MARK STRINGER

DELIVERING THE IMPOSSIBLE

Seven metaphors for successful software development

Delivering the Impossible

Mark Stringer

Chapter 1 - Introduction

This book is called "Delivering the impossible." But of course. The only things that can be delivered are possible things. So I might ask you, and you might ask yourself, Why on earth did you pick up this book? It literally has a contradiction, an impossibility right on the front cover.

But if I did that, I would be being more difficult than you want the author of a book that promises to help with impossible things. Here are some reasons why I think you might pick up this book.

How about this? You are working on a project, maybe you're a member of the project team, maybe you're the manager, and you feel stuck. It seems like the project is impossible. What's being promised can't be delivered, nobody really knows what's being promised, or what's being promised can't be delivered on time.

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

You have a bunch of people working on a project for you, and you've got this uncomfortable feeling that the project really isn't going well and, for the sake of the company or for the sake of your career, you really need this project to succeed. You'd wonder if there any gems of insight in this book that you could pass on to the team or maybe anything that *you* could do differently.

Even though we think the project is impossible, you haven't just given up. And it's worth asking yourself why? If the project seems so impossible, why hasn't it be stopped, abandoned, killed? Put out of it's misery. Why would anybody bothering carrying on with it?

Well, part of the reason is your inherent optimism. You know that a lot of things that seem impossible, are actually possible, give the right skills and expertise. You know from other parts of your life, that if you can only learn to look at things in the right way and do the right things at the right time, things that seem impossible can magically be made to work out.

So what you would really like and what you might expect to find in a book with a title like "Delivering the Impossible" is some kind of guide, some kind of handbook for dealing with seemingly impossible projects.

In such a book, you might expect to find methods for spotting potential issues that might make a project "impossible" as early as possible. You might then expect the book to go on to talk about what to do in these seemingly impossible situations to make a project reasonable and deliverable and give it a chance of success.

Finally, you might not be surprised if such a book also talked frankly about the things you might look for to satisfy yourself that a project is genuinely impossible and gave advice on what to do in such circumstances, to keep ourselves and the

people around us financially, physically and psychologically safe, so that you can walk away from a lost cause and move on to succeed on other projects, ride into the sunset to fight another day.

If you're getting really demanding as a reader, you might also expect the book to talk about how deliver, not just on impossible projects, but also on impossible programmes, even in impossible organisations. Yes, if you're really 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 of these expectations. This is especially that book if your project, programme or organisation is in some part dependent on the development of software.

If you read this book, you will definitely learn, how to quickly identify seemingly impossible situations, how to manage and manipulate those situations so that's there's a chance, not just of delivery, but also of success. You'll also get lots of advice about how to look after yourself and those around you while you're doing it.

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 repeatedly signed up for improvisation classes. Why? Because my experience is that every time I go to an improvisation class, I learn something new, more specifically, I have new experience, and these often somehow 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 options. Person 2 could be really 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 really 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 any initial idea, no matter 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 Pomeranian poodle.

or alternatively:

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

Person 2: Humph. Fine.

Generally in improvisation these kinds of responses are not recommended? 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 negate the “offer” of the cabbage, utterly ruining any chance unfolding of a couple who learn to fly on the wings of their flatulence or another possible story of the dangers of living with Kimchi that becomes sentient.

The second answer “Hmph. Fine.” Is perhaps even more dangerous to a good story because it just goes nowhere, what’s really happening in such a situation is that the person who is speaking is scared. They feel out of control, so they’re doing that absolute minimum possible, even though the result of this is very boring to watch, not very nice to their fellow performers and kind of defeats the whole object of improv.

Here’s another flavour of “Humph, fine.”

Imagine that instead of just two performers on stage for an improvised scene, there are a group, let’s say seven, give or take a few. As is often the way with improvised performances, the actors get a suggestion for a setting for the scene from the audience and get “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 totally “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.

And all of these are viable selections (providing trying to hide in the life boats is ineffectual).

But what can often happen in such a situation is that instead of saying one of the things above, that might move the story forward, someone says something like this:

Person 5: Hey! Let's scrub 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 option. But of course, for the audience, it's a very bad idea. Once the Pirate ship has been mentioned, they want to see the pirate ship arrive. They want to see what a chase between this ship and a pirate ship looks like? They want to see if this crew really 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 be treated to the sight of everyone on the stage miming mopping the floor.

But for inexperienced improvisers, the temptation to agree to swab the decks rather than wrestle with the implications of the approaching pirate ship is extremely tempting.

Why? In his books "Impro" and "Impro for storytellers" Keith Johnstone describes this kind behaviour as "agreed activity." Putting people on a stage in front of an audience makes them scared. And scared people try, instinctively to make themselves safe, even if what they're doing individually is making the collective endeavour less likely to succeed. Part of what constitutes the skill of being a good improviser is knowing how to avoid this primitive instinct for safety and instead, move the story forward. Accept that there's a pirate ship on the horizon and deal with the consequences.

What's particularly fascinating about this kind of behaviour is that everybody, I mean all the improvisers on stage, seem to instinctively know that it's the right thing to do. There isn't a long discussion. It happens in a second. And that is in spite the fact that it's exactly the wrong thing to do. What this really shows is how good we are as individuals and in groups at shying away from things that we think might be dangerous, or difficult, or really just anything that will make us having to change our behaviour and our thinking.

One way that Johnstone suggests to help the improvisers avoid "agreed activity" is to have a director who is watching the improvisers and can intervene during live shows. In this kind of set up, it's the job of the job of the director to spot which actions and suggestions by the improvisers will move the story forward - and even in desperate situations - suggest them herself. For example if a director were watching a scene where there'd been a suggestion of scrubbing the decks, she might have allowed the crew 10 seconds of deck scrubbing before shouting

"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 concrete practice has usefulness in the context of project management. As with improvisers on stage, there's a temptation amongst the people who work on a project to find some agreed activity as a way of avoiding having to think, or be changed by the realities of the problem that they're solving.

Of course, if we pick up on what we talked about in the introduction. If we want to deliver the impossible, the first thing that we have to do is stop ignoring the bits that look like they might be impossible.

In iterative, Agile, approaches to project management there are lots of opportunities for the team and the team manager to spot the pirate ship. Typically, every day there's a "stand-up" meeting where the team talk about what they did the day before, what they're going to do today, and, crucially if there's anything blocking their progress.

Of course, it's in the nature of agreed activity that if there's one big thing that's blocking progress, that might be the very things that no-one mentions at all in the stand up meeting. So the signs, that there is a pirate ship that needs tackling might very well not be the sight of a Jolly Roger and the sound of "Arrrrr!" Rather it might well be reports of agreed activity. Very often, the same report with mild variations for days and days.

Another sign of agreed activity and pirate ship avoidance is when a task or "Story" as they're often called in Agile, gets planned and then just 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 just agreed, possibly tacitly, 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 weird 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, scolded, threatened or even disciplined for pointing it out. This in improv the equivalent of someone on an improv stage shouting "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 before he gets one of the weirdest recurring problems that

I've seen - a highly skilled, highly paid team of professionals who've completely given up trying to think for themselves.

But no matter how many times they've been beaten down. If you run retrospectives and encourage people to speak, listen to what they're telling you and do your best to raise their problems with the people who either need to know about them, or can fix them, sooner or later, they'll mention the pirate ship. And when they do, it might be difficult 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 until the team were at least allowed to talk to the person who was giving them the details of what they were 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 (the business analyst that the team wasn't allowed to talk to was a technical business analyst) and then erm, business business analysts - these were the people who actually talked to the business, to the people who actually wanted the software to be built. These two groups of business analysts also weren't allowed to talk to each other - they were only supposed to communicate via emailed documents. A substantial part of making our project possible was to gently insist that the business analyst who was working with us, should sit with our team and be allowed to directly talk to the customer!

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? The product owner who is supposed to act as a representative of all the people who want the project to happen and any other interested stakeholders 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 employees of the bank all over the world. So for the first phone conference I could here 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" chorused in someone with a Glaswegian accent.t "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 fundamentally wrong. This problem hadn't been detected through months and months of analysis. But, literally 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 daily stand-ups with your team, if you do retrospectives and show and tells, even if you do these things *badly* - you will still find out what the problems are that are facing your project. Not having stand-ups, not having retrospectives and show and tells is the equivalent of shouting "Scrub the decks, don't look at the pirate ship!" at your team.

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

Well, it means something very basic. If you work to discover the problems that your project is facing and then work to solve those problems, you may well be able to transform seemingly impossible projects into projects that are not only possible, but actually get delivered.

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 scrubbing the decks and in the process are making possible projects impossible.

Why? Because problems are scary. Problems are humbling. Problems cause conflict. 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 a project unearths can be threatening to the sponsors of a project. What if the problem that you unearth 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," or "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 scrub the decks.

Oh dear. Well. You probably picked this book up because you wanted to deliver (seemingly) impossible projects. I'm delivering on that promise. I'm not delivering on the promise that delivering these projects is a walk in the park or a day at the beach.

Even though dealing with the problems that you find and making them clear to your team and your sponsors can be a rough ride, demanding tenacity, tact, diplomacy and a willingness to be called not very nice names, there are two reasons why you still should do it.

Reason number one is that, despite the resistence you might encounter, solving these problems is still the best chance you've got of delivering this a seemingly impossible project. That's a really good, solid positive reason.

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

Chapter 3 - Trench Warfare

Two members of my team were supposed to be installing their software on a clients machine. They'd written an install script to run from a clean install. They'd tried the install on a practice machine. It had run without any problems. They kept trying to run the install script on the client's machine. It kept failing.

They ran some diagnostic tests. The machine's that they were installing the software on were supposed to be clean. But running the diagnostics, it turned

out that they already had other software installed. 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 install their software on the clients machine. The client was implying very strongly that the reason we'd failed was because our guys were incompetent. Just like the guys in the server room, when I brought up the subject of software already being installed 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 installation failure was solved. The company that were providing the servers were in some kind of contractual dispute with the organisation that wanted the project. While they were in dispute, they were working to the letter of their contract - and so providing servers that already had some software installed, even though they knew it would break our software. That's part of why they were being so weird. But the other part is that the contract for the provision of 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 almost definitely going to fail unless it is substantially reorganised and restructured. 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, materiel and human misery and doesn't get anywhere. A key cause of trench warfare is a refusal (or prevention) earlier in the project to deal with the problems that it faces and a key effect (aside from failure of the project) is that there is every more demand for the "agreed" activity which is an alternative to facing the problems to become ever 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. As are high turnover of staff, staff being signed off long-term sick, or with stress.

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

A trench warfare project is a bad situation. And experience of working on one can be one of the best motivations for getting over your reluctance to tackle

issues as soon as they arise - call out and then tack the pirate ship - on the next project that you work on. But every now and then you are going to find yourselves working on a project which is simultaneously working towards a ridiculous, undeliverable deadline, whilst at the same time labouring under several other problems, meaning that it's undeliverable. This is in many ways, the archetypal 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? Televise them. Make sure everybody on the project knows what those problems are.

Why? Because someone else may, if they really want to the project to succeed, be able to solve them. Another way to look at this is that some of the people who are browbeating you and your team, you complaining that you're team don't look worried or stressed enough (this really happens) or are complaining that your team lack commitment because they took the whole weekend off, some of those people may only be doing those things because they have no idea what else they can do to make the project more successful. At least by making clear what the problems are that are stopping you delivering, you're giving those people more chance of helping the project succeed.

The second thing that you can do - and this isn't a second option, it should really be done at the same time as you're problem televising, is to deliver something to some of the people who want it. This probably won't be everything, or 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 really 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 come up with a way of tracking and showing the actual progress of the project 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, and I'll talk through in detail one particular way to do it in a later chapter.

Demonstrating that a project can't possibility be delivered by a particular deadline can result in some sudden sensible discussions about reduction of scope, which then again can result in the possibility of delivering a small bit of working software to some of the people that want it. So, no matter, how, dug-in, going nowhere and doing nothing but enacting damage on the members of its team a project might be, by pushing on all of these three "fronts" and getting some kind of virtuous circle going, it might be that something good could come out of even the worst trench warfare project.

It might be that something could be done to save any particular trench warfare project. But equally, it's important to remember that some projects simply 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 point out that there are multiple ways to depart from a a lover. 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 unique handle on which projects are going badly.

Chapter 4 - Flowers and Fruit

This is a distinction that I picked up from reading about Taoism. Put bluntly, 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 terms, the equivalent of flowers are ideas.

People have ideas, those ideas get funding and that's how projects are born. Something that ideas and flowers have in common is that they appeal is immediate and unthinking. It doesn't take any thought to like a rose. Of course, some people might not like roses. But the people who do, don't have to work it out.

Something similar seems to happen with ideas, the kind of ideas that get funded as projects seem to have a common structure that gives them immediate appeal, so that, like flowers, those who are expose to them respond to them positively and unthinkingly.

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 difficult 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 get funding. Clearly it isn't, otherwise everybody would have managed to get a project funded.

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 an unpleasant 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 and this involve 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 radically 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" fruit for users is one were, if it doesn't taste right, they will spit it out, and if it isn't grown right, 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 religious 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 funded is there instant, unthinking appeal. Often what creates that unthinking appeal is that they 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 demonstrates the work that it's been doing in the last two weeks to anybody in the organisation (sometimes also external users) who's 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 problems might arise for the "Flowers" aspect of a project, the idea that has immediate appeal? The minute that a team starts to implement

the idea of a project they will encounter problems and if the team is using an Agile approach like Scrum, those problems will start to be reported, literally on day one.

So? What's the problem with problems? The problem with problems is that literally 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 more difficult, slower, more expensive, different and partial.

So, this is the paradox at the heart of project management and project development 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 difficulties 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 support and value from the idea because we are starting to find out what is wrong with the idea, and the same time there is no support and value from the reality, because there is no reality.

As someone who is trying to help this project get delivered, it is important to do two things. Firstly, it's important not to hesitate to move a project through this phase, and this is what we talked about in Chapter two, when we talked about agreed activity and in Chapter three where we talked about the awful situations a project can get into if the problems with agreed activity 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 difficult phase.

OK, here's where it gets even more tricky. There are two kinds of thing that you should be doing to get the project through this trick phase and on the road to success. And here's what makes this tricky. Both of the kinds of things that you should be dong are the kinds of things that the sponsors of a project are likely are either a pointless waste of time or are actually damaging to the project and 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.

Also, as I'm writing this, I'm resisting from saying things like "find out as much as you can about the people who are funding this project" or "find out everything that you possibly can about your users." Why? Because as a seasoned and grizzled project manager I'm sensitive to statements that involve "all".

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

At the same time, you need a strategy for getting working software that somehow addresses these needs and is attractive to the ecosystem of stakeholders as near to real users using real data as you possibly can.

So how do we do this? How do we acheive this transition from the "Flowers", where value and support for an idea comes from the idea itself to the "Fruit" where value and support comes from a real thing, 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 mentions 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 suspicion is that they had all had the unpleasant 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 essentially an offshoot of other kinds of engineering. And in other kinds of engineering, nothing is built before the productions of meticulous plans.

But what is this second principle in the Agile manifesto saying? It's essentially 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 a quirk or eccentricity that you've developed. 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 difficult 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 palm 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 really 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 palm 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 security mechanism are that are out there in no man's land, and that's to try to trigger them.

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 mention are 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 tripwires that your software might trigger is to try to cater for 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 difficult 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 really 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 suddenly 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. Sometimes it's really important to re-state very simple things. 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 basic than that. These are more basic 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 and the infrastructure that you need to deploy this software.

Well, really there's only one way to find out the answers to these questions and embarrassingly, the answer 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. Of course these are problems that can be solved, but only once they're uncovered.

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

But there's a third reason. Software is magical.

Concrete Practice

Connections