MARK STRINGER

DELIVERING THE IMPOSSIBLE

Seven metaphors for successful software development

Delivering the Impossible

Mark Stringer

Chapter 1 - Introduction

The title of this book is "Delivering the impossible." But of course, you, or anyone, can only deliver possible things. So might ask yourself, "Why on earth did I pick up this book? It has a contradiction, an impossibility right on the front cover."

Here are some reasons why you might have picked up this book.

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 is impossible. Someone promised something. You know that your team can't deliver it. Or you don't know what they promised. Or you do know what they promised, but you're certain that your team can 't deliver it on time.

Every direction that you look in there seem to be problems that can't solve. You have found this book. You think it might be worth a read.

Maybe it isn't that. Maybe it's this. You have a group of people working on a project for you. You've got this uncomfortable feeling that the project isn't going well. For the sake of the company or for the sake of your career, or both, you need this project to succeed. You wonder if there any useful tips in this book that you can pass on. Is there anything that *you* could do to help the project succeed?

Even though you think the project is impossible, you haven't given up. And it's worth asking yourself why? If the project seems so impossible, why hasn't it been stopped, abandoned, killed? Put to sleep. Why would anybody carry on with it?

Well, part of the reason is your inherent optimism. You know that a lot of things that seem impossible can be done., given the right skills and expertise. You know this from other parts of your life. If you can only change your point of view, things can improve dramatically. If you look at things in the right way, they get easier to solve. If you do the right things at the right time, things that used to seem impossible can turn out to be to a success.

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 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. Is should tell you how to keep yourself and others safe. How to get away from a lost cause and move on to succeed on other projects.

How to ride into the sunset to fight another day.

Let's imagine you're a demanding reader. You might not just want advice on impossible projects. You might also want to know how to deal impossible programmes. Or even in impossible organisations. Yes, if you're demanding, you might want such a book to talk about how to deliver the impossible at scale.

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 will learn, how to identify situations that seem impossible. You'll learn how to distinguish them from those situations that are impossible. 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 succeed.

Why This

OK - so this book is about delivering the impossible. So why is it subtitled "Seven metaphors for successful project management delivery." Well, that's because the main thing that I'm going to do in this book is talk about different ways of seeing project management and software development.

The software pioneer Alan Kay is often quoted as saying "Point of view is worth 80 IQ points." Which I take to mean that if you can only figure out the right way to look at something, you can do much cleverer things. What I try and do in this book is provide the reader with several different ways of seeing the business of project management, especially of software development. The aim of providing these different ways of seeing is to make you much smarter.

What it's important to understand is that I'm not doing this because I think there's anything wrong with the idea that in order to be more successful with software development projects, teams should use Agile methods, and in order to make those methods most effective, at least at first, they need help from people who've made it work in other places - they need coaching. I believe both of those this are needed.

The trouble is, I don't think they're enough. It's true that following an Agile method like Scrum is a good way of improving your chances of a software development project's success. It's what I would almost always do if I were put in charge of delivering a software development project. It's true that if you add into this mix some extreme programming engineering practices like continuous integration, pair programming, test first development, then you're make success even more likely.

It's also true that if you haven't done this before as a team, you are going to need some outside help and coaching from somebody who has done it before.

But unfortunately, this just isn't enough. You need to do more than this if your project has any realistic chance of succeeding.

So, for example. In the first chapter, you need to be able to notice when you and your team are quietly agreeing to ignore the big issues that need to be tackled. Using the metaphor that I talk about in the first chapter, you need to stop you team from looking down at their feet and mopping the decks and get them to look at the horizon and face the approaching pirate ship.

Why is it important that you tackle "pirate ship" issues as soon as you see them? Because if you don't, you and your team will end up (metaphor number 2) in trench warfare. And once you're there, it's much harder, although not impossible to help your project to succeed.

Why do you need 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 the project? Why do you need to put working software in the hands of people who might genuinely benefit from using it as soon as you possibly can? I try to explain this using the metaphor of flowers which are appealing, but ultimately not sustaining-sustaining and fruit, which is difficult to grow and even more difficult to make palatable but can ultimately feed people.

By doing this, by providing these different ways of looking at software development, the promise of this book is to provide its readers with something, solid, nutritious, and sustaining. The genuine possibility of knowing what they're doing when they are project managing software development. And once you know what you're doing, a lot of things that previous seemed impossible can seem a lot less so.

Why now?

Well, another way of thinking about this is – "Why not *then*?" Why haven't we been trying to find better ways of thinking about software development since the very beginning.

We've certainly needed better ways of thinking about software development since it became a thing that people did in the 40's. I remember listening to a description of a "Waterfall" approach to managing software development in the early 90's in my first jobs as a software developer and thinking "That'll never work." And it didn't.

The first two projects that I worked on, one for big oil company and one for the military, had both been developed using what was then the accepted metaphor for software development - the waterfall approach. The project was tackled in stages. In a first stage, all the requirements for the project were carefully written down to produce a huge requirements specification. In a second stage, these were developed into first high level and then detailed designs. Following this, finally software was developed against the designs before final stages of first internal and then external testing. Then and only then did the software go live.

Unfortunately, both of the first two projects that I worked on went through an extra stage between external testing and launch - litigation. When the customers

finally got to see what they'd asked for months, if not years earlier, it really wasn't what they wanted. This was probably for a whole bunch of different reasons. In the two years since they'd specified the project, it's possible the world had changed. But it's also possible (very possible) that the specifications that were written in the specification hadn't quite captured exactly what the users wanted. It's also possible (very possible) that the users hadn't really been capable of saying in one long fluid document, exactly what it was that they wanted. By the time what the users had asked for had got through the systems designers, the developers, the internal testers and then the external testers, it's hardly a surprise that things had gone astray.

One intriguing thing, that I barely noticed about those first few projects that I was involved in, was that when it got to the crunch phase, when the customers finally saw the software, decided it wasn't what they wanted and threatened to sue, something interesting happened. The project would call in a guy whose entire job was to rescue projects that got themselves into this state. He 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 some room for manoeuvre. A bit of time and a bit of money from the client so that bit of the software could get built. And 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 they would drop the threat of a lawsuit.

And what those guys were doing, well it kind of sounds like iterative, Agile software development.

So, one answer to the question - "Why now?" Is well, well not ages ago? We could and should (and kind of were) doing this right from the beginning."

But there's another answer to this why now question. Why now? Because being good at writing software, being good at doing it in a timely manner, being good at putting software that people want to use in the hands of users, is become more and more important.

As Marc Andreessen pointed out more than a decade ago now, software is eating everything. Since he wrote that article, software has taken huge bites out of sex and dating, politics, social life music and television. Some of the companies that are good at developing software (Amazon, Facebook, Google, Alibaba) are very arguably more powerful that governments.

So, if you're interested in being good at something and even slightly susceptible to megalomania, getting good at developing and delivering software is worth a look.

But ironically, the craft of managing software delivery doesn't seem to be so susceptible to being eaten by software.

Why me?

So why should you list 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 German guys who were writing code for the first "smart" phone - the Nokia 9000. This was a really dumb phone. It had the same operating system as some sewing machines and it was a nightmare to write anything for it.

Working for Cambridge University gave me my first experience of using iterative, user centred design. We designed a system to help school kids put together discursive arguments. The funding was for an interface that was tangible, an interface that you could touch, pick up. This project taught me a lot.

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) electronic interface that used cards with statements like "People don't like graffiti," to search the internet.

Whatever you're building needs to fit in a particular situation. The situation that we were designing for was English schools. They were at the time 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 ultimately, our funders. I didn't think of it then like this, but that project was definitely a swamp.

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 or they turned out to be things that nobody wanted, or people wanted but nobody could make pay. Turns out that users are quite keen on personal loan websites, but most of the people who want a personal loan are exactly the kind of people that you probably should be loaning money. People need to keep re-learning that user research is a good idea.

Some of the projects that were successful made millions of pounds for their owners. Some of them weren't about money, they were literally about making the world a better place by helping their users do scientific research or helping their users make justice in courtrooms work more smoothly.

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

might not have been so successful, if I hadn't been involved.

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.

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

Why you?

Why you? Why should you read this book? Well, here's what's going to happen to you if you try to deliver a project and you haven't read this book.

If you try to deliver a project and you haven't read this book. There's a good chance that you won't spot the pirate ship (Chapter 2) 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 3) and it will take you way to long to realise that there's nothing you can do about it, and you need to leave.

Let's say you and your team get lucky and you do admit to yourselves that there's a pirate ship coming. If you haven't read this book and you don't know about flowers and fruit, you'll be really tempted to put off trying to put working software in the hands of users until way too late.

If you haven't read about the swamp in Chapter 4, there's a good chance that you won't have hired the best user research team you can find and yo won't have a detailed map of the user and stakeholder ecosystem. If you haven't, even if you tackled the pirate ship the minute you saw it, and even if you pushed working software as far as you could along the right dimensions, your project might still be as much of a success as it could be. There's less chance 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. Oh, wow, I just thought of this, 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" really is and what you can do to escape its evil clutches.

Chapter 2 - Agreed activity

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 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 blocking. For example. If an improvised scene starts with someone knocking on a door. The scene might go something like this.

Person 1: [Knocks]

Person 2: [Opens imaginary door]

Person 1: Hello! I've brought you a cabbage.

OK, now we've reached a key stage in the drama. Because Person 2 has lots of different things that they could say. Person 2 could be happy that Person 1 had brought them a cabbage.

Person 1: Oh you remembered that this is the week that I make all my Kimchi! Oh darling! You're so thoughtful.

Or they can be angry that Person 2 has brought a cabbage.

Person 1: And that's dinner is it? You know cabbage doesn't agree with me! Remember what happened last time. You're such an ass-hole Kevin.

The thing about both of these responses is that they move the action forward. They take someone's first idea, no matter how humble, and move it forward.

In improvisation, this is known as "Yes and." Accepting whatever your partner gives you and amplifying it. The opposite of "Yes and" is called blocking.

So for example, if we go back to what will forever now be known as the "Cabbage scene." and get Person 1 to knock again on the imaginary door. Person 2 could respond to the "offer" (as it's called) of the cabbage with some kind of bizarre argument.

Person 1: Hello! I've brought you a cabbage.

Person 2: No you haven't that's a Pomeranion poodle.

or:

Person 1: Hello! I've brought you a cabbage.

Person 2: Humph. Fine.

In improvisation classes, they teach that these aren't the kind of answers that result in an fun scene. Why? Because they "block" the scene moving forward. They don't build on it and move it forward. The poodle response is completely trying to stop the "offer" of the cabbage. It ruins any chance of an interesting cabbage story.

What could be an interesting cabbage story? The story of two people that learn to fly on the wings of their own farts. Or a story of the dangers of living with a pot of Kimchi that brews for so long that it becomes self aware.

The second answer is "Hmph. Fine." This is even more dangerous to a good story. It goes nowhere. Again the story stops dead in its tracks. What's happening

in such a situation is that the person who is speaking is feeling fear. They feel out of control, so they're doing the least that they think they can get away with. The result is very boring to watch. Also it is not very nice to the other performers and defeats the whole object of improv. The whole object of improv is to create an interesting story. But even so, the performer feels safer than if they had accepted the offer of cabbage. Because they didn't know where the cabbage might lead.

Here's another way of saying "Humph, fine."

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

Here's what might happen.

Person 1: [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?

As with the first scene that we talked about, the other performers have a decision to make. One obvious decision here is to "yes and" the Pirate ship.

Person 2: Oh my God the Pirates are coming. Haul up the sails, let's try to outrun him.

Person 3: Oh my God the Pirates are coming. The cannons! Load the cannons!

Person 4: Oh my God the Pirate 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.

But what happens in such situations is a different kind of suggestion. Something like this:

Person 5: Hey! Let's wash the decks!

Persons 6, 7, 8, 9 and 10: 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 mentioned the pirate ship, they want to see it arrive. They want 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.

For inexperienced improvisers, the temptation to agree to wash the decks is strong. It is much attractive than dealing with the implications of the pirate ship.

Why? Keither Johnstone invented his own style of improvised theatre in the 1950's. He also wrote several books: "Impro" and "Impro for storytellers." In these Johnstone describes this kind behaviour 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 group as a whole less safe.

Part of what 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 there's a pirate ship. They make the story intersesting and move it forward by dealing with what that means.

What's fascinating is that all novice improvisers seem to know that it's the "right" thing to do. There isn't a long discussion. It happens in a second. This is in spite of the fact that, 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. They also see which are "agreed activity" and tell the actors to avoid them. In desperate situations she might also make suggestions herself.

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 and nearer! Deal with the Pirate ship!"

Why am I telling you this? Because, of course, this idea of agreed actity is useful in project management. People who work on a project can prefer agreed activity to moving the action forward. Like improvisers, they do this as a way of avoiding having to think, or needing to change. Instinctively they shy away from the realities of the problem that they're solving.

We can start to deliver the impossible by avoiding agreed activity. We can start to deliver the impossible by spotting the pirate ships on the horizon.

In Agile project management there are lots of opportunities for 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, what they're going to do today. And in stand up team members talk about any problems they have that blocking their progress. This is the most important bit. These problems are the pirate ships.

Of course, if there's one big thing that's blocking progress, that might be the thing that no-one talks about. 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 "Arrrrr!"

Agreed activity might be a sign there's there's something everyone is ignoring.

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 the team plans a task and then it doesn't get done.

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 story and tackle it. As with the crew on the improvisational stage, they agreed, they might not have talked about it, not to tackle it. Of course, in these situations, it's the job of the project manager to hold up with story and ask what it is about this story that means everyone is avoiding it.

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. What went well, what didn't go so well, what could be done better.

One of the strange things about project management is that very often everybody who is working on a project knows what's wrong with the project. If they're not saying, it's because they've tried to say or have said and have been either ignored, told to back off, threatened or even disciplined for pointing it out. This is the same as if, in an improv show, one of the performers shouted "Look! A pirate ship!" and a director sitting next to the stage shouting "Shut up about the pirate ship! Clean the decks!"

A leader doesn't need to tell the team to shut up about the pirate ships and clean the decks many times. Once they've done it once or twice they will have already caused one of the strangest problems that I've seen. And I've seen it repeatedly: highly skilled, highly paid team who've given up trying to think for themselves.

No matter how many times they've been beaten down, a team will still respond to the right kind of opportunity to feed back. 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 notice see and appreciate you doing your best to raise their problems with the people who need to know about them and can fix them. And sooner or later, they'll talk about the pirate ship. When they do, it might be hard to keep your jaw off the floor.

For example during the retrospective for one project that I worked on, it emerged that all of the requirements for the project were coming from a business analyst. They made no sense. One of the main struggles that the development team had on any day, was making sense of what the business analyst had put in the stories. What struck me straight away was that the business analyst wasn't in the retrospective. In an agile project, the business analyst is part of the team -

they're not supposed to be outside throwing in requirements. Especially if those requirements turn out to be problematic.

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

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

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

Yes it was crazy. Yes, it was right. 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 succeed it its current state. The team needed to talk to the person who could tell them what the product was supposed to do.

Side note: This project was crazier than it even sounds. It turns out there were two groups of business analysts. Technical business analysts and business business analysts. The "business" business analysts were actually allowed to talk to the "business" - the people who wanted the product. The technical team was only allowed to talk to the technical business analysts. And these two groups of business analysts - technical and business - also weren't allowed to talk to each other. They were only supposed to communicate via emailed word documents. A substantial part of making this project possible was to make sure that we had just one business analyst. This person was allowed both to talk to the people who wanted the software and the people who were building the software. Things got a lot better after that.

So we've talked about stand up meetings and we've talked about retrospectives. There's one more Agile meeting where the "Pirate Ships" that are on the horizon might be detected. And that's the "Show and Tell." The "Show and Tell" is a meeting where the development team show the outputs of what they've been working on in the last "Sprint" - typically two weeks. Who do they show it to? Definitely you should show it to the product owner who is supposed to act as a representative of all the people who want the project to happen. But you should also invite other people who are interested and who want to come along.

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. Fraud was an issue that affected people who worked for the bank all over the world. So for the first phone conference I could hear just from the accents the geographical spread of interest. Northern Irish accents, Scottish accents, Cockney-sounding Southend accents and Indian 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 detection journeys.

One of the developers put up the login screen and then clicked through the first screen and started to talk it through. Even though most people who were on the conference call were on mute, I thought I detected some kind of 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 demoing the screen looked blank. The business analyst who we'd only just set eyes on and had only just joined the team started to look worried. "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 someone holds, and see the activity across all of them for this to make any sense," said someone 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 front end) in front of the people who might use it had found 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 *badly* - 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 understanding 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 possible of actually 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 rather than looking at the pirate ship and doing what needs to be done when it arrives, they are washing the decks and in the process are making possible projects impossible.

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 the job that they can't do? Maybe it means that someone hired the wrong people.

Highlighting the problems that starting to do a project shows up can be threatening to the sponsors of a project. What if the problem that you find is something

that they haven't thought of and they don't know how to fix? They will be tempted to avoid addressing it, possibly by attacking or threatening the team for even daring to raise the issue.

Too, too often, when someone on the stage shouts "Look a pirate ship," It's the director off stage that shouts something like "You're wrong, it's not there."

Other ways of saying this 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 why people don't raise problems and instead just wash the decks.

Oh dear. Well. There is a good chance that you picked this book up because you wanted to deliver what seems to be an impossible project. I'm keeping that promise. But notice I'm not keeping the promise 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.

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 not very nice names. But there there are two very goodreasons why you still should do it.

Reason number one is that, despite the resistence you might experience, solving these problems is still the best chance you've got of delivering this project that seems to be impossible. That's a solid reason.

But reason number two is possibly for me, just as important: I know what happens when you don't.

Chapter 3 - Trench Warfare

Two members of my team were supposed to be putting their software on a clients machine. They'd written a script to run on a clean machine that would put our software where it needed to be. 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. But running the tools we had to check for problems, it turned out that they already had other software running on them. When the pointed this out to the people who were supplying the servers, they began to act very strangely.

It was a Sunday afternoon. And I was in an emergency meeting. The topic of the emergency meeting had been what we were going to do about the failure of my team to load our software on the clients machine. The client was implying very strongly that the reason we'd failed was because our guys weren't up to the job. Just like the guys in the server room, when I brought up the subject of

other software already running on the servers he became evasive. He moved the topic of the emergency meeting onto what we should do about the fact that our three o'clock emergency meeting had overrun into our four o'clock emergency meeting.

Ultimately the mystery of server software already running was solved. The company that were providing the servers were in some kind of dispute with the organisation that wanted the project. While they were in dispute, they were working to the letter of their written agreement - and so providing servers that already had some software loaded, 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 officially 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 never heard of it again, I'm certain it never delivered.

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 it is set up, stuctured and managed is changed. Why are we talking about trench warfare now? Because in the last chapter, we were talking about pirate ships. Yes, I know, I know, I'm mixing metaphors.

What I mean by trench warfare is any project that is hugely expensive in terms of money, effort and human suffering and doesn't get anywhere. A main cause of trench warfare is a refusal (or active prevention) earlier in the project to deal with the problems that it faces. The main effect, besides project failure, is that there is more demand for the "agreed" activity which is just a way to avoid the problems to become more frantic, pointless and damaging.

Working late. Working weekends, holidays being cancelled or frowned on, "a positive, can do attitude" are signs that you're working on a trench warfare project. Other signs are lots of staff leaving drinks and staff being signed off long-term sick, or with stress. If you can't have a meeting with a manager because he has to give evidence in a law suit that people who use to work for the project are bringing against the company, this is also a bad sign.

One of the things that personally I've found most upsetting about working on trench warfare projects is when senior management say that the staff don't look stressed enough.

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've worked on a trench warfare project, you might be much keenr to call out problems the minute you see them on your next project. But every now and then you are going to find yourselves working on a project which is working towards a ridiculous, undeliverable deadline. At the same time

this project will struggling with several other problems. This is a very common kind of undeliverable project.

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 don't solve people's problems, I televise them."

If you're working on a trench warfare project, that's 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 can't solve? Broadcast them. Make sure everybody on the project knows what those problems are.

Why? Because someone else may, if they want to 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 you're team don't look worried or stressed enough. Those people who are saying that your team don't care about the project because they took the whole weekend off. 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 pproject more successful, they just don't know how. At least by making clear what the problems are that are stopping you delivering, you're giving those people more chance of to help the project if they do want to help.

The second thing that you can do is to deliver something to some of the people who want it. This might not be anywhere near everything that's 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. Pull from outside a team that comes from demand from real users is a truly wonderful thing.

Thirdly, one of the most revolutionary things that you can do, is to track and show the actual progress. 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. We'll go through it in detail, well at least one way to do it, in a later chapter.

Demonstrating that a project can't possibility be delivered by a deadline can result quickly in some sensible discussions about reduction of scope. This can then, sometimes, result in the possibility of delivering 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. Still, by pushing on all of these three "fronts" and getting some kind of virtuous circle going, it might be that something good can come out of even the worst trench warfare project.

It might be that something could be done to save a trench warfare project. But equally, it's important to remember that some projects cannot be saved. I don't

think I've ever worked on a project that was being deliberately targeted to fail, but I have heard talk of such projects.

It is quite 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. Paul Simon points out that there are multiple ways to get out of a difficult situation. There might not be as many ways of getting off project, but there are certainly a good few.

Of course, you can 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 who is looking. Recruiters have a very good idea about which projects are going badly.

Chapter 4 - Flowers and Fruit

This is an idea that I picked up from reading about Taoism. There are two ways that you can get paid.

You can get paid for looking good and you can get paid for doing good. In Taoist terms. Getting paid for looking good is "Flowers" and getting paid for doing good is "Fruit."

Don't get me wrong. It might seem as we go through this chapter that I'm down on flowers. I'm not down on flowers. Flowers are great. In project management, the part of flowers is played by ideas.

People have ideas, those ideas get money 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. 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 some ideas, the kind of ideas that get money to turn into projects seem to have a common structure that makes people like them without much thought. Just like flowers, those who are exposed to them tend to like them without doing much or any thinking.

Often this is in terms of easy, fast, cheap, same and all.

For example - "Does everything that the old system does, but cheaper and faster."

"Deals with all customer enquiries, without the need for human involvement."

"One stop shop for everybody who is trying to do this job."

Just like flowers, ideas get paid for being attractive to people before they even think about it.

What about fruit? What's the difference between fruit and flowers? It isn't that they are hard to grow. There's lots of effort involved in growing both. And although it might *look* easy to put together an attractive idea that can gets

money. Clearly it isn't, otherwise everybody would have managed to get money for their project.

No, the difference between fruit and flowers isn't that one is easier to do that the other. The difference is that the fruit has to actually be eaten. The users of fruit interact with it in a completely different way to the users of flowers.

Users eat fruit. The only look at flowers. Nobody ever died from looking at a flower. A sour apple can give you bad indigestion and some fruit is actually poisonous.

Similarly, implemented, delivered projects have to actually give value to their users. This involves an interaction with users which is different from buying and selling flowers. User have to actually 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, the process of "using" software for users is one where, if it doesn't taste right, they will spit it out. Also like fruit, if software isn't looked after properly, it could do them harm.

To be honest, I wish I were able to cite some other reference in support of this idea of "flowers and fruit" than a 2000 year old text.

Because for me, moving a project from its "flowers" state to its "fruit state" is one of the most important things that anybody is doing when they are trying to deliver a project. It's also one of the hardest, and it's one that hardly anybody talks about.

What makes this so hard? Well, one of the things that makes it so hard is the nature of project ideas that we talked about. What makes project ideas attractive - the kinds of things that get money - is that what makes them liked doesn't require conscious thought. Often what makes people like things straight away is that they don't require much thought, but they do suggest some of these features: easy, fast, cheap, same and all.

The problems start when anyone tries to implement the idea. This is especially a problem if someone tries to implement the idea using an iterative framework like Scrum. In Scrum the team meet at the beginning of every work day and talk about what they're going to do that day and anything that's stopping from doing what they want to do. They do this every working day. This is called the stand up.

Once a fortnight, the team shows the work that it's been doing in the last two weeks. The audience is made up of people in the organisation and also sometimes also external users who are interested. This is called the show and tell.

The team also has a private meeting where they share what went well, what didn't go so well and what they might try differently.

Can you see yet how there might be problems for the "Flowers" aspect of a project? This idea that people liked without thinking to much about it? The

minute that a team starts to implement the idea of a project they will get problems. And if the team is using an Agile way of doing things, like Scrum, those problems will start to be reported. Right away. On day one.

So? What's the problem with problems? The problem with problems is that any problem that the team finds is like to take the shine off the idea. Any problem that the team finds is likely to make the idea look harder to do, slower, more expensive, different and partial.

So, this is the problem at the heart of project management and project development. It's one that I wonder people don't talk about more.

> Nothing makes a project appear less attractive than actually starting to do it.

So what can be done? To solve this problem? What absolutely must not be done, as we discussed in chapters two and three, is to avoid the genuine problems that the project is facing. That way failure or possibly trench warfare lies.

The project is entering a vulnerable, cold phase, where it is not supported and valued for the appeal of the idea. This because we are starting to find out what is wrong with that idea. At the same time there is no support and value for the reality, because there is no reality. We don't have anything to show yet.

As someone who is trying to help this project get delivered, it is important to do two things. Firstly, it's important not to avoid the move of a project through this phase. This is what we talked about in Chapter two, when we talked about agreed activity. It's also what we talked about in Chapter three. A project can get into an awful state if the problems that agreed activity is there to avoid aren't tackled.

So, someone who wants a project to succeed shouldn't do anything to stop it moving into this phase. At the same time, you should know where you're going and what kind of things can should be done to get you out the other side of this phase.

OK, here's where it gets even harder. There are two kinds of thing that you should be doing to get the project through this tough phase and on the road to success. And here's what makes this hard. Both of the kinds of things that you should be doing aren't often popular or well-recevied. They are the kinds of things that the sponsors of a project are likely to think of as either a pointless waste of time or the kind of thing that should be stopped.

What are these two things? Stakeholder research, which I'll talk about in more detail in the next chapter, and working software.

What do I mean by stakeholder research? I mean as a product team developing a good understanding of who is interested in the product and what their interest is. Notice here that I'm not saying "user research" that's because stakeholder research doesn't just include users, it include other individuals, organisations, interest groups that might be interested in the product.

I'm resisting saying things like "find out as much as you can about the people who are paying for this project" or "find out everything that you possibly can about your users." Why? Because as a seasoned and grizzled project manager I'm very careful of statements that involve "all".

But I can say this. You need a strategy for mapping the ecosystem of stakeholders. You need a strategy for investigating the needs of users. You need a way of taking what you find and using it to inform what goes into working software.

So how do we do this? How do we acheive this move? From "Flowers", where value and support for an idea comes from the idea itself to the "Fruit" where value and support comes from a real working piece of software.

Chapter 5 - Working Software

> "sufficiently advanced technology is indistinguishable from magic" Arthur C. Clarke

Concept

In the Agile manifest, working software is talked about as the second key value.

> Working software over comprehensive documentation

It's worth asking possibly, why? What was the experience of those guys who got together to talk about "lightweight" software methodologies that meant that in the final 93 word manifest, working software took up five of those words.

I don't know for sure, but my thought is that they had all had experience of being involved in projects where the production of working software was delayed for a long time while the specification was agreed.

It's important to remember that 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, nothing is built before the production of detailed plans.

But what is this second principle in the Agile manifesto saying? It's saying the engineering equivalent of "have a go at building a bridge and see how you get on."

So why? Why would people who were experienced in the software development business and had got together for the express purpose of making the way that software development happened better, why would they want something about working software to be 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 about how do this thing. 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 totally 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 do this - trench warfare.

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.

Conclusion

I think there are three main reasons why any software development team should be trying to develop working software sooner rather than later.

The first reminds me of a joke.

> You will never be alone, if you take with you everywhere the ingredients and equipment to make a dry martini. Because, even if you think you are completely alone and stranded on a desert island, the 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. The more you try to get working software loaded up on the environments where it's going to be used, by real users, using real data, the more likely it is that people that you didn't even know existed are going to jump out from behind trees and say "you're doing it wrong."

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.

I'm mixing metaphors again aren't I? 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 what are known as “non functional requirements.” The software needs to be accessible. The software needs to be secure. The software is covered by some regulator in the industry that you have never heard of.

Of course, one way to tackle all the rules and regulation man traps that your software might set off is to try to take care of them in the specification before any software gets written. But in my experience (and I suspect in the experience of the people who wrote the Agile manifesto) it’s very hard to find out exactly what you can and can’t do without trying to do something. 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 quickly 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) but also known unknowns (like the guy jumping out from behind a tree).

OK. This metaphor is in a blender. But it still applies. Why try to do working software? Because if you do a man will jump out from behind a tree and tell you why you can’t and the only way to find out what there is in no-man’s land that’s stopping you and your software from escaping is to get 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.

I’m teasing this out from the kind of non-functional requirements, rules and regulations reasons that might be stopping you from getting some software working. This is more straight-forward 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 answers might not be the one 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 straight-forward 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, important and clever, whenever you can, to try to move a project towards delivering working software earlier rather than later. By doing this, by putting working software in the hands of the people who will use it, you start to solve the flowers vs fruit problem that we talked about in the last chapter.

Because you move the discussion and the dynamic of the project away from flowers - a discussion about the list of things that a piece of software will do to fruit - something that is in people's hands that will do something that's useful for them.

I worked on a project for an organisation that was still doing pretty much all of it's business using paper documents. This was a huge organisation, and it processed a lot of paper. The project that I was working on was trying to take just one of the processes that this organisation did and manage it using an electronic document management 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 new all the other clerks and she knew their business very well.

To start with the project had a tough time. We could get servers to put the software on. The document package we'd chosen wasn't as mature as we thought it would be. But we pushed on through some early design iterations and a load of technical problems until we got to one show and tell where the team finally had some 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 what this demo did 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 dialog window came up that said "Do you want to veiw your document?" and when the user clicked "OK" an error message appeared.

The look on the product owner's face! At this point were were about £500,000 into a £2 Million project. And all she had to show was a misspelled-spelled but that led to an error message. It was a hard time for her and it was a hard time 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 seems 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. It provided a good extra reason to negotiate with the people who were supposed to be giving us server space.

Once that was in place, the product owner went on the road with her laptop. The demo still wasn’t much. The demo still had bugs. We still had some spelling problems. But the demo by the product owner to her own kind went over very well. She could wave past any user interface issues, or problems and she could show her old work mates the 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, and a list of suggestions for features that had come from the clerks, but the main question that she came back with was a demand “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 and then adding to that the next obvious steps.

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

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 included in the contract, the client was arguing, and the client shouldn’t have to either commission them, pay for them or support them. But this became a much harder argument to make once people in their own company were asking for the software. Suddenly they were in the way, rather than helping the company by being tough on costs, they were stopping people who worked for their organisation from getting at something that could make their lives easier. The servers appeared, and then money for staff to support them appeared and 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

the there was no way the software could be released until it met a extra set of requirements that would make it usable according to a yet another set of standards that try to make websites more accessible to people with disabilities. The client claimed that we should have know about these requirements right from the start and so we should pay for them.

And by the way. This guy was pretty much 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 who you might not think of as disabled to use your site. Do it. If in doubt, pay a blind guy to look at your 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 an 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 born 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 was also not telling us what it was 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 improved the accessibility, but we went live with what we had. We submitted the software to outside security testing. We addressed some issues, but we went live with some others still being looked at.

Putting bad, barely working software in front of real users completely changed the project.

Connections

Showing barely working software to the clerks had created this magical thing, which gets talked about when people talk about other ways of doing Agile - like Lean, which is a lot like Agile, but for manufacturing. What is this magical thing? It's called "pull".

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: it just wasn't possible to make cars in Japan in the way they were being made in the 1920's in America? Why? Because Japan's economy at the time was tiny and it went in cycles of good and bad times. So from very early on, Toyota

tried match the rate that it made cars to the pattern of demand for cars that it saw from the market. Over nearly a hundred years, that process has become so sophisticated 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.

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

Thinking about things in terms of providing value to the customer results in another important idea which comes from the way that Toyota do things - waste. In Toyota's way of thinking. Anything which money has been spent on, which isn't on it's way to a 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 and don't do anything that isn't on its way to customers, you end up with a process that fits the labels that have been applied to it in the west - lean and "Just-in-time."

It's important to point out that making cars is *very* different from making software. It's also possible that reports and descriptions of how the Japanese, and especially Toyota, make cars, might differ a lot from how it's done today - or how it's ever been done.

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 is 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. Which feels about right to me. If you think your project is about 6 months long. I'll give you a month to do some set up. But if after a month you and your team don't have *something* that works, you've waited to long. By the end of a sixth of the time, you need tiny piece of software that starts to do the thing that you're supposed to be doing in this project that you can show.

And once you've got this tiny little thing, you should be looking to improve its status in some way. And there are lots of ways to do this. Sure, add functionality. But if this first working version will only run on a laptop, that's not the first way to go. Just as in the example that I gave. If you've got an example that works on a laptop, then one good move would be to show it to users. Another would be to move what's on the laptop to test servers and beyond that, to the live servers where it will eventually be used.

What's so 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 directions. 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 is 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 also, without the climb!

Gradually developing working software in all of those directions is the careful and effective way to deliver something which 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 in that process, we hope sooner rather than later, you will start to get "pull". You will start to get demand for the software and demand for functionality from the software from the people who are likely to use it, rather than "push" from the people who got the funding for the project.

Gradually developing software in these directions will also result in people jumping out from behind trees and helpfully providing you with extra rules and regulations that you need to follow before your software can finally escape and live free in the real world. It's (one of) the most powerful secrets to delivering things that seem impossible. And everybody, or nearly everybody, will try to stop you doing it.

What? Yes, that's right. Incrementally delivering working software is the thing that you absolutely need to do to succeed and 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 that could be shown, released and tested. 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 functionality which don't show the whole journey, that it's a waste of time and money.

How? Somebody possibly even some users will tell you that there's no point looking at the new system until it has their real data in it, or until when they press the "launch missile" 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? Nearly everybody (I've been lucky to work with some exceptions) who is responsible for product management will duck their responsibility for deciding what are the most important things that the software should do. They will utter the magic 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 very similar to the reason that the improvisers that we talked about in Chapter 2 don't want to deal with the pirate ship and would much rather just scrub the decks. Without having to think about it, they know that dealing with working software will mean that they have to *change*. They will have to change their thinking. 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 - this is the aspect of the idea that people think is good, without having to think about it. Remember that ideas tend to be of the form "all", "same", "faster", "cheaper."

Trying to get even a tiny bit of software working tends to undermine those 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 that they place on your project. Especially the people who got funding for the idea.

Yes. 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 and the effects of these reasons when you try to implement a small amount of working software will be immediate. It will be very tempting to stop.

So, when you start to feel these reasons for not doing working software, when you see other people being persuaded by them. It's really important that you understand the reasons why you should start delivering working software as early as you possibly can.

1. You're going to have to do it sooner or later.

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, problems with the technology, problems with environments, regulatory problems. There is no way around all of those problems. 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."

This may not be pleasant, but when that happens, you might want to be grateful that you found this out when you were a small percentage into the project, when you've spent just a small amount of your time, money and resources.

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 I can I have this?" And then, almost certainly, they will follow that up with "Could it do this? It would be really nice if 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 that's based on actual value to the users (fruit) rather that superficial appeal to the internal sponsors and funders of a project (flowers).

Once you've done this, once you've found something that users want that you

think you're software can provide, you have moved a long way down the road of shifting a project from being impossible, to being 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 and tell you that you're not allowed to put software out into the real world. There 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 from them which it's obvious to you, you need to satisfy 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 funded it?

All of a sudden, they're not as in control of the project as they were, the project has been let out in to the wild world. It isn't just their baby any more. People may not act to this positively or rationally. It may feel that you've deliberately pushed you and your team into a storm, when they could have stayed in calmer waters. And 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 in mind the the direction that you need to keep pushing the working software. If what you're doing is adding functionality to the software that can then be tested with users, then 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 being live with live data and really being used by real users, then you're moving the software in the right direction.

But you need to be moving forward in all of these directions. A little bit at a time, adding bits of functionality, pushing the software as near as you can to live. Moving from working with dummy data to working with real data. All the time getting feedback from users and stakeholders and dealing with the people who jump out from behind trees and tell you that you can't do what you're doing.

You need to move carefully along along all of these dimensions. Carefully, changing what you do in light of what you find. Why? Because where you're going isn't a nice wide, paved road. It's a swamp.

Chapter 6 - Driving a Car or The Empirical Process

"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 other senses. And that's what the word "empirical" means. It means relating to experience in the real world. But it's an 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 such a strange word, there's an exercise that I often do where I get people too look up the word, to discuss what it means. Then we do exercises. These exercises probably involve Lego and not being able to build as many models as the team initially thought. So half way through the exercise they have to do something different, based on this experience. At the end of exercise, I come back to this word "Empirical." Based on experience.

To make sure that my own performance can be improved based on experience, I get feedback at the end of the courses I run. One of the questions that I ask is "What's the most important thing that you think you've learned from this course?"

In that box, someone had written "The importance of Imperial Progress."

Yup. Clearly, it's a tricky concept with a funny name. Which is kind of strange, because it's also something that we all do every day of our lives, and it we didn't we wouldn't get very far, or live very long.

What are we doing that's so vital? What we're doing is honouring 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, is when we're driving a car.

Just think about it, when you're driving a car down the road, you need 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. Preferably, you'd like to also be able to see behind you through your mirrors - so the wing mirrors need to be there and the back windscreen needs to also 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 quickly.

But transparency isn't enough. If you're driving, it isn't enough for the windscreen to be clear, you need to actually look out of it. If you're not looking out of the windscreen and checking your mirrors, if instead, say you're checking messages on your phone, or distract and desperately trying to brush scalding hot coffee out of you lap, this is also not good. Transparency isn't enough. You need inspection.

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 noise. 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 in you're in the middle of committing a back robbery. But otherwise, you'll slow down.

So inspection isn't just a visual thing. It's paying attention with all the senses. And 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, or speed up, or swerve to avoid something, or put the fog lights on, is we're changing what we're doing because of what we see, hear and feel.

We couldn't do this without transparency - if the windscreen were covered in mud, we would see the obstacle that we needed to avoid. We couldn't do it without inspection - if we didn't look up from our texting and see the obstacle we wouldn't feel the need to swerve. And finally we wouldn't be swerving if it weren't for adaptation. Because that's what adaptation is, it's swerving, or slowing down, or even stopping, to avoid the things that you can see due to inspection. That's possible because of transparency. And that's all we're talking about when we're talking about empirical process.

At this point, I might mention that I've crashed two cars, OK three. I'm far from an expert on driving. But I am an expert on what this principle of empirical process means for software development teams. What does transparency mean for software development teams. Well, there are at least three meetings where there's opportunity for transparency.

Members of the team can - and should - in the daily stand-up let others know what they're doing, what they're going to do and any problems that are finding. This gives everybody in the team the opportunity for to know what's going on around them.

In retrospectives, the teams gets to look in the rear view mirror at the road behind them and talk about what obstacles could have been negotiated better, what things they've got the hang of, that the might be able to tackle more quickly now and that things they still have no way of tackling at all.

And in show and tells, demonstrations of working software show exactly how it's going. And of course, that can really be a problem.

I didn't witness this for myself. So this story may be entirely made up. 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's been a show and tell meeting. The publishing company was international and 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 apart from this - at some point one of the Scrum Masters said that he was going to share with everybody on the project the progress that had been made through the backlog. The backlog is the Agile way of saying the list of requirements for the project.

He had a presentation with some slides. And there was a bit messing about setting up the presentation - and then making sure that it was shared with the office in New York. Finally The Scrum Master was set up and could move to the slide that showed progress of the project through the requirements. 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 backlog.

This is brutal transparency. Any it probably came out of desperation. Having worked on that project I can guess that promises had been made to the bosses in New York that the project would be finished by some 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 Scrum Master was never allowed to speak at a show and tell again. The bosses in New York hired a "traditional project manager" to communication progress across the Atlantic. After that, guess what? The news was always good.

This is what one reader said about this story:

> I was there and I remember this meeting very well. It is all true. And it was followed by 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.

So this story that I'd heard really is true. 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. Mmmm. Actually, I don't need a funny, and I don't need the strange noises. If you have a project, and this project has yet to deliver it's first live release and that release has a deadline, I already know one thing about it.

> Your project is supposed to deliver N times as much work by the than it can do.

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!

Maybe I'm wrong. Maybe your project is fine. Maybe you're reading a book called "Delivering the Impossible" out of curiosity of what it would be like to work on one of those projects that seems impossible. 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 fact it resulted in the hiring of someone to specifically stop the bosses in New York being told the truth.

The first thing to understand is that if your project has over-promised, it's never

a bad thing for the team that's delivering that to know it. If you're in charge of that team, you should making sure that all of the chances to keep everything transparent (stand ups, retrospectives and show and tells) are happening.

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 2" 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. They literally hired someone to stop him telling them the truth ever again.

By starting to talk about the rather jolly metaphor of driving a car as a way of thinking about empirical process, we've ended up right at the absolute nitty-gritty of project management.

In order to manage a project, you need transparency. When you achieve that transparency, you get bad news. You have to do something with the bad news. What do you do with the bad news?