projects that genuinely are impossible and get out of them as soon as you can.

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 on the front cover, in its title?

Here are some reasons why you might have done.

How about this? You are working on a project. You're a member of the project team or you're the manager, and you feel stuck. It seems like the project you've been asked to deliver is impossible.

You're not quite sure when this happened. But at some point, someone told someone that the project would be finished by a certain date (probably Christmas). Instinctively, 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 needs to be finished by a certain date. But when you, or your team try to get a definitive answer of what finished would mean, you don't get the answers you need.

Even if you do know what should be delivered, you're certain that your team can't deliver it on time.

In every direction, there seem to be problems that you can't solve. You have somehow found this book. You thought it might be worth a read.

Yes, maybe it's that. You're the project manager. Well good, because this books is specifically targeted towards you. Obviously this book may be of interest to other people but if you're a project manager, managing a software development project, this books is especially for you.

You have picked up this book. That probably means that you're starting to get this uncomfortable feeling that the project isn't going well. And for the sake of the company, well, that and the sake of your career, you need this project to go well. You realise, you don't know what you can do to help the project succeed.

You wonder if there any useful tips in this book, for you, or for your team. Is there anything that *you* could do to help the project do better?

Maybe you're working on a project, but you aren't any kind of boss. Maybe you're one of the people who does the work. Yes, yes, you know you're going to get paid no matter whether the project is a success or not. But still, you'd rather work on projects that are a success.

So, let's get right to it. How do you deliver a project that is impossible? The quick answer is that you look at it from a different point of view, you find a way of seeing it, that makes it so that it isn't impossible.

Of course, that's another reason why you might have picked up this book. Even though you think your project is impossible, you haven't given up.

Well, part of the reason for not giving up is that you know that a lot of things that seem impossible do actually get fixed, given the right skills and expertise. You know this from other parts of your life. That it's not a good idea to give up immediately when things get difficult. Often things do get better if you persist. If you look at them from a different angle, if you see them through a different lens, suddenly things that seemed impossibly start to click into place.

The software pioneer Alan Kay said, "Point of view is worth 80 IQ points." What I take this to mean is that if you can only figure out the right way to look at something, you can do much cleverer things. What I'm trying to show in this book are different ways of looking at the problem of project management. The idea is that if we can only find the right point of view, we can be much cleverer about what we do.

So, you might expect a book called "Delivering the Impossible" to have a bunch of different ways of looking at problem projects. You might even expect some useful ways of looking at the ones that seem totally impossible.

What you would like and expect from a book titled "Delivering the Impossible" is some kind of guide. You'd expect some kind of handbook for dealing with projects that seem impossible.

You might expect to find methods for spotting issues that make a project "impossible". You might then expect the book to go on to talk about what to do in these situations. How to make a project reasonable and deliverable and give it the best chance of success.

What if a project is genuinely impossible? A book called "delivering the impossible" should tell you how to spot it. And it should give advice on what to do in such circumstances. It should tell you how to keep yourself and others sane and safe. How to get away from a lost cause and move on to do well on other projects. How to ride into the sunset to fight another day.

Well, this is that book. And it tries to meet all these expectations. This is especially that book if what you do involves the development of software.

If you read this book, you'll learn how to improve your management of impossible-seeming situations. Things might look bad. But there are often some quite simple things you can do to turn projects around and help them do well.

Why This?

What might be a shortened version of why you picked up this book? Because you're involved in a project and you don't know what you're doing. On the whole, in your life and your work, you would really like to feel that you *do* know what you're doing.

My experience is that knowing what you're doing isn't that different from having no idea what you're doing. You're doing very similar, simple things, but being guided by the right way of looking at things. And that's why this book is subtitled 6 metaphors for successful project management. Metaphors are ideas, ways of looking at things. Doing the right things is literally about having the right ideas, seeing things in the right way

The reason for describing these different points of view is to make good on Alan Kay's claim. The aim is to make you much smarter, to make you feel like you "know what you're doing" when it comes to managing projects.

And in this way, this isn't like most books that offer to improve your success at project management. Most books like that suggest a particular Agile method. For example, Scrum [Ref], or Extreme Programming [Ref]. Other books talk about how to coach teams in the use of these methods. Other books talk about a particular part of the Agile process. It could be writing "user stories" or running retrospectives. And there are books that focus on the technical aspects of the process. You can read about test driven development and wonder how often it is "honoured more in the breach that the observance [Ref]". You can read about Dev Ops [Ref] and continuous integration [Ref].

I'm not knocking these books. Some are brilliant and I've referenced some of the ones that I like best in the bibliography. But I don't think they're enough.

Following an Agile method like Scrum is a good way of improving your chances of project success. It's what I almost always do if I'm in charge of delivering a software development project.

And Scrum works best when it's combined with XP engineering practices. For software development, it's not really going to work at all unless it's combined with some form of automated test coverage. And then on top of that you probably also need continuous integration and continuous delivery.

And if you haven't done this before as a team, you are going to need some outside help. It is really hard just to learn it from books. Bringing in an Agile coach, somebody who has done it before, will seriously improve your chances of success.

But unfortunately, this isn't enough either. You need to do more than this if your project has any realistic chance of doing well. You need to develop the ability to see things from different points of view. And, having developed that ability, you need to be able to act, and help and inspire others to act on what you see.

OK, I think it's time for a real example of what I'm talking about. In the next chapter, I talk about pirate ships and "agreed activity". I talk about how you need to notice when you and your team are agreeing to ignore the big issues. You need to spot when you are all repeatedly doing something simple and safe, because in the end this can be very dangerous.

I talk about how you need to stop your team mopping the decks. Rather you need to help them to look at the horizon and face the approaching pirate ship.

Why is it important that you tackle "pirate ship" [Chapter CHREF)] issues as soon as you see them? Because if you don't, you and your team will end up [CHREF] in trench warfare. And once you're there, it's much harder, although still not impossible, to help your project do well.

Why do you need to show progress and call out problems as soon as you can in a project, even in the face of hostility from the people who are paying you to do it? Why do you need to put working software in the hands of people who might get some value from using it as soon as you can?

I explain this using the Daoist idea of flowers and fruit. Flowers appeal to us straight away. But they aren't sustaining. Fruit is difficult to grow and even more difficult to make tasty. But it feeds people.

Maybe you're thinking: "My software development project is in trouble. It's already cost millions of pounds. I can't find any way out of this that doesn't cost millions more. The last thing that I need is some dreamy guff about pirate ships and trenches, fruit and flowers."

But this book tries to provide something solid, nutritious, and sustaining. It sets out ways of seeing projects, those points of view that Alan Kay talks about. The contention of this book is that this is what you need when your project looks impossible - different points of view. It offers the genuine possibility of knowing what you're doing. And once you know what you're doing, once you have the right way of seeing, the right idea, a lot of things that previously seemed impossible can seem a lot less so.

Why now?

OK, you might think. OK, I get it, seeing things in a different way is useful. But this 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 what I want to show you in this book is that now is the time. Well, now is way, way too late. But better late than never.

We've needed better ways of thinking about software projects since the 1940s. Humans have been writing software for at least 75 years and we're still not very good at it. That is because it is very hard.

I remember listening to an explanation of a "Waterfall" approach to project management. Waterfall is the traditional approach to project management. It was regarded as the absolute best way of managing projects until very recently. There was just one problem with it. It was hardly ever a success.

My first job was in the mid-nineteen nineties. I remember thinking "this will never work." And I soon saw that it didn't.

In my first two years as a software developer I worked on a software project for a big oil company. We managed it using the waterfall approach. This was the way nearly everybody managed software development at the time. And it's called the waterfall approach, because work happens in stages, like the steps of a waterfall.

Stage one: all the requirements for the project were carefully written down. So, at the end of this stage we had a huge requirements document. I didn't join the project until this had already happened, and it had taken a year.

There were two problems with this stage. Firstly, how did we know we were finished? We didn't have any way of testing when we had *all* of the requirements. Secondly, in the time that it took us to write down the requirements, the world changed. What changed? All sorts of things. It was an oil company, then, as now, what goes onin the world of politics, wars and natural disasters affected its priorities. And all of these events are difficult, if not impossible to predict. Added to that, inside big organisations big corporations, there are also political manoeuvrings. There is a hierarchy and the people in that hierarchy are continuously struggling for influence and position. So, the people in that hierarchy change.

We were trying to write software that would work in this sea of changing. But as we did that, guess what else was changing? The software! Not the software that we'd written, the software that we were trying to write it on. The project was developed at the point when Windows 95 started to be available. And this caused a different set of problems. Should we use the new technology? If we did, we had to deal with all of the teething troubles that came with working with something that was brand new. Should we use old, green screen, more reliable technology? If we did, we had to deal with clients who were uncomfortable about paying so much for software that looked so old fashioned.

All this change was going on in the background, while we tried our best to put together a high-level design so that we could move to stage two.

Stage two: we turned these high-level requirements into a detailed design. In one of the projects this stage took a year and £6 million pounds. And it only

stopped, not when there was any feeling that the design was ready, but when the budget for design ran out.

Stage three: after the first two stages, which took years, we started to write the software. And that? That took another year

Stages four and five. Then came separate stages of internal and then external "factory acceptance" testing by the customer. Then and only then did the software go live.

Unfortunately, on this project, (and next one that I worked on) then went through an extra stage between testing with users and launch. What was that extra stage? It was litigation.

When the customers finally got to see the final product of what they'd asked for it was four and a half years since they'd made their original request. And so it substantial parts of it weren't what they wanted. Not to mention that the software only ran on old-fashioned "Green Screen" terminals, and not on the new, modern windows 95 GUI.

In the long years since they'd specified the project, it's the world had changed. But had specification document captured exactly what the users wanted, even all that time ago. It's almost impossible that it did.

But now, nearly five years had passed. The customer had had those long years to think more about what they wanted. What the users had asked for had gone through several stages of work. The systems designers and developers had done their bit. The the internal testers took the software through its paces and then finally testers working for the customer. By the time those initial specifications had got to the end of that pipe, it's not surprising that this wasn't exactly what the customer wanted.

But there was one intriguing thing about this whole sorry, expensive process. Something that I only fully appreciated about that first project that I worked on, much, much, later. When the customers finally saw the software, decided it wasn't what they wanted and threatened to sue, something interesting happened. The management of the project called in a guy whose only job was to rescue projects that got themselves into this state.

His name was Terry. Terry would sit down with the customers and find out which bit of the software was most important to them. Which bit did they *really* need most? He would then negotiate a bit of room for manoeuvre for our company. A bit of time and a bit of money so that the software that was most valuable to them could get built. And then, when that was done, he would repeat the process. Bit by bit, the software would start to resemble what the client really wanted. At some point it would become valuable enough to them that the threat of a lawsuit wouldn't get mentioned any more.

And what Terry was doing, well it sounds like iterative, Agile software development.

Yes, this book is going to advocate an iterative, incremental approach to software development. But obviously, it isn't the first book to do that.

We could and should have been doing this right from the beginning. And in the most extreme circumstances after a lot of time and expense, under threat of legal action, we kind of were.

Seventy-five years ago, it would have been great for us to have really got the hang of delivering software projects. Still, it's better late than never. Being good at writing software has never been a more important competitive advantage. For organisations, teams, and even individuals, it really helps to be good at writing software. Being good at developing software that customers want, that is genuinely valuable to them, in a timely manner, has never been more valuable. Being good at software development, has become a way of, literally, conquering the world.

Am I overstating things a bit? I don't think I am. As Marc Andreessen pointed out 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. Some companies are good at developing software – Amazon, Facebook, Google, Alibaba. And some of those companies are now more powerful than some governments.

But ironically, the craft of managing software delivery doesn't seem to be so susceptible to being eaten by software. It remains a resolutely human pursuit.

Why me?

Let's say that you believe me. You agree that now is a good time to get very good at software development. You realise that that also includes managing software development. Why should you listen to me?

Well, I've been working in the software development industry since 1994. First as a developer, writing software for oil companies, the military and then a new-fangled thing called the internet. Then I worked as a researcher, first for Xerox, then for Cambridge University.

Working for Xerox gave me my first experience of project management. I think they gave me the project management job because they'd seen how well I could write code. I managed a couple of guys from former East Germany who were writing code for the first "smart" phone – the Nokia 9000. These guys were some of the only people prepared to put up with how awful it was to write for this phone. They'd learned to program on Commodore 64s smuggled over the wall into the DDR.

This was a really dumb phone. It had the same operating system as some sewing machines and it was very difficult to write anything for it. But working with these two guys, we managed to get a working prototype of our document access application. And at one point during the madness of the first internet bubble,

rumours of our product were said to have put tens of dollars on the Xerox share price. Then the bubble burst.

The Xerox lab closed and I found myself working for Cambridge University. This gave me my first real experience of using iterative, user-centred design. We designed a system to help school kids put together discursive arguments. The European Union funding was for an interface that was tangible. That means an interface that you could touch, pick up. This project taught me a lot.

It taught me that ideas evolve. We started out with brightly coloured boxes with topical ideas on them "People don't like graffiti." We ended up with a fancy (for its time) hybrid physical and electronic interface. It used rfid-tagged cards with statements like "People don't like graffiti," to search the internet. When you put a plastic card that had an rfid tag hidden in it on a special tile that could read the tags, a website associated with the statement appeared on a screen. This might have been the only piece of software that I've ever written that got "oohs" and "ahs" when we demonstrated it.

Taking a user-centred design approach means believing that whatever you're building needs to fit in a particular situation. The situation that we were designing for was English state schools. At the time they were closely regulated. Any activity that we were going to get the kids to do with our fancy new interface would have to fit in a lesson plan. But because this was a research project, we were also trying to please our other research partners and in the end, our funders. I didn't think of it then like this, but that project was definitely a swamp, as we discuss in Chapter [N].

Look this is all very well, but it's starting to sound a bit "I was born at an early age."

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. The projects that weren't successful were either "trench warfare" projects when I arrived (e'll talk about those kinds of projects in chapter [N]) or they were projects that turned out to be things that nobody wanted. We'll talk about how to avoid those in chapters [N] and [N]. Or people wanted them, but nobody could make them pay, again - chapters [N] and [N]. For example, it turns out that users are quite keen on personal loan websites. It also turns out that most of the people who want a personal loan are exactly the kind of people to whom you probably shouldn't lend money. That was a project that would have greatly benefitted from user research up front. People need to keep re-learning that user research is a good idea.

Some of these projects that 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. They helped solicitors defend their clients in court run more smoothly.

And I really believe that some of those projects wouldn't have done well, or

might not have done so well, if I hadn't been involved. That's about as much as you can ever say if you're a project manager.

And I learned from all of them. By working on so many projects, good and bad, my thinking about what project management was and how to do it well, changed.

One of the things that I learned is that all projects are different. You might think that was obvious, but it's understanding this that probably made the Japanese (for a while at least) the most successful car making nation in the world. And it's forgetting this that allows some boss or other in every project that I've ever worked on decide that it's a good idea to try to compare the "productivity" of different teams. Now's not the time to talk too much about this, but really. This isn't that obvious and it is very important and we'll come back to it in [Chapter N].

Another thing that I learned is that there's a lot more to learn.

And part of the reason why I've helped these projects is that I try to think about them in the ways I'm going to talk about in this book.

Why you?

What might happen if you try to deliver a project and you haven't read this book? Or, you aren't aware of the ideas in this book? Well, there's a good chance that you won't spot the pirate ship [Chapter N] until it's too late. If that happens, there's a very good chance that you'll end up in a trench warfare project [Chapter N]. It might take you far too long to realise that there's nothing you can do about it, and that you need to leave.

Let's say you and your team are really good, or really lucky or both. Team members can admit to themselves that there's a pirate ship coming. They alert others who need to know of the looming problem. They tackle the problems that they need to tackle. Even so, if you, or they, haven't read this book you won't know about value streams, flowers and fruit [Chapter STREAMS]. And not knowing about that, there's a good chance that they won't push in the right direction. They might be tempted to delay putting working software in the hands of users.

If you haven't read about the swamp [Chapter SWAMP], there's a good chance that you won't have hired the best user research team you can find. You might even be telling yourself, or agreeing with others, that you don't really need any user research.

So, you won't have a detailed map of the user and stakeholder ecosystem. That means that even if you tackled the pirate ship the minute you saw it, there might still be trouble. Even if you pushed working software as far as you could along the right dimensions, your project might still not go well. Without a map of the stakeholder ecosystem to influence decisions, the chances are that your project

won't do as well as it could. There's less chance that you'll have moved from "flowers" to "fruit" and you'll have no idea what that means.

Finally, if you don't read this book, there's a good chance you're going to feel bad, even when you're doing the right things. In fact, if you don't read this book, there's a chance you're going to feel good, even when you're doing the wrong things.

Why? because if you don't read this book, you probably won't know about how wrong and damaging the metaphor of "Project management as keeping a promise" (Chapter [N] really is and what you can do to escape its evil clutches.

So, by now, we might as well admit to each other, you are reading this book. And if you carry on, very soon, you're going to know about agreed activity because we talk about it in [Chapter SHIP]. This means that in the early stages of the project, as you and your team start to encounter problems, you would try to "televise" those problems. You would make them obvious to everyone who needs to know. Once we've talked about agreed activity and what happens if you don't try to disrupt it, we'll talk about the swamp [Chapter SWAMP]. When you start to understand the implications of the swamp, you will be looking to involve user researchers in your team, and you'll be looking to release working software as soon as you possibly can. You and your team would have been trying to "promote" working software, to get it as near as it could possibly get to the hands of real users.

And all that you've learned so far, agreed activity, the pirate ship, the swamp, the need for user research and working software all of this is going to prepare you for understanding the different between flowers and fruit.

In chapter [N] you would read about the pirate ship and the dangers of avoiding it. That means when you do come across a big problem that the team is ignoring, there's a good chance you would tackle it. By tackling problems early, there's a much better chance that you'll avoid the pain and misery of trench warfare.When you've read this book, you'll know, because you've read all about commitment and consistency in [Chapter COMMITMENT] about the dodgy metaphor of project management as the keeping of promises, why it makes you feel so back, even if you never promised the things that it's claimed you did.

When you've read this book do you know what? Not all, but more of the projects that seemed impossible might turn out to be possible. You might just get lucky. Yes, it still needs a lot of luck. You might manage to put working software in the hands of users. And when you and your team do that guided by user researcher and feedback from frequent releases for work software ,you'll have found something that is not only valuable to the users, but also creates value for the organisation. Your project will be a success.

Chapter SHIP - Ignoring the Pirate Ship - Agreed activity

The main danger to any project is the refusal of the people working on it to recognise the risks. If the risks are recognised and acknowledged, the team can do something about the ones that can be addressed and figure out how to work around the ones that can't.

One of the great things about living in London is that you can sign up to all sorts of classes. Over the past ten years, I've signed up for improvisation classes over and over again. Why? Because my experience is that every time that I go to an improvisation class, I learn something new. I have new experiences, and these often, in some way or other, turn out to be useful in real life.

One of the key ideas in improvisation is that you should avoid what's called "blocking". For example, an improvised scene starts with someone knocking on a door. The scene might go something like this.

Imagine that there are a group of performers on stage for an improvised scene. Often to start a scene, the actors will ask for a suggestion from the audience. Someone in the audience suggests "The deck of a tall sailing ship."

Here's what might happen.

Kevin: [Putting mimed fake telescope to their eye] Look over there! On the horizon? There's a ship, and a flag! Is that a skull and cross bones?

Now the other performers have a choice to make. One obvious choice here is to "yes and" the Pirate ship.

Janice: Oh my God the Pirates are coming. Haul up the sails, let's try to outrun them.

Leena: Oh my God the Pirates are coming. The cannons! Load the cannons!

Mei: Oh my God the Pirates are coming. Quick! Hide in the lifeboats!

All these are good selections. Of course, the audience hope that hiding in the lifeboats won't work but will result in lots of comedy.

Sometimes in this kind of situation, someone makes a different kind of suggestion. Something like this:

Nadia: Hey! Let's wash the decks!

Oliver, Pam, Quentin, Rana and Shug: Yes! Let's!

For some of the performers on the stage, this can seem like an attractive thing to do. But of course, for the audience, it's a very bad idea. Once somebody mentions the pirate ship, the audience wants to see it arrive. The audience wants

to see what a chase between this ship and a pirate ship looks like. They want to see if this crew can load a cannon, point it in the right direction and fire it. They want to see the pirate king and see what he'll do to the crew members who are hiding in the lifeboats.

They don't want to see everyone on the stage miming mopping the floor.

Improvisers who don't have a lot of experience are really tempted to just scrub the decks. It is much more attractive than dealing with the implications of the pirate ship.

Why? Because no one wants to be changed. That was the realisation of the theatrical innovator Keith Johnstone who developed his own style of improvised theatre in the 1950s. He also wrote several books: "Impro" and "Impro for storytellers." In his books, Johnstone talks about people who are new to improvisation. He describes them wanting to do safe, repetitive activities. Washing the decks is an example of this. He refers to this as "agreed activity."

He explains that putting people on a stage in front of an audience makes them scared. Scared people try to make themselves safe, even if what they're doing is making the group as a whole less safe. Even if what they do bores the audience.

Part of the skill of being a good improviser is knowing how to avoid this need to stay safe. Instead, good improvisers have the courage to move the story forward. When they hear the suggestion, good improvisers accept that there's a pirate ship. They make the story interesting and move it forward by dealing with what that means.

What's fascinating is that all novice improvisers seem to know that mopping the decks is the "right" thing to do. There isn't a long discussion. It happens in less than a second. Even though, from the point of view of the audience, it's exactly the wrong thing to do. What this shows is how good we are at shying away from things that might be dangerous. Especially, we tend to avoid anything that will make us change our behaviour or our thinking.

Johnstone has one suggestion to help the improvisers avoid this "agreed activity". This is to have a director who is watching the improvisers and can speak out and direct them during the show. The director spots which suggestions will move the story forward. She also sees which suggestions are invitations to "agreed activity" and tells the actors to avoid them.

Let's say a director is watching a scene where there'd been a suggestion of washing the decks. She might allow the crew ten seconds of deck washing without advancing the action. But then she might shout "The deck's clean! The pirate ship is getting nearer! Deal with the Pirate ship!"

So if everyone's first instinct is to mops the decks, rather than to do the "right" thing and tackle the pirate ship, how can anyone who is learning improv improve? It easy to shout "deal with the pirate ship" if you're offstage. If you're on stage, improvisers call this the "red mist descending", the lizard part of your brain

takes over. You're scared, you're going to do the thing that makes you safe and that's boring and so ultimately, not safe.

So how do you make sure that then next time that you're in this situation, you don't

Often improv classes take the form of a set of games, and improv practice for mature improvisers, take the form of games. The purpose of the games isn't only to have fun (although of course that's important). Good improv games are also drills, that focus on improving one particular aspect of the improviser's abilities.

There is one game that helps get improvisers out of the habit of agreed activity. It's called "new choice."

It can be used as guard rails, or training wheels, to help improvisers avoid blocking and agreed activity and move the action of a scene forward. It gives them a chance to go for that instinctive agreed activity option, but then to have that choice immediately wiped out and replaced by a new, better one by an independent referee.

The way that the "new choice" game works is that two improvisers get involved in a dialogue. A third improviser acts as a referee. The two improvisers start a dialogue. If the referee isn't happy with one of the responses that the improvisers give, she can shout "new choice" and the improviser has to say something else.

With this simple exercise, an improviser can make sure that a scene keeps moving forward and avoids blocking. Also, the improvisers involved in the scene get to hear the suggestions that are rejected. They start to get a sense of what "good" suggestions and responses look like. They start to get better at making them first time.

Coming back to the ship example that we've been using, with three improvisers: Kate, Leena and Mei - if Mei is the referee, a scene might go like this:

Leena: Look! A pirate ship!

Kate: Oh no, we'd better be ship shape if the pirates visit. Let's scrub the decks.

Mei: New choice!

Kate: Let's polish the brass handles!

Mei: New choice!

Kate: Let's hide in the lifeboats.

Mei: [Nods indicating that's fine]

Leena: Here we are. The lifeboat.

Mei: New choice!

Leena: Oh my God! Where are the lifeboats?

And so on.

Why am I telling you this? Because, of course, this idea of agreed activity is useful, not just in improvised theatre, but also in project management. On a software development project, when a team comes across something that's the equivalent of a pirate ship, there's a strong tendency to avoid it. And it's the job of the project manager to either play the game of "new choice" with the team and get them to select a better option that moves the action of the project forward, or in other cases, like the director role that Johnson played in some of his improv productions to just shout "deal with the pilot ship."

People who work on a software development project will often choose agreed activity rather than moving the action forward. They will fill their days with simple problems that they know how to fix rather than tackling big, complex problems. Like improvisers, they do this as a way of avoiding having to think or needing to change. Often this "decision" to avoid tackling the problems is taken without much discussion, or even conscious thinking. It is instinctive. Just as it is with improvisers on stage.

You might think what I'm saying here is that it's the nature of software development teams to be lazy and cowardly. I am not saying that. Not at all.

The important thing to understand about agreed activity is that we all fall prey to it. In new, uncomfortable and unpredictable situations we will all be tempted to revert to the easy, the unthinking and the familiar. But, as in improv, there are things that we can do to help get us out of the deck mopping rut and move us on to tackling the pirate ship.

You might think that this is an interesting idea. But you might not yet be seeing how this idea of agreed activity and the story of the pirate ship can be directly applied to the more prosaic and practical world of project management.

Agile project management provides lots of opportunities for you as the project manager and the team to spot the pirate ship. Every day there's a "stand-up" meeting. The team talks about what they did the day before and what they're going to do today. And in stand-up, team members also talk about any problems they have that are blocking their progress. Reporting these "blockers" is the most important bit of the stand up. Because these blockers are the project's pirate ships.

Similarly, retrospectives at the end of every sprint provide bigger chunk of time for the team to reflect on how things are going and call out anything that might clouding the horizon.

But just because there's an opportunity, that does mean that it's one that anyone takes advantage of. Of course, if there's one big thing that's blocking progress, that might be the thing that no-one talks about. Remember, agreed activity is instinctive. It happens often nearly silently and instantly. The signs that there is a pirate ship that needs tackling might not be obvious at first. There might not be an obvious Jolly Roger flying and the sound of "Arrrrrgh!"

So sometimes, the way to look for the pirate ship might be to look for agreed

activity. If some members of the team have the same update with no variation for days and days. That's a sign of agreed activity. Another sign is when the team plans a task and then it doesn't get done for days, weeks, possibly months.

Nobody on the team decides to pick it up. It may well be that no discussion has been had within the team about why they don't want to pick up this task and tackle it. As with the crew on the improvisational stage, they made an agreement. They might not have talked about it. But they all agreed not to tackle that task. Of course, in these situations, it's the job of the project manager to point to that task and ask what it is about it that means everyone is avoiding it.

When they do that, the project manager is acting as the director. They are sitting "off stage" and encouraging the team to tackle something that they've all quietly agreed to ignore. Sometimes, the project manager must take on this "director" role. But of course, it's much better, and much more powerful for the team to do this for themselves.

So, one place to spot the pirate ship is stand-up. Another is the retrospective. That's a meeting that happens, typically every two weeks where the team talks about how the previous sprint went. In the retrospective, the team asks these questions:

- What went well?
- What didn't go so well?
- What could be done better?

Here's one of the many strange things about project management. Very often everybody who is working on a project knows what's wrong with the project. They know what the pirate ship is. So why aren't they saying what they know? Maybe it's because they've tried to say or have said and have been ignored. Maybe it's because they've been told to back off. Maybe they've been threatened with disciplinary action, or even disciplined for pointing out the problem. Or maybe, as we've said, none of the above have happened on this particular project. It's just the same silent and nearly instantaneous collaboration that happens when an improv troupe agrees to scrub the decks.

Just imagine that in an improv show, one of the performers shouted "Look! A pirate ship!" and a director sitting next to the stage responded by saying "Shut up about the pirate ship! I don't want to hear any more of this negative thinking. Clean the decks! Any more talk of a pirate ship and you're off the show."

A leader doesn't need to tell the team to shut up about the pirate ships and clean the decks very many times. Once will be enough to do the damage. If the leadership response to reports of problems isn't positive and helpful, the leaders will already - from just one example - be on their way to creating one of the strangest problems that I've seen. And I've seen it repeatedly: a highly skilled, highly paid team who've given up trying to tell the bosses what's wrong.

The team identified some problems. They explained to the leadership what those problems were. The leadership ignored them, or got angry, or suggested that if the team knew how to do their jobs, there wouldn't be a problem. The team noticed this. They understood. They'd found an agreed activity - doing as they're told. They're smart people, so they've learned really quickly when it's better to not try to speak out or think for themselves.

I saw this phenomenon so many times when I was working as an Agile coach that I gave it a very depressing name - "the beaten down team." Often, I would have been asked by the leaders to give this team an Agile training course. The idea being that a course, all by itself, would magically improve the performance of the team, or the performance of the project.

No matter how many times they have been beaten down, a team will still respond to the right kind of opportunity to feedback. If you run retrospectives and give people an opportunity to speak, they will speak. They will talk if they think you are listening. They will see and appreciate you doing your best to raise their problems with the people who need to know about them. And sooner or later, they will talk about the pirate ship. Whatever it is that is the main problem that is preventing the project from succeeding. The thing that no one is talking about. And when you do find out what it is, it might be hard to take your jaw off the floor.

Here's a story. I worked as a coach on a project that was really struggling. We had a retrospective.

One big problem emerged. All of the requirements for the project were written by a single business analyst. The team's main difficulty was that they couldn't understand what the business analyst had written.

What struck me straight away was that the business analyst wasn't in any of the Agile meetings, not in the stand ups, not in planning and not in the retrospectives. In an Agile project, the business analyst is part of the team - not outside of the team sending in requirements. Especially ones that cannot be understood.

So, my response to this problem was to suggest that the business analyst should be invited to stand-up meetings, and the planning meetings. And retrospectives and show and tells. This was where I noticed the developers looked at each other sideways - this is always a sign that you are finally getting to the real issue.

"We're not allowed to talk to the business analyst."

"What? That's mad! I'm sure that can't be right."

Yes, it was mad. Yes, it was right. Well, it was what was happening. Of course, it was very wrong.

I tried to talk to the business analyst. I emailed him asking for a meeting. I got a phone call from his boss saying I wasn't allowed to talk to him.

And of course, there was no way that the project was going to get anywhere in its current state. The team needed, at the very least, to talk to the person who could tell them what the product was supposed to do.

My suggestion was that we get the business analyst to sit with the team and to attend stand ups and retrospectives. When this finally happened, progress on the project started to improve.

Right here, in this first chapter, we're getting down to the nitty gritty of what project management really is. One thing that project management really is, is a willingness to look on the horizon and look out for problems. Another thing that it is, is willingness to spot when teams have slipped into a pattern of agreed activity and help them to get out of it.

But we aren't sailors, and we aren't fighting pirates. The only tools we have at our disposal are conversations. Meetings. And yes, one crucial meeting is the stand up because it's there that team members highlight what's stopping them getting things done. And another important meeting for tackling the pirate ship is the retrospective where broader, more structural problems with the project often come to light. But there are a couple more meetings that are also a vital part of identifying the pirate ship and tackling it. One is a standard Agile meeting - the "Show and Tell." This is a meeting where the development team show the outputs of what they have been working on in the last "Sprint" - typically two weeks. Who do they show it to? At the very least, they show it to the product owner who acts as a representative of all the people who want the project to happen. But they can also invite other people who are interested. Pretty much anyone who is allowed to see it should be able to come along.

Going back to the story of the incomprehensible requirements that I mentioned earlier, after the minor victory of getting the business analyst to sit with the team, I then suggested that we start to have "show and tell" meetings. This was a fraud detection project. Fraud was an issue that affected people who worked for the bank, and their customers, all over the world. So, for the first phone conference, I could hear just from the accents, the wide geographical spread of interest. We were sitting in an office in central London. But on the phone, there were Northern Irish accents and Scottish accents. There were also Cockney-sounding Southend accents and accents from the offshore call centres in India.

In that week, the team had been doing some work on the user interface for one of the very early screens in one of the fraud reporting journeys.

A developer put up the first screen in the journey. She started to talk it through. Even though most people who were on the conference call were on mute, I thought I detected a change in the silence. Finally, someone on the line with a cockney accent said:

"Erm, I thought we'd agreed that we were going to arrange cases by account name rather than by account number?"

The developer who was demonstrating the screen looked blank and slightly panicked. The business analyst who we'd only just set eyes on and had only just joined the team started to look worried and scroll wildly through the requirements document.

"No, it's in the requirements that cases should be arranged by account number."

"But that makes no sense" someone with a Northern Irish accent joined in.

"It's people who are victims of fraud, not just individual accounts," added someone with a Glaswegian accent.

"We need to see all the accounts that a customer has. And we need to see the activity across all of their accounts, for this screen to make any sense," said another voice with an accent I didn't recognise.

Yes, that's right. The structure of this project was wrong from the very beginning. This problem hadn't been detected through months and months of analysis. But it took just five minutes of putting the working software (OK, I'll admit, it was only a clickable GUI) in front of the people who might use it to find the problem.

If you do stand-ups every working day with your team, if you do retrospectives and show and tells every sprint, you will find out what the problems are. Even if you do these things relatively - you will still find out what the problems are. Not having stand-ups, not having retrospectives and show and tells is just like shouting "Scrub the decks, don't look at the pirate ship!" at your team.

So, what does this idea of "agreed activity" mean for our overall aim of delivering the impossible?

Well, it means something simple. If you work to discover the problems that your project is facing you will find them. If you then work to solve those problems, you may well be able to transform projects that seem impossible into projects that are possible. This gives you the best chance of delivering them.

This seems so obvious as to be laughable. Why then, in project after project, have I found teams that aren't articulating their problems and aren't tackling them? In improv terms the teams aren't looking at the pirate ship. Why? Why aren't they doing what needs to be done before the trouble they can see coming arrives? Why are they washing the decks? Why are they making potentially entertaining improv shows boring? The same way of behaving occurs in software development. It's a way of behaving that makes challenging-but-possible projects impossible to deliver.

Why? Because problems are scary. Problems are humbling. Problems cause the ways in which people don't agree to come to light. Professional people who are hired to do a job are supposed to be able to do it, aren't they? What does it mean if they openly admit that there are parts of a job that they can't do? Maybe it means that someone hired the wrong people.

Highlighting the problems that a project throws up can be threatening to the

senior people involved in that project. What if the problem that comes to light is something that they haven't thought of and they don't know how to fix? They will be inclined to avoid addressing it, possibly by attacking or threatening the team for daring to raise the issue.

In software development, members of development teams do the right thing and point out the problems that they see on the horizon. They do the equivalent of shouting "Look, a pirate ship!" all the time. But unfortunately, the response of the of senior people who should acknowledge these problems and the thank the people who point them out is something like "You're wrong, it's not there."

Other ways of responding that are just as bad are "I'm tired of this negativity," or "Maybe you're not up to the job if you think that's a pirate ship." And that's the main reason that ships and projects sink.

And that's why people don't raise problems and instead just wash the decks.

But why? Why would the people who want a project to succeed most be the very people who put it at most risk by not acknowledging the problems that need to be solved when they arise.

In recent years, the concept of psychological safety has been developed and popularised. It states in a more formal way something that we already really knew. Safety isn't just about physical safety. We also need psychological safety and when we don't have it, we will change our position and behaviour until we do.

Psychological safety imagines that there are four basic levels of safety that sit one on top of the other.

Level 1 is being allowed in the group - inclusion safety.

Level 2 is being allowed to learn in the group - learning safety.

Level 3 is being allowed to contribute to the group - contribution safety.

Level 4 is being allowed to challenge what the group are doing - challenger safety.

These ideas, that Timothy R Clark outlines in his book *The 4 Stages of Psychological Safety* are very useful. As a first step, it's good to understand that one stage of psychological safety needs to be built on top of another. And, from the point of view of our metaphor of the pirate ship, we can see that we need to be at the top of this staircase, all the way up at level 4, before, as a crew member, we can comfortably shout "Hey look! A pirate ship."

For Clark, fostering psychological safety in creative teams is about trying to do two things at the same time. In a creative, problem-solving team, we need disagreements, we need different ideas. As Clark puts it, we need increasing intellectual friction. But at the same time, we don't want to spend all our time falling out and storming off. Again, as Clark puts it, we need to be decreasing social friction.

This is something that I've been struggling with since I've been a project manager. When I'm working with a team, I find myself very often having to point out that there are disagreements and reassuring the team that this is OK.

This connects with another idea that Clark discusses - the idea that psychological safety is about a relationship between cover and candour. The more reassurance that the leader of a team can provide that there won't be any negative effects for being honest, the more honest team members will be. The more cover the leader provides, the more candour she will get from her team.

But there are two issues I have with the idea of fostering psychological safety as a full solution to the problem of the pirate ship. Firstly, real creative situations often aren't psychologically safe - and they never are going to be. There is always going to be a potential conflict between doing the right thing in terms of pointing out the pirate ship and being safe. If you point out the pirate ship, you run the risk of not only being shouted down or ignored, but even possibly kicked out of the team.

Second, there's someone missing from it. The boss. Yes, increasing psychological safety for the team makes it more likely that the team will contribute and challenge. But what about the boss? Is she part of the team? Or is she part of an entirely different organisational setup? What is her role in *that* team? Is she trying to get her feet under the table and just get "member safety?" Is she trying to learn the ropes? Or is she really at the point where she can say to other people in her peer group, "My team have identified this problem. It's something all your teams must also be experiencing. I'm surprised you haven't mentioned it. We need to fix it." What if your boss doesn't have challenger safety in the group that she's in? If she isn't at the point where she can say things like that in whatever forum she operates in, then we have a problem.

This is the idea. The boss is in a different team, and in that team, they might not be, probably won't be, the boss. This helps explain something that I'd known for a long time but hadn't completely understood. It's what the writer Robert Anton Wilson calls "the SNAFU principle."

> Adequate communication flows freely between equals. Communication between non-equals is warped and distorted by second-circuit Domination and Submission rituals perpetuating communication jam and a Game Without End (From Robert Anton Wilson, *Prometheus Rising*).

What does this mean? It means that even though it's the right thing to do, to avoid agreed activity, it's absolutely right for members of a team to point out problems, to stop mopping the deck and point out the pirate ship. Even though this is what a team needs to do to be successful, what happens as a result of acknowledging the problems, is always going to be vulnerable to being warped and distorted by whatever power structures the boss is involved in.

To explore this a little bit more, it might be useful at this point toput the

metaphor of the pirate ship aside and concentrate on the real problem of software development. What actual bad news might you need to communicate to a boss? Here are some examples, starting with the most common.

"There's no way that this project is going to deliver for the proposed deadline." This is of course, the absolutely classic, that in some form comes up in almost every project. But here are some others which are also very common.

"This project is entirely dependent on Department X's system Q being ready. We just talked to someone from Department X, system Q doesn't exist."

"What we're doing is probably illegal. We'd need to spend at least six months thrashing out what we can and can't do with the regulatory authorities."

"The people who can tell us how this system is supposed to work won't talk to us, or can't talk to us, or don't exist."

"My team don't have access to (one or more of) internet connectivity, computers, desks, toilets."

"You aren't paying us."

"We showed this interface to some potential users, and they hated it."

These are real common problems and these are problems that we can't fix. We need to discuss them with the boss, we need the bosses help to get them fixed.

When we're doing this, it's extremely useful to understand that the boss might be in a team where she doesn't have psychological safety and that may lead her to not be in a position, psychologically, to acknowledge, let alone deal with the problems that we're bringing to light. But now what do we do?

This again reminds me of some of the ideas that I've come across in improvisational theatre, and especially in Keith Johnstone's book *Impro*.

Johnstone had been trying to get the students that he was working with to have natural-sounding conversations on stage. He'd tried all kinds of things and hadn't managed to get anything to work. Finally, after another day of struggling, he went home and turned on the TV. There was a programme by the biologist and writer Desmond Morris. Morris was talking about a community of chimps. He explained that pretty much everything that chimps do - who they sit with, who they eat with, who they have sex with, can be explained as a status transaction. When one chimp grooms another, the chimp that is doing the grooming is affirming that they are lower in status than the chimp that gets groomed.

The point that Morris made in the TV programme was that most of these status exchanges aren't huge. There are very few fights involved. There isn't much chest beating. The status exchanges are tiny, fractional indications of who is higher and lower status. Who sits highest on the tree? Who is has their hair picked for fleas first when two near equals agree to pick through each other's

fur? How these questions are answered show the precise status of all the chimps in the group

The next day, Johnstone went back to his studio and asked his students to improvise scenes. He had only one instruction for his students. Whatever one person said, the other person should try to lift themselves, or lower themselves, just slightly, in status with their reply.

Johnstone was delighted with the results. Suddenly, his students were having conversations that sounded like the way people normally talked.

"Are you going anywhere nice on your holidays?"

"Tahiti."

"Ooh, that sounds amazing, I'd love to go to Tahiti."

"Well, my husband is working out in the South Pacific, so it's easiest if I go out and holiday near where he is. Are you going anywhere nice?"

"Ibiza."

"Oh! I love Ibiza. We used to go there all the time before we had kids."

"It's my first time."

"Oh, I'm sure you'll love it."

OK - now read this again. Who is high status? And what does the high-status person do to (slightly) lower their status? Does this sound like a realistic conversation?

Think of a conversation that you've had recently with a friend of a colleague. Who was lower status (just fractionally) who was higher status.

Now let's come back to the business of bad news and taking problems to the boss. The difficulty that project managers are always wrestling with is that bad news is humbling. The giving and receiving of bad news is a status transaction. Johnstone's insight (and Robert Anton Wilson's as well) is that in real life nearly everything is a status transaction. But this makes life really difficult for project managers and also for their bosses. But it explains a lot of the negative and unhelpful behaviour that we see from bosses when they get bad news. It also explains why so many teams recognise the bad news and then tell nobody about it.

Why do kings shoot the messenger? Because in that moment when the messenger arrives and spills the bad news, the messenger is lowering the status of the king. This is intolerable for the king. And in that moment, he can easily solve the status problem. Sure, the battle is lost. But to recover his status and show that he's still the king, he can have the messenger shot. So, guess what happens a lot of the time

What we've established from the beginning of this chapter is that if you don't tackle the pirate ships that you see on the horizon, your project will be a disaster. What we've also established is that the people who came up with the idea for a project will always be resistant to your spotting of the pirate ships. As a project manager, you might see your main job as managing this contradiction.

If we all wait until we aren't scared to start tackling the pirate ship, the pirates will already have boarded and we'll be walking the plank. If we're working on the team, or if we're managing the team, we need to act, even when we're afraid.

Timothy R Clark points out this important difference. Social friction is going to happen - arguments, fall outs are going to happen. But too much is bad for a team. It's better if the team gets along. But another kind of friction is also going to happen. It's the kind of friction that comes from people seeing a situation differently based on their experience. This is friction caused by differences of opinion. In new situations, when dealing with new problems, we need this. We have to expect disagreements, we have to be comfortable with different points of view. Managing the psychological safety of a team is about allowing intellectual friction. At the same time, we need to ease social friction.

But he also points out the importance of a trade-off between cover and candour. If we're one of the bosses, it's our job to reward courage. What's courage? It's doing the right thing, in spite of our fears and providing cover for candour. And in turn, that requires courage on our part as bosses, because *our* bosses might not like the truth either.

And here it comes. The admission. I think I messedthis up for most of my career as a project manager.

How? I think sometimes I've been unaware of this status transaction. And I think sometimes, I've mistakenly thought that I can exploit it. That I can raise my stature relative to my boss by pointing to the bad news. This has never turned out well. This is a bit of a revelation for me, just now, writing this book. The challenge is to point to the pirate ship on the horizon and tackle it. What makes this even more of a challenge is to find a way of doing that that doesn't undermine the bosses. This is my revelation. If you're trying to use the fact that you've spotted this problem as a way of elevating your own status, things will go very badly indeed.

I've spent most of my career as a project manager pointing out the pirate ships. I'm not brave. But I am clear that *not* pointing out the pirate ships and not dealing with them will not work as a strategy. Once you've realised this - that keeping quiet when you see a problem is going to sink a project - it no longer becomes a real option.

But still, you have to find a way of bringing up these problems that doesn't attack the status of your boss. This seems to be a variation on the idea - nobody seems to be entirely sure where the quote originates - "Culture eats strategy for breakfast."

Another way of thinking this (I wish I had known this 10 or 15 years ago) is this.

> If you try to use the severity of the situation as a way of challenging the hierarchy, the hierarchy will win, no matter what the cost to the organisation. Even if it means that the project will fail.

When you're looking at project management methodologies, it's worth looking for the ones that *do* publicise trouble. For me the one that does this most is Scrum. Another method that shows up where there are problems is Kanban. Kanban's idea of visualising the work in progress, and where it is in the process is extremely powerful. Scrum is good at throwing up problems like "We can't access the office" or "The database that we're supposed to interface with doesn't exist." Kanban is good at throwing up problems like "Half the work in the system is waiting for answers to questions from a senior stakeholder." Most commonly, Kanban also shows "Work is taking so long to get from start to finish because there is just too much work in the system."

As we discussed in the introduction, there is a good chance that you picked this book up because you wanted to deliver what seems to be an impossible project. And in this chapter, I think I've given you an important way of seeing projects that will help with this. Find the big problems that everyone is ignoring and, with the help of the team, tackle them. I think I've also explained why the team, and why the bosses might be ignoring those problems. Because they don't feel safe. Because they might have to change. Because they might have to think.

I'm keeping my word. But notice I'm not saying that I'll show you how delivering these projects is a walk in the park or a day at the beach. It might not be.

In conclusion then, what should you do? Well, if you're the project manager on a project, you should look for agreed activity. When you find it, you should look for the pirate ships, the real problems that this low conflict, low impact work is hiding.

When you find the pirate ships, you need to fix what you can in the team. But you also need to find ways of bringing these problems to the attention of your boss in a way that doesn't threaten their status, and perhaps even enhances it.

Dealing with the problems that you find and making them clear to your team and your sponsors can be a rough ride. It can be hard to stick to it. It needs an ability to talk about the real problems. It also needs the ability to be understanding when people call you names. They will call you "Negative," or "pessimist," or "unprofessional." Even if they aren't likely to actually shoot you. But there are two good reasons why you still should do it.

Reason number one is that solving these problems is still the best chance you've got of delivering this project. Spotting the pirate ships and avoiding agreed activity is the single best way of making a project possible. Making it possible rather than impossible. That is a solid and compelling reason.

But reason number two is possibly for me, just as important: I know what happens when you don't and that's what we'll look at in the next chapter.

Chapter SHIP2 – Tackling the pirate ship

We've acknowledged that ignoring the pirate ship is a bad idea. We've established that we're an authority on the problem? But what about the solution? Once you've spotted the ship on the horizon, what do you do about it?

Connections

I think I made it pretty clear in the chapter on agreed activity and the pirate ship, that it's very important to deal with the pirate ship. Let's just be clear again what the pirate ship is. It is the problem that everyone can see on the horizon that nobody wants to deal with.

What I didn't really address in that chapter is how to deal with the pirate ship. Why? Well, here's the main reason. Most of the problems that software development projects encounter are easy for someone to solve. They are mostly of the kind that I call "Bricks without straw" problems. And there is an obvious solution to them. What do I mean by "Bricks without straw" problems?

Here are some examples.

- The wifi in the office doesn't work.
- The people that we need to talk to so we can find out what the software should do, refuse to talk to us.
- The list of things that need to be done by the deadline can't be done by the deadline.

Most of the time these kinds of "bricks without straw" problems can be solved. If they are made clear to someone who has the capability to fix them, they can be solved relatively easily. Someone has sufficient seniority to call the company that is providing the wifi and internet access and get it fixed. Someone is senior enough to email the subject matter experts and tell them that they should make time to talk to the development team. Even to make the tough decisions that need to be made to either reduce the scope that needs to be delivered for the deadline, or change the deadline.

But this is a bit of an Agile dirty secret. Yes, Agile has a focus on using transparency to fix problems. And yes, it's really effective as a way of fixing a lot of problems. But there comes a point when you've fixed all the problems that can be fixed. For a lot of problems, simply giving them enough air, daylight, and publicity works. They get fixed. But then, guess what you're left with? Yes, that's right.

The problems that you can't fix.

Some problems are hard. Some problems are hard because their solution would require innovation - a new answer that nobody has tried before. Some problems are hard because they are inherent contradictions - they don't have a solution. They can't be solved, they can only be managed. And many problems are hard because it's not clear whether they have a solution and we need to look for it, or they don't and we just need to manage them.

Concrete practice

It is worth asking, what would a solution *look* like. What *kind* of thing would be a solution.

Is it a technology? Is it a relationship? Is it a form of communication?

Another thing to think about is the dynamic of the problem. Is there a threshold beyond which this won't be a problem anymore? For example, how fast would the internet connection have to be for a developer in the office before it ceased to be a problem? How infrequent would outages have to be before they stopped being a problem?

Let's say that we commit to a course of action to improve something. We agree to push to get access to the people we need to talk to. How long is that likely to take? How long will it take for us to feel the benefits. Some kinds of capability take months, or years to develop.

Do a bunch of things need to happen, together, or in sequence, for this to get better?

A lot of the problems that we encounter in project management are trade-offs. There are lots of trade-offs in software development with speed. Speed is traded off against accuracy. You can have something fast, but it might not be exactly right. Speed is also traded off against something called "tech debt." Tech debt is the difference between doing something fast and and doing something "right." Where "right" means tidily and in a way that might make sense to other people.

Speed of development is also, of course, traded off against security. But speed can also be traded off against things we might think are unreservedly good. An example? User research is great, understanding the swamp and the denizens of the swamp is vital. But at some point you have to start writing some software.

Concept

Congratulations. You have solved all the easily solved problems. You have pointed out the pirate ships. Now you've hit bedrock.

Conclusion

What kind of problems might the "pirate ship" metaphor represent? When we start to list the kinds of problems, we get more understanding of why people instinctively avoid them. The pirate ship might be a relatively straightforward "Bricks without straw" problem. These are just very frustrating for the teams who have to labour under them. And they are often embarrassing for the senior people who need to get them fixed. But they can be fixed.

But some other problems are just downright *hard*. What's the right trade-off between the complexity of the software and maintainability? What signs should we look for to know that we're adding too much functionality too quickly? Or even what signs might we see that suggest we're not being adventurous enough and we need to go faster?

Well, then we can actually start to build something that is valuable to its users and valuable to the people who paid for it. And when you get to that stage, what you needs as well as general ideas is technique. How do you run a project using the principles of empirical control?

How do you make sure that the "pirate ship" whatever it is, is front and centre and being address by both the team and the stakeholders? How do you involve the users at every stage of the project? This is what we're going to talk about in the next chapter.

So the book up to now, been how to look at a project strategically. How do you deal with a project that seems to be impossible? You look at it from the point of view of a crew on a ship. Where are you scrubbing the decks in some kind of agreed activity? What pirate ships are there on the horizon that you aren't dealing with. You find the agreed activity in your team and disrupt it.You help your team, you help your stakeholders to focus on the problems in front of them.

And a good way to think of those problems is as the exploration of a swamp. It has submerged dangers and opportunities. It has a ton of people who are interested in what goes on there, from different points of view. It isn't a known quantity. If it were, it wouldn't have required an expensive and risky project to explore it.

When do we listen to the people of the swamp? When do we ignore them? When do we decide that we've done enough talking and we have to start building something? Getting that right is the result of experience and technique.

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

There are some things in life that we're immediately attracted to, without thinking. There are some things which are valuable but require work. The problem with and the challenge of, project management, is that we need to move from the idea to a reality. The idea is immediately attractive - like a flower. But the reality, which does you good, like fruit, requires effort to cultivate. This move from flowers to fruit is fraught with difficulty. One way of understanding this and knowing what to do about it is by using the idea of "value streams."

In this chapter, I'm going to talk a lot about "value streams". And also, about "flowers" and "fruit". Of course, of those terms, "value stream" is probably the less familiar. So, I'll try to explain value streams first.

Value streams are often talked about with regard to "Lean" approaches to manufacturing. And what are they exactly? In the mid 1970's there was an energy crisis, oil prices were high. American car manufacturing 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 from the business schools in America began to write about *why* Japanese made better cars than than the Americans. A key idea in explaining why Japanese industry, and especially the Toyota car company 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's 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.

American observers also drew attention to the Japanese understanding of what *didn't* add value. Activities that are going on in a factory that don't contribute to the value stream are regarded as waste. There's an example in Taiichi Ohno's book, "The Toyota Production System". There is a machine that is part of the production line that needs to be changed when a different model of car is made. All of the time that it takes to swap the machine, is, from the point of view of a value stream, waste. This seems easy to understand. But there are other kinds

of waste that Taiichi Ohno identified at Toyota which perhaps aren't so intuitive. If completed cars, having rolled off the production line, are waiting around in a yard, they aren't delivering any value. That's also waste. If parts are stacked high in warehouses and for long periods of time aren't being added to any car – that's also waste.

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

This investigation of how the Japanese made cars resulted in at least three interesting ideas.

1. The manufacturer of a product can be arranged into a series of value adding steps.

1. Any waiting around between these value adding steps is waste.

2. Any waiting around before the final step of getting the product to the people who use it, that's also waste.

These ideas were interesting to a lot of people. They offered a suggestion for how any organisation might improve its process and make it more efficient. Although this is quite a leap. Not all organisations, you might have noticed, make cars.

And one main idea came out of these discussions. It's possibly to map a value stream. It's possibly to see what all the steps in a process are. So then, 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, comes 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 especially look at reducing waiting times. Then we can notionally 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 people have made good solid efforts to apply these ideas to software development. Some of these attempts work [Ref: Poppendiek] and some really don't. We'll talk about where these ideas are exceptionally useful in a following chapter [Chapter ESCAPE].

But right now, I'm going to take this idea of value streams and stretch it in a couple of directions. Because, I'm going to argue, that when it comes to development of software for a new product, there isn't just one value stream. There are (at least) two value streams. A further complication is that, at the beginning of a project, of these two streams, one of them doesn't exist (yet) and the other is opaque and 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, and will actually do that, if you don't manage to stop them,

So, what are these two value streams? The first of these streams is what I'm going to call the "virtual" value stream. This is a stream that delivers value by looking good, sounding good, or by coming up with a new, attractive, 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 the people 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.

The 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 (I already have) 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," or "will have the same functionality."

Sometimes the idea's appeal is difference:

"A fresh look," or "A different take."

Sometimes all of 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, almost always, is 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 is a substantial part of why I say that the virtual value stream is weird. Yes, there might be an application process for getting ideas funded, and, as I said, earlier, there must be some steps to the process. But there might just as well not be, this could just be the idea of someone really senior, or someone who has the really 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. Maybe another organisation is doing it, and they just have a strong inclination to do whatever they do. And 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 the people s going to be vaugue.

Even though the appeal of ideas might not have any obvious logic to it. From the examples I've given, we might see that are some common themes in the ideas that get funded. Ideas that get funded often have these concepts lurking in them – "all", "same" but also "new".

Where ideas come from is a fascinating subject but I'm going to side-step it for now. 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 it needs to start being fascinated by the reality. We need to stop smelling the flowers think about how we're going to grow the fruit.

And there's an 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 actually 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 an interaction with users which is different from looking at, or smelling, flowers. Users have to get some good, some value out of using a software product. What 'good' that value is, is different for different kinds of products. For example, it's very different for a social media product than it is 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 (grown) properly, it could do them harm.

So, what about fruit? What about this other value stream - the virtuous value stream? The one that actually gives people what they want? In the Toyota product system mentioned 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 of this that often gets missed when people try to transfer these ideas to the development of software. Most software development projects are developing a 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? I'm exaggerating a little bit. 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 need 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.

1. Discover value – explore and find out what the potential users of this product might want.

3. 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 the chapter after that [Chapter ESCAPE], we'll talk about 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 two steps.

3. Deal with all the flack and fallout that comes from discovering value and constructing a value stream.

This 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 actually doing things results in uncovering 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, particularly a Scrum approach. In Scrum, every day, the team have an opportunity to report any problems that they're finding in their attempt to realise the dream, the idea that has been funded. Add into this mix user researchers who are talking to the users about their needs - and we should add them. Then, pretty soon after the project starts, you start 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 a really good reason why it won't work. They think about it in a way that you really didn't anticipate.

That's the thing. However, users think about it doesn't really matter. What matters is that it's almost guaranteed that they way that they think about it won't exactly the same as the people who had the idea. And to them – the "keepers" of the idea - unless they have deep and strong experience in product management and development, this is going to appear as a threat.

And this is a sticky, difficult, delicate and business. And we're going look at it from a series of different points of views in the hope of getting a smarter way of looking at 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, is a value stream. But it's a value stream that, if it delivers at all, will deliver something that is slightly different from the dream. The reality, if it's to be a success, will be something that is intimately involved with users. It will engage with their enthusiasms, capabilities and limitations. It also, if it's got a chance of success, it should also address the interests and concerns of lots of other stakeholders. Regulators and spectators and possibly 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 dreams and reality. One the one hand are the dreams of the people who had the idea and funded it. On the other hand are the wants and needs of the people who experience the reality. And between these two is a hard place. This is where support from the people who had the idea is running out and support for them 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 (a prophecy?). The hero has no real idea how they're

going to realise that vision. But, after some reluctance, set off to try any way. 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 actually be about. They have lots of struggles, setback. They think about giving up. But they don't. In a final scene, they put all of 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, and 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 90 IQ points."

I believe that thinking of a project in terms of value streams is a interesting and different point of view. One value stream is a virtual one. It deals in dreams. The other is a virtuous one. It needs to be built through course of the project. I think this point of view does make it easier to understand what's actually going on in a project. For example, this view of a project explains the team's reluctance to tackle the pirate ship that we 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. Especially, 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 said anything. Maybe they're just relying on intuition. The leader's feeling about taking risks was shown by the way they reacted to even the tiniest thing going wrong.

In order to go 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 give them encouragement. They want someone to acknowledge their struggles and try to give them the best chance of success – that's you!

Taking this point of view, of a project as the construction of a path through a wild and unknown space, helps. It helps with one of the major causes of feeling bad for any project manager - 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.

Word and phrases like “new”, “fast”, “just like the old system, but better”, “Using AI and blockchain.” to some people, 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 all a funded project is, is 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 can commit. 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. And you can commit to finding what’s useful, valuable and attractive there [Chapter SWAMP]. 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 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 substantially explore the area that it points at. But then, having explored, you need to decide which of the things you find are valuable. Then, of valuable things you’ve found, you and your team need to decide which of those things 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 whilst somehow not upsetting the people who had the idea.

So, lots of bad news. Is there any good news? Well yes, there is good news. If you follow this approach, you will have given yourself and the people you work for, the best chance 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 90 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 on almost any project, no matter how crazy the idea sounds.

Finally, there’s one bit of news that you could get, that 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 written in some sales copy. Gary Halbert tells a story of a question that he asks people who come along to his copy writing 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", "sour dough bread" and "Ethical meat grown in a lab." 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. If you have a hungry crowd, that's 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 a lot different from "enthusiastically support."

So, there's a simpler idea, past all the talk of fruit and flowers, virtual and virtuous value streamsr. 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, but the challenge is to decide which of those to connect together.

CHAPTER_SWAMP_START

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. There is untapped potential. There are pitfalls. The way to succeed in this kind of environment is to acknowledge that you're exploring. One aspect of doing this exploration is user research. Another is stakeholder research. It's almost always a struggle to do either of these. For different reasons.

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 there. 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. They 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. And beyond government, there might even be global bodies that care about this swamp. There are people who 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 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 who already in the swamp? Maybe there are new species there, interesting 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 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 of those are directly associated with the project, some of those will directly benefit from the project. 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, they might not have heard of you, but you're still important to each other. If the project is going to be a success, 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 and in doing so, would make you a fortune.

On the other hand, it might be a really 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 we talked about in [Chapter STREAMS]. Projects are things that are done in a complex environment. There are lots of people who interested in that environment. And the interests that they have are extremely varied.

I didn't really get this completely until I worked on one project. On this project, I worked with a woman who did research with users. 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 made 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 with them, 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 user's 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.

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 were surprising about it for me. The first is 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 lurch back 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 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 be a success. Many of these would not be at all obvious if you hadn't done these investigations. And you also start to see where there are dangers and problems and potentially unhappy people who could cause you and the project trouble.

This project was the first time that I really understood the value of user research. But I really should have understood it about fifteen years earlier. Because I had previously worked on a user centred design project. 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 connected 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 a European Union funded project. The first aim of the project was to design an application that used tangible user interfaces. Tangible user interfaces 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 school children construct discursive arguments. My organisation was doing the user research for the project. "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 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 who were 10-11 years old) 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's the thing that we found out really quickly. And this was an important way that we could help the residents of this particular swamp. We found it out by doing direct research with them in schools. They didn't need any help understanding logic. What they needed help and support with was gathering evidence and putting it in a structure that sounded like a persuasive argument.

That was the great thing about doing research in real schools. But when we started to do research, we found out about some other concerned parties who didn't live in the swamp. They were a long way away. Even so, they really cared about what happened there - the government. At the time when we did this research, the government insisted that all the schools in England follow a "National Curriculum." Pretty much every lesson that 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, 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.

Fortunately, we had an ex-teacher on our team. So, we could design our research sessions so that they also made sense as national curriculum-focused lessons.

A long time later, I found myself working on 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 to develop 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 may lift your eyebrows slightly at a company that didn't talk to the regulators before starting up in the money-lending business. You can also see that research with users and mapping of stakeholders would have made something as important as that very clear, very quickly. User research is great, and this project would have seriously benefitted from user research. 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, commentators on the ethics of your industry. To come back to the swamp metaphor, these are people who don't live in the swamp. These are people who live miles from the "swamp" and who will never

visit it. But they still care what goes on in there. And if these people aren't happy, sometimes, they can kill a project.

You might say that I'm just exaggerating. Software development projects aren't that different from one another. Swamps aren't all that different from one another. You've seen one swamp you've seen them all. Do you know what this sounds like to me? Well, it sounds like ignoring the pirate ship, for a start [Chapter SHIP].

But also, in later chapter [Chapter Ref] we're going to talk about driving a car as a metaphor for managing a project. And here's just an early view of that way of seeing things. Some people might say "One project is pretty much just like any other, I don't see why you need to do any more investigation with users." They're saying something that's the equivalent of "Yeah, I've driven cars before, there's no need for me to look at the road."

Yes, controlling the car, and managing the team might be similar from project to project. But outside of the car, or the team, the environment can be very different. And my main point in using this metaphor of the swamp is that if you look out through the windscreen, you may well find that there isn't a road. Or that the road is blocked by monsters. Or that there is a very nice road, that it looks very easy to drive down, but if you do, you will get a massive fine or, actually, in some cases get sent to jail.

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 done by professionals. But one of the main aims of doing this research should be to find out surprising stuff. Then maybe, you can take what you find back to whoever is paying for the project. Maybe you can show them what you found and use it as a case for more user and stakeholder research.

Another criticism of my insistence on user research is what might be called the "faster horse" objection. Henry Ford said something like "If I'd given the people what they want, 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 ask for. What I *am* saying is that whatever you do has a much greater chance of success if it's informed by what they've asked 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.

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 very 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. It's even harder than getting support for development of working software.

There's something I feel I should own up to here. I'm convinced of the value of continuing user research through 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 in 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 is well-informed is important. The best chance of success for a project is when the team are aware of what the 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 all, or some 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 that comes 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 psychological safety of the team. Reassure your team that conflicts that are emerging as a result of 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? There are still things that you can do.

Talk to your stakeholders about risks.

This is something you should do even if your project has a full complement of user researchers. I'd 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 another 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 came back with a suggestion he had found on the internet for an "Agile" risk register. This was a Kanban – style board with three columns. At the beginning of the project, we got together. Who was we? It was me – the delivery 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 "cost" to the project of fixing the fallout of the risk if it actually became an eventuality. We expressed that it, 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. Then, we also gave each risk a probability as a percentage. As a group, we thought that there was a small, but not insignificant chance of this happening, so we gave it a probability score of 5. Then the overall score of the risk was 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" (>1000 points). Another had "medium risks" (>500, <1000) and another had "small 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 changed the scores. And we added any new risks that anyone in the group identified.

In one sense, none of this discussion about scores on the risks mattered. But in another sense, it was a life saver 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.

Do show and tells.

Scrum is the most popular Agile framework. And "Show and tell" is the Scrum name 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 doing user research. 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 on that week. 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 you insist that they do it.

Invite people from all over your stakeholder map to your show and tells. Remember the story about the bank and the fraud project? 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. 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* boss of *their* back. But you don't have to see them like that. You can see them as powerful, if a little 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? 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. We'll talk more about this in the next chapter [Chapter ESCAPE]

Exploit the crisis.

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

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 who fund 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 is because they are in power, or because they want to preserve some aspect of the environment or possibly for some other reasons. Some other reasons that, without research, it would be difficult to really understand.

The message of this chapter is that the place that's pointed to by the dream of a project that we talked about in [Chapter Streams] is a mysterious and complex place. And the way to give your project the best chance of thriving in this mysterious place is research and investigation. Talk to as many of the interested and potentially interested parties as you can. Understand how their interests and values interact – 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, user researchers who can can talk to the interested 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 as a facilitation of conversations. One way of thinking about the chances of success in a project is by increasing the quality of the conversations that you have with the people who have interests and stakes in the project. The people who own, live and care about the swamp. We'll talk about improving the quality of conversations again in [Chapter CONVERSATION]

But there's another, complimentary way of exploring the swamp. And that's to build something in the swamp. To let interested parties, see touch, even feel what it's like to use this new thing. And that's what we're going to talk about in [Chapter SOFTWARE].

CHAPTER_SWAMP_END

CHAPTER_ESCAPE_START

Chapter ESCAPE - Working Software

In software development, the other key method of exploration is working software. The more working software you put in the hands of users, the more likely it is that your project will succeed.

> "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 and wrote the Agile manifesto. They were all people

who, for a good long while had been wrestling with the question of how to make project management methods for software development simpler, more lightweight and more successful. The emphasis on simple and lightweight is possibly why the manifesto isn't a huge, long document. It's less than a hundred words. And in the Agile manifesto, working software is talked about as the second key value.

> "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 discussions between people. And that's what we talked about in the last chapter. But right after that, the founding fathers of Agile recommend working software. It's worth asking. Why? What was the experience of those guys who got together to talk about "lightweight" software methodologies that made them mention "working software" as one of the four key values.

My guess is that they had all had experience of being involved in projects where the production of working software was delayed or had never happened. They had worked on projects where months or years could go by before the specification was agreed and software development could start.

For the first forty or fifty years that software development existed, 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. And in other kinds of engineering, at least in theory, nothing is built before the production of detailed plans.

This is what makes the second principle in the Agile manifesto so revolutionary. As if in the first principle saying you should talk to people and understand what they want isn't revolutionary enough. The second principles is saying the engineering equivalent of "have a go at building a bridge and see how you get on."

So why? These guys got together for the express purpose of making the way that software development was done and was managed better. Why would they want something about working software in the manifesto?

OK, let's make this about you for a minute. Think of something that you know a lot about. There will be something. It doesn't have to be anything to do with work. But think of something that you know how to do. OK, now think of some aspect of that thing that someone who wasn't an expert would think was strange. Why do you do that thing?

There are at least a couple of possible answers. Maybe you were taught to do this thing by whoever taught you. Maybe it's just your own way of doing things. But there's one answer which is very likely and very compelling. You do things in this way because you've seen what happens if you don't. You've learned the hard way.

And I think that that's the reason why "working software over detailed documentation" is in the Agile manifesto. And this is a very similar reason to one that we've discussed when we were talking about avoiding agreed activity. Why was I pushing to avoid agreed activity and work with the team to tackle whatever the problem was that was looming on the horizon? Well, there are lots of good reasons, but one of the main ones was that I've seen what happens when you don't [Chapter TRENCH].

I think it's absolutely the same reason that working software is one of the four main things that are discussed in the Agile Manifesto. The people who put it there have seen what happens if you don't push for working software. But when it comes to working software, it might also be that they've seen the good things that can come from producing working software.

So, what is it about working software that make it so important?

You can sum up what's so important about software with these three ideas. Magic, discovered value (leading to pull), discovered trouble.

Picking up on the quote at the beginning of this chapter, working software is "sufficiently advanced technology" and people respond to it as if it is magic. People respond to working software immediately, automatically and emotionally. When they press a button, or click on a picture, they have an expectation of what will happen. Working software creates needs, expectations and it evokes emotions: delight, joy, frustration and anger. This is a very different response than a checklist of functionality in a requirements document. That means that the only way to find out how they are going to respond, is to put actual working software in their hands.

And when you do put software in their hands, users will find ways of using it that you didn't expect, users will value aspects of the software that you didn't expect. Remember the swamp? When you build in the swamp, you will start to discover value, people will use what you've built in unexpected ways. And when people like something, they want more of it.. And this leads to something called "pull".

Up to the point where actual users of your software find something in your software that they like and ask you to expand that function, or do more of it, by delivering it in different ways, the sole driver for the functionality in your software was the stakeholders who had originated the project.

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

And this is a huge change. It means that your project has moved from being just someone's idea to something that people actually want, something that has value. This might not be the only possible criteria for success, but it's certainly one of the most important.

And this is one of the most important reasons for making working software available as soon as possible. When we make working software available early, we're giving it the opportunity of connecting with its audience, of letting it create pull, of being guided by the people who are using it in how to make it deliver more and more value.

But there other very good reasons for making your software available as soon as possible. And one of them is perhaps best explained using a joke.

> You will never be alone if you take with you everywhere the ingredients and equipment to make a dry martini. Even if you think you are completely desolate and stranded on a desert island. Because minute that you start to make the dry martini, someone will jump out from behind a tree and say "that's not how you make a dry martini."

Something like this is true of developing working software. Ideally, you want your software to be in the environments we it will finally 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 people that you didn't even know existed will jump out from behind trees. These people will tell you that "you're doing it wrong" and try to stop you getting working software in the hands of users.

In my head, I always imagine the job of getting working software out in the world on a working environment like trying to escape from a prison camp! There's only one way to find out what all the traps are that are out there in no man's land, and that's to try to set them off.

OK, I'm mixing metaphors. Let's stick with the man behind a tree for a moment. Most of those things that the man who jumps out from behind a tree will bring up are "non-functional requirements." These are requirements that the software needs to meet but which 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 is covered by some regulator in the industry that you have never heard of.

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. Before any software gets written you might try to think of everything that might possibly be required. 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 these requirements without a piece of working software, you're only dealing with "known knowns". 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 find about "known unknowns". You knew there would be other

security measures out there in no man's land, but you didn't know what they were.

This is the slightly counter-intuitive thing I'm trying to tell you. 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. The only real way to find out what there is in no-man's land that's stopping you and your software from escaping is to push your software through it. Let's move on to the second reason why trying to create working software is a good idea.

The second reason is this, we might call it the "there's only one way to find out," reason. How do you find out if you can do something? By trying to do it.

There are lots of rules and regulations relating to - well - everything, it seems. And software is no exception. Rules about security. Rules about hosting. Rules about performance. In software, this red tape is often called "non-functional requirements." And your software might get stopped dead in its tracks by someone wearing a metaphorical peaked cap because it doesn't meet these. And we've talked about these. But I'm not talking about those right now. This is more straightforward than that. These are the simplest questions that trying to get working software answers.

Is your team capable of writing this software?

Does your team have access to the tools and resources that they need to write this software?

Does the technology that you've decided to use work?

Is the organisation that you're working for capable and willing to pay for the servers, people and set-up that you need to deploy this software?

Well, there's only one way to find out the answers to these questions and the answers might not be the ones that you're hoping for. I've worked on teams where the team members don't have access to the office. I've worked on teams that don't have access to the internet - "To make internets, you need internets," one developer was forced to explain. Of course, these are problems that can be solved, but only once they're uncovered.

So, this is the most straightforward reason why a team should be trying to develop software as soon as possible, because there is only one way to find out.

But there's a third reason. Software is magical and the way that people respond to it is magical. People don't respond to working software in the way that they respond to feature lists or specifications.

Think about it. Think about the pieces of software that you interact with every day. Are you thinking about them in terms of lists of features? When you're using some piece of software, you are using it to do something. You have other things on your mind. A funny joke that you want to share, a report that you want to write or a podcast that you want to listen to.

And this, final reason is the main reason that it's useful to try to move a project towards delivering working software earlier rather than later. By putting working software in the hands of the people who will use it, you start to solve the Flowers and fruit that well talk about in [Chapter Ref]

When you put working software in the hands of users, you move the discussion. You move the discussion and the dynamic of the project away from flowers. And it isn't just the discussion, it's the actions. The focus isn't any more about the shiny list of things that a piece of software might do. Now the discussion is about fruit. What does the software actually do? Now that it's in people's hands, does it do something that's useful for them. What can we do to make this thing that users are already using even better?

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 make it manageable using an electronic document handling and storage system.

And the project had a couple of good things going for it. Firstly, it was using an Agile way of doing things. Secondly, the product owner was a former clerk of the company. She knew all the other clerks and she knew their business very well.

Even so, to start with the project had a tough time. We couldn't get servers to put the software on. The open-source document package we'd chosen wasn't as mature as we thought. But 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 managed to pare down the demo to a view of a collection of documents and 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 pound 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 and 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 chance 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. But the product owner’s request to have a version that she could take around the country and to show off was a powerful help, if we could actually 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. The demo still wasn’t much. The demo still had bugs. We still had some spelling problems! But the demonstration by the product owner to her own former workmates went very well. She could wave past 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 come across 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 questions - “When will it be ready?”

From that point on, the nature of 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 road of what we needed to do rolled out before us, with the people we delivering to cheering us on.

There was another very interesting thing about getting some 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 a dispute. The price of providing and supporting the servers was not included in the contract. But the client argued, that they shouldn’t have to commission the servers. Neither should they pay for them or support them. This became a much harder argument to make once people in their own company were asking for the software.

Suddenly the people who were trying to negotiate free stuff were in the way of their own colleagues. Now they weren’t helping the company by being tough on costs. Rather they were stopping people who worked for their organisation from getting something they wanted. Something that could make their lives easier. Suddenly the servers appeared, and then money for people to support the servers appeared. A real roll-out of the software started to happen.

To the second question, “If we try to do this, is anybody going to jump out from behind a tree and stop me?” The answer 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. 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 right. We should have built in accessibility right from the start. It takes no more 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 this 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.

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 costs of any changes we made needed 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 others still being looked at.

Putting bad software that just about worked in front of real users completely changed the project.

Connections Showing working software to the clerks awoke

something powerful – pull. It's a thing that gets talked about when people talk about different ways of doing Agile that aren't focused on software development. Ways of doing Agile like Lean, which is Agile, but for manufacturing. In Agile in manufacturing, pull is a key concept.

You may have heard the saying "just in time." It's a strategy for managing supply chains in all kinds of industries. One industry that has perfected this approach to managing its work is the Japanese car industry, and particularly the Toyota car company. The Toyota company succeeded because from the start it understood one thing. Its senior engineers realised it just wasn't possible to make cars in Japan in the way they were being made in the 1920s in America. Why? Because Japan's economy at the time was tiny. And the owner of the Toyoda sewing machine company, who was looking into moving into the car business, knew that the Japanese economy was very cyclical. There were good

times, but there were also bad times. There were booms but they were always followed by busts.

So, from very early on, the new Toyota car company, matched the rate that it made cars to the level of demand for cars. The market decided how many cars it made. Over nearly a hundred years, that process has become very developed. The result is that many different kinds of cars, with the many different extras that modern cars have, can roll off the production line of the same plant. The cars that come out of the factory almost precisely match the demand outside it.

Each car is "pulled" off the production line by a specific request for a car which comes from a dealer, which in the end comes from a customer.

Thinking about things in terms of providing value to the customer results in another important idea – "waste". In Toyota's way of thinking, anything on which money has been spent, which isn't going to result in value that will be delivered to the customer, is waste.

If you put these two ideas together: make things in response to demands from customers and don't have anything hanging around the factory that isn't on its way to customers, you end up with a process that fits the label that has been applied to it in the west - Lean. It also fits the other name that is often used - "Just in time."

It's important to point out that making cars is *very* different from making software.

Even so, taking these two ideas - only deliver things which the user wants and don't do anything which isn't going to either directly benefit the user - are powerful. These two ideas are two sides of the same coin. If taken seriously, they are a powerful way of making projects which seem impossible start to suddenly seem possible.

Concrete Practice

So how do we do this? Every project is a little bit different. But I'm going to arbitrarily invent a rule of sixths. This feels about right. If you think your project is about 6 months long, I'll give you a month to do some set up. If, not long after a month you and your team don't have *something* that works, you've waited too long. By the end of a sixth of the 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 finally use it.

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 rather than the software being run on just one machine, or in just one development environment; set things up so that the software is in a live, or like live environment.

What's important with working software, is not so much what you've got, but that you've got something and it's moving in all the right direction. Sure, it needs to be increasing in terms of what it can do. But it also needs to be moving from developer environments, to test environments and onto live environments. And it needs to be moving from being tested by the team to being tested by "friendly" users to being tested 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 pull on it, to make sure it's firm and would hold you if you fell, then you can climb a bit more.

Yes, delivering software is a bit like that. It's a bit like climbing a rock face. The odd thing is how many people think that you can get to the top without a rope. But even odder are the people who somehow imaging that you can get to the top of the cliff in one single bound.

Gradually developing working software in all of those directions 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 functionality from the software. Not from the people who sold the idea - 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 helpfully provide you with extra rules and regulations that you need to follow. Also sometimes, they'll tell you that you can't proceed until you're certified in some way, and won't tell what rules you have to follow to get certified.

You will need to negotiate these before your software can finally escape and live free in the real world. Push working software as far as you can towards being live with real users. It's the most powerful secret to actually delivering things that seem impossible. And everybody, or nearly everybody, will try to stop you doing it, even people you thought were your friends, colleagues and allies

What? Yes, that's right. Incrementally delivering working software is the thing that you need to do. It's the thing that's most likely to help your project succeed. Still, nearly everybody will try to stop you doing it.

How? Why?

OK, let's deal with the how and the why, but separately.

How? Developers will tell you that there's no point breaking big bits of functionality into smaller bits. Even though, if they did do this, those bits could be

shown, released and tested sooner. They'll say that it "only makes sense" to release some bit of functionality in one big piece.

How? Somebody will tell you that there's no point troubling users with small bits of functionality. They'll tell you that users only want to see the whole, finished, journey. They'll tell you that showing them anything sooner is a waste of time and money.

How? Some users will tell you that there's no point looking at the new system until it has their real data in it. Some other users won't be interested in using the software until it really works. They won't want to use it until when they press the "launch missile" (or business equivalent) button something actually happens.

How? Somebody will tell you that test servers are expensive.

How? Somebody will tell you that the live environment is only in the budget from the week before the project is about to finish.

How? Product managers, even good product managers, that you like and trust, they will utter the dread, tragic words "we need it all, so I don't think it's important what order it gets done in."

But why? Why don't people want working software?

The reason people don't want working software is something we've already talked about. It's the same reason that the improvisers that we talked about in [Chapter ref] don't want to deal with the pirate ship. Without having to think about it, people know that dealing with working software, or with the pirate ship that just appeared on the horizon, will mean that they have to *change*. They will have to change what they think. They also might have to change what they do.

The other reason is that trying to get software working attacks the "Flower" of an idea that we talk about in [Chapter ref]. This is the aspect of the idea that people think is good, even without giving it any thought. These are the aspects of ideas that tend to be of the form "all", "same", "faster", "cheaper."

Trying to get even a tiny bit of software working tends to undermine some aspects of an idea. If just getting *something* to work takes so long, and that something is so slow and has cost *how much* money? Suddenly the idea doesn't look so shiny and appealing.

And what about the people who jumped out from behind trees and told you couldn't do things, or you shouldn't be doing them that way? Nobody wants to see them. Nobody wants to deal with whatever extra requirements they place on your project. Especially the people who got funding for the idea.

Moving towards working software does a whole lot of things that in general people want to avoid. It throws up a lot of technical problems that require thinking. It throws up a lot of rules, regulations and restrictions about what can and can't be done that require even more thinking. It makes obvious how

slow and expensive it is to just get *some* of a product. And in doing so, it tends to undermine the "simple", "fast", "all", "cheap" appeal of an original idea.

These are all very good reasons for not getting software working. Getting software working throws up all kinds of surprises. It will always be very tempting to stop.

You will start to feel these reasons for not doing working software. You will see other people being persuaded by them. At this point it's really important that you understand these reasons why you should carry on.

1. You're going to have to do it sooner or later – or never

And sooner is much better that later. Why? Because of all those problems that you encounter when you start to deliver small amounts of software. You will get problems with the technology. You will get problems with environments, unheard of rules and regulations. You will get problems I haven't thought of, problems you haven't thought of. There is no way around all of those problems. The only way to deal with them is to get through them. If you start to deliver something early, then you can deal with these problems in bite size pieces. If you put off dealing with these problems until later, you'll be forced to try to deal with them all at once.

2. Working software teaches you things about the problem that you're solving.

Trying to do working software gives you enormous amounts of information about the environment in which you're working. Putting working software in front of potential users teaches you about your users. But trying to get software onto live and like live environments also teaches you about your stakeholders. Who values this project? How is it seen in the rest of the organisation? When you do stuff, you learn stuff.

3. You create pull - or you don't.

Sometimes, when you put working software in the hands of users you get a response that you weren't expecting. "What the hell is this? This makes no sense at all." Or even worse, apathy and silence. Nobody says anything, everyone just ignores it.

This may not be pleasant. But when do you want to find out? You might be grateful that you found this out when you were a small percentage of the way into the project. You've only spent a small amount of your time, money and resources. There's still time to change your mind, take account of the feedback and do some things differently.

And having users object to your software, or even hate it, isn't actually the worst response they can give. The worst response is utter indifference. You show users your software and they really don't care. They don't like it. They don't not like it. They are just indifferent. This is the most difficult kind of feedback to deal with because it doesn't give the project any guidance about where to go next. But even such a non-response is useful earlier rather than later. Again, it's worth asking yourself, when would you, or your sponsors like to find out that

nobody cares about your project? Now, when you've spent less than a quarter of the time and money that you'd budgeted? Or later?

Of course, there's another response that you're hoping for if you put small amounts of working software in the hands of users. If you're lucky they will ask just one question - "When can I have this?" And then, almost certainly, they will follow that up with "Could it do this? It would be really nice if it also did this." This is what you're looking for from working software. You're looking for pull. You're looking for a way of prioritising the things that the software currently does and the things that the software could do. And you want this based on actual value to the users (fruit) rather than superficial appeal to the internal sponsors and funders of a project (flowers).

Once you've found something that users want that you think your software can provide you will have made substantial progress. You will have moved a long way down the road of moving a project from impossible, to possible. But by doing that you've also made the problem a lot more complicated. By trying to get software out into the real world, you will probably have had people jump out from behind trees. They will have told you that you're not allowed to put software out into the real world. These people are now on the list of people that you have to please. By putting working software in front of users, if you're lucky, you'll have created demand and expectation. Now it's obvious to you that you need to satisfy some of these demands and expectations if this project is going to be a success. But at the same time, what about the people who got this project funded? And what about the people who actually funded it?

All of a sudden, they're not as in control of the project as they were, the project has been let out into the wild world. It isn't just their baby any more. People may not react to this well. It may feel that you've deliberately pushed you and your team into a storm, when they could have stayed in calmer waters. Because that is exactly what you've done. This may feel like a very stupid thing to do, so it's important to remember why you've done it. You've done it because staying "safe" not being changed won't get you where you, your team and your project need to be.

What's important as you move through these difficult waters, is to keep pointed in the direction that you need to be travelling. You need to keep pushing the working software in the direction of the real world, real users and real data. If what you're doing is adding functionality to the software that can then be tested with users, then good, you're going in the right direction. If what you're doing is making the software fit better with non-functional requirements, that's moving in the right direction. If what you're doing is moving the software nearer to live with live data and being used by real users, then you're starting to make this project look possible.

Moving from working with dummy data to working with real data. All the time getting you're getting feedback from users and stakeholders. And when people jump out from behind trees and tell you that you can't do what you're doing,

you're ready for them.

CHAPTER_ESCAPE_END

Chapter CAR - Driving a Car or The Empirical Process

Project management is all about facilitating the empirical process. Be transparent and encourage and facilitate transparency in others. Look at what's going on, encourage others to look at what's going on. Change what you and your team do in light of what you see.

> "Seeking what is true is not seeking what is desirable." - Albert Camus

"Empirical" isn't a common word. But empirical process refers to something that we're all doing all day every day. All day every day we change what we're doing in response to what we see and experience through our senses. And that's what the word "empirical" means. It means relating to experience in the real world. It's a slightly odd word that doesn't sound as down to earth and practical as it is.

For years I've run training courses to introduce people to Agile ideas. Because "empirical" is a slightly strange word, there's an exercise that I often do where I get people to look up the word and to discuss what it means. Then we do exercises. These exercises often involve Lego.

I give the team a bunch of Lego models. I ask them to estimate how many of the models they can build in a short, fixed space of time. These short, fixed amounts of time are called "iterations," in Agile.

Almost always, at the end of the first iteration, the team realise that they can't do as many models in the time as they initially thought.

So, part-way through the exercise this forces the teams to do something different, based on this new experience. Either they reduce their estimates of the number of models they can build or, possibly simplify the models. At the end of the exercise, I come back to this word "empirical." Empirical means based on experience.

That is the key point that I'm trying to get across in this exercise. Learning from experience is important. Modifying what you do in light of what happens is a fundamental approach to controlling and improving your performance. And this doesn't just apply to your performance, it also applies to the performance of projects.

What are we doing every day that's so vital? What we're doing is honouring what are referred to as the three pillars of empirical process - transparency,

inspection and adaptation. When are we doing this? All the time, but one time when we're especially doing it, that I'm going to use as an example, is when we're driving a car.

Imagine that you're driving a car. The first thing that you need is transparency. You need to be able to see out of the windscreen. If the windscreen is covered in mud, or being washed with buckets of rain, or frosted up with ice, that's bad. And it isn't just looking forward that's important. You'd like to see behind you through your mirrors. So the wing mirrors need to be there and the back windscreen needs to be clear. When you're driving, before you even start moving, you need transparency. And if you don't have it, things can go wrong very fast.

But transparency isn't enough. If you're driving, it isn't enough for the windscreen to be clear, you still need to actually look out of it. You need to look out of the windscreen and check your mirrors. If instead, say, you're checking messages on your phone, or distracted and trying to brush scalding hot coffee out of your lap, that's no good. Transparency isn't enough by itself. You need inspection as well.

We're focusing here on the visual aspects of driving. But we all know that when you're driving, you're also really checking with your other senses. If you feel an unusual vibration accompanied by a dull thudding sound there's a good chance that you'll slow down. If you smell petrol, or burning, you'll probably slow down. If you hear police sirens, you'll check in your rear view mirror and check if the police seem to be interested in you. If they are, you'll probably speed up if you're in the middle of committing a bank robbery. If you're doing nothing to be ashamed of, you'll probably slow down.

So, inspection isn't just a visual thing. It's paying attention with all the senses. Talking about slowing down and speeding up brings us to the third pillar of empirical process - adaptation. What we're doing when we slow down, is adapting. That's also what we're doing when we speed up because we see the road is clear. It's also what we do when we swerve to avoid something or put the fog lights on when it's foggy. Adaptation is when we change what we're doing because of what we sense, what we see, hear and feel.

Notice how, for these aspects of empirical control to work, they have to be arranged correctly. They are dependent on each other. We can't drive successfully without transparency - if the windscreen were covered in mud, we would not see the things that we needed to avoid. We couldn't do it without inspection. If we didn't look up from our texting and see the thing we were about to hit, we wouldn't feel the need to swerve. And finally, we wouldn't be swerving if it weren't for adaptation. All of these things together are empirical process.

At this point, I might mention that I've crashed two cars. So nobody is going to say that I'm an expert on driving. But I am an expert on what these principles of empirical process mean for software development teams.

Let's start with transparency. What does transparency mean for software

development teams? Well, I'm afraid it means meetings. There are at least three meetings add transparency into the software development process.

The stand-up is a short meeting that all the members of the team have every working day. Each member of the team says what they did the previous working day, what they're going to do today and highlights anything that's blocking them. It's pretty obvious how this maps back to the driving analogy and empirical process. The stand-up has all the pillars of the empirical process. It has transparency, members of the team are telling each other what they're doing and pointing out any problems that they're having with it. It has inspection. Well, it has inspection, so long as team members are listening to what other team members are saying. And the stand-up is also a crucial opportunity for adaptation. When they hear about the progress, or lack of progress of other members of the team, it them becomes relatively natural for the team to change their behaviour in response to what they hear. To adapt. Either by focussing on the problem and fixing it, or understanding that the problem can't be fixed right now (or ever) and routing round it.

Members of the team should let others know what they're doing, what they're going to do in the daily stand up. And they should also point out any problems that they are experiencing.

Another meeting that implements the three pillars of empirical process is the retrospective. In retrospectives, the team looks in the rear-view mirror. They look at the road behind them and talk about what progress they have made down it. They have time to reflection the things that blocked their path. They get to talk about how they could have dealt with the obstructions more easily. What things they've got the hang of now, that means that next time they might be able to tackle obstacles faster in the future. They also get to talk about the things that they still have no idea how to handle at all. The obstacles that are still in the way and the things that are still slowing them down.

The final Agile meeting that implements the pillars of the empirical process, is the show and tell. I'm not sure what the reason is, but this is a meeting that goes by a lot of different names, all of which mean the same thing - "Sprint demo" or "showcase."

The show and tell is a regular demonstration of working software that shows exactly the real progress that the team are making. And of course, that can really be a problem.

The first ever Agile software development project that I worked on was for a publishing company. When I joined the project, it had been going for over a year and had cost many millions of pounds. The way that I heard the story, a few months before I joined, there had been a show and tell meeting. It was an international company and it had offices in London and in New York. There were people with American accents dialling in on a conference call.

I don't know much about what happened in that meeting. I do know that at

some point one of the Scrum Masters said that he was going to share project progress. This was progress that had been made through "the backlog." The backlog is the Agile way of saying the list of all the requirements for the project. But I'm not going to use that word any more throughout the whole of this book. Simply because it's one of those words which is used in Agile in a specific way but seems to cause more confusion and difficulty than it does clarity and progress. So from now on, I'm going to use the phrase "To do list." And for each of the items in the "To do list" I'm going to use "To do list items."

He had a presentation with some slides. And there was a bit of messing about setting up the presentation - and then making sure that it was shared with the office in New York. Finally, everything was set up and the scrum master could move to the slide that showed progress of the project. He clicked on to the slide. In the middle of the slide was a giant "2".

I don't know how dramatic he was at this point. I don't know if he asked anybody in the room or on the phone if they could guess what the two meant? 2 weeks to finish? 2 Months? 2 Years? But I know that at some point he did get the point across. The 2 was percent. Progress on the project, after six months and many millions of pounds was 2 percent of all the work that was in the to do list.

This is brutal transparency. And it probably came out of desperation. Having worked on that project I can guess that someone had told the bosses in New York that the project would be finished by a certain date. Nobody had mentioned to the bosses the problems that were being found. Problems that were slowing progress to a crawl.

The main result of that number 2 was that that Scrum Master was never allowed to speak at a show and tell again. The bosses in New York hired a "traditional project manager" to communicate progress across the Atlantic. After that, guess what? The news was always good.

This is what one friend of mine said about this meeting:

> I remember an explosion of anger over the phone where the entire team (50+ people) were dressed down and told to get on with it or heads will roll.

But what does it tell us? What does it tell us about empirical process, transparency and adaptation? And in the end, what does that tell us about delivering the impossible.

OK. Let's start here. I'm going to put on my magic fortune teller's hat and I'm going to look into your project. Mmmmm. Ahhh. Mmmmm. Actually, I don't need a funny hat and a cloak, and I don't need the strange noises. If you have a project, and this project has yet to deliver its first live release and that release has a deadline, I already know one thing about it.

> Your project is supposed to deliver N times more work than can possibly be delivered by the deadline.

What's N? On a "good" project N is two or three. I've seen lots of projects where N is 7 or 8. On the project that I just told you about where the Scrum Master brought up the "2" on the screen, N was somewhere between 25 and 50!

But maybe I'm wrong about your project. Maybe your project is fine. Maybe you're reading a book called "Delivering the Impossible" out of idle curiosity. You wonder what it would be like to work on one of those projects that seems impossible. But your project? Your project is fine! Maybe.

But if your project isn't fine, what are you going to do about it? Telling it like it is, as the Scrum Master in the story tried to, doesn't seem to work that well. In that case, it resulted in the hiring of someone to specifically stop the bosses in New York being told the truth.

Someone senior might well have promised something that is impossible to deliver. Getting them to deliver anything near that might be a difficult process, requiring careful handling. Yes, that's for the bosses. Diplomacy, tact, careful handling and an understanding of the politics may well be required.

But for the team that's doing the work? It's never a bad thing for the team to know the truth as soon as possible. If you're in charge of that team, you should keep the windscreen clean. You should take every chance to make sure things are transparent. This means making sure that the meetings which aid transparency are happening. It means that people feel free to talk in the stand-up about any problems that they're coming across. It means people feel comfortable speaking out in retrospectives about anything else that might be bothering them. It means that show and tells are honest about progress and show working software.

The second thing is to understand that shouting out bad news in front of fifty people might not be the best way of communicating it.

Here is where you might be a lot better at this than me. Like the guy in the "2" story, I'm not very diplomatic. Like the guy in the "2" story, I've nearly gotten fired for telling it like it is in front of the wrong audience. You might be more diplomatic. You might have a better way of dressing up bad news.

But here are some other things that it's important to understand.

Everybody in the room knew that "Mr Two" was right. The bosses in New York heard what he said. Unfortunately, the way that "Mr Two" said it was so confrontational that their reaction was exactly the wrong one. Their response was to hire someone to stop him telling them the truth ever again.

We started to talk about the rather jolly metaphor of driving a car as a way of thinking about empirical process. And by doing that, we've ended up right at the absolute nitty-gritty of project management.

To manage a project, you need transparency. When you achieve that transparency,

you get bad news, almost always. Then, as a team, you have to do something with the bad news. What do you do with the bad news?

So what was bad about what Mr Two did? What was bad about what he did was that he upset the bosses. He made them feel stupid in front of the whole team. And the result of that was that they made very sure that they didn't hear any news from him again. They appointed a "project manager" who never ever told them any bad news. Please note that this didn't mean that there was never any bad news.

Of course, another thing that was bad about the situation was that it had been allowed to reach such a point. My guess is that this wasn't the first time that "Mr Two" had tried to point out actual progress. He'd probably tried several times to make clear what measures of actual progress meant for when the project would finish. Mr Two's action was a reaction against a powerful, top-to-bottom (and probably top-down) system of agreed activity. In the end he was forced to shout out "There's a Pirate Ship! Oh my God! There's a pirate ship! Can't you see it?".

What was good about what Mr Two did was that he did manage to get a message to the bosses. And even if the result was that they made sure that they never heard from him again, there's no doubt that they heard it. And it's interesting to note, they didn't fire him.

Is there anything else he could have done? that would have made the situation better. It seems that everybody in the organisation didn't want to hear what he was saying. So what could he have done?

Well, there are are few things. All of them are partial. But they're all powerful.

He could, and should have continued to keep track of the project's progress. Even when the bosses had ignored him. Why? Because this is still useful information. Actual progress tells you not only when a project is likely to "Finish" but also how long any particular piece of work should take.

He could have pushed more to get working software to where it finally needs to go. He could have done more to get it into the hands of real users. Because real users were the people that the bosses had to listen to.

And he shouldn't have taken it personally. It's easy to say that. From my experience, it's difficult to do it.

Transparency, inspection and adaptation is the best chance that you have of delivering a project. What every trouble your project is in, it's normally a good idea to increase these things. Are you stand-ups getting repetitive? Is nobody really saying anything in them? Have battle lines been drawn over some disagreement, so nobody is saying anything about it? What can be done differently to get around the problem?

Chapter CHAOS – Cynefin and the Edge of Chaos

Software development is "complex" – that means that even when we have lots of experience and do things in the right way, problems might still occur. A lot of business is "chaotic" – experience can count for nothing and it might be that there just isn't a right way to do things. Software development sits on the border between the complex and the chaotic. Almost the only absolutely certain thing is that if you treat it as if it's a simple problem, you're guaranteed to fail.

Concept

There is a model of project management that has chaos on the diagram. This is the Cynefin framework. This is a framework that was developed and is still being advocated by a working consultant, Dave Snowdon. And it keeps changing, Snowdon keeps refining the details, but when I first came to know about it, the Cynefin framework had four sectors. Four sectors which represent different kinds of problem-solving environments. This is how I explain those four sectors.

Simple Simple. This is the first sector. Some problems are simple. They are understood. Pretty much anybody can solve these problems.

The type of problem that I like to use as an example in this sector is cake. Ingredients for cake are well understood. The equipment needed to make cake is well understood. And there are recipes. Also, people understand the value of cake. Cake is nice. People like it, is has straightforward value.

So most people can make the simplest kinds of cake. These are problems that almost anyone can solve.

Another thing to notice about cake – the value of cake is obvious. Most people like cake! OK, there might be some people who don't like cake because eating it makes them put on weight. OK, there might be some people who for various dietary reasons can't eat it. But for most people, the value of cake is very clear. Crucially connected to this, the relationship between the effort involved and the value produced is relatively well understood. People know the value of cake and the roughly know the amount of effort involved to make it.

Complicated Next to the "Simple" sector on the Cynefin diagram is the complicated sector. There are lots of different kinds of problems in the complicated sector. But they have one thing in common. Like the problems in the simple sector, they are understood (by some people at least) and there are agreed, effective, solutions that can be applied. But there is also a crucial difference

between complicated problems and simple problems. Complicated problems can be understood and solved, like simple problems. But complicated problems can't be *easily* understood or *easily* solved. Complicate problems can only be solved by someone – as the phrase goes that's used in patent applications "Skilled in the art."

Anyone who wants to solve a complicate problem will need to go on a training course, they might need to go on several training courses, those courses may take years. But if you pay attention and do well on the courses, at the end of them, you will be able to understand and solve these complicated problems.

This level of training an education is the big difference between simple and complicated problems. But, as well as solutions being known and available, there is another similarity between simple and complicated problems. Their value of solving them is understood. And as with simple problems, the relationship between value and effort is understood. We have respect for doctors especially surgeons because we know that they studied for a long time and that the outcome of that study is to be able to do lifesaving operations.

Complex OK, here's where it gets interesting. Next to the complicated sector, we have the complex sector. In the complex sector most of the certainty that we had in the complicated and the simple sectors disappears. Problems aren't perfectly understood, and solutions don't perfectly work. Obviously these two uncertainties are connected. In the complex sector, it isn't that nobody has any idea how to solve the problems. There are still suggested approaches that work a substantial proportion of the time. There is a *degree* of success. But there are some situations for which there aren't any clear solutions.

And, I think you were probably expecting this, unfortunately for us, this is where software development sits. Why? Here are a few reasons.

1. The "tech stack". Any project that involved software development doesn't just involve one piece of software. It involves lots of different pieces of software, which, ultimately, then run on hardware. All of these layers of hardware and software depend on each other. So that a change at the hardware level can mean that software doesn't run any more. A change at the operating system level can mean that networking doesn't work exactly as expected. And change to the language that the project is being written in can affect how the software behaves and performs. The instinct then, might be to stop any of these things changing. But that too, is fraught with its own problems.

2. There's no perfect way of expressing what we want the software to do. This is what, in philosophy we might call a "strong claim", but it's the kind of thing that's easily shown by asking someone to write down all the steps involved in making a cup of tea. Pretty soon you realise that you're making a *lot* of assumptions, or you have a lot of dependencies, if you want to see them that way. Assumptions about the quality of the water, about

the availability of electricity, about what the right amount of milk is. A requirements document for making a cup of tea would be a non-trivial thing. And remember, when you try to write down what you want to happen when making a cup tea, you're trying to describe the solution of a *simple* problem. If you're trying to describe any reasonably complicate interaction between humans and software, which probably also involves business rules.

3. Empirical evidence. Software development projects go wrong. They go wrong a lot. If there were a way of making sure that software projects were successful, even if it took as long, or longer than training to be a surgeon or an architect, someone would have put together that training and the success rate for software development projects would be higher than it is. Even using agile methods, research indicates that at least twenty percent of software development projects fail completely. Another thirty percent are "challenged," meaning that they are either materially late or cost materially more than was expected.

The complicated problem-solving space isn't just where software development sits. It's where all kinds of consultancy sits. especially "management consultancy" sits. But it's also where all sorts of fakery and quackery sits. This is why people can be so rude about consultants. And this is why this book is neither going to be the first nor the last about the difficulties of managing software development.

So, once we move away from simple and complicated, to the complex sector of the Cynefin framework. Most of the certainty that we had when solving simple and complicated problems is gone. And with it disappears any obvious relations between the effort put into solving a problem and the value that's returned. [The value of solving complex problems gets sucked into this vortex The cost isn't obvious, and the relationship between cost and value isn't obvious]

Chaotic Then there's a fourth sector. Chaos. In the chaotic sector, the problems aren't understood at all and the solutions to the problems aren't understood at all. Unlike in the complex sector, there are no recognised solutions that work in this sector. In the chaotic sector, you just have to try things, celebrate when things work, or shrug it off when things don't work.

~~T~~his is my experience:

> Software development works best in the complex sector. Software development is always tending towards the chaos sector.

And actually, sometimes:

> Software development is always tending towards another sector, that I haven't told you about, called "Disorder". Disorder is when you don't know which sector you're in, and so you have no idea how to behave or what to expect.

What? Why? How?

Reason #1 - the business is chaotic.

Software development itself isn't a chaotic activity. But software is often trying to track or profit from activities which are chaotic. A lot of entrepreneurial business is chaotic. In fact, if it weren't chaotic, it probably wouldn't be a properly entrepreneurial business.

Let's say you've come up with the genius idea of a social network for pets. Nobody has done this before! There might be a good reason! Is it a good idea? Who knows. The only real way to find out is to do it.

Reason #2 - misapplying a different problem space

What happens if you treat flying a passenger airliner as if it's baking a cake? Bad things probably. Treating a complicated problem like it's a simple problem doesn't work that well. Treating a complex problem, like software development, as if it's a complicated problem doesn't work that well either. Trying to manage the chaotic space with the kinds of approaches that work in the complex space doesn't work. But I've just realised, this isn't chaos. This is what Dave Snowdon (the developer of the Cynefin framework) puts in the middle of the Cynefin diagram. It's the black hole in the middle.

This isn't chaos, this is disorder. It isn't chaos that I'm scared of. It's disorder.

What's the difference between chaos and disorder? In chaos, there is an understanding, even if it's tacit, that the environment that you're in is chaotic. In disorder, the wrong kind of methods are being wilfully, sometimes desperately applied.

Connections

The screenwriter William Goldman, the man who wrote the script for "Butch Cassidy and the Sundance Kid", said in his book "Notes from the Screen Trade" that "No one knows anything."

He was, of course, talking about the film industry. And what he's saying is slightly strange, because of course, lots of the people who are involved in making films know lots of things. There probably aren't many simple skills involved in filmmaking, but there are lots of complicated skills. Make-up? Special effects, sound, lighting and cinematography – these are all skills that are complicated but can be learned. Then there are some skills which, when they're put together have a good chance of working but can't be guaranteed to work. We might put acting, directing and the fundamentals of screen writing in this category

But there's one aspect of film-making that is always going to be, to some degree, chaotic – it's reception. This is a huge risk for movie studios and the people whose reputation is attached to the film being a success.

Concrete Practice

The concrete practices for this are the core pillars of the empirical process that we've already talked about repeatedly. Transparency, inspection and adaptation. And also working software. Pushing working software as far as possible towards its real users with real data making real transactions.

Conclusion

I'm not scared of chaos. I'm scared of disorder. Actually, I think that might be bollocks. I think, like most people, I have a healthy fear of chaos. With chaos comes the possibility of success and failure without rhyme or reason. Also, most people have a healthy fear of failure. And people like things to happen for a reason. This is why there are now lots of films that are sequels, this is why there are now some many films that are part of a "universe".

What does drive me crazy is people trying to control chaos using the wrong methods. Simple methods won't work in the chaotic space. Complicated methods won't work in the chaotic space. Complex methods won't work in the chaotic space. But the fundamentals of an Agile approach, inspection, adaptation and iterative and incremental delivering of working software. Those all do work.

If you've developed a new product, there's a good chance that you're in a chaotic space, in some sense, you're *there* for the chaos. Even though you're telling yourself that you're not. You're there because. But if you're in the "disorder plughole" in the middle of the Cynefin framework, you're there because somebody really doesn't understand what's going on.

The system as imagined vs the system as found

Imagine a triangle. The triangle is your project. Now imagine that there are forces pushing on the triangle. One of those forces is for functions – stuff. There's pressure from the people who want this thing for it to happen for it to do stuff that works in the world. Then there's another pressure on another side of the triangle. This is a bit weird, it's a negative pressure, a pressure to do this stuff with as few resources as possible. Fast and cheap. So, there's pressure to do stuff, and there's pressure to do it quickly and cheaply. But then there's also pressure from the third side of the triangle. There's pressure to do it safely. Safety. What kind of safety? Well, if the software is for an aeroplane, it might be the "get it wrong everybody dies" kind of safety. If the software is for a financial institution, it might be the "get it wrong everybody loses their money" kind of safety. But there are lots of other kinds of safety. Loss of reputation, getting in trouble with the regulators.

Beyond that safety line, is the real world. It's where bad things happen, like planes falling out of the sky and banks losing all of their money. It's also where all the good stuff happens, planes fly to exotic destinations. People get to spend

and send money how they want. Pictures of cats doing funny stuff get shared all around the world.

And so we're left with a tricky question. Where is the right place to be on this safety side of the triangle?

Chapter EGG - Fried Egg Agile – with knobs on

Agile is a collection of lightweight, empirical methodologies. The way you do it for your team will be slightly different than the way others do it with their team. That's fine. In this chapter, I share what has worked for me.

I saw a TV programme by the chef Rick Stein. He said that whenever he interviews a chef who might work in one of his restaurants, he asks him to fry an egg. Stein says that all the best chefs - the ones that he's likely to hire - do the same thing. It isn't that they fry the egg in the same way. It's that they have the same attitude to it. They shrug and say something like "I don't know how you fry an egg. This is the way that I do it."

You can tell a lot about a person by how they do the simple things in your line of work. But there are a lot of ways doing it right.

There are many ways of frying an egg. There is also more than one way of running even the basic aspects of an Agile project. So, here's my description of how I do the Agile equivalent of frying an egg. But here is also a description of the "knobs" – the variables that you can change depending on your circumstances to make sure that these tools fit with the circumstances.

Concepts

Initially, examples of good ideas that work look a lot like each other. One car looks a lot like another. One aeroplane looks a lot like another. There's a central idea that works and everyone sticks to that.

But then after a while, that central idea gets adapted in different ways. Think of all the different kinds of cars that there can be. You can have really fast cars. Really stylish cars. You can have huge cars (like the dumper trucks that are used in quarries). The designers of cars start to realise that there are parameters – knobs - that they can tweak.

Meetings There are four meetings.

1. Stand up
2. Planning

3. Show and Tell (or Sprint Showcase, or Sprint Demo)
4. Retrospective

Stand up For me the power of Scrum is that it tells you to do some simple things – meetings. And it tells you to have some things and they are quite simple things - artefacts.

The first thing to start doing with any team that you're working with is to have a standup. The standup is a meeting that happens at the same time every working day in the same place.

The original idea of the standup was just that. The participants should stand up. And the idea behind this is that standing up discourages long conversations. I think there might also be something else about standups that was pretty powerful. They involved a change in stance.

Even so, I still think there's a lot to be said for the act of standing up. It's doing something *different* from what the team does for the rest of the day. I think there is some power to the simple act of adopting a different physical stance when you do something.

Anyway. Get your team together and get each of them to roughly answer these three questions. What did they do yesterday, or the last working day? What are they going to do today? And is there anything that's blocking them? That means, is there anything that's stopping them from doing what they want to do?

And try to keep the whole thing under 20 minutes. Try to push things forward if there are obvious digressions. Sick cats? Sympathy yes, long discussion of ailments no. Someone's team did well in the sports? Brief chuckle yes, blow by blow analysis, no. Somebody mentions a problem that you think you know how to solve? Agreement to talk to them in detail about it after standup yes. Getting right into the nitty gritty right there in the meeting? No.

That's it. I don't know what the Scrum guide says these days. Maybe it says something different. The stuff I've just described?

Planning Get the team together and discuss what you're going to do in the next fixed period. The world seems to have settled on two weeks as the agreed fixed length of time. And I don't have much against it, except to point out that in my experience, sometimes work is obviously moving on a larger "cadence". By that I mean on a different rhythm, a different fixed period, to a different beat. And it's worth noticing these cadences as the team progresses and to discuss with the team what you want to do about them.

Sometimes things are so crazy that a week works better. I avoid any options other than multiples of seven days, because calendar arithmetic is beyond me. And I truly believe in the power of getting into the rhythm. If this is Wednesday and it wasn't planning last Wednesday, it must be planning today. That's it.

That's all the effort anyone should ever need to put into figuring out if today is planning day or not.

People are such dicks about commitment consistency. See the chapter [chapter ref] about your other boss, reality if you don't believe me. And for this reason, I don't push planning as being any kind of commitment. Again, this might seem like a heresy. There is this idea that the team should plan what they're doing in the sprint and then "move heaven and earth" to do it.

Do you know what "moving heaven and earth," means? It means working late, it means missing lunch, it means doing something other than working at a sustainable pace. And so it probably means making mistakes or taking short cuts. It probably means making the project harder to manage, harder to deliver and certainly much harder and more unpleasant to work on.

It's really easy to fall into the trap of thinking that rushing to meet deadlines *is* real work.

A long time ago I read a book called *Sources of Power* by Gary Klein. Klein spent a lot of time studying people who really know what they're doing. Experts. The book is a series of examples of critical situations where experts worked successfully and their expertise won out. One distinction that Iremember he draws is that a strong indication that you're in the presence of an expert is that they know what they're capable of.

Of course, experts weren't born magically knowing what they're capable of. They had to learn over time. Learning what they're capable of is a fundamental part of their expertise. And it's the same with teams.

Teams, including the product owner, will initially tend to plan too much. Of course, when this happens, it's important to make this clear at the end of every sprint. But not in any kind of accusatory tone, rather as point of practicality. "We planned thirty things. We finished twenty things. Do you think there's much point in planning more than about twenty-five things this sprint?"

Here's something I hope that all the people involved with the team can understand. This includes the developers and the product owner. Let's look back to the previous planning meeting. When we got to the point where were discussing issues number 21, 22, 23, but especially when we were discussing numbers 26, 27, 28, we were just wasting our time. Because we were talking about things that just weren't going to happen in that sprint.

One thing that I never used to bother with was a sprint goal. But in recent projects I've found a sprint goal, or goals, extremely useful.

Why? Because they're a good way of describing to people outside of the team what the team is doing. If a goal is something like "Release candidate for Feature X", I think that's a good goal. But also, "Start Feature Y." For me, that's a good goal.

There's an argument that often surfaces when you first talk about goals that goals should be "SMART".

There are variations on what this acronym stands for, but it's something like: specific, measurable, achievable (or attainable), relevant, and time-bound.

So often some of the members of the team will baulk at a goal like "Start feature Y," because they'll say that it's not measurable.

I simply don't agree. The team will absolutely know if they've started a feature or not. And at the end of the sprint, if the goal has been met, that's useful information. If the goal hasn't been met - if the feature didn't get started - that's also useful information.

Fiddling with the knobs.

At some point it will dawn on you that a lot of these things, focus on detail in stories, focus on a general goal for the sprint, the emphasis that you put on delivering stories within a sprint, the frequency that you plan, the focus that you put on estimating the complexity of the stories that you plan. When you've been on a few projects, you're start to notice that where these where these knobs and dials are set is different for each project.

If things are a bit too slack, well then tighten them up. Set up some meetings during the sprint that ensure that stories for the next sprint have sufficient detail. If you get to planning and it turns out that the detail in the stories was either impossible to fill in, or wrong because there's too much uncertainty, well the loosen things up. With the team, set some more general goals for the sprint.

And all of this, needs judgement. But there are obvious things to look for. If hardly is getting finished, radically reduce the amount of work that you plan. If the team are running out of work half-way through the sprint – well, then plan more. If there seem to be endless discussions about what precisely goes in a story, as I said mentioned, put more work into refining those stories before they get planned, but on the other hand, if, when the developers get to a ticket, a lot of the detail that's in it is wrong, or needs revising because of new information that's come to light, it's OK to reduce the detail.

Show and Tell Demonstrate the working software to people who might care. That will probably be the product owner. It might be other people related to the project, all sorts of people who have an interest in the swamp.

Understand that dealing with feedback is, at least, a two-step process.

The first step is capturing the feedback. It's very important that you show that you're listening. Write down the feedback from every show and tell.

The second step is to decide what to do about the feedback. That's something for the product owner, primarily, and possibly the rest of the team.

What you're doing by having a show and tell is fulfilling the "transparency" and "inspection" aspects of empirical process.

What that really means is that if you've done something dumb, someone is going to tell you. Also, if you've missed something important, someone is going to tell you. That's what you're hoping for. That's the payoff.

But this isn't just a negative thing. You might find out in the show and tell that you've just done something that will make people's lives much easier. Your team might just have done something that marketing think they can make a lot of money from, or win a big reputational improvement. You might get to know that if it weren't for the show and tell.

The price for getting this valuable information is that people might say stupid things. We demonstrated an application that helped a bunch of users fill in a big, complex form. We were asked this question: "Can you make it so that users aren't allowed to submit the form if the English has spelling or grammatical errors?" To some people this might sound like a great idea, if the answers that you're supposed to put in those boxes contains the names of chemical compounds and scientific formulae, it gets a bit more tricky.

This isn't too serious. But it can get a little wearing.

What can be more of a problem is that by showing working software you can unearth a real conflict in the organisation. You want to avoid putting controversy in the software. Which means you need to surface the controversy before it gets in there. But this can be a rough ride.

So, feedback, no matter how brutal, non-sensical or contradictory, is the first reason that you're having a show and tell.

What's the second reason? The second reason is working software. If you have a meeting where you're supposed to demonstrate working software, it's going to become obvious if you aren't demonstrating any.

If you've read and remembered the chapter about exploring the swamp, you'll recall that these were the two activities that I recommended. Drawing a map that shows the people who live in the swamp and the people who care about the swamp. Exploring values and creating pull with working software.

On a healthy project, on a project that is really working well, both of those activities are happening in separate sessions that have been arranged with users and stakeholders. On a project that isn't working so well, they both happen in the show and tell, it's important to the health of any project that we make sure that they happen there.

It's important that the show and tell works well. You need a good subset of the stakeholders to pay attention to what new things are going into the software. If the team isn't demonstrating new working software on a regular basis, you're creating a problem for the future.

What kind of problem? Well, maybe the team isn't really building anything. That's one kind of problem. Maybe they're struggling so badly with a bunch of different issues that they're not managing to get anything to the point where they can demonstrate it. If the team can't demonstrate any working software, they can't create enthusiasm and pull from the users. They can't get an important signal about what's valuable about what they're doing.

But, even worse, maybe the team do have working software to demonstrate, but because no one is looking at it, it's the wrong thing.

Retro The retrospective is a chance for the team to express how they feel about how things are going.

Yes, notionally, it's supposed to be a more forensic look at what went on in the last sprint. But really, I want it to be a meeting in which the team get chance to vent.

If there are definite actions that come out of it, that's great. If there is a short list of things that need to change that can be taken to senior management, also great.

You will be able to find lots of descriptions of how to run a perfect retrospective. There is one thing that it's important to understand. If you're not willing to get it wrong by trying different ways of running retrospectives, you probably won't get it right.

Other meetings that I find helpful These are some other meetings that I find useful. I'm not saying you'll need these for your teams. I'm not saying that *at all*. But I find them useful on some teams.

Mid Sprint Check

I find it useful in the middle of a sprint to have a meeting where the team just gets together for half an hour and checks in to see how things are going.

Here are two or three things that I do in that meeting.

1. Check the goals - we set some goals at the beginning of the sprint, in planning. How are we doing against those goals? Sometimes, I know this sounds crazy, but we'll all have forgotten what the goals were!
2. Looking at the list of stuff still to do in the sprint, does any of it now seem impossible? Have we found out things that mean that something just can't be done? What do we want to do about that? Is there someone we need to tell.
3. Look at the roadmap.

Risks and Issues

This is a new one that I learnt on my last project. My heart sank because I was told that I needed to report risks for my project every week.

My experience of risk registers is that they aren't useful. They tend to become, almost instantly, "information fridges." A place where information goes to remain unchanged and unnoticed and have no effect.

But my product owner was an employee of the organisation and wanted to make sure that we were doing everything we could to show good governance. He found a description of an Agile approach to risk management that I had no problem getting behind.

Each sprint we'd meet with the senior stakeholder. For half an hour we'd walk through a Kanban board of risks. Each risk would be scored using an "exposure" score. This was worked out multiplying the cost of the risk happening and the chance of it happening.

For example:

Risk: We don't deliver the service on time.

Cost (in days for the whole team): 40

Percentage chance: 30%

Total Exposure score: 1200

Of course, the scores are (almost) complete hokum.

But having a short meeting every fortnight where we discussed what was worrying the senior stakeholders? That was useful. And that's what this meeting allowed us to do. It was also a way of us gently raising our own concerns. But it was also a way of doing research into what is happening in the organisation. What's going on with the owners of the swamp? Very often, this isn't something that stakeholders will tell you straight out. But, for example, let's say that a risk gets added to this list which is something like "There is no funding beyond December" (and it's October). And this issue starts off with a high score. Discussing this single item, even if it's only every two weeks, tells you a lot about what's going on in the organisation.

If, over time the risk of no more money goes up – well, that's very useful information! If the risk goes down, it tells you that there's commitment for the project inside the organisation.

The knobs? So, what are the knobs and dials on having a risks and issues chart. Firstly, I think the score. The exposure score was a great start. And what I like about the number is that you can then put together a chart that has an overall number. If the overall "risk score" for the project is going up, well, that's something else to talk to the product owner and stakeholders about. Similarly, if it's going down, or even if it's all over the place. I haven't actually tried this (yet) but I think another kind of score might be better. Something that goes like this.

1 = Mildly irritating

2 = Irritating

4 = Mildly annoying

8 = Annoying

16 = Worrying

32 = Very worrying

64 = Bad

128 = Very bad

256 = Disastrous

512 = Catastrophic

1024 = End of the world / the project / the company

Even talking about what the names of these scores would be with the product owner and stakeholders would be interesting – it might be a way of getting them to name their worst fears.

But there are other aspects of the risks and issues meeting that can be varied. Who turns up to the meetings? Is it just the product owner? Or are there more senior stakeholders who would like to turn up? Maybe they can only come once a month? Who gets to see the chart? Is it just talked about in the risks and issues meeting? Or does it go in the weekly report?

Backlog refinement

There are lots of different ways of doings this. On any project that I'm working on, I'm expecting this to evolve through the course of a project. There is the "Three Amigos" approach. Several members of the team maybe a designer, a developer and a user researcher, get together once a week. They look at the project to do list. They discuss from the point of view of their discipline what might be the challenges with items in the to do list. Technically, is this possible? What else do we need to know? From a user research point of view, what might we need to explore before we develop it? From a design point of view, is there anything new and different? And of course, from a business point of view, why is this important?

In my experience, this might be needed early on. So that these points of view are made clear to the other "amigos." But over time, this can be streamlined, and the reviews of the project to do list might only need the product owner and the Scrum Master. Maybe. Think about it this way. Part of the work that you're doing on a project is figuring out what shape the work is in. Where does the work need detail and specification up front? Where can it be left to be figured out as it's picked up to be developed.

Which bits need to be explored using user research and testing? How much? Where? With whom? Which bits need a lot of design thinking and input. Which bits can be developed without much design input?

(On live services) Support board

So, you went through the whole heartache and pain of developing a service to the point that it went live.

Congratulations! Now you have to maintain it.

One thing I've found useful on a recent project is a regular weekly meeting where we look at the support tickets. These are the tickets that have been raised as a result of issues reported to a support email address. By the time we look at it, all of the frequently asked questions have been filtered out and replied to by the team that read the emails. What we're left with are either bugs or feature requests. Part of what we do in this meeting is decide which of these the issues are. If we decide something is a bug, we then decide on some kind of urgency for fixing it.

That's a *lot* of meetings Yes, I know that this is a lot of meetings. And I'm not suggesting that you need all, or any, of these. You might need a different meeting. In fact, what you need to be doing all the way through your project is inspecting and and adapting on your inspecting and adapting. Yes, that's right, you need to be continuously using transparency, inspection and adaptation to change these meetings if you see the need.

Artefacts So, you've got your four meetings. What else do you need?

For me there are two, (ok, maybe three or four) artefacts.

1. To do list items.
2. The to do list
3. Maybe a roadmap
4. Maybe a burnup chart
5. An emerging definition of ready and done

Stories At the beginning of a project, I'm very happy to start collecting a list of any things that need doing.

And I don't want to be too prescriptive about what kinds of things are allowed to go on that list.

Why am I not saying the "S" word? Why am I not saying stories? Not because I don't think you should call them stories, I'm fine with calling them stories. I'm also fine drawing distinctions between tasks, bugs and stories. Roughly a task is something that needs doing that isn't going to change the functionality

of the code. Make some changes to the cloud server architecture? That's a task. Run a one-off report from the database, that's a task.

A story is something that, when it's done, will result in the software doing something differently. "Add middle name to sign-up page" - for me that's a story.

Am I going to say that all stories should be of the form "As an X, I can Y, so that Z"? No, I'm not saying that. Why?

Because I think the team need to get a feel together for what constitutes a well-formed story. It's fine for them, together to agree on a story of the form: "As an X, I can Y, so that Z." It's fine for the team to come up with that as a suggestion - or for you to suggest it, if they're really struggling. It's not fine to insist on it. It's especially *not* fine to insist that everything is in this form. Insisting that everything must be a story rather than just a thing that needs talking about. That's not fine. Insisting that all stories must be of the form "As an X, I can Y, so that Z." That's not fine.

And drawing distinctions between stories and bugs. I'm also fine with this. Once you get going, this is fine. Roughly, a story is a change that the product owner wants to make to the way the software behaves to make it do something new. A bug is something that went wrong. The software doesn't work as expected.

So, "When I click on this button, I get an error" is a bug.

"Add help text that appears when I hover over this button" is a story.

And there are two other kinds of items that you might find in a backlog.

A task is something that needs doing which isn't directly going to affect the behaviour of the software "Book the accessibility assessment".

And there's a final type of story – "Epic". I've got to admit I'm a bit leery of epics. The idea is that an Epic is a BIG story (geddit), but also that an epic is a collection of stories.

The project to do list (or backlog) The only other thing that you *really* need is a list of things to do.

What you do in planning is that you look at that list of things to do with the team and the product owner and decide, all together, on a shorter list. The sprint to do list. These are the things that the team is going to do in this sprint.

And where do you put this list of things to do? To be honest, I've got a way that I like to do it. But I try not to force them on the team. This is tricky because most of these tools aren't free. They involve signing up to something and spending money. But I think the team should figure out all together what kind of thing they should use to communicate.

They might want to start out with a spreadsheet, but somebody will probably find out quite soon that that's too difficult to maintain. They might want to start out with a physical wall, but in these days of everyone working from home, I'm not sure how you could make that work.

I'm going to come out and admit that I like Jira. Well, I don't really like it.I curse it a lot. But it seems to be the best application for keeping track of the status of an issue. Also, it seems to be pretty good at keeping hold of the discussion that's gone on around an issue.

A roadmap I've found that roadmaps can be very useful. They don't have to be perfect to start with. What they do need is some kind of regular cadence to make sure that they're maintained.

Any kind of tool that can be used to make a Kanban board (Trello or even Jira) works for this. All you need is a list of the big things that need to go into the product. They you need a way of ordering them so that some of these things are being dealt with "Now." Then some of them are "Next." Finally, some of them are "Later." You might also have a column for "Done."

As I said, this is fine also, so long as it's maintained. If it's written once and then never looked at, it's just been thrown in the bin, it's a waste of time.

One occasion that I've found works well, for looking at the roadmap, is the mid-sprint check. I don't know quite why I've found that this works, but it does. After checking how things are going in the current sprint, the team, gets to lift their head up and look at the roadmap. Actually, what the team does with the roadmap leads me on to my nearly final artefact "A burnup chart"

Maybe a burnup It's interesting that this might be a bit of "Agile" practice that I like best. It's the bit that has worked well frequently for me. But it's also the bit that I'm a bit ashamed of and a bit nervous about. It's also something that I feel might get me drummed out of the Scrum Master guild or whatever the secret society of Scrum Masters would be called.

Here's the situation. We were six or seven months into a greenfield, brand new, project. There'd been some vague discussions at the beginning of the project about deadlines, but no definite date had been mentioned. Then, all of a sudden, there was an absolutely definite date. Fortunately, it was about twelve months away. But by that deadline date we had to have a system that could do enough of what the existing system did to turn the old system off and the new system on.

The minute I heard this, I knew that there were substantial features that we just weren't going to be able to deliver for the deadline. I knew that, if we were to have a chance of meeting that deadline, we would also need to drastically reduce, not only the list of features, but what we did for any particular feature.

In Agile language, what we needed to do was reduce the scope. Not just in terms of the numbers of stories we implemented, but also internally, we need to reduce the scope of the stories that we implemented.

But there was a problem. I'd tried to do this before. And I knew that discussions around scope reduction were always extremely painful. But it was something that I had to do if the project had any chance of being a success. I also knew that pointing out that there was no way that the promised scope could be delivered by a particular deadline had gotten me fired from jobs in the past. Yes, fired.

I liked this project, I really liked the people I was working with, I didn't want to be fired.

So, here's what I did, over the course of about three months.

I ran a workshop with the team and the product owner. We listed out all the features that we could think of that the product needed to do.

Then we did a "bucket" sort on the complexity of those features. Using the Fibonacci series.

The Fibonacci series is a series of numbers where the next number in the series is generated by adding the previous two numbers in the series. So, strictly speaking, the first two numbers in the Fibonacci series are 1 and 1. Getting started with the Fibonacci series is a bit weird. But once we get going, it's more straight-forward.

1+1 = 2

2+1 = 3

3+2 = 5

5+3 = 8

and so on. This results in this series.

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Of course, you can keep going. But for the purposes of the job I'm describing, these numbers are normally sufficient.

So, what you end up with at the end of this session is a wall (real, or virtual) that has big ticket items on it.

And all these big ticket items are grouped under Fibonacci series numbers.

This is the really important bit. You add up the total of the scores on all of the tickets. This gives you an overall number for the size of the to do list.

This gives you important number 1 – the total number of points in the backlog.

Then what you need to know is a rough order of priority for these "big ticket" items. A really easy way to do this is use a package like Trello to create a Kanban board with four columns and then label those columns "Done", "Now",

"Next", "Later". The next job is for either the product owner or the product owner together with the whole team to put those tickets in those columns. When you've done that, you have a sized and roughly prioritised road map.

Then every fortnight at the end of the mid-sprint check I would walk through the Kanban board. I asked the team to estimate the percentage completion of the things that were in the "Now" column. I multiplied the percentage complete scores that I got from the team by the fibbonaci numbers on the ticket items. And that gets me important number 2 – the number of points in the backlog completed in a particular fortnight.

Important number #2 is the number of points that have been "completed".

Warning: a lot of people will tell you that this is absolutely what you shouldn't do. I'm kind of worried even to admit this is what I've done. Even though it worked beautifully for me. And I will definitely try it again. This is the particular way that I fry an egg. Actually, this is the particular way that I fried an egg on a particular occasion.

Before you know it, another two weeks have gone by. You're sitting in the mid-sprint check with the team looking at their big ticket items and asking for estimates on percentage progress. This mid-sprint check and all the others that follow it start to give you important numbers three, four, five and six. So, 8 weeks into a software development project you have all the numbers you need to really cause some trouble.

You could even get yourself fired. Why? Because now you have the numbers to put together a burn up chart and a burn up chart gives you some kind of indication when the project might actually finish.

And guess what? The news is never good. the news is never "Oh look!" We're going to have this finished in about half the time!

After three or four sprints 6 to 8 weeks you have a pretty good idea of how much work there is and how quickly the team are getting through it. Of course, the first reaction of product owners and stakeholders will be denial. The second will be some kind of bargaining; actually this normally takes the form of some kind of bad arithmetic.

"Yes, but if you add all these numbers together divide them by this you get X. Then take away the number you first thought and add in the *special Mayan number for fate and good luck* you get Y. And Y is the number that shows we'll get everything done on time."

The more eagle-eyed amongst you will realise that what's happening here is that we're walking through the five stages of grief: denial, bargaining, anger, depression and acceptance.

In my experience the bargaining stage is where people say the maddest stuff. I used to argue with them. When they said, "we're still going to be able to deliver for the deadline. All we have to do is work twice as fast," I used to argue

with them. When they said, “I talked to this crazy person on the street. They said that this project is easy. They said you don’t know what you’re talking about,” or “my nephew put up a website in a weekend I don’t see why this is any different.”

Now what I do, is I just keep counting. Another two weeks you have another data point. You put that on your chart and things will become clearer. Now it’s obvious. You have twice, three times, four times – the worst I’ve seen is seven times - more work than you can deliver in the time required.

And that’s when the anger will come in. Why are you being so negative? Why didn’t you tell me about this before? “I thought I’d hired professional people who knew what they were doing. Clearly if this is going to take twice as long as I want it to you can’t possibly know what you’re doing.”

There might also at this point be an attempt to change the way that you count things so that things look better than they are. It’s quite important to resist this. How do I know? Because I’ve failed to resist it and then I’ve seen what happens when you don’t.

It was a long time after my first attempt at this manoeuvre that I read *Never Split the Difference*, by Chris Voss.

Now having read that, I realised what I’m trying to do. I’m trying to show the product owner and the senior stakeholders a realistic picture of the project. Then I’m asking them an open question.

What do you want us to do?

The difficult bit of this is dealing with the answers that come back that won’t work.

So we often start with denial.

“I don’t see what the problem is, if only you and your team could be more positive about this, I’m sure you could manage to deliver things on time.”

Then we get anger.

“I thought I’d hired a professional team? But clearly it looks like you’re not up to the job.”

Then we get some sort of bargaining. Often at first it’s a bit ineffective.

“If we let you have some help from our new intern, do you think that would speed things up?”

It may not sound like it, but depression is a sign of progress.

“I am totally regretting this project. This is the worst project I’ve ever worked on. OK. What do we do?”

And then finally. Finally, we get to acceptance.

"We can probably do without feature G at launch if we know that it's going to be coming very quickly afterwards. And we only need a tiny bit of feature C. If we just had these bits of feature D and feature E, we could probably manage with the old system for just a bit longer."

OK, that's the dream kind of acceptance.

What really happens is that ~~it~~ often ~~comes~~ (as I'm sure is also the case with grief about other, more important aspects of life). ~~Is that~~ the acceptance stage, if it comes at all, comes mixed up with anger, bargain, denial and depression.

"OK, we can schedule that feature for after release - but when I signed you guys up, I thought you knew what you were doing. And I'm still seeing some of you going home at about 5:30pm, looking relaxed, as if you don't have a care in the world. And Jeff? Yesterday, I saw Jeff took a whole hour for lunch! I would never have done this if I'd known it was going to take this long. Oh my God software is expensive. Are you sure there's no way we could just do this faster? Are you sure more people wouldn't help?"

That? That's what success sounds like.

Definition of Ready, Definition of Done What about a definition of ready? I don't have one. That is, I don't insist on one. Again, as I write this, I'm aware that it's the kind of thing that could get you drummed out of the Scrum Masters' guild.

The idea of the definition of ready is that it's a checklist of things that need to be in a story before it's ready to be planned in a sprint.

Trouble is, I don't have a degree in computer science, I have a degree in philosophy. Philosophers start with Plato. There are others before him, like there are other rock and roll singers before Elvis, but you get the idea. The problem is that when you read Plato, you get to see the trouble that he gets himself into trying to think about very basic ideas like "true" and "good." And a lot of that trouble comes from trying to solve the problem by coming up with definitions. This makes me very suspicious of an approach to solving problems that starts out with "first write your definitions."

So, I'm kind of fine with a definition of ready. I just don't want it to be written down. Or if it is, I want the word draft written across it and I want to revisit regularly. I want it to emerge through practice. I want it to emerge through a discussion between the people who are writing the stories and the people who are developing the stories. And if as part of the process of discussing what needs to be an story before it's ready for development, they realised that, the main this is that there needs to be regular conversation between the people who write the story and the people who develop the story, I'm more than fine with that. For anyone interested, this is a Wittgensteinian approach rather than a Platonic one.

I'm not saying that we should forbid anybody from writing things down. But if the team notice that they don't have anything to plan. And the reason for this is because none of the items in the to do list are "ready" according to their own definition of ready, well, I might suggest they don't focus so much on the "definition of ready". Rather they just have a chat about the things that are most nearly "ready". What would need to happen to them for them to be ready?

And so you, might have already figured out what I'm going to say about "definition of done"? Something very similar.

Concrete Practice – cook for yourself

On the projects that you work on, there will be a tweaked version of Scrum, that will perform better that the "ideal" one that is generally talked about. It might be the length of the sprint. The whole world seems to have settled on two weeks, but the initial books talked about a month. For some projects, the best sprint length might be a week. It's probably best if it's a whole number of weeks. I've seen projects that have tried a different number of days. An awful lot of time then ends up being spent trying to figure out when the current sprint starts and when it ends.

You can dispense with that "cadence" completely, or have daily sprints. I'm only really comfortable using daily sprints with teams that are working on support projects. That's when it's possible that urgent things might come in at any time that will need to be dealt with immediately.

So, the length of sprint. That's definitely a knob on the scrum. Once you realise that it's a knob, you can talk together as a team about changing it.

Another knob that you can turn is the depth of the backlog. What do I mean by that? I mean how long is the list of stories, issues, to do items, whatever you want to call them. How long is the list of things that need to be done on the project? On some projects, this list will be everything that anyone can imagine being part of the project. On some projects, it will be only a list of things necessary for the project to go live. On some projects, the backlog might be even shorter than that. While the current sprint is running, what's going to go into the to do list for the next sprint is still being discussed.

And of course, both kinds of backlogs have benefits. A long to backlog would be one that captures either everything that needs to be delivered, or everything that could be delivered. It gives a sense of completeness. It's also relatively easy to use backlogsto predict when a project is going to be delivered, and to negotiate changes in deadline or changes in scope.

Alternatively, working off a short to do list is useful for projects that are trying to negotiate uncertain times. This allows a team to change direction quickly and to incorporate new ideas as they emerge from research

We want reliability and manoeuvrability in both cars and projects. We'd also

probably like speed. But the fastest cars aren't the most reliable, or the easiest to steer. And it's the same with projects. A project that's set up to be manoeuvrable probably won't be the most reliable and the fastest.

There's also another knob on the to do list. Actually, there might be two. That's the required depth of detail and maturity in its stories. Sometimes this is known as the definition of ready.

How complex and detailed the definition of ready is, is something that can vary. It's a knob. On some projects the definition of ready may need to be *very* detailed indeed. Maybe all the content needs to have been combed by lawyers before it can be put in the story. Maybe each story needs a list of acceptance criteria and conditions of satisfaction. These need to be there so that tests for the story can be written at the same time as the story is being developed. Maybe the story needs to go through several different refinement stages before it's ready to be "developed." These could be user, research or design. For some companies, there might be also stages with risk and legal.

Maybe. But maybe a story can also be a single line. Each approach has its advantages and disadvantages. There is a level of detail which is "just right" for a story which will mean that the story encounters the minimum of trouble as it goes through development. But the definition of ready can get so complex that no story is ever "ready." And progress grinds to a halt. Alternatively, the description of the story can be so vague that no one knows what it means. Lightweight descriptions can also mean that *something* gets developed quickly. Users get to see working software. They get to use it and respond to it. They point out bugs. They are excited about some parts and bored with others. The stage is set for another, better, next iteration.

Beyond agreement about how detailed any given story should be, there's a separate discussion about the gradient of this detail.

What? How many stories are fully detailed? How many stories are partially detailed? How many stories have no detail at all?

These might seem like insignificant questions. But they're not. Actively managing the profile of your backlog is one of the best ways to use a backlog to make sure that a project can be delivered on time.

Conclusion

Everything about Agile is about facilitating the empirical process. The agile meetings are about transparency and inspection (standup, show and tell, retrospective) and inspection and adaptation (show and tell, planning).

Chapter SAFETY - Psychological Safety for you, your team and your Boss, for everyone

In a very early chapter [Agreed Activity] we talked about agreed activity. The phenomenon of doing something that seems safe. Doing something that seems safe, rather than doing something that helps a project and moves it forward. Then we talked about the antidote to agreed activity - creating psychological safety.

One of the things that bothered me when I was reading the Timothy Clark's book on psychological safety. In quite a lot of the stories in that book the boss didn't look good, in fact, the boss looked frightened. And the boss's fear was the reason that psychological safety didn't emerge in the team.

A thought drifted vaguely through my mind. If *everybody* has a need to feel psychologically safe, well then, *everybody* also applies to leaders and bosses. The thought also occurred to me that what constitutes psychological safety might be different for bosses than it is for team members.

Looking at things in this way provided me with a different way of looking at some of the very worst experiences I've had whilst working as a project manager. And it also made me realise if you're a project manager, you ignore the psychological safety of your bosses at your peril.

Concept

So, let's just remind ourselves what the four levels of psychological safety are that Clark talks about in his book.

1. Membership Safety
2. Learning Safety
3. Contribution Safety
4. Criticism Safety

And let's just take these one by one and think about what these mean if you have some kind of leadership role.

1. Membership Safety - showing the boss that they're the boss

It isn't enough to be a member of the team if you're the boss. You have to have everyone in your team recognise that you're the boss. But as with membership safety, this is just the first, fundamental level of safety that a boss needs. No one can usefully participate in a team unless they feel they are part of the team. And no boss can usefully provide leadership if there's any doubt about their authority.

This is a problem for bosses in software development for two reasons. First, all of the people who are actually doing the work know how to do the work better

than the boss.

Second, in software development, reality is your other boss. If you're doing it properly as a team (i.e. using Agile, iterative methods), you will soon discover the really hard problems that need solving. It will be very difficult to air these problems without undermining the authority of the boss.

And here we uncover a central contradiction of project management. It's one that you're not going to ever resolve. The best that you can hope for is that you're going to be able to manage it.

> The thing that you have to do to get the project done is the thing that you should never do to your boss (i.e. point out the fundamental problems with their ideas).

2. Learning Safety

Leaders need to feel safe to learn. But somehow part of the rules of being a leader is that they can't simply go to school. They can't take a course or sit at the feet of the expert who actually knows. So again, the team is faced with a paradox. They need to figure out ways to teach their leaders what they need to know. And they need to do this without flagging what they're doing as teaching. They can't make it too clear that it's obvious that there are certain areas where the leader in lacking in knowledge.

Of course, to some degree, the things that a leader is discovering about the project are the same things that the team are discovering about the project. What is easy? What is hard? What aspects of the project are genuinely appealing to users. What aspects of the project are totally uninteresting to users. Which other groups of people care about the project, or have regulatory power over it. But it's very possible that, for the leader, learning these things is not going to result in a safe experience.

The project is going to cost way more than the leader expected. The project is going to take a lot longer than the leader expected. Potential users don't seem that excited by it, or are actively hostile to it. It turns out that there is a regulator that, if you don't make expensive changes, could stop the project dead in its tracks.

Yes, this is learning. Yes, it is good to know these things, rather than not know them, if you really want the project to be a success. But sometimes, if not often, it's difficult to separate the experience of learning from the value that the learning gives you.

3. Contribution Safety

Leaders need to be able to contribute. From our point of view, as managers of software development projects, they need to be able to suggest things that they want the software to do. And they need to feel comfortable doing so. This is tricky. Because of the swamp. The people who live in the swamp, the people who own the swamp, the people who might want to live in the swamp and the

people who will never go to the swamp, but still care about what happens there. All of these people need to be taken care of to some degree. How also, to fit in the person who is in charge of the project?

4. Critical Safety

Perhaps this is radically different for leaders than it is for team members. Perhaps not. In this fourth level, where the leader is made to feel as safe as possible by the team. So safe that the leader can admit that they're wrong and can even be open to criticism from the team. Maybe you're never going to get to this level. Maybe you're only going to touch on it.

Unfortunately, what this shows us is that there is yet another side to the trade-off of project management. And this is something that I think I've got wrong, repeatedly, on several projects. What I think I've done, is used the bad news on the project as a way of "levelling" me up with he bosses. Of lowering the status of the bosses, of doing the opposite of providing them with psychological safety. This is a really dumb move.

The challenge is to deal with the pirate ship on the horizon whilst at the same time not rattling the psychological safety of your boss. It means that the role of project manage isn't that of Jerry Springer – you aren't just televising people's problems. You're making sure that the bosses are aware of the problems. You're keeping those problems, in the consciousness of your bosses. Using Daniel Kahneman's principle of "What you see is all there is." You want to be creating artefacts and having meetings where these problems are seen. Especially, if they are problems that only your bosses can fix. What you're trying to do is to engage the problem-solving mind of your bosses.

One of the things that makes this so difficult is that there is always money is saying that everything is fine. A method like SAFE provides psychological safety for your bosses, but it's an illusion of psychological safety.

Chapter 12 – How to kill your boss

OK this is what I wanted to say and I'm just going to say it.

I think this book is described in an arc.

I think when I started this book, I was angry at bosses.

And this is something that I know intellectually is a stupid idea and a bad idea and I need to get over it but that doesn't stop me feeling it. I feel it viscerally.

How do I know it's a bad idea? Because of the doctor-patient lawyer-client thing

Doctors see sick people. Lawyers, criminal lawyers especially, only see people who have managed somehow to get themselves into trouble. Most sick people and most people who have been charged with a crime have probably got some opinions on what needs to be done. They think they know what needs to be done

to get them out of their predicament. But at the same time, they've decided to consult a professional.

Maybe part of the reason that I have shied away from this in the past is that I don't want to get too pretentious about the role of project manager.

This isn't me saying that being a project manager is anywhere near similar in terms of education and skills. This is me just pointing out now there's a similarity in this dynamic between patient and doctor, client and lawyer.

The people who come to you and ask you to manage their projects are in trouble. Do they know that they're in trouble? That will probably vary from case to case. And it's your job as a project manager to get them out of trouble.

And do you know one of the ways that I got to this understanding? I consulted a coach. Yes, I put myself in the same kind of relationship with someone else and I talked through with her the problems that I have had and the problems that I'm having. Out of these discussions, two key ideas have emerged.

Status

There is an in-built status relationship between a client and the professional. The professional has to at least pretend that the client is the boss.

This has often bothered me. I'm sure that there's some plain vanilla psychological reason why people asserting their authority over me annoys me so much, but it does. I mean, I think that it probably annoys a lot of people, but I also see a lot of people that seem to be dealing with it better than me.

And here's the mistake that I've made repeatedly. I've used bad news as a way of trying to lower the status of the client. And let's just think about this in doctor or lawyer terms. Do you think that that would work? Of course, it doesn't work. It's a disaster. It makes the entire discussion about the client reasserting their authority.

What's the solution? Never use how bad things are as a way of elevating your status (really, this is attempt to lower the status of your boss). It will depend on the boss and the circumstances, what status they will tolerate from you. Some bosses very definitely want the people who work for them to behave exactly like "people who work for them", all the time. They never relax. Some bosses want the people who work for them to be almost like equals. It will take some experimentation to know exactly where the right point is with any given boss. Some bosses want one, but pretend they want the other.

What's always going to be difficult if you're a project manager and you're doing the job right is this. The "right" point for doing good things for the project is different from the "safe" point. The "safe" point is forlock-tugging and just doing as you're told. That's the same as mopping the deck. It's agreed activity. It's the same safe feeling, but it's actually dangerous behaviour.

Timing

I'm in the timing business. Delivering things on time. That's what I do. Right? Well. Right and wrong. Just because I'm in the timing business doesn't mean that I can force my timings on the bosses. To force my timings on the bosses will always be seen by them as an attempt to lower their status.

And this is a tricky one. Let's say for example that some boss tells me that they want something by a particular date. Christmas is the one that comes up most often. Bosses always want things by Christmas. In my head, the minute I hear this, I'm thinking of a bunch of things that need to happen if there's a ghost of a chance of this happening by Christmas. And a bunch of things are things that the boss needs to do. And the boss has just said Christmas, so, if that's a real deadline, he needs to do these things by a certain time. Right? Nu uh. No.

You would think it would be like that. It ought to be like that. It isn't. It isn't and it never will be.

Open questions

I don't think there is much in the literature to help with this. There is mention in the writings of Robert Anton Wilson of something called "the snafu principle."

"But a man with a gun is told only that which people assume will not provoke him to pull the trigger."

This is the problem put most graphically, and most brutally.

And you might object that managing most projects doesn't involve any actual violence. But I've said the wrong thing, or nearly the wrong thing, to the "man with the gun" several times. And several times it's resulted in me nearly being fired. Once it did get me fired. And being unemployed, while not directly and automatically a violet experience, can still be a tough one.

So how can we possibly deal with this? This is the fundamental problem. This is the abyss right at the centre of project management.

We need to have honest conversations with our bosses. We don't ever want to tell an armed man something that will make him fire his gun.

OK. Let me tell you some things that I know have worked. Then I'll tell you some things that I *think* might work. Even though I haven't yet exactly figured out how to apply them in what you'll see is a slightly different context.

OK. What works?

Showing actual progress.

OK, here's the situation. I was working on a project. It was really important that the project succeeded. There was probably more at stake on this project than almost any other project that I've worked on.

When we started the project, I don't remember that it had a deadline. Actually, that's not true, it did have a deadline, but it was a deadline that was so ridiculous that nobody took it seriously. We were going to have to build an entirely new system to manage a very complicated licensing function for a government department. The initial deadline (I mean how did anybody keep a straight face?) was seven months. This was obviously impossible. But the contract with the current supplier was running out and ideally, for the organisation, they would have loved to stop at that point. But still, it was impossible. The fact that it got mentioned at all tells you how powerful organisational imperatives can be, even if they are completely at odds with reality.

Then the deadline suddenly became the old deadline plus 12 months. I wasn't part of any discussion around this. And there was a solid organisational reason for it. The old contract could be extended for 12 months in a relatively straightforward way. But after that, a new contract would have to be negotiated with the old suppliers. And the outgoing suppliers weren't happy that they hadn't won the new contract. They were making expensive noises regarding keeping the old system running beyond the 12 month extension.

So, it totally made sense for that to be the deadline.

There was just one problem. It was a big problem. To turn the old system off, we needed to be able to turn the new system on. The new system needed to be able to do enough of what the old system did to allow us to do that. Which means the new system had to already be running and have been tested with some users. This was a licensing system. The licences were licences that granted to the owners' permission to do highly controversial things. It needed to be very secure. We had to be confident of this. But we also had to persuade security people inside government of this.

When I heard about this new deadline, I realised that it was probably very difficult to move. I thought of all the things that we'd have to do before we got to the point where we could turn the new system on and turn the old one off. But I also thought of something else. We couldn't possibly do all the things that we had talked about doing before the deadline. At that point, we didn't even have a basic working system for doing the main thing - form submission - that the system was supposed to do. Rather, we'd been concentrating on seeing the world through the second metaphor that I've talked about above, "exploring the swamp." We'd done lots of user research, we'd developed a good rapport with most of our users. But we'd done almost nothing to limit the scope of what we were going to do relative to what we'd found out that our users wanted us to do.

As soon as I heard the deadline, I knew that we'd have to do three things, and we'd have to do them relatively quickly.

1. We needed to put together a release plan. It needed to show the dependencies between the security certifications and the testing we had to carry out.

2. Come up with a strategy for limiting the scope of what we agreed to deliver so that we had reasonable confidence that we could deliver it in time.
3. At the same time, sooner rather than later, put together a piece of skeleton working software that did the very least that the system was supposed to do.

I was really worried. My rough guess was that we had probably 2-3 times as much work that we'd identified, compared to what could be delivered in the time. But that wasn't the worst thing. The worst thing was that we didn't have a strategy for reducing scope. We didn't have any way of stopping the mountain that we had to climb from simply getting bigger and bigger..

Understanding the status impact of bad news,asking open questions

Right at the start of this book, I talked about improvisational theatre. My experience of improvisation classes is that whenever I do them, I learn something. One thing I realised about this "sticking point" – of delivering bad news to bosses is that I wasn't going to get around it, over it or above it without some experimentation.

So, I decided to run some one-on-one improvisation sessions with actors to try to really understand the ins and outs of the business of delivering bad news.

These are the immediate impressions that I got from running several sessions with an improvisation partner who I asked to play a boss. Let's call him Dave.

1. Definite bad news is easier to deliver than hunches. This is a problem, because identifying problems when they've just cropped up, on the very edge of the radar, is probably the most valuable time.
2. It's an interesting question, how to find naturally high status people that can take part in this kind of improvisation. Dave wasn't naturally high status. This is just a sample of one, but he was naturally low status (at the end of the call, he even said that it was OK if I didn't pay him).
3. Getting people to say nasty things about other people is really, really difficult (or I'm not doing it right). Also, when you've only just met someone, they tend to be remarks about appearance,. I'm wondering if the way around this is the Cyrano de Bergerac strategy i.e. go through all the possible kinds of insult. And also, grade the insults, also possibly just give the person that you're improvising with a set of insults on cards.

E.g.

You're saying this because you're racist.

You're saying this because you're sexist (this one has happened to me in real life).

You're just saying this because you're old and you don't have the balls to take on a challenge anymore.

You're just saying this because you're young and inexperienced.

If you're not up to the job, just say, I'm sure I can find somebody else who is.

If you're saying this, I'm wondering if you're competent.

That's exactly what someone who's trying to take all my money would say.

Why are you always so negative?

You can't possibly know that.

I heard from somebody else that this was going fine.

How long have you known about this? Why didn't you tell me about this before?

But if things are so bad, why aren't your team working longer hours to fix this problem?

Your team don't seem to be working hard enough, they don't look worried or stressed.

4. Asking open questions.

According to the Chris Voss author of *Never Split the Difference*, the secret to negotiating with kidnappers is to ask them open questions. The bosses in these kinds of situations are sort of like kidnappers.

I was doing this today. But I think there are important differences between three different versions of the same questions.

a) What are you going to do?

b) What are we going to do?

c) What do you want us to do?

d) As a team we'd appreciate your advice on this matter.

5. Wanting to be smooth

This is a feeling that I had as we entered this final phase. That Dave - my co-improviser, who was playing the boss, was trying to find any "hold" that he could over me. And I wanted to be "smooth" to not have any fault that he could catch hold of to gain the upper hand.

But as I'm writing this now, I'm wondering if this is quite stupid. What are you trying to do from this kind of difficult conversation?

What you're trying to do, this is what you *should* be trying to do, is to get the person that you're talking to, to think. We've already figured out that telling senior people bad news is status lowering, so they will naturally try to regain their status. This is tricky. Because you need a certain level of status in order

to continue to make them think. But is that the same as the status you need for self-respect?

So do you need to resist this "wanting to be smooth" feeling? Do you need to let your boss find some way to get a hold over you? I think you do.

I'm wondering about the time that somebody suggested that a member of my team was committing fraud. They thought he wasn't working full hours, when in fact, he was turning up early and leaving early and this had been agreed.

What possible responses were there to this accusation?

1. Anger. This is how I responded, pure, white, incandescent rage. "How very fucking dare you? We agreed that Jerry was going to come in early and leave early, you know that." Needless to say, this didn't go down well.
2. Serious concern. Really? I'll check with Jerry, but I think he's been coming in early. That's what we agreed. I'm very sorry that this has come up as a concern and we definitely need to check.
3. Overkill and an attempt to gain the upper hand. "Really? Fraud, that's a very serious allegation? Should we call the police? Oh? You don't think we should call the police? Well, exactly how serious do you think this is then?"

It took me a long time to realise that tit always has to be number 2.

The more psychological safety you can get in your team. The more successful it will be.

Chapter COMMITMENT - Commitment and Consistency and Your Other Boss

We are driven to make sure that our actions consistent with what we say. We have a powerful urge to keep our promises and commitments. But if we don't understand what that really means when for project management, we'll probably break the project and make ourselves miserable in the process.

> A foolish consistency is the hobgoblin of little minds
>
> Ralph Waldo Emerson

Concept

In general, as human beings, we like to keep our promises. When we say we're going to do something, we like to honour the commitment. Also wee don't like

our behaviour to look like it's all over the place. We want to appear consistent. If we tell people that we're vegetarian, we don't want to be caught eating a chicken burger the size of our head.

This is one of the most powerful principles that controls our behaviour - it's known as commitment and consistency. We like to keep our promises - commitment. We like to behave in a way that looks like it's in line with what we believe - consistency.

This is, on the whole, a good thing. It's a good thing when people do what they say they're going to do. It's a good thing when we behave in a way that fits with what we say we believe. On the whole.

But project management isn't the whole. Project management is an unusual activity, not like most of the rest of our life. And this is especially true of project management when it comes to software development? Why?

Connections

Do you remember the swamp? That's why. Think about it just for a second. What if I'm right about the swamp? What if I'm right that we don't really know what we're going to find in there until we actually go in there and start digging? And what about those people who care about what happens in the swamp, that live nowhere near it? Didn't we admit to ourselves in the last chapter that the only way to understand what's going on in the swamp is to explore. Talk to people who might want to live there. Talk to the people who already do live there. Start to build up a picture (a map) of who the stakeholders are and how they relate to each other.

Didn't we also admit that the only way to understand what we can do in the swamp, and what might be of any value, is to actually do stuff?

Translated back into non-swamp language, what does this mean for software development? It means that we need to do user and stakeholder research and we need to deliver working software. We won't really know what's possible, valuable, legal and attractive (i.e. what has value) in a new area until we've done that kind of exploration.

But guess what? Guess when as a team, as a project, we're often asked to say what our software is going to do? Guess when we're asked, how long it's going to take? When are we expected to even know how much it will cost? And what value it will deliver? Not after we've done all that exploration of the swamp, but before we've even started.

Concrete Practice

I think I had just been reading Robert Cialdini's book *Influence: The Psychology of Persuasion*. That was where the problem started. I hadn't been working for this company long. It was a small company with fewer than a dozen people

working on that many projects. I'd been brought in as their Scrum Master. We were having a meeting on the phone, almost at the start of a new project. We'd discussed some of the things that needed doing to move the project further forward.

It wasn't the kind of meeting that I liked. It wasn't an Agile meeting. There wasn't just one kind of thing that we were talking about. It was more free-form and rambling than that. But it was with the chairman of the company, and this project was his baby, so I didn't feel I could interrupt too much. The meeting ran long (as meetings that don't have a single specified purpose and aren't time-boxed and aren't facilitated by someone who cares about these things are wont to do). But then, just when I was hoping that the meeting was going to be over, just when the chairman gave me hope by saying "Right then!", he snatched that hope away.

Instead of stopping the meeting, he said, "Let's go round the table and I want you to promise the team that the actions that you've been given will be done in time for the next meeting."

What? Promise? I could hear the other members of the team around the virtual, conference phone table solemnly committing to do their actions. These had been captured during the meeting and written down with team members' names against them. What was I going to do when he got around to me?

I knew that the actions that had my name against them were vague. I knew that I could probably have some story to tell that they were done by the next time we had a meeting. At the same time, I knew something else. I knew that some members of the team were committing to things they had no business committing to. Either they had no chance of doing them by the next time we met, or they simply had no idea whether they could do them or not. They weren't committing because they thought they could do the tasks. There were committing because they wanted to avoid the difficult situation they would be in if they refused to commit.

Also, in the back of my mind, were some things I'd been hearing about how this guy, Mr Chairman, was working with the rest of the team. He himself worked from home most of the time and got up at 4 am. I had vaguely heard some worrying discussions in the stand-up meeting. He'd been getting some of the other, younger members of the team to have meetings at 5 am. He had also been encouraging people to work through the night to meet his deadlines.

You know what? Fuck that.

So, when my turn came, I said, "I'm sorry, I can't commit to doing my actions from this meeting. I'll do my best obviously. But some of those things, they're just too uncertain."

And. Oh my god! Mr Chairman put the phone down! I really wonder what would have happened if it had been a face-to-face meeting.

Now I'm wondering if I should have told this story. Because I'm not exactly sure what the moral of it is. I think a bunch of things were going on at the same time to make this go so badly. We'd only just started working together. The chairman wasn't to know that pushing to people to make undeliverable commitments was a hot button issue for me.

Getting people to make commitments that they couldn't possibly honour was clearly a management approach that had worked for the chairman before. Really, in the Tuckman level stages of "forming, storming, norming and performing" we were definitely in the storming phase.

I suppose the real take away from this is that for many people who are used to "getting things done." Getting people to make commitments that they are in no position to make is pretty much the beginning and end of their management strategy.

Actually, since then, I've come to realise that things are actually worse than that. To give the chairman his due. At least he was trying to get us to hang ourselves with our own words. In many other situations that I've been in since, the boss tries to use the power of commitment and consistency *even when no commitment has been made*!

Sometimes this takes the shape of what I find myself calling "commitment and consistency Jujitsu."

Normally this is something like "You committed to A, and I took a to mean, B, C, D and P, Q, R – oh and also X, Y and Z."

Sometimes it's just flat out brute force rank-pulling "I've committed to this, so you've committed to it."

I used to think that if I were going to write a book about how to successfully deliver software development projects, I would have to have some kind of fiendishly clever way of diffusing these claims that a commitment has been made. That this would the point where I tell you how to avoid being on the hook for a ridiculous commitment.

Here's what I know. I don't know how to do that. Rather what I've realised is, that, like death and taxes. Stupid deadlines are always going to be with us.

Conclusion

When I run Agile training courses, I do an exercise at the start of the very first day, just to get people out of their seats, standing up and talking to each other.

I ask the group to split themselves into smaller groups. I then give each group a pad of flip chart paper and some pens and ask them to draw a "stick person" in the middle of the piece of paper. Then I ask them to give that person a smiley face. Then I ask them to write down as many words and phrases as they can which they associate with a "happy" project - with a project that is going well.

Almost always, the first thing that gets written down is "On time, to budget." Do you know what almost never gets written down? I might have seen it twice in ten years of running training courses. "Valuable to its customers." Do you know what gets written down even less often? I think I've seen it once in those 10 years. "Makes money."

Maybe this is because I'm asking the wrong question. Maybe if I talked about a happy product rather than a happy project, I would get better answers. But this simple exercise shows something project managers already know. People think that project management is almost entirely about keeping promises. They think it's about being on time and keeping to budget. They forget what's actually important. Whatever the project is producing needs to be valuable to somebody. Sometimes that value isn't cash. Often that value isn't just cash. Of course, in a lot of cases, it would be good if the project made money. And if it did, it wouldn't matter, or certainly wouldn't matter so much, if it were late and over budget.

For me, commitment and consistency is the electric cattle prod of project management. People who don't have experience of delivering projects often think that keeping promises is the most important thing, if not the only thing. They forget about "valuable to customers", they forget about "makes money." But they also seem to think that it's the best and most powerful way of making the people involved in a project do their bidding. They are certainly right that "commitment and consistency" has power.

It's so powerful that it still has power, even when it's been used against people who never made a commitment. It's still powerful even in situations that aren't consistent. This is what I call "commitment and consistency Jujitsu."

For example, I was recently in a discussion about a new feature for an existing, successful, product. The owner of the product - the guy who paid the bills - wanted this new feature. He hadn't mentioned it before. But now he was saying that it was really important and that he'd promised it to clients at a meeting.

The logic of what he was saying was this: "I've promised this - so you have to deliver it." But of course, this is, totally bogus. Just because somebody else has promised something, doesn't mean I'm on the hook for delivering it. Then again, this guy is my customer, he pays the bills, he's my boss, I want to help him if I can. And the force of this argument is "I've promised this, if I don't deliver it, I will appear to be untrustworthy, don't do that to your boss."

And unfortunately, as a project manager, I have another boss. Reality.

We don't know what this new feature really is. Because we don't know what it is, we don't know what it involves. Maybe we can deliver it in that time. Maybe we can't. We're certainly not in a position to promise anything.

What is happening here is that my boss is trying to use commitment and consistency jujitsu to get some extra software done. This software is a change to

what we've already planned. But that's fine. When we're using Agile, iterative methods, we accept change, even late in the day.

But this is a "I promised this, so you promised this, so it has to happen" argument. It's also trying to avoid something. Let's say that we have "promised" this thing that we only just found out about. We're going to have to *not* do something else. And it's very possible that the other thing, the thing we're not going to do, has also been promised.

Actually, I'm very careful not to promise things unless I really have to. So more likely, it's been discussed, it's been seen on plans. What we will have said will be something like "this will be the next thing that we do, unless priorities change." But rest assured, what somebody will have heard is "we promise to deliver this by a certain date."

What this boss is really trying to do is talk himself into a situation where he can have his cake and eat it.

Fortunately, the boss in this case is a reasonable person. He acknowledges that we can't do both of these things at the same time.

But lots of bosses aren't that reasonable. Lots of bosses would try to avoid making this connection.

The poem that keeps coming into my head when I'm thinking about deadlines and commitment and consistency is "The Hound of Heaven." Which is about the poet trying to avoid the inevitability of his own death. As we've just discussed, death is inevitable, I think I've realised that bosses playing silly buggers with deadlines is also inevitable. But do you know what's also inevitable?

Criticism

> Well, none of this would be a problem if you did what you said you were going to do.

Yeah. OK. Actually no, this is not OK.

Here's why. What I was always going to have to do was explore the swamp. In order get funding to do the project, despite all of my best efforts, I might have been manoeuvred into saying something that sounded like a promise.

When me and my team started exploring the swamp, we found stuff that we could never have imagined would be a problem. Or maybe, when we started exploring the swamp, we found stuff that the customer already knew was a problem. They knew it was a problem, but they kept quiet about until after we'd made our commitment.

So, I don't agree with this statement. By agreeing to do this project, we agreed to encounter problems. We might not have said that out loud. But we did. By agreeing to this project, we agreed to the possibility that we wouldn't be able to do what we said we were going to do because of those problems.

Confessions

It still hurts. I can tell myself all I want that being attacked by some senior manager for not "delivering what was promised" shouldn't affect me. But it does.

I felt the sting of this kind of attack only recently. We'd been asked to do a follow-up piece of research to a feasibility study. In the proposal that we'd put forward, we'd suggested that we'd be able to get some working software together within two months. Note to self. No matter how short the engagement, always deliver *some* working software.

The initial feasibility study, which our bid was based on, wasn't quite as described. It was supposed to be a survey of possible users across the industry. In fact, it was mainly a survey of, well, possible users inside the survey company. Note to self. If you're promising work on the back of something that's already been done, make sure that you caveat your work. Say it will only be done if the other piece of work is as described.

Even though the boss admitted that there were problems with the initial study, he repeatedly brought up our original commitment. He repeatedly used what he portrayed as our broken promise to get more free work out of us. And even though I knew what he was saying was complete rubbish, it still hurt.

I would dearly like to avoid this "commitment and consistency" dance. I would dearly like for it not to hurt when people accuse me of breaking my promises. I would like people to stop using contorted logic to supposedly show that me and my team have broken our promises.

But I've realised that it can't be avoided. Most projects simply won't get off the ground and won't be approved unless some kind of promise is extracted from the people who are going to deliver.

Chapter CONVERSATION - The Conversation

There is no point shouting at our clients, there is no point shouting at the team. Similarly, there's no point keeping quiet when we see that things are going wrong and if our clients avoid us, it's pretty much game over. Successful project management is about continually, actively working to keep the conversation going.

> The only successful way of rendering this disturbance inoperative is to keep on breathing quietly and unconcernedly, to enter into friendly relations with whatever appears on

> the scene, to accustom oneself to it, to look at it equably
> and at last grow weary of looking.

Eugen Herrigel - Zen in the Art of Archery

Project management is all about improving the quality of the conversation. That's the idea that I want you to passionately believe by the end of this chapter.

This is the realisation that I've come to. I've had hints of it in the past, but writing this book, working to communicate what I know about project management, has made me much more certain.

We're all probably agreed in theory that communication is a good thing. But we're also probably agreed that not all communication is of the same quality. There's a Monty Python sketch that involves performing the novel Wuthering Heights using semaphore flags. What that graphically shows is that some communication, for some things, is probably better suited than others. There are good and bad kinds of communication. But for project management, what *kind* of communication is a good thing? What characterises it and how can we make sure that that's the kind that we use in our projects?

And there's another, related question. If there is a kind of conversation that's a good kind, why is there so much bad communication?

What kind of conversation is good communication? The ideas that I'm going to talk about mainly in this chapter are based on the work of Patsy Rodenburg and her book *Presence*.

Rodenburg's idea is that there are three basic kinds of communication. She calls these first, second and third circle.

The first circle is going inwards, it's introspective. First circle is either not talking or talking to yourself. If first circle is talk at all, it's probably mumbling. It's either talking quietly, or using a language that only really makes sense to the person who is talking. First circle is quiet, timid. Someone who is in first circle might look like they're listening. But in some fundamental way, they aren't. They aren't engaging with what's been said.

The opposite of first circle is third circle. Third circle is extrovert. It's shouting and pushing. It's presenting and performing to others. It is telling people things that you want them to hear. It is showing people things you want them to see. Third circle is loud, it is flamboyant and possibly aggressive, possibly aggressively friendly. It doesn't want to hear what other people have to say. In fact, as we'll see, that's a large part of the reason for third circle.

The second circle is genuine dialogue. Each side is paying attention. They are being changed by what the other side says. Each side is coming to an understanding of the other's position. That doesn't mean that everyone in second circle accepts that the other side's position is right or true. But if two sides are in second circle, they at least accept that the other side has a position.

From the point of view of project management, by engaging in second circle, you can start to understand what's valuable to your client. When you do that, you can see if there are things that you can offer, even if these things aren't valuable to you. This is closely related to the discussion about the swamp that we had earlier.

By revealing the current situation through discussion, you can manage to make something magical and alchemical happen. You can get your "opponents" to negotiate, not with you, but with themselves. Again, this is closely related to something that we've already talked about - the pirate ship. There is bad news on the horizon. It needs dealing with. The crew by themselves can't deal with it. They need insight, strategy and decisive action from whoever is in charge.

Whatever the problem is, it is very dangerous to ignore it. And, although everybody might feel better for a while, it's also dangerous and ineffective to rail against it, and just complain that the world shouldn't be like this.

So, here's my thesis about what project management is about: it's about working to improve the quality of second circle conversation.

Understanding and accepting this can be good news if you're a project manager. Because if you accept this, your job description changes. Your job is no longer about delivering the impossible. It's about improving the conversation between the people who are asking for something that's, on the face of it, impossible and the people who are trying to deliver it. Of course, there's a secondary goal of trying to improve the communication enough so that what's being asked for starts to be something that's possible. Because possible things are deliverable, and so actually capable of delivering value in the real world. But since the only way to do that is to keep improving - and maintaining second circle conversation, that really isn't the *main* concern.

My conclusion is that if you focus on the quality of the conversation, you can be comfortable working on seemingly impossible projects. Why? Because what you are doing when you're working on these projects is focusing on keeping the conversations in second circle.

Projects will only work if there's genuine dialogue between the team and the client. Now obviously, all the practices that we've talked about in this book are still important. It's important to understand the swamp. Who owns this area? Who's interested in this area? Who lives in it? Who works in it? Who regulates it? What unexpected value have you unearthed through exploration?

It's important to work in a way that encourages transparency, inspection and adaptation. It's important to call out the problems, the pirate ships, which are visible on the horizon. These things are vital to improving the quality of the conversation.

In every organisation, and maybe every project, the nature of this conversation will be different. You will need to start from the beginning. Sometimes things will be familiar enough from other projects that you can start to pick up speed.

Sometimes everything will be strange and different. Sometimes, it will seem like nobody will want to talk to you.

But in every new project and in every new organisation, there will always be work that needs to be done. There will always be effort to figure out how and when it's best to have dialogue, second circle conversation.

So, if the solution is second circle conversations, why don't people just stay in those kinds of conversations the whole time? If that happened, wouldn't so many projects be so much more successful?

The reason people don't stay in second circle the whole time is that it can be dangerous and exhausting.

Why is it dangerous? Because being in second circle is revealing and the more people know about you, the more likely it is that they can use what they know to harm you. Even if people don't intend you harm. Second circle is revealing. It's revealing of your real values, your real motivators your real strengths and weaknesses. Instinctively we avoid being this revealing.

Why is it exhausting? Because, when you're in a genuine give and take dialogue, you have to think. You can do first circle in several different ways. You can simply be quiet and not listen to what others are saying to you. You can listen to what others say, but not engage with it. You can even just accept whatever others say and cease to have any opinions yourself. None of these options involve anywhere near as much effort as trying to remain in second circle and engaging.

Engaging means asking questions, not only of the people that you're talking to, but of yourself. Questions like, am I right? Am I seeing this straight? Are my intuitions valid in this situation? What can we do about these issues that seem to make this project impossible? What can we do with the contradictory behaviour from the client? How can we deal with their insistence that progress on the project needs to be faster whilst at the same time they are refusing to help us remove the "bricks without straw" problems?

OK, let's say you're with me to this point. So, what are the concrete actions that we can take to make sure that we're talking reasonably in second circle with our clients and not yelling or mumbling?

First of all, it's important to understand that all of the Agile ceremonies create opportunities for second circle conversations. Notice that I say that they create opportunities. They don't guarantee it. In the chapter where I talk about "Fried egg Agile" I would say that most of the ways that I've modified the way that I do Agile are connected with improving the conversation, or creating artefacts that can be used to improve the conversation.

It's a very good way of seeing the Agile meetings - as opportunities for conversation. Blockers that come up in stand up? Opportunities for conversation.

Questions from the product owner as to why the team haven't planned as much as they had imagined? An opportunity for conversation.

Surprise, shock or horror at a demonstration of working software? This is certainly an opportunity for conversation.

A recurring theme in retrospectives about some infrastructural, "bricks without straw problem?" This is also a chance to have that talk.

Count things and then use the numbers as conversation pieces.

Count things? Count things like what? Here are some things that have worked for me in the past.

How long does it take for a piece of work to get from the point where it's agreed that it should be worked on, to the point where it is in the software?

How much work is the team completing on average in a two-week period? When does that mean the project is likely to complete all the work that's needed?

How long does a piece of work, on average, have to wait for an expert opinion before it can be worked on? How long is the list of things that are currently waiting?

What's the rate at which the budget is being spent?

All these numbers are the kind of numbers that a Lean approach to manufacturing and development suggest you count.

Of course, carrying out user research at any stage in the project is actively having second circle conversations with people who might use the software. And then reporting research findings about current users, possible users and other interested parties to the client is another crucial set of second circle conversations.

Discussing these things in second circle will probably increase the chances of a project being successful. I'm putting this mildly. It will dramatically increase the chances of the project being successful. In fact, it's probably the only chance.

Discussing these things also brings with it the chance of a first circle response from the client (ignoring you) or a third circle response (shouting at you). And that in turn, I'm ashamed to admit it, leads to similar kinds of reactions from you as a project manager, or your whole team.

I've reacted many times to first circle reactions from the client with third circle responses, asking awkward, angry questions in meetings, putting the pirate ship issue that the client doesn't want to hear in reports with a wider audience.

I've also lost my temper when either me, or my team have been addressed in shouty third circle. Many times. And I don't feel guilty about these reactions. I feel shame. A more powerful emotion. Guilt is about what you've done. Shame is about who you are. And when I think about these things, in the past, I've felt shame at who I am. I'm the kind of childish, uncontrolled individual who reacts emotionally to attacks.

But two things have started to make me feel better about the responses I've made in the past. Seeing these things through the lens of first and third circle

helps a lot. It has helped me to progress to a feeling of, not shame, not even guilt, just a kind of old fighter's acknowledgement of those times when my opponent made a move that I in my naivete couldn't counter.

The second thing that has helped, is to realise that when I've reacted with anger (third circle) or silence (first circle) to some kind of response from the client, I'm doing what they want me to do. I'm responding in a way that is fine with them. Almost always, in these situations, the way that they want me, and my team to respond is by shutting up and not bothering them with the problems that they need to give some attention. Sometimes they want us to respond by faking an "agreed activity" of running around and looking busy and getting visibly stressed i.e. "working hard."

Because you're the supplier and they're the client, because they're senior and you're junior, they don't expect you to get angry. At least, they don't expect you to show that you're angry. But it's really no skin off their nose if you do.

Here's the kind of dialogue that can happen.

PM: We can't really make any more progress on the payment function until we get details of the transaction database. [Second circle]

Client: I'll put you in touch with Jeff.

PM: Jeff won't talk to us. He's ignoring our emails and our calls.

Client: I'm just tired of your excuses and your negativity. I thought I'd hired a team who could get things done. Maybe I need to get that team over there from X Corp to look at this, they'll probably be happy to take over the work.

This is third circle. It's angry and shouting. And the purpose of it isn't to solve the problem, it's to stop you talking about the problem.

The mistake that I've made in the past with this kind of response is to deal with the content.

PM: I think if you asked X Corp, they would say the same thing.

Also, I would get angry. I would get angry at the threat to take the work away, and I would get angry at the suggestion that me and my team are either being dishonest or don't know what we're talking about.

Now, I think I would try to keep the conversation in second circle. I would do my best not to get angry.

PM: We can't see how we're going to solve this problem without access to that database. I think if we could just talk to somebody from the team that is building the transaction database, we can get a better idea of when it's likely to be ready. And if that's going to be a long time, we can maybe start to talk about what we can do without it.

I don't expect the client to behave well or rationally in response to this. I don't think it's impossible that the client will go and talk to X corp. If you were the

project manager are you scared now that they might go and talk to X Corp? Is it going to shut you up? Because that last thing the client wants to do is to go through all the rigmarole of changing suppliers, but what they do want you to do is shut you up so they don't have to think. Of course, losing the project to another supplier is really scary. It's perhaps made a little bit better imagining that this isn't just one conversation. You can back off and come back another day.

Here's where I'm going make an admission. And it's quite a big one. I don't think I'm that good at second circle communication. Second circle communication, especially between me and clients - actually, it's with everyone - is something that I continuously struggle with. The thing I'm probably best at is spotting the pirate ships on the horizon and insisting that they get talked about, rather than ignored.

There are always going to be people who are better at having difficult, second circle, conversations than me.

Maybe you are a natural at having these kinds of conversations (the evidence that we see from most projects is that most people aren't). Maybe you work with people who are better at having those kinds of conversations than you are. Maybe you all find it hard, and you need to work together to make sure that these conversations happen. Even so, if you understand and accept that this is what you have to do, you can agree to inspect and adapt as a team to get better at these kinds of conversations.

Because this is an important thing to understand. As a team you need to be having second circle conversations with your client. Not just once, but repeatedly throughout the project, all the time through the project.

You also need to be clear that, for those of you who do this communication, it might be very difficult. The responses that you get from the client may well be third circle. What are some third circle responses that you might get from the client?

They might shout at you to go faster. They might complain that your team isn't looking ill enough and stressed enough. They might point out (whether it's true or not) that you promised to do a certain thing, by a certain time and you haven't. They will cast doubt on your ability and your professionalism. They will often accuse you of lying and deception. A lot of what's said in these situations isn't about solving any of the problems on the project. It's about making you shut up.

These are all third circle responses. And to a degree they may work. The fact is that they don't really mean the things that they're saying - much. What they're really doing is just saying anything to shut you up. And, if you can stop yourself in time, this might be something to bear in mind when you find yourself reacting emotionally to what they say. You may find yourself angry at the idea that your clients want to physically and psychologically harm you and your team.

You might feel anger and shame at the suggestion that you haven't kept your promises. You might get a sinking feeling in your stomach that your name will be mentioned in connection to your bosses in connection with losing work to a competitor.

And it's important to have those emotions, because if you don't, they will "have" you. They will highjack you. They will stop you from thinking clearly and they will prevent you (or your team) from talking to your client in second circle. And guess what? That's what accusing you and your team of being incompetent, dishonest or lazy is designed to do. This might not be conscious on the part of the client, but it's exactly what's going on. Most people when they are attacked like this will either shut up (first circle) or be stung into a bout of frenzied agreed activity. Looked at in this way, agreed activity is a kind of manic, performative first circle.

In the worst, career-threatening cases, you might answer third circle attacks with a third circle response.

I've been stupid enough to argue back in third circle in the past. It doesn't work. It doesn't do anybody any good. It doesn't do you any good, as a project manager. It doesn't do your client any good. It's just another form of having the wrong conversation. It doesn't do the project any good.

How to deal with show trials and loyalty declarations?

I'd been working on the project about two months. We were supposed to be going live at Easter. The obsession of clients to launch software to coincide with Christian festivals made a bit more sense this time. It was a product that was used by university students. It made sense to deliver when they were taking a vacation. My boss had promised a certain list of functions would be delivered before Easter.

And the result of that promise was that he wanted everyone working on the project (and that was around 70 people) to tell him on a scale of one to five how sure they were that we would deliver for Easter.

To start with, he picked his favourites. The people that he knew would be most likely to say "Five." And sure enough, by the time he got around to people who were more independent minded and knew the state of the project, there was an immense and building pressure for everyone to say "Five." Every time a team member said "five" he would beam. Occasionally a brave soul would say "four" and this would bring about a period of gruff interrogation. Sometimes this even resulted in the poor soul sinking completely into submission and changing their score to "five." Even if that didn't happen, it would pretty much ensure the next few team members got the message and would cheerfully trot out "five." Then it was my turn. "Two."

My boss's face clouded. "Why two?"

"As you know, we've put the things that we said we'd deliver into three groups.

We've got the things that we've guaranteed. Then we've got the things that we've said we have a reasonable chance of delivering. And we've also got things that we've said we'll deliver only if we have time. Current progress suggests that we won't quite make all of the things that we've guaranteed. And I think a lot of the board think they're going to get all of the things that we've mentioned. In fact, I've heard some of them say that."

When the "meeting" was over, I had the inevitable one-on-one with my boss.

"Why did you embarrass me in front of the whole team?"

"Because they weren't going to tell you the truth, and it was unfair to ask them to."

"Never embarrass me in front of the team like that."

"OK, if you don't ask such a question like that in front of the team there's a good chance I won't."

Could I have handled it better? I'd be astounded if there weren't at least a dozen better ways. But I'm still not exactly sure what they would be.

I think I should have probably said "four" in front of the team. Then I should have asked for the one to one.

I should have walked through all the things that I was "assuming" would be fine that had allowed me to talk about a four. But I should have also tried to understand what the background and motivations were for him doing what he'd done. Making everyone say out loud that they were certain, or almost certain, that the project could be delivered on time was, of course, aggressive third circle communication.

Encouraging public declarations like this are calculated to avoid any kind of dissent or discussion. And they're always a sign of desperation.

So, this pulls us back to the guru of Toyota, Taiichi Ohno's, question - what business are you in? Yes, you're in the business of keeping the channels of communication open. You're in the business of providing opportunities for second circle conversations. But you're also in the business of finding new ways to keep second circle communication with the client going. But how exactly do you do that? The Agile meetings are part of that. And this is also the reason that the success of a project is directly related to the quality of the relationship that the product owner has with the client. Running the Agile meetings, counting things and showing progress are part of this.

Another good tactic is a dedicated risks and issues meeting.

This is a tactic that's worked very well for me recently. We were working on a project in a big organisation. The product owner told me that we needed to submit a risks and issues log to a programme-level governance function. I was sceptical. I hadn't had a lot of luck with risks and issues logs. In my previous experience they had been "information refrigerators." That is documents that

got produced because someone or something (like a programme-level governance function) has asked for. But once they had been produced. That was it, they didn't get reviewed or upddated.

Now it occurs to me, that as a project manager I should in general be a lot less sceptical about suggestions for anything that changes what people see - because what you see is all there is.

What I see now is that if you're going to create any artefact that is intended to be used as part of the process of managing your project, you need to also have some kind of process for maintaining it.

With risks and issues, it's the process for maintaining it that's far more useful than the document itself.

Ignoring my doubts (quite rightly) the product owner came back with a suggestion from one of the big names in Agile - Mike Cohn. The suggestion was for a Kanban-style board showing all the risks but then, providing each risk with a score. How did we calculate the risk score? Very roughly. The score was the cost of fixing a risk if it became a genuine issue. This was in terms of days for the whole team. So, for example, if the risk was "We can't find a rich text editor plug-in that meets our needs", the cost of that might be three months of work for the whole team - 60 days. And the likelihood of this might be low - 20 percent. Then the total score for that risk would be 20 times 60 - 1200. Scoring things with this slightly (almost totally) fantastical score meant that we could do several important things.

Firstly, and most importantly, we could have a fortnightly meeting where we discussed each of the risks - starting with the column which had the risks which scored the most. Then we could discuss if the score on the risk had changed. This was by far the most important aspect of having the risks board. Changes in score tended to indicate changes in understanding of the project or, more importantly for me, changes in political circumstances within the organisation. Talking about risk scores turned out to be a very good way of getting into a second circle conversation, a useful conversation, with the client about what is going on in the organisation.

One of the risks that we had on that project, that we talked about every two weeks, was "The project isn't delivered on time." Another was "We don't get funding to continue." Risks were also a way of safely containing rhetorical attacks on the team and general vague worries. Some examples of these: "The project causes reputational damage to the department." and "The department loses confidence in the development team." Discussing these risks every fortnight gave the client an opportunity to let off steam. But tracking these risks, and scoring them, allowed for a secondary benefit.

We could plot the overall risk score over time. We could see when the risks were sky high, which at one point they were - not having the money to continue is a big risk. At one point there was a high chance of not delivering and a high

chance of not getting the money to deliver. But over time, the risks reduced.

Setting up a risks and issues board and discussion is one quite straight forward way of encouraging third circle discussion.

Something that, isn't really a tactic, but is more of a consideration, is to think about, what I call, the words and music. Something that occurs to me when thinking about these instances of third circle barrages is that the main mistake is to engage directly with the content, rather than the fact that it's a third circle barrage. Nothing useful can be done until we've acknowledged that it was a third-circle attack. As we've mentiond before, once you've noticed that it's a third circle attack, your next move shouldn't be to engage with the content of the third circle attack, it should be to look for some way to have a second circle conversation.

And, in general, that should be the approach that you're taking the whole way through a project. You need to be looking for the places where there are "conversations" happening in first circle. These, of course, might be conversations that aren't happening at all. This is the agreed activity of scrubbing the deck while the pirate ship is on the horizon. But they also might be conversations that are happening very quietly. It's quite surprising how someone can get away with mumbling essentially the same update in stand-up. It can happen for days, sometimes weeks.

At the same time, it's important to be looking for situations which are in the third circle or are turning into third circle. Again, this can happen in stand up. The client can decide that the stand up is the meeting which he needs to use to take the opportunity to shout at the team and tell them to go faster. The client can decide that he wants to use show and tell to talk about how great the software *will* be rather than show how it currently is.

The client can bring his boss, or his boss's boss, or the chairman of the company to the stand up.

And it can happen with planning. For years, I've known what a good planning meeting sounds like and I've struggled to express it. Finally, by thinking about second circle, I think I've got a way of expressing it. A good planning meeting is absolutely about entering into second circle. The product owner walks through what they'd like to get done in the next two weeks.

Then the team ask questions. Sometimes they suggest that something is too small and it can be bundled in with something else. Sometimes they point out that something is huge and will either take several sprints or will need to be broken down into smaller bits. Sometimes there's heated discussion which ranges quite widely before it settles on an agreed understanding. That's a second circle planning.

But, as with all the other opportunities for second circle discussion, it can go wrong in lots of ways. The product owner can think that planning is about telling the team what they are going to do. It can even go wrong the other way.

The team can start telling the product owner what they're going to do, ignoring the product owner's own priorities for the backlog.

But by far the most common way that planning slips out of second circle is that it becomes a discussion between the product owner and the lead developer. They might be in second circle with each other. Of course, that's better than the product owner lecturing the developer, or the other way round. But the rest of the team are saying nothing. This is still bad; involving other developers and team members in second circle discussion is just as important as involving the client in second circle discussion.

So, to come back again to the Taiichi Ohno question. If you're a project manager and you're tasked with delivering the impossible, what business are you in?

This is the business that you're in. You're in the business of encouraging and initiating second circle discussions. Your in the business of the places where people involved with the project are mumbling or shouting and figuring out, with the help of the team, how to make those discussions more useful. What you want is second circle discussion. No shouting, no mumbling. Actual valuable exchange of information.. And to get it you'll need to have a box of tricks.

The agile meetings are part of the box of tricks. Support for user research and the actual conduct of user research is another part of the box of tricks. Various kinds of artefacts that *show* the progress of the project rather than *tell* the client about the project are also part of that box of tricks. Risk meetings, where all kinds of changes in the organisational structure, changes in understanding about the project and subtle shifts in focus and value come out during conversation, are also in the box, but so is developing the skill of listening to not only the words of a rant from someone senior, but also the music.

Chapter EMOTION –We don't have our emotions they will have us

Why don't we want to tackle the pirate ship? Why don't we retreat into silence or brashly state our opinion? Why do accusations of not delivering on commitments hurt so much? What could the reason possibly be, but our instinctive desire to avoid negative emotions.

Let me be completely open with you. I've been getting coaching. Full-on Agile coaching of the don't say much and let the client work it out for themselves kind.

I don't know how many sessions we've had, but it's more than ten. At the very end of my last session, she asked me if she could "share" something with me. That's Agile coach speak for mentioning that she has an opinion.

And the thing she wanted to share with me was this. I'd started the coaching

sessions with a problem. I still have that problem. What was the problem? The problem was and still is that I find it difficult to talk to senior, "C-Level" people.

The observation that she was making was that I had a lot of skills in project management, I had a lot of experience of project management. I clearly don't have any problem talking to her. I don't seem to have any problem talking with my team. So why do I have so much difficulty talking to "C-Level" people?

And my answer, maybe it was too frank was that when I'm talking to her, and most of the time when I'm talking to members of my team, I'm not angry. And almost all of the time when I'm talking to C-Level people who are involved in the projects that I'm working on, I am angry. What am I angry about?

I think one thing that I'm annoyed about is that they don't want to listen to me. They are not doing the things that they need to do in order to make a project a success. When a boss says that he or she wants a project to succeed, but then does a number of things that will almost guarantee that a project won't succeed, are they lying?

And mixed in with the anger is shame. I'm ashamed that I'm angry.

And now, writing this, mixed in with *that* is fear. Fear that there isn't any way to fix this, that this is always how I'm going to be in these situations. That there is no cure.

What I want to be like is like Columbo. Mild mannered, polite, but gently pushing the matter towards its inevitable conclusion.

Bullshit

You have to tell them something. And it has to be something shiny. But those are the things that you have to care about most. You don't have to worry so much about it being true. In fact, you don't have to worry at all about it being true. Bullshit isn't designed to play in the world. Bullshit is design to play in the truncated, limited resolution, limited imagination, model of the world that sits in people's heads.

Fuck it

What are the risks? What are the potential benefits? We need to weigh these carefully and then decide on a course of action. Don't we? Do we? Is this really how we decide how to behave?

What if it isn't?

What if our main guide for how to behave is not some careful reasoning from the facts, what if our main guide for behaviour is to behave how everyone else does?? But then periodically, we have a crazy dream, or we realize that the only way out of the particular situation is to offer someone else a crazy dream.

Nothing happens unless at some point, someone shrugs their shoulders and says "Fuck it."

Bounded Rationality

I think this concept is best explained by the genius Spike Milligan in a episode of the Goon Show.

Bloodknock: You Chinese think of everything

Grippipe Thin: But I'm not Chinese!

Bloodknock: Then you've forgotten something?

Milligan in his unique way is making a joke about the fact that nobody can think of everything. The reason that every now and then someone can get away with shrugging their shoulders and taking a risk that isn't calculated is that we have to engage in something called something called "bounded rationality." We literally can't think of everything. At some point we have to shrug our shoulders and give it a go. Of course, this would have been stamped out as an evolutionary behaviour long ago if it were always fatal. But it occasionally pays off.

The hiding hand

And as a society, as a *species* we're curiously, sometimes terrifyingly, tolerant, even encouraging of this. The economist Albert Hirschman spent a lot of time observing projects in Latin America that were funded by the World Bank. Many of these projects were set up in ways that made them unlikely to succeed. But he noticed that, even so, the people who were working on these projects would play down the risks and overstate the potential benefits. Hirschman called this "the hiding hand." This is a reference to a phrase used by an early economist, Adam Smith. His idea was that a "guiding hand" seemed to be deciding the price of products in a market. Of course this isn't really what's happening. Where there is a shortage of something, prices go up, when there's a glut prices go down. But it seems like there's a single force that's deciding the price.

Hirschman is suggesting that something similar is happening with new projects. Rather than a guiding hand, there is a hiding hand. It looks like all of the people in a project are conspiring to look the other way. They are pretending to not notice the huge risks and the low likelihood of success. Someone says fuck it and they go along.

Legibility

A crucial part of this is legibility. Or rather illegibility. When a project starts off, and especially when a project is in progress, it's not a good idea to be totally clear about the risks and costs. This is what everybody is cooperating with in the hiding hand. Clarity and legibility will not help garner support. Funding is

unlikely to come from outsiders. And is going to be even more difficult to get if every potential problem is starkly highlighted.

Heads I Win Tails you lose

Why are some people much better at kicking off projects than others? Is it because they have better ideas? Is it because they're better at execution? Or is it because they are relatively confident that no matter what happens to the project, they will be OK?

This seems like I'm being critical. But it's one explanation of the hiding hand, and it's a very good explanation of why anyone might want to carry on on a project that they regard as impossible. Why are you carrying on? Because you think you can profit from it, even if the project itself is a failure. This is a perfectly valid strategy while working inside a project.

1. Create a project that promises (vaguely) a great deal.
2. Maintain contact with the project.
3. If the project looks like it's going to be a disaster, distance yourself from it.
4. If the project looks like it's going to be a success, make sure that you're associated with it.
5. If the project crashes and burns, make sure that you're nowhere near it and heavily involved in the creation of the next project.

Of course, I forgot a step.

0. Say Fuck it.

This is an absolutely vital thing to understand about any project that isn't totally boring and simple. Any project of any complexity is simply too complicated for anyone to have considered all of the risks, costs and issues involved. Rationality has bounds. Like Grippipe Thin, you can't think of everything. If a project is going ahead, someone, somewhere has said fuck it.

But a lot of the time, and this is the same for the people working on the project, they will get paid if the project succeeds and they will get paid if the project fails. And most of the time, they also have the option to jump ship and find another project. So that's what the hiding hand is, it's a lot of other people joining in and saying, "fuck it."

Why should anybody care?

The people who kicked off the project, if they've kicked it off in the right way, are adopting a "Heads I win, tails you lose attitude." The people who are working on the project are getting paid anyway. Why should anybody care about a project?

That would seem to be a pretty dumb thing to do, to care about a project when it's irrational for anyone else.

The main reason is that projects can discover value. They can make people's lives better, more entertaining, easier. And of course, if they can do this, they can make money or prestige for the people who own those projects. But have we been too cynical and cruel about the people who work on these projects? Yes, they want to get paid, but most of them also want to do good work and want to be associated with a project that's a success.

And we have probably been too brutally hard on those people who said "fuck it." They got the project started when others would have worried too much about the risks and costs. They would also dearly love to be associated with a project that's a success.

What makes a project fly?

When a project delivers value that outweighs the risks and costs, it's flying. Initially, that value is vague perceived value. But at some point, there has to be a transition to actual value. This is the tricky phase on all projects. Because it requires clarity and legibility. And clarity and legibility undermine and attack perceived value and perceived cost.

Why am I so angry with the people pushing these ideas?

So, if this is the way that projects work? Why am I so angry with people who talk in terms of the shiny dream and the values? I also like to get paid. When I start a project, I've pretty much no idea whether it will succeed or not. I'm part of the hiding hand. And part of my strategy is "Heads I win, tails you lose." I haven't gone down with every failing project that I've been involved in. If the project is obviously going to be a failure, I can, and have, walked away.

Part of my anger is that I'm trying to get the project to "fly" to generate enough "lift" from real value that it can succeed and have a life of its own.

The other end of this. What about Steve?

Jeff is my friend. Jeff used to be a Scrum Master like me. Jeff has been kicked upstairs. He now manages literally hundreds of Agile coaches and scrum masters. Jeff said something to me that was interesting. But I didn't like it, I didn't want to hear it. Jeff said that he would be more impressed with Scrum masters if any of them were delivering their own software projects. But he didn't know of any who were. Why might that be? Possibly it's because they don't have the skills, connections and courage to put together the first half of the project. That's the bit where people are willing to invest time, effort and money in something that is just a dream. They don't have whatever it takes to say, "fuck it." Is that it?

Another reason of course is that they're too emotionally invested in the reality end.

These things are in opposition in projects.

The dream is in opposition to the reality.

Expenditure is in opposition to revenue, and ultimately profits.

All is in opposition to some

This is a reference to the theories of a Chilean Pyschologist and Psycholanalyst – Ignacio Matt Blanco. His idea is that there's an opposition between an "all" way of thinking and a "some" way of thinking.

More specifically, he connects the "all" way of thinking to our subconscious and the "some" way of thinking to our conscious mind. There is a similar opposition, and a relationship between the subconscious and the conscious mind and the concepts of "Same" and "different".

Most projects have some kind of "all" or "same" in their description. And that "all" or "same" is a connection to the subconscious, the dream world, the things that we know that we responds to , that we categorise without thinking.

And now it's a bit more obvious why this book has such a contradictory title. Because it's trying to capture both ends of this see-saw.

I think it goes the other way round.

What's impossible is in opposition to what's deliverable.

And I want to write here "Money or sense."

Because the phrase that keeps coming back to me is by Buckminster Fuller - "you can either make money or sense."

So maybe the way through this is to get on the other end.

I was thinking, become a product owner. Indeed, it would be interesting to become a product owner. Because the product owner sits in the middle of this nonsense or sense contradiction.

Now I'm thinking that that isn't far enough back up the pipe.

What I'm also thinking is that maybe I need to go "Full spend-everything nonsense dreamer who wants it all and says fuck it."

Maybe I need to be that person.

There's another thought to blend into this mix. This is that the software that dominates the world at the moment has been instigated by people who have sight of both sides of this divide. Mark Zuckerberg was a techie. Bill Gates was a techie. The two guys that built Google were techies.

Chapter REDUCTION - Harm Reduction - Safe drugs, safe sex, safe SAFE

Despite your best efforts, people will do mad stuff. But there are still things that you can do to stop the mad and bad stuff being fatal

Connection

This chapter might sound a bit crazy. It suggests drug addiction, unsafe sexual practices and project management have some things in common. That sounds crazy right? And yes, I am being a bit dramatic. Most projects don't result in death if they go wrong. But they can result in extreme expense. And they can be very miserable to work on.

Concept

If people want to do something because they like how it makes them feel, they will do it, almost no matter what. They will ignore what bad things will happen as a result. If people want to avoid doing something because they don't like how doing it makes them feel, they will avoid doing that, no matter what. They won't think about the bad things that will happen.

Part of the reason for this is the relationship between the delayed gratification and the instant gratification.

On the one hand there is the long life free from sexual disease. On the other hand, there is a quickie.

On the one hand there is a healthy old age free from self-inflicted psychological and neurological problems. On the other hand, there is a chance to get totally out of it and forget all of your immediate problems.

On the one hand...

Does this even work for software development? Am I pushing a metaphor just too far? What's the huge payoff for "dangerous behaviour" in project managment? The huge, instant payoff is in being able to pretend that your problem is solved. That this new thing will solve, if not all your problems, then a lot of them. That this new thing will make the company a lot of money, or save the government department a lot of criticism and boring manual processing. Make the reputation of the CEO or founder, allow the company to be sold off for millions.

What's the risky behaviour? The risky behaviour is buying dreams. Persuading others to buy dreams. That doesn't sound that so bad does it?

And what's the "safe" behaviour? The safe behaviour is iteratively and incrementally trying to implement those dreams. The safe behaviour is modifying the dream in light of actual experience. The safe behaviours are the things that

we talked about right at the beginning of the book. There is no way of being safe without "tackling the pirate ship." There is no way of being safe without identifying the big threats to the project and then you need to tackle them. You need to understand the swamp, its inhabitants and its landlords. You need to understand the values and needs of the people who are funding the project. And you also need to know about the people who might regulate this system, the governors. Last but not least, you need to understand the people who might use the system. If you want to practice "safe project management" a lot of your activities will be focused on increasing knowledge and understanding.

But there are other dangerous behaviours which don't relate to the immediate moment of gratification. And as with other areas where people are tempted to behave riskily, these have to do with bad news and contra-indications.

If you're a senior person who has started a project, what do you do when that project is showing signs of going wrong? Do you ignore it and hope it will go away? Do you try to front it out with positivity? Or do you take it seriously? And here the meta of dangerous behaviour does hold across, sex, drugs and project management. Do you get tested and understand that the results of those tests might require a change in behaviour?

Concrete example

SAFE is supposedly an acronym for "Scaled Agile Framework." Although of course, it's a "backronymn". SAFE draws on Agile, Lean and systems thinking to paint a picture of how an entire organisation might go about doing software development.

Key aspects of safe are that a programme is divided into projects. Each of these projects is called an "Agile release train." In SAFE the Scrum Master has morphed into an "Agile release train manager."

Agile release trains are chunks of work that are agreed at quarterly PIPs "Programme implement planning" meetings.

To connect this the subject of this chapter, SAFE, is about instant gratification for some people, although it causes pain and misery for others. SAFE is designed to appeal directly to senior managers. I have seen the hypnotic look that senior managers get on their face when they talk about SAFE.

This is the message that SAFE whispers in the ears of senior managers.

"Pay lots of money for an army of SAFE-trained and certified coaches."

"You can have vague ideas for entire programmes of work."

"They can be implemented in the way that you think that programmes of work should be implemented - top down and waterfall."

"You give the orders, they're obeyed."

This is the thing that it's taken me way too long to realise.

You can't stop senior people engaging in "dangerous" behaviour. And in software development, please turn off your irony meters, because one of the most dangerous behaviours is called "SAFE".

Why is SAFE so dangerous? Is there anything that can be done about it? SAFE is dangerous mainly because it is such an such an exquisitely attractive boardroom sales pitch. It deliberately ignores the most important thing about any agile method. There is nothing on its diagrams about feedback. It says close to nothing about inspection and adaptation. It pays lip service to the idea of Agile. But then there is no real implementation of the key Agile idea of "Responding to change instead of following a plan."

Why do we need feedback? Why should we respond to change instead of following a plan? Think about a classic kind of control problem, a computer controlling a robot arm.

Sure *control* consists of being able to tell the arm to move forward, lift up and crucially, to grab things. But if you're getting a robot arm to pick up something - like, say, a cup, the robot arm needs *feedback*. It needs to know when it's touched the cup. To pick the cup up, the robot arm needs to know when it's exerting enough pressure on the cup to get a grip. And it needs to stop exerting pressure before it breaks the cup to smithereens.

As it lifts the cup up, it doesn't want to spill whatever is in it. This means that it needs to move at a different speed, and with a different pattern of acceleration depending on how heavy the cup is. What kind of liquid is in the cup? The arm might need to move differently depending on that. So now we might be thinking that it isn't enough to have pressure sensors on the fingers and weight sensors in the muscles. It also might be good to have some visual sensors. And also, some high-level, overall analysis of what we're getting from the sensors.

OK, that was just one arm. Imagine that you're trying to control an octopus. Eight arms. Or a really big organisation with 20 arms, 100 arms. What if you design a structure to send out instructions to those arms and this structure doesn't involve any feedback? There are no sensors, no sense of touch. No sense of smell.

One of the things that makes recreating human arms with robots so challenging is that we have senses that provide us with feedback, that were hardly aware of. We have senses that sense how fast our arm is moving, and sense where it is in space, which way is up, and which way is down. This sense, sometimes called the sixth sense, is proprioception.

Without proprioception to sense the position of each of these arms and all of the other senses. One arm literally does not know what the other is doing. When one arm is punching into the side of the other, neither knows what's happening (no sense of touch, no sense of proprioception). What if one arm knocks over a naked flame and starts a fire? The arm can't hear the noise that it's knocked

something over. It can't smell the smoke. One of the other arms doesn't know that it's burning. This is what any control system looks like that doesn't have feedback.

SAFE, as it's advocated in the diagrams that sell SAFE, is, literally, senseless.

Conclusion

This is what's missing from SAFE. A discussion about feedback. Why? If feedback is so necessary, why is it missing from SAFE diagrams.

Do feedback loops look scary? Do they put off the decisions makers who buy SAFE? Let's imagine for a moment that the SAFE marketing process is trance induction. The expression that I've seen on senior managers' faces when they talk about it rather makes me think that it is. The waterfall top-down diagram is matching the senior manager's thinking about how a project should work.

What if we add feedback loops to this diagram. What if we suggest that the senior manager's idea about how a project will work might need correction? What if we mention some obvious problems that we've already found? Now we're not pacing. Now we're arguing. We're going to bring the senior manager out of a trance.

Do you know what this reminds me of? Although I feel slightly crazy saying it, it reminds me of the objections to condoms. The objection that involving condoms in the "love-making" process somehow spoils the moment.

I don't know anything about heroin users. But maybe there's a similar objection with heroin users. There's a camaraderie to sharing needles. Going and getting clean needles from a clinic is, well, clinical and also, going and getting needles from a clinic means admitting that you have a problem.

So, the challenge is to get horny people, drug takers and senior executives, to incorporate safe practices into their risk-taking.

Chapter TRENCH - Recognising trench warfare and knowing when to walk away

Some projects are never going to deliver. Or they're going to deliver so slowly and with so much damage to the people who are working on them that they still should be avoided. An important part of being able to deliver possibly projects is being able to spot and get yourself out of the ones that are impossible.

> Following World War I, "trench warfare" became a byword for stalemate, attrition, sieges, and futility in conflict.

From the "Trench Warfare entry in Wikipedia"

I'm going to use the term "trench warfare" in this chapter. And I'm going to use it to mean any project which is going to fail unless the way it is set up, structured and managed is changed.

What I mean by trench warfare is any project that is hugely expensive in terms of money, effort and human suffering and isn't getting anywhere. A main cause of trench warfare is a refusal (or active prevention) to deal with the problems that the project faces. The main effect of this is pretty much guaranteed failure. But on the way to that inevitable conclusion, there are more and more strident demands for "agreed" activity. Activity at almost any cost, no matter how frantic, pointless and damaging it might be.

Let me give you an example. We were trying to meet a ridiculous deadline. Two members of my team travelled to a secure site where they were supposed to be putting our software on a client's machine. They'd written an installation script to run on a clean machine. The idea was that would put our software where it needed to be and install everything else that was necessary for it to run. They'd tried the script on a practice machine. It had run without any problems. They kept trying to run the script on the client's machine. It kept failing.

They ran some diagnostic tests. The machines that they were putting the software on were supposed to be "clean". They weren't supposed to have any other software installed on them - just have the base operating system. The tools we had to check for problems showed that there was already software running on the machines. And this was clashing with the stuff that we were trying to add. When the guys from my team pointed this out to the people who were supplying the servers, things started to get very weird. The guys from my team wondered if there'd been a mistake. They pointed out that there was software that was running on the servers that they didn't expect. Maybe some other software had been installed. Could it be uninstalled, so the guys from my team couldget their installation working.

The guys who owned the server wouldn't say. They were very tightly lipped. They refused to acknowledge that was any extra software running on the servers. Something was clearly wrong, but we didn't know what, and the guys from the server company weren't saying.

A few days later. A Sunday afternoon and I'm in an emergency meeting. The topic of the emergency meeting started out as "What are we going to do about the failure of the team to load our software on the client's machine?" The client was implying very strongly that the reason we'd failed was because our guys weren't up to the job. But just like the guys in the server room, when I brought up the subject of the other software already running on the servers, the client became evasive. He moved the topic of the emergency meeting onto what we should do about our four o'clock emergency meeting. Our three o'clock emergency meeting had overrun so long that we were now in it. Or were we?

Clearly this was something that we needed to spend time on our weekend to discuss.

Ultimately the mystery of the software already running on the clean servers was solved. The company that was providing the servers was in dispute with the client organisation. While they were in dispute, they were working to the letter of their old written agreement. The original agreement had been written months ago. It mentioned some software that should be installed. In the nature of projects, as they progress, it had turned out that it was better if the server guys didn't install software and left it to us. But the contract, the one that the server guys were arguing about said that they should install software, so that's what they were going to do. The server guys were providing servers that already had this software, even though they knew it would break ours.

That's part of why they were behaving so oddly. But the other part is that the written agreement to provide the servers was classified as secret. So even as they saw our guys failing, and they knew why, they felt they weren't legally allowed to tell us.

Eventually we found out. Eventually it was admitted that there was no chance that the project could go live that week. I left the project the next week and I never heard of it again, I'm certain it never delivered.

How did I know to walk away? I've worked on other projects that were struggling and I haven't walked away. I've stuck with them. Some of those projects succeeded, some of them didn't. But something made me absolutely certain that I needed to get off this project. Even if I wasn't explicit about it, I think I was using this rule. I wouldn't say it's as strong as a law, but it's a very good rule of thumb. I call it the "Three nonsenses rule."

The rule goes like this.

> Most projects are going to be labouring under some delusion. There will be something about the project that doesn't make sense. Often this will be a delusion about how long the project will take to deliver, or how much it will cost. Sometimes, the delusion will be that this new product is guaranteed to be a success. Sometimes the delusion might be that some new technology that hasn't quite been developed yet will change everything. Sometimes the delusion is that successful projects can be delivered at all by the organisation that's funding it. Despite all evidence to the contrary. Despite all the still-visible wreckage of previous disastrous projects.

It took me a long time to realise the chances of a project having no delusions is nearly nil. You probably need to let a project have at least one delusion. Fighting to make a project completely delusion-free is a lost cause.

But bitter experience has taught me that delusions when combined, don't add,

they multiply. Crazy and crazy isn't twice as crazy. It's crazy squared. And if you add in a third delusion, that's crazy cubed. That's crazy in every direction, forwards, backwards, up, down, left and right.

Working late. Working weekends, holidays being cancelled or the boss making sideways comments when team member take them. There is an insistence on a "positive, can-do attitude." These are signs that you're working on a trench warfare project. At the same time, trying to deal with the problems that the project faces is not well received. It's regarded as "negative" or "defeatist".

Leaving drinks every Friday? People on your team being signed off long-term sick? Or with stress? Ex-members of the team bringing lawsuits against the company? These are also signs.

I must confess, I still haven't found a way of remaining calm when I get a certain kind of complaint from senior management. The complaint is that the team isn't looking stressed enough.

I have not found a way of communicating - without also giving away how annoyed I am - the basic truth about software development. Software development is about the "three Ts" - typing, thinking and talking. None of these are helped by being tired. None of these are helped by being stressed. None of these can be done well at three in the morning. The shouldn't be being done at all at three in the morning. Software is an activity for the daylight hours when everyone is rested and everyone is available for a conversation.

A trench warfare project is a bad situation. Experience of working on one can be one of the best reasons to get over your reluctance to tackle issues as soon as they come up. Once you have worked on a trench warfare project, you might be much keener to call out problems the minute you see them. But most projects have ridiculous timelines. You find them everywhere. They are the norm.

OK, so this project isn't being realistic about the deadline. That greatly increases the chances that they are also not being sensible about some other things. Maybe they are ignoring all the indications that the fancy new technology that the project uses isn't ready. Or maybe they are trying to get the project delivered "under the radar". Even though they know full well that at some point the giant "Monty Python" foot of regulation is going to stop it dead in its tracks.

I don't know if the real-life Jerry Springer actually said this or not. But in *Jerry Springer the Opera* the character of Jerry Springer says something that has stuck with me ever since I heard it: "I don't solve people's problems, I televise them."

If you are working on a trench warfare project, that is exactly what you should be doing. Sure, go one better than Jerry, and solve the problems that you can solve, but the problems that you cannot solve? Broadcast them. Make sure everybody on the project knows what those problems are.

Why? Because someone else may, if they want the project to succeed, be able to solve them. Another way to look at this is to look at some of the people who

are giving you and your team a hard time. Those people who are saying that your team dosn'tlook worried or stressed enough. Those people who are saying that members of your team don't care about the project. I mean look! They took the whole weekend off!

Why are people behaving like this? Because,let's be very clear. It doesn't work. It doesn't make the project more likely to succeed. It makes the project more likely to fail. Code written at 2am on a Sunday morning is not going to make a project succeed. Most of Monday and Tuesday, if you're lucky, January and February if you're not, is going to be spent finding and taking that exact same code out again.

It might be that some of those people are only doing those things because they have no idea what else they can do. They want to make the project successful. They do not know how. What if you do make clear what the problems are that your team is encountering? You are giving those people who do want the project to succeed a good solid opportunity to help.

The second thing that you can do is to deliver some of the things that some people want to some of the people who want it. This might not be anywhere near everything that has been asked for by the deadline. But if you can find one bit of software that you can release to some people who want it, you can completely change the nature of a project. In Agile terms, this is called "pull." Pull from outside a team that comes from demand from real users can be a wonderful thing.

Thirdly, one of the most revolutionary things that you can do, is to track and show the actual progress of a project.

This might be tough, but few things are as powerful as showing actual progress relative to the project's needs and expectations. In a way, this is just a more extreme version of the "Jerry Springer" principle. This is one of the most powerful manoeuvres that I think an Agile project manager has in their toolbox.

If I'm a one-trick pony, this is my trick. Capturing and then showing actual progress.

What? OK. Here's what you do. And it can work on most projects.

1. Get a list of things. For now this can be a list of everything that anybody has ever thought this project might do. This might not be easy. The list of things that product needs to do, might not be collected together anywhere, it might be in emails, it might be just in the air with promises for functionality discussed briefly in meetings but never written down.

2. Give these things a score. A number showing how big they are relative to each other.

3. Start to track progress. How many of these things have been done? If something is big? What percentage of it is done? Can it be broken up into things that are done and aren't done?

4. Put this progress on a chart.
5. Deal with the wailing and gnashing of teeth. Deal with the personal attacks, threats, denial and backstabbing. Watch out for underhand manoeuvres.
6. Over a period of, probably not just weeks, probably months, track progress. After a month, how big is the backlog? Did it get bigger? Did more stuff get added in? Did it get smaller? Did senior stakeholders see the chart and realise that there was no way they were going to get some of the features that had initially been discussed?

By doing this, by somehow quantifying the size of the problem and putting it in a chart,you're tapping into one of the fundamental ways in which people think. This is identified by the Nobel prize-winning psychologist Daniel Kahneman in his book *Thinking fast and slow*. What's this way of thinking?

> What you see is all there is.

Daniel Kahneman's point is that a lot of our irrational behaviour and poor thinking can be explained by this principle. We don't do our thinking based on all the facts. We do our thinking based on what we can immediately bring to mind. This has one very interesting implication. If you can change what people see, if you can change what they bring to mind, you can change what they think.

This is what we're doing by collecting together a list of all the work that needs doing and then tracking progress through it.

Actually, you know when I said that I was a one trick pony. Well, I've remembered a few more tricks. Actually, three more tricks.

1. A risks and issues table – if possible, together with a chart.

Whatever is bothering you – put it in a risks and issues chart. For a long time in my project management career, my experience was that risks and issues documents were "information refrigerators" – they were places that bits of information went to grow mouldy and shrivel up.

But in a project that I worked on recently, the product owner found a format for risks and issues that seems to work extremely well and I've used it on all the projects I've worked on since.

2. Working software

   Get something that works and release it somewhere. Ideally, if it's possible, release it to the world. This is so important that we're going to devote a whole chapter to it.

3. User research

   If you're project is a full-on trench warfare project, it's very unlikely that it has dedicated user researchers. That would be ideal. But even if it hasn't that doesn't mean that you can't put working software in the hands of potential users and get their feedback on it. There

> may well be extreme restrictions on who you're allowed to show the software to. It maybe government or commercial classified, but there are almost always people inside your organisation who are cleared, or can be cleared to see what you're working on.

This can then, sometimes, result in delivery of a small bit of working software to some of the people that want it. It may well seem that a project is dug-in, going nowhere and doing nothing but damage to the members of its team. But it's still worth doing these fourthings.

1. Be honest about problems the project is facing.
2. Track actual progress.
3. Push as hard as you can to get working software in the hands of end users.
4. Do user research.

Actually there's a fifth thing.

4. Look after yourself. Go home on time. Take your weekends off. Take holidays. See friends. Kiss your partner. Kiss your kids. Kiss the dog! Take it easy forget about the shit show you'll be walking into again on Monday morning.

Keep doing these things. I mean all of them. Points one, two, three and four and absolutely, definitely point five. Doing these things is the best way that I know to save the trench warfare project that you're working on. But equally, it's important to remember that some projects cannot be saved. Not by you. Not right now.

It's also important to understand that trench warfare was, at the time, an insoluble problem. Eventually the war ended. And there was the idea developed in English language commentaries that the allied soldiers were "Lions lead by donkeys."

But one of the points made by Norman Dixon in his book *The psychology of military incompetence* is that this is unfair for two reasons.

Firstly, the selection process. This might not have been explicit, or official. But the process by which officers rose to the most senior ranks in the British Army was one that didn't think innovation was important. Initiative and a willingness to challenge the status quo might be really useful in a war. They're pretty dangerous qualities to encourage in the military in peacetime. The result was that the kind of people who were leading the British army weren't innovative, creative problem solvers.

Secondly, trench warfare was a hard problem.

Trench warfare was "emergent behaviour". It came out of a combination of technological and strategic techniques. The very short version is that guns got a lot better. The technological way out of the impasse probably involved the use

of tanks. Probably, but not obviously. Tanks weren't technologically far enough evolved. Their potential as battle-winning weapons wasn't obvious or, crucially, immediately exploitable.

A solution might have been the devolution of some decision-making to small semi-autonomous groups. Maybe. This was something that the German side did experiment with late in the war. They were starting to have some success when the whole German nation ran out of food.

So, if your project is in a total impasse, it might be that there is a way out of the trenches using innovation. A novel combination of practice and technology might result in some powerful improvement.

An entirely different way of thinking about a trench warfare project is that it's an opportunity to try anything else.

I don't *think* I've ever worked on a project that was being deliberately targeted to fail. But maybe I have. I have heard talk of such projects.

I worked for a global consultancy. I was talking to the guy who was our main sales contact with a UK bank. I happened to mention a project that I'd been trying to help. They were in deep trouble, I could see, but I couldn't exactly figure out why.

"Oh, yeah," he said, grinning. "We're letting that project fail."

"What?"

"Yes, when we sold them the team to work on the project, we tried to sell them our banking engine at the same time. They bought the team, but they insisted on using this other banking engine. So, we're going to let the project fail. Then we'll suggest that they use our banking engine."

It is important to understand that you don't need to stay on any project that is failing. The people in the real trenches had to stay. If they didn't, they'd be shot for desertion. You don't have to.

Of course, you can nearly always just find a new job. When you start to look you might be astounded to find that you're not the only person on that project. Others might also be looking. Recruiters have a very good idea about which projects are going badly.

But if you're in an organisation big enough to have more than one project on its books, you can probably get moved off a trench warfare project.

Of course, you don't have to leave. You could just stay and get paid. You could look like you're paying attention whenever the bosses are around. You could pretend that the endless deadlines that come and go mean anything. Every now and then, you could stay late, or work all weekend to show that you're really dedicated to the cause.

Of course, none of this is going to make the project any more likely to succeed. At some point, someone is going to realise that the project is going to be a failure and pull the plug. But you're getting paid, right? You're inside, out of the bad weather, and there's no heavy lifting involved.

I don't know, I can't do it. But I've seen enough software development projects to know that a lot of people can. In fact, for some people and some companies, it seems to be pretty much their business model.

What if you're one of the bosses of this project? What if you own this mess? What can you do?

First: find out what the real problems are that the project is facing. If a project is really failing, it's very probable that your teams won't be running retrospectives, so you might not know. Ask your teams to run retrospectives. Ask them to just tell you the top three things that are limiting their progress. Actually, get them to tell you just one thing that's limiting their progress.

What is that thing that's stopping them? How can something be done to address it? And then be Jerry Springer about that thing. Televise it. Let everybody who should be interested and everyone who could possibly do something to fix that problem know about it.

The first of these is to understand the value environment that you're working in and delivering to. And the metaphor we use in this book is exploring the swamp. The second is to deliver working software.

Chapter CONCLUSION - Conclusion

Tackle the pirate ship. How? Use agile. Agile is nothing more than a bunch of tactics for delivering the an empirical process. Transparency, inspection and adaptation. Understand that every project is a swamp. It needs exploring. User research is a big part of that, but so is the adventure of releasing working software.

So, this book has discussed a lot of different ideas. A lot of different ways of looking at the world and specifically a lot of different ways of looking at projects. And, in the end, all of these different ideas boil down to a series of questions. Questions that you can ask yourself. Questions that the team can ask themselves. Questions that you could even possibly talk through with the client, if you have a close enough relationship. And the answers to these questions are mostly definite actions that can be taken to increase the chance of delivering your project.

The idea: what you see is all there is, which we get from Daniel Kahneman.

Imagine a woman being sawn in half. You see her head. You see feet. You see the casket being split into two separate parts.

If what you see is *really* all there is, what do you do if you want to change what people think? You probably have to change what they see. And so, the question that comes out of this way of seeing that suggests a potential course of action is: What can we change that people see, so that their behaviour also changes?

In my own account of "Fried Egg Agile," I've talked about how I've used burn-up charts to help to deliver several projects on time.

Imagine that rather than looking at your team, you're looking at that team over there. This isn't a team that you're connected with. This is a team that you don't have any stake in. You don't desperately want them to do well.

How are *that* team going to do? Are they going to do well? What problems might that team have?

Also from Daniel Kahneman, the planning fallacy and associate with the planning fallacy, the inside view and the outside view.

In terms of Triz:

Which of the problems that you're trying to solve are in fact insoluble? They simply aren't the kind of problems that can be solved, they are trade-offs that need to *managed*. If they are trade-offs, how do you find a sweet spot?

Are there any parts of the process which require something to both exist and not exist? Is this an opportunity for invention?

Is there is a situation in which you both need something and don't need it? How could you go about doing that? Are you doing it already (like having a "servant leader" as the manager of the team).

This is really both the ideas of Keith Johnston and the "SNAFU problem" as stated by Robert Anton Wilson. Everything is a status transaction.

In terms of Status, how is what the problem that you're trying to solve being affected by issues of Status? Are you trying to push / change the timings of your clients? Are you trying to make them do things that they don't want to do? What ways of approaching this problem are there that makes status less of a problem?

Looking at this through the ideas of John Boyd: are you being asked to speed up quicker than you possibly can? Are you being asked to run at a top speed that will always be beyond the capacity of the team? Are you actually been asked to slow down and you don't want to? These are ideas connected to Boyd's Energy Manoeuvrability Theory. What's important about an aircraft, especially a fighter plane, isn't its top speed, but its manoeuvrability. How quickly it can speed up, how quickly it can slow down and, as well as its top speed, what is the lowest speed that it can run at and still be manoeuvrable.

Another of Boyd's ideas is the OODA loop, which is one of many versions of the ideas discussed in the "Car driving" three pillars of the empirical process. What are you observing that's weird or unexpected on the project? How does it

change our thinking? How does it change our decisions; how does it change how we act?

In terms of the Cynefin framework, where are we? Are we in a chaotic situation? Are we in a complex situation? Where does the client think we are? Where are we being pushed? Are we being pushed from a complex situation into a chaotic situation? Are we being pushed into disorder? Are we already in disorder? How do we get out of disorder? What small steps can we make to take us to somewhere where we know where we are?

Driving the car:

Transparency

Inspection

Adaptation

Some and all

Same and different - Ignacio Matte Blanco

What problem are you solving? What is it? Taiichi Ohho.

Patsy Rodenberg's idea of first, second and third circle.

Which circle are you living in most? As an individual? As a team?

Theory of constraints (Goldblatt, et al.)

Is there a bottleneck? How could the bottle neck be managed?

Emotional Labour - part of the the work of a project manager is in dealing with other people's emotions.

Arlie Russell Hochschild

Hungry crowd - Gary Halbert

Childish Thinking - Marion Milner

Journalling - Marion Milner

Wanton go for it - Richard Bandler

Bullshit - Harry G Frankfurt

The Hiding Hand

The Fata Morgana - Albert O. Hirschman

On the evolutionary function of sociopaths or Sooner or later, someone needs to say "fuck it."

Chicken Soup for the Shirt

Journalling

Agile as a way of liberation

You can make money or sense - Buckminster Fuller

Chapter MANIFESTO – Epilogue – A personal manifesto

A Manifesto

All is the crevasse - partial is the path. Always? Ha! No, sometimes. Probably most times.

I want to help you and me get to the point where we're all more confident that we can deliver the seemingly impossible. That's going to be a long trip. On that trip, I want to be a reliable guide. So, to help keep myself straight, and to reassure you somewhat, here's my manifesto.

1. Tell the truth.
2. Don't over promise.
3. Be clear, don't know it all.
4. Point to sources and interesting things.
5. There is a great temptation to try to write books like this from the point of view of an absolute expert. Why? I don't know. I'm not an absolute expert. I've been doing this a while. I have some strategies that work fairly reliably. I don't have a rigid method that I follow.
6. Pretty much, see point 1. Obviously, I can't promise that you will deliver the impossible. The thing that you're looking at that seems impossible may well be exactly that. I worked on a project once that had all the normal hallmarks of an impossible project. It had an "aggressive" deadline. It had a strict separation between subject matter experts and the development team. There were no members of the team who were trying to talk to users or trying to talk to people in the wider organisation. But this project also had something else. It had an absolute reliance on a database that didn't exist. All the diagrams and sketches of this project showed this database. This database was essential to the functioning on the project. This was a database that the development team had to draw up data protocols to communicate with. It didn't exist. This project was never going to succeed.
7. Think about it for a second. Nobody knows it all. I'm pretty much trying to tell you everything that I know. There might even be a chance that combining that with what you know makes you much better at managing and delivering software development than I am.

8. “There are more things in heaven and earth, Horatio, Than are dreamt of in your philosophy.”

References

Keith Johnstone. Impro for Storytellers.

Keith Johnston. Impro: Improvisation and the Theatre.

Norman F. Dixon. The Psychology of Military Incompetence.

Edward Yourdon. Death March

Robert B. Cialdini. Influence: The Psychology of Persuasion.

Robert Dilts. Sleight of Mouth.

Gary Halbert. The Boron Letters.

Timothy R. Clark. The 4 Stages of Psychological Safety: Defining the Path to Inclusion and Innovation.

Robert Anton Wilson. Prometheus Rising.

David J Anderson. Kanban: Successful Evolutionary Change for Your Technology

Jennifer Rode , Mark Stringer , Eleanor Toye , Amanda Simpson and Alan Blackwell: Curriculum Focused Design (2003) in In Proceedings ACM Interaction Design and Children

Gary A. Klein Sources of Power: How People Make Decisions

Venkatesh Rao - The Gervais Principle

Ohno, Taiichi – The Toyota Production System

Chris Voss: Never Split the Difference - Negotiating as if Your Life Depended on It.

Genrich Altshuller: The Innovation Algorithm: TRIZ, systematic innovation and technical creativity.

The Machine That Changed the World Paperback – 4 Jun. 2007

by James P. Womack (Author), Daniel T. Jones (Author), Daniel Roos

Mary Poppendieck, Tom Poppendieck. Lean Software Development: An Agile Toolkit: An Agile Toolkit

Appendix A

The Tolstoy principle – all unhappy families are different (and all projects are unhappy families)

Appendix B

Feedback from readers

Feedback

I decided to write this book in the open as "working software". And one of the joys of that is that I get feedback from real users. This feedback is from someone I've known for years. I'm not sure what his title is these days. But he's a senior tech guy who still writes software. Interestingly, It's not enough to have a clean windscreen. You have to look out of it. And you need to change your behaviour depending on what you see. There is nothing that can protect you from your own stupidity.

> *[Quoted text] I've been reading Chapter [Chapter Ref] and I have some observations that you may wish to write about.*
>
> *Currently I am working at [Huge Organisation], where there is a dedicated department that does user research - lots and lots of it. All compliant with best practices. But. . . .*
>
> *Recently they've been cutting costs, and so have been reducing the experience in their user research department. Most people there are now new graduates or apprentices, and are now guided by a 'user research manual' that is effectively a step-by-step instruction book on how to conduct user research sessions - nothing about gleaning anything useful.*
>
> *Second, although user research is done; almost every project has some user research at some point; it gets ignored. It's all 'thank-you very much for this' and then they just go ahead and build crap.*
>
> *So, maybe you want to write something about customers who pay lip service to UR and how you can still make progress when faced with such atrocities. And also, when the team KNOWS they are building crap but are told "we are paying for it so we can dictate how it should operate" [/Quoted* text]

Two things I'd say about this. Firstly, it's a really bad idea to have the user researchers working in a dedicated department. They need to be in the team. Yes, they might also need to regularly meet up with people of their own skill set. But they need to sit with the team and be part of the meetings that deliver transparency (stand-up, planning, show and tell, retro). My experience is that, at least initially, nobody will like this.

The user researchers would much rather just get on with doing their user research and then write a report. The developers don't want to have to change the UI. They don't want to change how a screen loads or totally rethink the logic of an application, because of research feedback from users. So, both groups are much more comfortable sitting apart. Why do you think that is? Does this sound

to you a lot like one group agreeing to mop the ship's floor and another group agreeing to stay in another corner polishing the woodwork? That's because that's exactly what this is. Double agreed activity.

If you want the project to be a success, you need to get user researchers in and amongst the developers. You need to get them to listen to each other. The more you do this, the better your chance will be.

The people who are building the swamp need to speak to the people who are talking to the people who are interested in the swamp.

You might feel bad about this, because, if you know what you're doing, you will know that you're creating conflict. But the kind of conflict that you're creating is the kind that we talked about when we talked about the pirate ship in chapter [Chapter ref].

One way of getting the improvisers to do the interesting thing rather than the boring thing is to have a director. Is to help them out of their agreed activity. Have a director who shouts "deal with the pirate ship." What we're doing here is putting two points of view together and insisting that rather than just go off and do their own thing, they figure things out together. It's like someone shouting from the edge of the stage "come together and sort this out."

Don't worry too much if the user researchers are juniors. Just worry about getting them to sit with your team and tell your team about what they're finding. Most software development projects don't have any user research at all. Also, you never know when some of the people who seem to you impossibly green will turn out to be stars.

Some more feedback from a reader. I've kept the swear words because I think it's instructive that people are actually using this kind of language when they're talking to user researchers.

> *[Quoted text] A little story. I was on a User Research presentation for [Big Org] where they presented the user research. It was harrowing. Basically, the entire user base said they wouldn't use the software (even though the law will make it mandatory soon). We had warnings before some of the recorded snippets of the language used by the users.*
>
> *"If Big Org introduce this then I would just say"Fuck You Big Org" and I will rebel by doing everything on paper."*
>
> *"I'm not doing this. I'm just not. What the actual fuck is this shit"*
>
> *Yet, they are still building it - knowing full well that NO-ONE is going to use the software. [/Quoted text]*

Well, let's think about this in "Swamp terms." Yes, there are some people who are very rich and powerful. Ultimately, if they want to build something in the swamp that nobody wants to live in, they probably have the power to do that. A good question to ask is if there is *any* group of stakeholders who could have some

influence over these rich and powerful people. The more stakeholder research you do, the more chance you might have to discover who these people might be. For example, in this case, if a form is mandated by government, then there are MPs and ministers who have some interest.

Users who are forced to comply with regulations often form an interest group or pressure group. There might be more than one. They represent those who are being regulated. Often whoever it is who is ignoring individual users may well still listen to representation from these groups. Think of it this way. Everybody has a boss, well almost everybody. And everybody has a constituency, a wider group of people that they would really like to please and would dearly love not to annoy.

So, user research isn't enough. You need to be mapping the stakeholders on your project and figuring out how to get to them. Maybe it's with user research, maybe it's with working software. I used to be very scared of ridiculous deadlines (I still don't love them). But what I've started to see is that they can be used as an opportunity to push working software further down the pipe. What if part of that process is getting the software working on a "live" test server? Obviously, it wouldn't be using real data. But this could be the kind of thing that pressure groups could get a look at and have an opportunity to provide their - ahem - feedback.

Style guide

Scrum Master

Agile

stand up

Is "team" singular or plural – looks like it's singular.