Allan Kelly

Xanpan

Team centric Agile software development

Combining Kanban and XP – inspiration for creating your own hybrid

Xanpan

Team Centric Agile Software Development

Allan Kelly

This book is for sale at http://leanpub.com/xanpan

This version was published on 2023-08-02

ISBN 978-0-9933250-2-1

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Twitter & other online media

Please Tweet!

Please help Allan Kelly spread the word about this book on Twitter.

Allan's Twitter handle is @allankellynet, and the hashtag for this book and all things Xanpan is #xanpan.

Find out what others are saying about this book on Twitter with this link:

> http://twitter.com/search/#xanpan

Websites

There are two websites to accompany this book. The first is the LeanPub page and the second the author's own:

> http://leanpub.com/xanpan

> http://www.xanpan.org

And the author's own website is:

http://www.allankelly.net

What people say about Xanpan....

"This book is an important contribution to the agile literature. It tackles the challenges that most teams face, head-on, by proposing alternatives that some might consider heretical." **Seb Rose**

"Xanpan is a method based on the best from Scrum and Kanban. It reflects Allan Kelly's very pragmatic approach to agile software development, based on extensive experience. This book will help you avoid waste and turning up the good." **Nicolas Umiastowski**

"Allan Kelly's Xanpan is now required reading for our team" **Dom Davis** Head of IT, Virgin Wines.

"A well written and insightful reference material, packs a punch throughout!!!" **Sunish Chabba**

"Finished @allankellynet 's book XanPan http://nud.gr/1fNSSxF if you're doing agile, lean, Kanban, xp a mix of any or just starting read it" **Kev McCabe** @BigMadKev on Twitter

"Unhappy with the strictness of Scrum he crafted his own method called Xanpan, a combination of Kanban and Extreme Programming (XP). XP focus is on engineering practices to write good code and achieve a sustainable level of quality, while Kanban (or Scrum) covers the organisational aspects. Using

the best of both makes Xanpan an interesting new player for organising software projects." **Johnny Graber**

About the author

Allan Kelly has held just about every job in IT. London-based, he works for Software Strategy, where he provides training and consultancy in Agile practices. He specialises in working with software product companies, aligning company strategy with products and processes.

He wrote his first program at the age of 12 on a Sinclair ZX81, in Basic. He quickly progressed beyond the ZX81 and spent the mid-80's programming the BBC Micro in BBC Basic, 6502 Assembler, Pascal and Forth. As well as appearing in several hobbyist magazines of the time, he was a regular on BBC Telesoftware, with programs such as Printer Dump Program (PDP and PDR), Eclipse, Snapshot, Echos, Fonts, FEMCOMS and, with David Halligan, Demon's Tomb, and EMACS (Envelop Manipulation and Control System, nothing to do with its more famous namesake!).

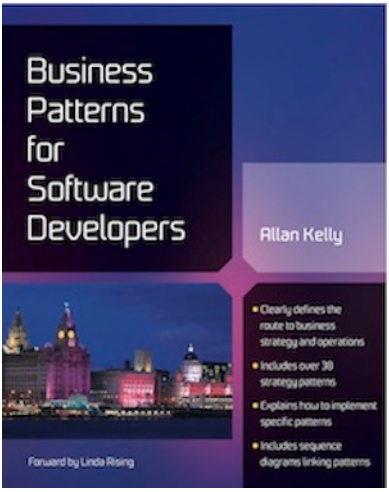
The low point of this early career came in 1986 when Cambridge Examinations docked a mark from his GCE 'O' level Computer Science project for not using the GOTO statement in his code. The high point came five years later when he held an internship at Distributed Information Processing in Guildford, working on the Sharp PC-3000.

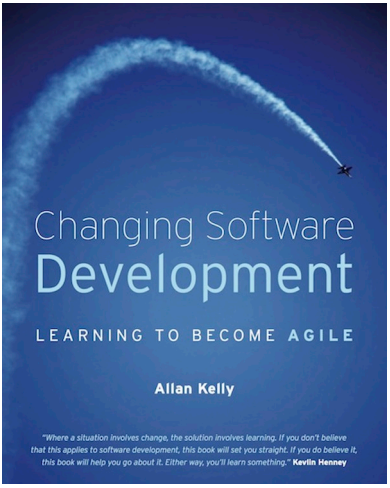
He believes his first Agile project was in 1997 - although it might have been 1994. His Agile journey began after the Railtrack Aplan ISO-Waterfall death march in 1996 and reading Jim McCarthy's Dynamics of Software Development (1995). Since 2000

he has helped numerous companies - particularly in Cornwall - adopt Agile and Lean ideas.

In addition to numerous journal articles and conference presentations, he is the author of *Business Patterns for Software Developers* (2012) and *Changing Software Development: Learning to be Agile* (2008), both published by John Wiley & Sons. He is also the originator of Retrospective Dialogue Sheets (www.dialoguesheets.com).

More about Allan at http://www.allankelly.net and on Twitter as @allankellynet[1].





[1] https://twitter.com/allankellynet

Prologue

Dear Customer: The Truth About IT Projects

Dear customer,

I think it's time we in the IT industry came clean about how we charge you, why our bills are sometimes a bit higher than you might expect, and why so many IT projects result in disappointment. The truth is that when we start an IT project, we don't know how much time and effort it will take to complete. Consequently, we don't know how much it will cost. This may not be a message you want to hear, particularly since you are absolutely certain you know what you want.

Herein lies another truth, which I'll try to put as politely as I can. You are, after all, a customer, and, really, I shouldn't offend you. You know the saying "The customer is always right"? The thing is, you don't know what you want. You may know in general terms, but the devil is in the detail - and the more detail you try to give us beforehand, the more likely your desires are to change. Each time you give us more detail, you are offering more hostages to fortune.

Software engineering expert Capers Jones believes the things you want ('requirements', as we like to call them) change 2% per month on average - thats close to 27% over a year once you

compound changes. Personally, I'm surprised that number is so low.

Just to complicate matters, the world is uncertain. Things change, and companies go out of business. Remember Enron? Remember Lehman Brothers? Customer tastes change. Remember Cabbage Patch Kids? Fashion changes, governments change, and competitors do their best to make life hard. So, really, even if you do know absolutely what you want when you first speak to us, it is unlikely that it will stay the same for very long.

I'm afraid to say that there are people in the IT industry who will take advantage of this situation. They will smile and agree with you when you tell them what you want, right up to the point when you sign. From then on, it's a different story; they know that changes are inevitable, and they plan to make a healthy profit from change requests and late additions at your expense.

While I'm being honest, it is true we sometimes gold-plate things. You might not need a data warehouse for your online retailer on day one. Yes, some of our engineers like to do more than what is needed, and yes, we have a vested interest in getting things added so that we can charge you more.

It is also true that you quite legitimately think of features and functionality you would like after we've begun. You naturally assume something is 'in' when we assume it is 'out'. And, in the spirit of openness, can you honestly say that you've never tried to put one over on us? (Let's not even talk about bugs right now: it just complicates everything.)

Frankly, given all this, it is touching that you have so much faith in technology to deliver. But when IT does deliver, does it deliver big. Look what it did for Bill Gates and Larry Page, or Amazon

and FedEx. Isn't it interesting that when the IT industry develops things for itself, we end up with multi-millionaires? When we develop for other people, they end up losing money.

How did we ever talk you into any of this? Well, we package this unsightly mess and try to sell it to you. To do this, we have to hide all this unpleasantness. We start with a ritual called 'estimation' - how much time we think the work will take. These 'estimates' are little better than guesses. Humans can't estimate time. We've known this since at least the late '70s, when Kahneman and Tversky described the 'planning fallacy' in 1979 and went on to win a Nobel Prize. Basically, humans consistently underestimate how long work will take and are overconfident in their estimates.

To make things worse, we have a bad habit we really should kick. Between estimating the work and doing the work, we usually change the team. The estimate may be made by the IT equivalent of Manchester United or the New York Yankees, but the team that actually does the work is more than likely a rag-tag bunch of coders, analysts and managers who've never met before.

Historical data - data about estimates, actuals, costs, etc - can help inform planning, but most companies don't have their own data. For those that do have data, most of it is worse than useless. In fact, Capers Jones suggests that inaccurate historical data is a major cause of project failure. For example, software engineers rarely get paid overtime, so tracking systems often miss these extra hours. Indeed, some companies prohibit employees from logging more than their official hours in their systems.

So we make this guess (sorry, 'estimate') and double it - or we might even triple it. If the new number looks too high, we

might reduce it. Once our engineers have finished massaging the number, we give it to the sales folk, who massage it some more. After all, we want you to say "yes" to the biggest sticker price we can get. That might sound awful, but remember: we could have guessed higher in the first place.

Please don't shoot me: I'm only the messenger.

We don't know which number is 'right', but to make it acceptable to you, we pretend it is certain and we take on the risk. We can only do this if the number is sufficiently padded (and, even then, we go wrong). If the risk pays off, we get a fat profit. If it doesn't, we don't get any profit and may take a loss. If it's really bad, you don't get anything and we end up in Court or bust.

The alternative is that you take on the risk - and the mess - and do it yourself. Unfortunately, another sad truth is that in-house IT is generally even worse than that provided by specialists. For a software company development is a core competency - such companies live or die by their ability to deliver software, and if they are bad, they cease to trade. Evolution weeds out the poor performers. Corporate IT on the other hand rarely destroys a business - although it may damage profits. Indeed, Capers Jones' research also suggests specialist providers are generally better than corporate IT departments.

Sales folk might be absent, but the whole estimation process is open to gaming from many other sources and for many other reasons. The bottom line: if you decide to take on the risk, you may actually increase risk.

I know this sounds like a no-win scenario. You could just sit on the fence and wait for Microsoft or Google to solve your problems with a packaged solution, but will your competitors

stand still while you do? Will you still be running a business when Google produces a free version?

Beware snake oil salesmen selling off-the-shelf applications. Once people start talking about 'customisation' or 'configuration', you head down a slippery slope. Configuring a large SAP installation is not a matter of selecting Tools, Options and then ticking a box. Configuring large packages is a major software development activity, no matter what you have been told. The people who undertake the configuration might be called 'consultants', but they are really specialist software developers, programmers by another name.

There really isn't a nice, simple solution to any of this. We can't solve this problem for you. We need you, but you have to work with us. As the customer, you have to be prepared to work with us, the supplier, again and again in order to reduce the risk. Addressing risks in a timely and cost-effective manner involves business-level decisions and trade-offs. If you aren't there to help, we can either make the decision for you (adding the risk that you disagree), or spend your time and money to address it.

You need to be prepared to accept and share the risk with us. If you aren't prepared to take on any risk, we will charge you a lot for all the risk we take on. Sharing the risk has the effect of reducing the risk, because once the risk is shared you, the customer, are motivated to reduce risk. One of the major risks on IT projects is a lack of customer involvement. You can help with that just by staying involved.

Ultimately all risk is your risk: you are the customer, you are paying for the project one way or another. If it fails to deliver value, it is your business that will suffer. When you share

risks, when you are involved closely, risks can be addressed immediately rather than being allowed to fester and grow.

Finally, you may have grand ambitious, but we need to work in small chunks. I know this may not sound very sexy, but software creation works best when small. Economies of scale don't exist. In fact, we have diseconomies of scale, so we need to work in tiny pieces again, again and again. If you are prepared to accept these suggestions, then let's press 'reset' on our relationship and talk some more.

Yours sincerely,

The IT Industry

Originally published in the now defunct Agile Journal, March 2012

1. Xanpan Principles

Boiling Xanpan down to a core results in a few principles that guide thinking, and lead to a number of specific practices. Most of these principles and practices will look familiar to anyone who has worked with Agile and studied Scrum, Kanban or XP. However, Xanpan's formulation is subtly different from all of these: it mixes these principles and practices differently.

Deeper still, underlying all of these are even deeper principles and laws. Some of these are specific to the software industry - I will elaborate these later - and some are broader still. The key, deepest, principles I refer to as 'Philosophy'. These make up the "I think therefore I am" bedrock.

1.1 Work in iterations

The iteration, or sprint, is a well-established Agile practice. Working in regular cycles provides a rhythm to work, and also imposes a deadline - actually, a recurring deadline. As I have discussed elsewhere, humans are actually quite good at working to deadlines. Xanpan seeks to harness this effect by having teams work in fixed length iterations.

I like to use the train metaphor. A traditional project is like a train leaving London King's Cross for Edinburgh Waverley station. Trains are infrequent, about every two hours if memory serves. And to get the cheapest tickets, you need to book in advance. As

a result we check the timetable in advance, we book our ticket, and if there is any doubt about who needs to go on the train, or what we need to take, we take it. The risk of leaving something behind is too high.

I arrive at the station in plenty of time. If while I'm waiting a colleague phones and says "Hey, we're having a beer in the Euston Flyer - want to join us for a quick half?", then I look at my watch and say "Sorry, my train leaves in 15 minutes". The risk to my schedule is too high, so I avoid changes. In the extreme, if one of my colleagues is late arriving for the train, I try to hold it back, I argue with the guard, I beg, I hold the door open in the hope my colleague makes the train.

When the train does leave I immediately call Edinburgh to say "I'm on my way"; when the train becomes late, I deny it or hope it will fix itself. Then I think "Well the taxi at the other end will only take 5 minutes, not the 20 I allowed". Eventually the whole train is late; with other passengers, I argue with the guard to throw some people and packages off the train to make it go faster.

Eventually the train calls at Edinburgh Haymarket, and everyone gets off saying "That's close enough".

In contrast, working in iterations is the equivalent of getting the London Underground. It isn't perfect, it has problems, but generally trains run, and they run regularly. I enter the Underground when I am ready; except for the first and last trains of the day, nobody checks the timetable.

If a train pulls in and it is over-crowded, I can decide: squeeze onto a packed train, or wait two minutes for the next. If I am lucky there is a sign that tells me how many minutes the next one will be. And if while I'm waiting someone calls me and says

"We are in the St Pancras Champagne Bar, fancy a drink?" I have an option. I can decide: "Get the next train and get home when I planned, or enjoy a drink and get home 30 or 40 minutes later." I have options. There will always be another train.

1.2 Team-centric: flow the work to the team

Xanpan is team-centric: the team is the production unit, need goes in, working - even valuable - software comes out. This is the machine, the goose that lays the Golden Egg.

Teams everywhere differ: some are large, some are small, some have testers, some don't, some have requirements people, some don't; some are collocated, some are distributed. And some work on one stream of work - call it a 'project' if you will - and some work on multiple streams. In fact, most teams of my experience - both as a team member and an observer - have to deal with multiple streams of work.

Multiple streams might be two projects at once, project A and project B. Or it might be working on a new product while maintaining an old one. Or it might be the team working on project A but individuals being pulled some days to work on something old. These streams might come from different sources, different business units. Or they might come from the same source.

Consequently some work can be known about and planned in advance, and some work just appears, unplanned. There seems to be some unspoken law which mandates that the later work appears, the more urgent it is.

Xanpan aims for stable teams that accept both planned and unplanned work on multiple streams.

1.3 Work to improve Flow

Hand in hand with team-centricity is *flow*. Work arrives at the team from somewhere, somehow. This is inbound flow. Ideally work is flowing into the developer (see Coplien and Harrison 2004 for the Work Flows Inward pattern). Then it has to flow out - perhaps via testers, operations, deployment, and finally to the customer. The actual time needed to do the work may form a very small part of the end-to-end elapsed time.

As with Kanban, Xanpan aims to reduce the overall end-to-end time by improving the flow, making it smoother, more regular, more predictable. Improving flow can also mean levelling flow - reducing the peaks and troughs in work patterns - and constraining work to deliver an overall improvement.

The key to improving flow in Xanpan is allowing work to span more than one iteration. This is heresy in some circles and some texts.

1.4 Quality is free (provided you invest in it)

Philip Crosby wasn't writing about the software industry, but he could have been (Crosby 1980).

There are those in the software industry who believe there is a dial on the wall marked 'Quality'. If you turn the dial down,

quality falls and work happens faster. Turn it the other way, dial up quality, and work slows down. They are wrong. This might be true in some industries, but not software.

In software development the dial is wired in reverse: if you want to go fast you dial quality up; when you dial it down, you go slow. Capers Jones has devoted most of his career to studying software development quality and metrics, and he is unambiguous:

> 'Projects with low defect potentials and high defect remove efficiency also have the shortest schedules, lowest costs, and best customer satisfaction levels.' (Jones 2008)

To be clear, when I use the word 'quality' here I am not talking about features, polish, leather upholstery or just about any other external attribute. Indeed I'm not specifically talking about software design or architecture characteristics - although they are implied.

When I say 'quality' I am specifically referring to defects, bugs - or to put it another way: *low quality implies rework* fixes. High quality work does not need rework.

This is not an excuse to gold-plate and over-engineer systems; work may well need to change as we learn things, but we should not be delivering work which quickly requires fixing. All software needs rework. That is the nature of successful software. The question is, how easy is the rework? Low quality makes rework harder, and therefore slower. High quality makes rework easier, and therefore faster.

In the same book Jones says:

> 'IBM was also the first company to discover that software quality and software productivity were directly coupled and that the project with the lowest defect counts by customers were those with the shortest schedules and the highest development productivity rates. This phenomenon, discovered by IBM in the early 1970s and put in the public domain in May 1975, is still not understood by many other software-producing enterprises that tend to think that quality and productivity are separate issues.' (Jones 2008)

Of course rework occurs not just for defects: requirements changes and unforeseen changes can result in reworking existing code. And ones man's bug is often another's change request. I am perfectly happy to have these debates, but only when a team is approaching zero defects. Until then the team needs to work to improve quality and reduce rework.

Quality is important, because when quality is low there needs to be rework, and when there is significant rework other parts of the Agile jigsaw just don't fit together. High quality is essential to Agile working:

- Rework destroys flow: work must move backwards, work creates work because work creates rework.
- The need for rework means stories and tasks can't truly be considered 'done'.
- When stories aren't truly 'done' iterations are an illusion, because hidden work is flowing between the iterations.
- Metrics are destroyed because work that looks 'done' isn't.

- Developers, testers, managers and others spend inordinate amounts of time prioritising, reporting, managing and even doing rework rather than delivering value.
- Organisations spend inordinate amounts of money on testing resources and cycles: the need for test demonstrates that quality has been sacrificed.

As far as I am concerned anyone who thinks reducing quality is a good way to speed up software development shouldn't be working in the industry.

The need for high quality is why Xanpan embraces all the XP technical practices explicitly. These practices, specifically test-driven development, raise quality, thereby reducing defects and rework. (The software craftsmanship movement continues to advance this cause of quality, and I encourage all to follow the latest thinking here.)

Quality

Any mention of quality, especially high quality, demands a proper discussion of what is quality. Quality will be discussed in more depth in an appendix.

For now when I speak of quality I am essentially speaking of two attributes - *qualities* if you prefer - of a software product:

- Defects: quality is inversely proportional to the number of defects seen in a system, i.e. lots of defects imply low quality; few (or no) observed

defects do not by themselves imply high quality, but such a status is a precondition for high quality.

- Maintainability - changeability and extendibility: successful software lives and needs to change over time. If software is not changeable, then it cannot change as it needs to during its lifetime, and it will be hard to remove any defects that are found.

There are other attributes that customers, users and the development team might like to include when talking about quality for a given product. Indeed I would encourage all teams to think about what constitutes quality for their product. However I believe that these two attributes of quality apply universally for any software product.

That said, this definition of quality is not a reason to over-engineer, over-design and gold-plate software. The Xanpan approach is to achieve these attributes, not through 'big up-front design', but through repeated design sessions, 'little and often', constant attention to technical excellence and delivery of working products in the short term.

Such an approach leads to a form of rework, but this is rework because things have changed, knowledge has improved, requirements have changed, new demands have emerged. This is not rework because fault has been found with existing work.

1.5 Visualise

There is surely no team sport in which every player on the field is not accurately aware of the score at any and every moment of play. Yet in software development it is not uncommon to find team members who do not know the next deadline, or what their colleagues are doing. Nor is it uncommon to find managers who have no insight into the work coming their way, or indeed what happens to work when it leaves them.

There are multiple ways in which teams can visualise their work: whiteboards, flip-charts, burn-down charts, cumulative flow diagrams, stickies, posters and many many more.

Software development is an extremely abstract activity. We do things that cause sub-atomic particles to move about in a machine. We work in obscure languages that might look mathematical to some, but look decidedly like natural language to pure mathematicians.

Visualising work helps people learn, it helps people improve flow; visualisation makes it difficult for problems to hide; it creates shared understanding and shared learning. If we cannot see, we cannot learn. If we cannot learn, we are doomed to failure.

1.6 References

Coplien, J. O., and N. B. Harrison. 2004. *Organizational Patterns of Agile Software Development.* Upper Saddle River, NJ: Pearson Prentice Hall.

Crosby, P. B. 1980. *Quality is freeâ€¯: the art of making quality certain.* New American Library.

Jones, C. 2008. *Applied Software Measurement.* McGraw Hill.

2. Board 1

In Xanpan all work is represented on a physical board, usually a magnetic white board, sometimes called a 'Kanban board'. You might decide to use an electronic equivalent, but I always strongly advise teams to start with a physical one until they are familiar working in this way. Working physically is magical - the learning experience is so much stronger and faster. Even if an electronic system is adopted, the team should keep the electronic board publicly visible at all times.

There are many different board designs. Indeed, each team is encouraged to design their own board. Think of this like the light sabres used by Jedi Knights in the 'Star Wars' films: every Jedi must build their own light sabre, and every Xanpan team must design their own board.

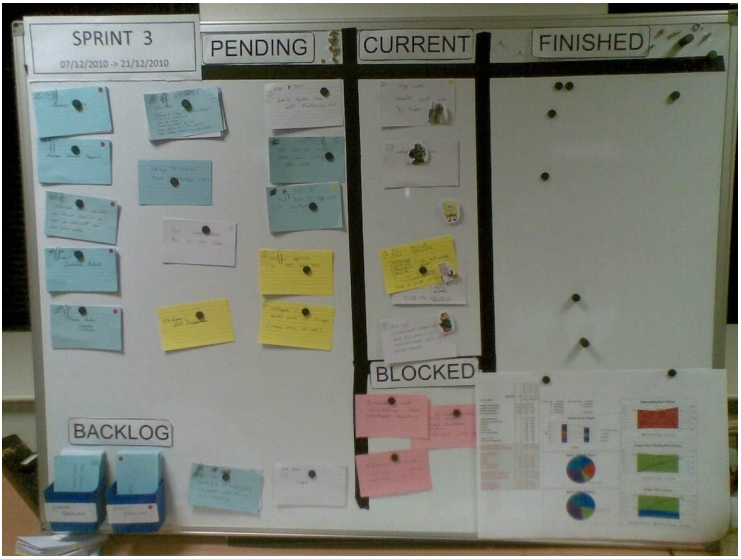


Figure 1 - Basic board design

Figure 1 shows perhaps the most basic board design, and will probably look familiar to many readers. This board is at a laboratory equipment maker in Falmouth (UK). The small - five person - development team creates desktop and embedded software used in products for IVF treatment. In addition they create some embedded software used in the company's own product line facilities.

On the left of the board is the work to be done (the pending column), in the centre is the work in progress (current on this board, commonly known as 'work in progress' or WIP in Kanban), while on the right is the work completed. This photo was taken the day after the team reset the board in the planning meeting, so it is quite clear. Over the course of a sprint, cards

move from left to right. The objective is to get as many cards as possible from the left-hand side to the right.

At the bottom left is the 'product backlog'. As with Scrum, the product backlog contains all the work that might be done. In this case the product backlog is physical, and is attached to the board. Personally I like working with a physical backlog, but even I accept it has its limitations. Rows in an Excel spreadsheet can be used, while a GoogleDocs spreadsheet is even better for sharing. Plenty of electronic Agile and 'requirements management' tools are available, but there is value in simplicity.

From the product backlog a subset is selected to do in each sprint. In keeping with Scrum terminology, this is the 'sprint backlog', but may equally be called the 'iteration backlog'. This team has two product backlogs for two 'projects': one embedded, one desktop. In the right-hand corner are graphs showing status. There are actually three burn-down charts here, so the team has three projects in flight. (The third backlog is not visible here for other reasons.)

Each sprint/iteration is two weeks long. They run back-to-back: the end of one sprint is the start of the next. Each sprint begins with an iteration planning meeting and ends with the next such meeting.

In each planning meeting the board is cleared of work done: cards counted, graphs updated, remaining work reviewed and removed if no longer needed, or left if still important. Then the team refill the left-hand side of the board to slightly more than its expected capacity.

This board also shows a blocked column. Whenever the team want to work on a card but cannot - they are blocked - the card

is put in the blocked column. This is a signal that escalation may be needed. Or, as in this case, the team is waiting on the customer for additional information before work can continue.

The colour coding of cards is significant. While teams are free to follow their own colour convention, most follow the one laid out in 'blue-white-red':

- Blue cards are development stories - often in user story format, but not always. Blue cards mean something to the business; each card should represent some item of functionality that is valuable to the business/customers.
- White cards are tasks - developer tasks are the most common, but there may be test tasks, analysis tasks or any other type.
- Red cards are bugs - if a problem is found in work done on a white or blue card before the end of the sprint, the card simply moves back to pending or current. If a bug escapes the sprint and a formal bug report is raised, it re-enters the board as a red card.
- Yellow cards are unplanned work. Xanpan allows for both planned and unplanned work. More about these cards in a moment. Bugs requiring an urgent fix may well be unplanned work and should be on the board. Arguably they could be yellow, but red is more commonly used.

Xanpan teams generally follow the Scrum/XP iteration cycle: work is planned in a 'planning meeting' and the team accepts a reasonable amount of work. Unlike Scrum, the team is not asked to commit to performing all the work, neither is the cycle locked; work may continue to enter the sprint.

(The words ‘sprint’ and ‘iteration’ are taken to mean exactly the same thing, namely a fixed period of development activity. Scrum introduced the word ‘sprint’ for this, and Extreme Programming, ‘iteration’. In the Xanpan context there is no difference.)

Ideally the team would be able to plan all work up front: no work would be added or removed. However in many environments this is not possible. Bug reports - reds - are one way work may enter. In the case of this team yellows are largely IT support tasks the group needs to look at.

In this picture three IT support yellows are in pending. They have occurred since the start of the sprint, or perhaps they occurred in the last sprint and have been carried over into this one. (Also, unlike Scrum, work can be carried from sprint to sprint: more of this later.) One yellow is in the pending column: someone is actively working on this issue.

While every effort is made to limit work in progress - a default of one task per person - such limits are sometimes broken, and team members need to suspend one task to deal with something more urgent.

Close examination of this picture shows that the yellow in the pending column is on top of a white. One of the developers was working on the white, but has suspended work to deal with an urgent yellow. When this is complete the yellow will be moved to the ‘finished’ column and work will restart on the white.

Unlike whites, the yellows have no effort estimates. Unlike Kanban, tasks - and perhaps stories - are estimated, most probably using the standard planning poker technique. Yellows are retrospectively estimated by the person who completes the tasks

when they move to the finished column.

Red, bugs, may or may not be estimated. They may be estimated in advance - as part of the planning meeting - or retrospectively by the person who undertakes the work. Whether and how they are estimated depends on the team's approach to bugs. The aim is to have no reds - no bugs.

On this board the team has chosen to use Avatars to indicate who is working on which card. Many teams don't bother indicating who is working on what - they remember or don't need to know. Other teams use colour-coded magnets (usually the colour of the magnet is meaningless); one team used coloured star stickers, and another coloured shapes.

Regardless of how or whether the team indicates who is working on what, it is best to avoid allocating work until the last possible moment. This moment is when someone becomes available to work on a task. Ideally the tasks to be done are prioritised. The next time someone finishes a task and becomes available, they simply take the highest-priority task and work on it.

This is the ideal, and it is easy to imagine why this doesn't always happen: someone may lack the skills to work on the next highest task, someone may be working on a related task and it would conflict for two to work on the same thing, and many more reasons.

Allocating tasks as late as possible ensures that priorities are worked on, and reduces bottlenecks around individuals. For this to work, team members need to have similar skill sets.

Most teams - and certainly the team illustrated here - frequently fail to work in strict priority order: however, it should be an aspiration for every team. The first step in this aspiration is

to avoid deciding in the planning meeting who will work on each story or task. Instead, wait until at least the daily stand-up meeting, when it becomes clear which stories and tasks will be worked on during the day.

Finally, on this board no attempt is made to indicate how much effort is remaining on a card. The cards have their initial estimates, which are not reduced. It is possible to get an idea of how done a blue is by looking at how many of the associated whites are done, but even this is not very useful. Cards are either done or not done: in programming it is usually impossible to accurately tell how much work is remaining, you don't know when you are about to hit a problem. Therefore we only consider 'done' and 'not-done'.

Similarly there is no attempt to capture 'actual effort' or measure the difference between 'actuals' and estimates. For reasons discussed elsewhere, actuals are only retrospective estimates. When a yellow or red is undertaken without an initial estimate, then a retrospective estimate is put on the card; this is not perfect, but is good enough.

Figure 2 - Basic board five months later

Figure 2 shows the same board five months later. There are fewer yellows in play because the team has hired an IT support engineer to handle most of these tasks. A few support issues or other unplanned work still persist. On the whole support tasks are managed on another board dedicated to the work of the support engineer.

The team has expanded in this picture by joining with a related team that has more engineers and projects (five backlogs now). The merging of the two teams was not entirely successful and was later reversed. The team has added another colour card to the convention: green cards represent process-improvement tasks, perhaps originating in a retrospective, or perhaps from general conversations.

The board has also acquired a new ‘review’ column. When a card is complete it is first moved here before progressing to the finished column. This serves two purposes: first, to demonstrate what items were completed yesterday when the stand-up meeting is held. Second, it allows the developer to ask for someone else to review the work before it is moved to the finished column, although this is not always necessary.

Key points:

- Cards are colour coded: blue, white, red, yellow and green.
- Iterations/sprints are two weeks in length, and start and end with a planning meeting.
- Xanpan iterations contain both planned and unplanned work.
- The board represent the state of the team and their work, not the state of a particular project.
- The product backlog contains all the work that has been requested for a particular product.
- The sprint backlog contains a subset of the product backlog that is currently in play.
- Xanpan is team-centric, so the team may be working on more than one product or project at a time.
- Estimates are made for work planned in the planning meeting; unplanned work is estimated in retrospect.

3. Iterations

As in XP and Scrum, work is conducted within the framework of time-boxed iterations - also called 'sprints'. These are normally two weeks in length, sometimes one week, sometimes longer, although iterations of three weeks should be avoided. Iterations of four weeks or more, while once common, are often seen as as too long now.

Changes to iteration length should be rare. Teams may occasionally decide to move, say, from two-week iterations to one-week iterations, or to four-week iterations. Once changed, the team will stay in this mode for several iterations before considering another change.

Ad hoc changes to iteration length should be avoided, although at times such as Christmas it might make sense to suspend iterations, or have just one iteration of a longer length then usual.

What is not allowed is extending an iteration by a few days because the work planned for the iteration has not been finished. Not only does this disrupt the following iteration, it also destroys the value of doing work in iterations in the first place. It destroys the deadline effect, it destroys benchmarking, and it negates the 'get good at doing things in short bursts' ethos.

Having to work to a deadline not only focuses individuals' attention, it also limits the work in progress, forces teams to address problems that might otherwise be hidden, and stops

procrastination. People are actually quite good at working to deadlines.

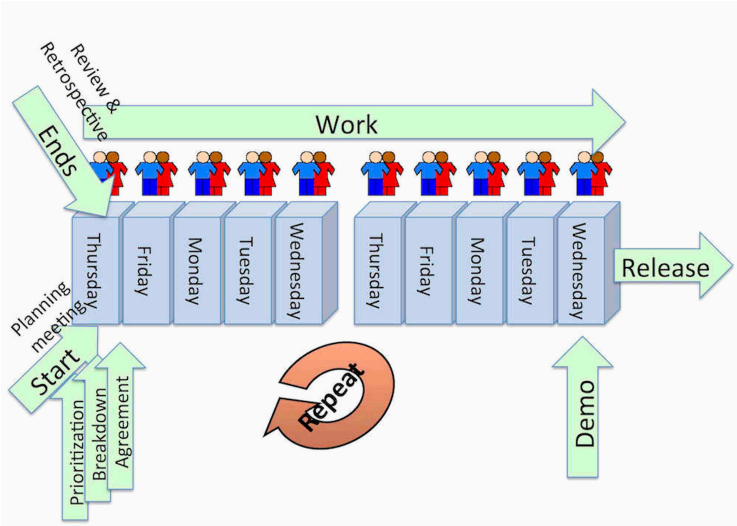


Figure 3 - Typical 2 week iteration (sprint)

The next iteration starts immediately after the previous one ends. Often the end of iteration meeting for one iteration simply flows into the planning meeting for the next iteration. The planning meeting (described below) marks the start of the next iteration. During the meeting the board is reset for the next iteration of work. Once the planning meeting is complete, work begins.

The first big difference with Xanpan iterations compared with Scrum or some other methods is that work can, and does, flow between iterations. Taking work into an iteration does not mean that the team is in any way committed to doing the work.

During the end of iteration review, work still in the 'to do' section

of the board is examined and may be returned to the backlog, or removed (de-scoped) completely if the team and Product Owner agree. In most cases though work remains in 'to do' and is simply left in play and carried to the next iteration, forming the starting point of planning.

The second major difference from Scrum is that Xanpan allows both planned and unplanned work. If teams can plan all the work for the next two weeks and stick to the planned work, great; if not, then the team accepts the work and does it. By tracking the planned versus unplanned work, this can be factored into forecasts.

Xanpan differs from Scrum on this point for several reasons:

- Allowing work to flow from iteration to iteration creates Kanban-style flow mechanics.
- Forcing stories to be small enough to fit within an iteration regularly results in stories that are too small to have any business value - especially for teams new to iterations. It is more important that a story delivers business value than that it fits in an iteration.
- Over time teams will get better at slicing stories small enough to fit into an iteration and big enough to have value. Over time the team will get better at delivering stories in one iteration. However, this is hard when a team is moving to iterative working.
- Teams in some domains, such as embedded and telecoms, or using some technologies, such as C++, and those working with unfamiliar, often legacy systems, find it harder to reconcile small stories with business value.

- Rather than use Scrum style commitment, teams are encouraged to try and do more work than they expect to do, and it is accepted that not all work taken into the iteration will get done. There is little to lose by trying for more. Statements about what will be delivered at the end of the iteration are statements of probability, e.g. "The highest priority story is almost certain to be delivered, the next is probable, the next maybe and the last unlikely."

In an ideal world a team would size every story to fit within an iteration and deliver value. Teams are encouraged to aim for this. But even here there is a problem:

> Assume there are two stories: A and B; each will take 7 of the 10 working days in the sprint. Both are acceptable: the team starts on A and completes it as expected. On day 8 the team is ready for a new story. Do they pick up B? Some Scrum teams would not, as they may never have scheduled the work in the first place. At some point, in planning or when A is done, they will look for a piece of work that is 2 days long.

This is the Scrum equivalent of dry-stone walling. A suitable story may not exist: even if it does, who is to say it is a priority? Why wouldn't the team start on the next highest priority story rather than go hunting through the low priorities for one that happens to be the right size?

3.1 Releasable

The key point about the end of the iteration is that the team should have a releasable product. This does not mean that the product is released: hopefully it is, or it moves straight to release after the end of the iteration. Essentially whether a product is released at the end of an iteration or not is a business - most likely a marketing - decision. At the end of every iteration the team gives the business an opportunity to release the product as is.

When work flows from iteration to iteration regularly there should always be some stories that are completed in an iteration even if some are carried over. Occasionally there may be an iteration where nothing is complete; this isn't good, but it might happen. If it happens regularly then there is probably a problem that needs to be addressed.

Increasingly teams are not only practising continuous integration, but continuous deployment. As work is ready it is pushed to release. When teams get good at doing two-week releases, this is a logical next step. This mandates that each story - each blue - is in its own right releasable.

3.2 Iteration sequence

Although an iteration follows the same basic sequence as a Scrum or Extreme Programming iteration for completeness, it is best to talk through the iteration.

An iteration starts with a planning meeting; these are described in detail in the next chapter. During this meeting the Product

Owner will present the work they would like done in the next iteration. This work should be presented in absolute priority order - 1, 2, 3, … - no Moscow rules, no equal priority.

The development team will break this work down into tasks. The team and Product Owner then need to reach agreement on what will be done. Xanpan uses a variation of the XP velocity system to gauge how much work the team should achieve. Teams are advised to accept slightly more work than they expect to do, but to expectation-manage the message. Nothing is guaranteed, so it is important that those who want to know what the iteration will produce understand that some things are more likely than others, and some things won't be delivered.

Once the iteration planning meeting is complete, work begins. Each day - preferably first thing in the morning - the team hold a stand-up meeting of no more than 15 minutes to synchronise and share their progress and issues. Ideally no other meetings are needed.

At the end of the iteration the team should at the very least undertake a demonstration to stakeholders. It might well be that the Product Owner has seen the work, as it is completed and has no need for a demonstration: they may even be the one running the demonstration.

The best teams will go straight to release - after the demo or perhaps skipping the demo altogether - as the iteration closes. It is then time for the review and retrospective.

The review is purely a bookkeeping exercise: count the cards and points, update any tracking systems, file any reports and so on. Retrospectives are intended to help the team improve their working in the next and subsequent iterations.

Retrospectives may be as informal as a once-around-the-table opportunity for everyone to say how they thought the last iteration went and what they suggest for the next one. Alternatively, they may be formal, with set exercises such as dialogue sheets (http://www.dialoguesheets.com) or from a book (e.g. Derby and Larsen 2006; Kerth 2001).

With the retrospective complete, it is time for the next iteration planning meeting. There might be a lunch or coffee break between the two meetings, but there is no need for a lot more.

3.3 Mid-week to mid-week

Figure 3 deliberately shows an iteration running mid-week to mid-week. My experience, and the experience of others assisting teams, is that running iterations mid-week to mid-week is more effective than running them Monday to Friday. Why this should be so is difficult to pin down, but there are several reasons that I believe contribute:

- The 'Friday afternoon effect'. If an iteration resets on a Monday, then anyone completing a piece of work on Friday will not be particularly motivated to start another piece of work. In many environments it is common to finish early on Friday, or take a long lunch. If the iteration has several more days to run, there is less incentive to 'slack off' on a Friday afternoon.
- Monday refresh. I believe there is research showing that people are more effective at the start of a new week. I hasten to add I have not read this research myself, which is a little lax of me. Anecdotally many people, if asked, say

their most productive day is Tuesday. Assuming there is truth in these tales, it is a good reason to avoid meetings on Monday and Tuesday.

- Public holidays. In the UK and many other countries many public holidays occur on Mondays. This means that there is more chance of an iteration meeting being disrupted if it is scheduled on a Monday.
- Personal holidays. Again in the UK and many other countries individuals are more likely to take a Friday or Monday off as a long weekend. Consequently planning meetings on these days are more likely to be missing a team member.
- Meeting rooms. In a surprising number of organisations booking a meeting room can be difficult, either because there is a shortage of rooms or the booking process is complex. Therefore it is worth striving to book rooms further in advance on a regular basis, and minimise changes. Indeed, it can be impossible to book a meeting room in some companies for the day after a public holiday, because all the meeting conveners are trying to move their meetings back one day.
- Regular schedule. Keeping to a very regular schedule has many benefits, one of which is that meeting rooms can be booked weeks, months, even years in advance. If you have an electronic booking system, set the meeting to repeat with no end date.
- Time zones. Although it is best to collocate teams, many find this is not an option. A team in Holland has programmers in Houston; a team in London has analysts in New York, a team in Pennsylvania has members in Bangalore and so on. Thus holding any whole team meeting means someone may need to work late or arrive early. Asking

> team members to work late on a Friday, or arrive early on a Monday, encroaches on family time and space.

There are probably more reasons that could be added to this list. None is in itself conclusive, and you might choose to argue with some of the points. As I said, experience - not theory - shows mid-week to mid-week is best. Perhaps the best advice is to experiment for yourself.

3.4 Iteration length

The default length of an iteration is two weeks. You need a pretty good reason to do anything different. If in doubt, do one-week iterations at first. You may later extend them to two weeks or four weeks.

I find monthly iterations to be too long. Remember that software has diseconomies of scale: pushing up to a month allows too much to change, and means that too much needs to be planned and scheduled in a planning meeting.

There are natural human rhythms and rituals based on weekly, fortnightly and monthly iterations. The working week is a week long, many people take two weeks for their summer holidays, the lunar month is almost 28 days. I have never heard of a natural cycle that takes 21 days. In English and many other languages there are words for these time-frames: week, fortnight and month. What we do not have is a word for a three-week period - and I have yet to find a language that does! It is hard to think of any human or natural cycle that takes 21 days to complete. Yet it is not uncommon to come across teams that

run three-week iterations. Usually these teams exist within a corporate IT environment, and it is not uncommon to find mini-waterfall processes within the three-week iteration.

Typically when asked “Why did you choose three weeks?”, they reply “Because we don’t think we could do anything useful in two weeks”. This misses an important point of short iterations: they exist to make you better.

Working in short iterations has many benefits: predictability and routine, reference benchmarking and more. However the iterations themselves are not the biggest benefit. The biggest benefit comes from the changes that need to be made, the problems that need to be resolved, the processes that need to be changed, and so on, in order to work effectively in short iterations.

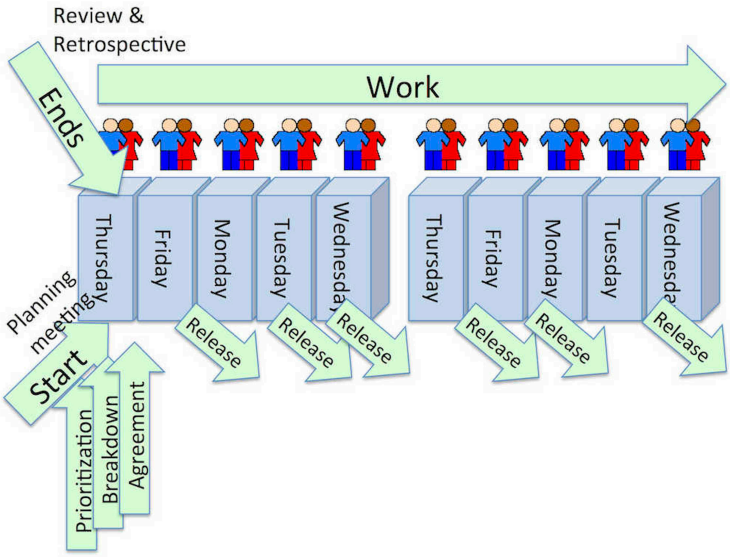
Iterations are not an end in their own right - rather they are a tool for making you better. Forcing yourself - and others, processes, tools and so on - to produce something useful, valuable, in two weeks, forces change for the better.

A team that is just starting with Agile is well advised to start with one-week iterations in order to practise the starting and stopping activities, see problems and resolve them.

3.5 Release schedules

Many teams will synchronise software releases with the iteration, so a new version of the software appears every two weeks. Better still, individual blue cards would be releasable, so the team could release multiple times during an iteration. Stories of Kanban, XP or Scrum teams releasing many times during a

week are increasingly common: this is the start of 'continuous deployment'.



Iteration with multiple releases

Before trying for releases during the iteration, a team need to be capable of releasing at the end of the iteration (or even releasing at all). For a team which currently finds it difficult to release the first objective is to be able to release every two weeks. Once this can be done fairly painlessly they can then consider more frequent, even smaller, releases.

For every team releasing during an iteration there are probably more teams releasing at the end. And, as far as I can see, there are more still who release after several iterations. For example, six two-week iterations followed by a quarterly release used to

be fairly common. Releasing every second iteration, so once a month, is also common.

I myself ran a team that made one monthly release after four one-week iterations. Occasionally there was a fifth iteration to stay with the publicised release schedule of 'first Tuesday of the month'. The first planning meeting of the new cycle was longer than the subsequent ones. While features did change, it was broadly possible to see what was coming for the next month and work on the details iteration by iteration.

3.6 The CEO test

Imagine you have just finished an iteration, perhaps yesterday or the day before yesterday. The CEO walks in and says "Bad news, we are out of money, we can't make payroll". While you are taking in this shocking statement he adds "Of course, if you have something I can release today, we can get some revenue and we have another month".

If your answer is "Sure, here's one we just finished", then everything is OK.

If on the other hand your answer is "Well we just finished something, we are code complete on some stories, but we need another month or two of test-fix-test-fix", then you are out of work and out of pocket.

Never be tempted to do more work which looks done but isn't. Get good at finishing things completely, to release standard. Do fewer features and more quality.

Do this and you not only give your CEO options - you change the game.

3.7 References

Derby, E., and D. Larsen. 2006. *Agile Retrospectives*. Pragmatic Programmers.

Kerth, N. L. 2001. *Project Retrospectives*. New York: Dorset House.

4. Planning Meetings

One of the things I enjoy about my work as a consultant on Agile is visiting teams and observing iteration planning meetings. If you've never observed another team's planning meeting, or if you've never seen one, you might expect them to be fairly similar. Indeed they are but, as is so often the case, the devil is in the detail. I am routinely surprised by the ability of teams to interpret, find and invent different ways to doing things in the meeting.

Take for example deciding how much work a team can undertake. Some teams just have the product owner propose stories and accept stories until they feel they have 'enough'. Some teams are strict in only accepting stories they are sure they can get done - the Scrum idea of 'commitment'; other teams will take on more work than they expect to do.

Some teams will use velocity to judge how much they can do. They determine how many points they can do at the start of the planning meeting and then accept stories up to that limit, give or take a bit. Some teams set the upper limit by simply looking at how many points they did last time and rolling that forward, called 'yesterday's weather' in XP. Some teams play planning poker to decide how many points they can do. Some teams think about who's on holiday, who's not, who was ill, what got in the way and a million other things.

Seeing these variations is often educational for me, and vice versa, I sometimes suggest changes to a team's current practice.

However it does mean that when describing a planning meeting to someone, there are a lot of details that could be different without necessarily being wrong.

Personally I advise teams not to spend a lot of time deciding how many points they can do. I don't see much point in asking "Who's in this week" or "Who is out?". You could add more and more detail without adding any more accuracy. All the team needs is a rough guide to judge how much work it is worth planning in detail.

To keep it simple, I would say just take the average velocity from the last four or five sprints, and ignore holidays, illness and other factors that might have, or will, disrupt work. Recalculate this average at the start of each iteration, i.e. use a rolling average.

Then - and this is both important and slightly controversial - schedule slightly more work than the team expects to do. That is, if the team expects to do 10 points, then schedule 12 or 13. If they expect to do 20, schedule 24 or 25.

The team needs to accept enough work that it doesn't run out of things to do, but should not accept more than is reasonable to expect. Preparing work that will not get done is wasteful.

What follows is my take on a planning meeting and how it runs. This description matches the A1 Planning Sheet <www.dialoguesheets.com> I have devised to help new teams run planning meetings. This is the core of Xanpan.

After this description I detail some of the activities which might occur outside the planning meeting but which contribute to a smooth running planning meeting.

I will also detail some variations I've seen.

4.1 The Players

- The *Product Owner*: this role is usually played by a product manager, or a business analyst acting as a proxy for the real customer. On occasions the real customer might play the role. Some companies employ subject matter experts who can play the role of Product Owner at times. Sometimes someone else needs to play the role. Whoever plays Product Owner needs to do their homework (see below) and be ready to propose stories, answer questions, prioritise and make decisions as the meeting progresses.

  While it is common practice for there to be only one Product Owner, there may be more than one. If more than one Product Owner (for the same product) attends the planning meeting, they should agree priorities and approaches beforehand .

- The *Creators*: software engineers and testers mainly, although sometimes others, such as user interface designers, are involved. These are the people who will build the things the Product Owner asks for.
- The *Facilitator*: sometimes there is a dedicated facilitator who is not the Product Owner or a member of the building team. They may be, for example, a project manager, Scrum master or Agile coach.

  Some teams are too small to have a dedicated facilitator, so a developer steps into the role - in which case they wear two hats during the meeting. Experienced

> teams may not need a facilitator, but inexperienced teams who lack a facilitator may find the meetings long and difficult. Whoever plays facilitator should have some experience of planning meetings and facilitation. They should also have respect and authority from the team to play the role.
>
> Usually it is not a good idea for the Product Owner to double up as facilitator, because someone needs to watch for and resolve disputes between the Product Owner and developers. The Product Owner usually has more than enough to do during the planning meeting in any case.

I consider the total of all the people in these roles, and possibly some more as well, to be: the development team. The Product Owner and facilitator are, in my book, as much part of the team as the coders and testers.

4.2 The Artefacts

There are several artefacts, or props, which are normally used in a planning meeting. When teams rehearse planning meetings in training courses, I use dice to simulate the work in an iteration. While dice are not used in real-life planning meetings, they do simulate the nature variability that occurs in reality.

- *Blue cards*: describe bits of functionality that are useful to the business in a language the business representatives understand. These are vertical slices of business functionality

that are conceivably deliverable on their own. Often a 'user story' format (Cohn 2004) is used and the cards are usually created before the planning meeting.

- *White cards*: each card describes one task needed in building the thing described on the blue card. White cards are normally written during the planning meeting, and there are usually multiple white task cards for each blue card.
- *Red Cards*: these are bugs and other expedited items. Normally reds are not broken down into tasks, but they are occasionally. What colour you use for tasks related to a red card is up to you: white or red would both be appropriate.
- *Planning board*: usually a magnetic white board divided into columns. Often these boards are 1.2m by 90cm but the size is part particularly important, other sizes and types of boards may be used. While many teams use electronic tracking systems, I strongly advise teams initially to adopt physical boards and cards, and only progress to electronic systems when they have some physical experience. Even then both systems may be run in parallel.
- *Planning poker cards*: the description that follows assumes the team is playing planning poker. Not all teams play planning poker; it isn't compulsory, so feel free to use whatever method works for you. That said, whatever method you use should address some of the point discussed below.

Special sets of planning poker cards are available - these can be surprising difficult to buy, but are often freely available from Agile training and consulting companies - just ask! Different planning poker sets have slightly different sequences.

The exact colour of the story and task cards can vary from team

to team. Keeping to the colours outlined here does keep things consistent.

Some teams use additional colours to signal other types of work. One team started using yellow cards to signal unplanned work; this Xanpan convention has been described elsewhere. By their nature yellow cards will only appear at a planning meeting when they represent carry-over work.

4.3 The Meeting Sequence

The basic format of the meeting is shown in the diagram below. The team agree how much work they will attempt, the Product Owner presents the work they would like done and the team works through each item in priority order. They discuss each item, break it down to tasks and estimate the tasks.

After each item is done (i.e. discussion and breakdown has ended) team members count up how much work they have accepted and compare this with what they expect to do. If they have spare capacity, the next highest priority item is pulled from the Product Owner and the discussion, breakdown and estimation process repeats.

When the team has accepted work to its capacity, its members review what they have with the Product Owner and agree any changes.

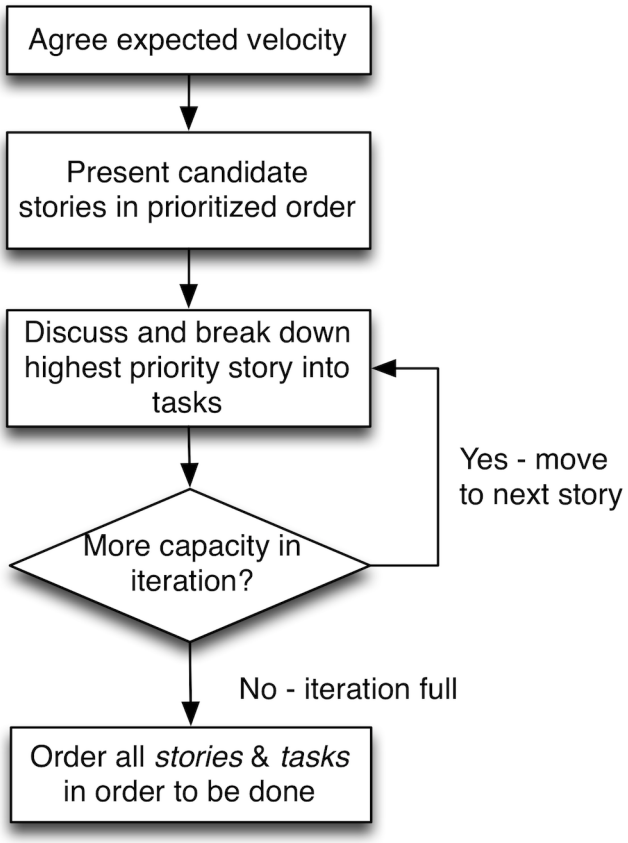


Figure 4 - Basic meeting sequence

First meeting

The first planning meeting a team holds is always the hardest. This is when their experience is least and the unknowns greatest;

naturally it will take longer to navigate the meeting. In the worst cases the team's lack of experience can derail the meeting altogether should it encounter difficulties.

The first meeting is also the most difficult for another reason: the team has no reference points. They have little idea of what they need from a story, how long it will take to do a story, or quite what acceptance criteria should look like. Even if they have practised for these questions during training, doing it in real life will be harder.

Significantly the team will also have no data on how fast it can go - it will only have a vague idea of how big 'one point' is or how many points it should accept into an iteration. In my experience teams tend to accept far more work into the first sprint than they will ever get done: they are inherently optimistic.

Second and subsequent

Because the first meeting takes so long and over-plans, the second meeting, two weeks later, tends to be one of the shortest. So much work is carried over in one form or another that the second meeting finds little to plan. After that the meetings start to settle down. The team has two meetings under its belt and two data points.

The meeting format also changes after the first meeting. At first the meeting is entirely forward-looking, but subsequent meetings have a backward-looking element. Prior to the start of the meeting, or right at the start, the team will demonstrate the work done in the previous iteration. It will then review the work done, usually by counting the points and updating any charts.

Formally the demonstration, review and retrospective might be separate meetings. But they are likely to be arranged back-to-back, perhaps with a short break between each. So whether one regards them as one long meeting or several shorter meetings is debatable.

When teams only delivered at the end of iteration - or less regularly - the demonstration of developed functionality used to be an essential part of the end-of-iteration and start of the next iteration. As more teams move to continuous delivery, it is worth questioning whether the demo adds anything if people can already use the actual software.

Teams may also hold a retrospective as part of the iteration end routine, although not all teams hold retrospectives, and even those who do may not hold them at every iteration.

The schedule of the second and subsequent meetings is something like the diagram below.

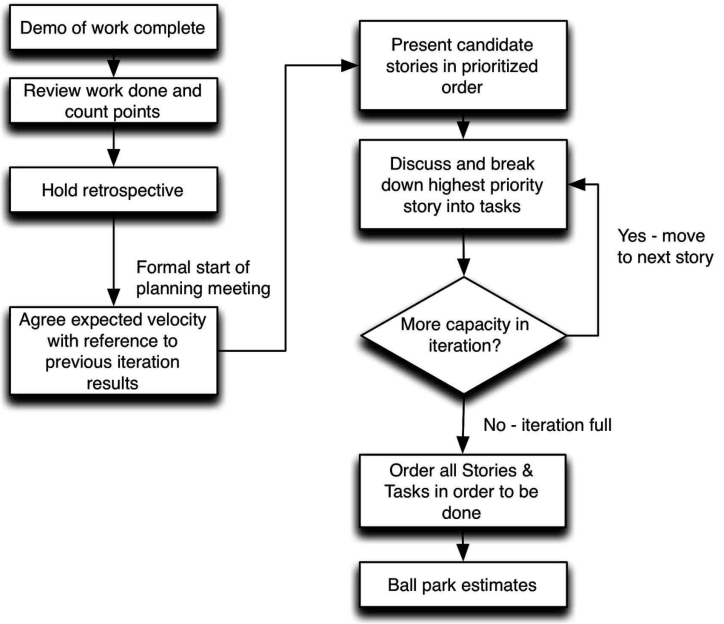


Figure 5 - Subsequent meeting sequence

In the second meeting the team has a rough idea of how many points of work it can accept, because members can sum up the points completed in the previous iteration. In the third meeting it has a better idea because it can average points from two iterations. By the fourth meeting the average is fairly accurate, plus reasonable high water (best case) and low water (worse case) marks can guide capacity thinking.

These meetings should happen regularly at the end/start of every iteration - typically every two weeks. They can be scheduled in everyone's calendar and room bookings made months in advance. If you are using an electronic scheduling system (such

as Microsoft Outlook) you can set up a recurring meeting with no end date.

4.4 The Planning Game

The game opens with the Product Owner(s) presenting to the team the blue (business facing) stories they wish to have developed. These are presented in absolute priority order - 1, 2, 3, 4 etc. No duplicate priorities are allowed, i.e. there can be only one priority one, one priority two and so on.

If two items are deemed to be of equal priority by the Product Owner (e.g. two cards are both assigned priority three), then the development team is allowed to decide the ordering. If the Product Owner disagrees with the ordering then they have, by their disagreement, determined the ordering. In general it is considered an abdication of responsibility if a Product Owner does not guide the team in prioritisation.

Each blue card is broken down by the development team to a set of white technical tasks. The white cards describe the things that need to be done in order to build the blue. One might think of the whites as the building blocks for the blues, or of the whites as *verbs* and blues as *nouns*.

Blues are the domain of the Product Owner, whites of the technical team. The breakdown is partly an act of design, partly an act of requirements elaboration, and partly an act designed to produce the smallest practical work items. (I will discuss the breakdown in more detail later.)

Of course sometimes only one white is needed to create one blue. In these cases the white is omitted and the team can

work directly at the blue level. In some ways this represents the idea scenario; however, for this to work, the blue must be no bigger than one white, i.e. if the blue can be broken down to multiple whites, it should be. (Of course there comes a point where further breakdown is excessive, but to start with teams find breakdown difficult. Once they have mastered it, they might consider whether some tasks are too small.)

The breakdown is a cooperative process between the development team and Product Owner - both should be present. There should be conversation between both sides: developers should ask about the requirements in detail, Product Owners may promote white cards to blue cards if they think the task itself has business value, and remove technical tasks if they want - even against the advice of developers - although in general the two sides should strive for consensus.

Giving Product Owners the ability to promote whites to blues can be a useful tool in extracting business value. It also gives the Product Owner the power to override much of the technical team's advice. On occasion this may be useful, but it may also indicate problems. For example, a Product Owner who frequently finds whites to promote may not be devoting the time and attention to preparing small blues in advance. Or it may be a sign that they do not trust the technical team and are attempting to micro-manage the work.

When white task cards have been broken out, those who will be responsible for undertaking the work - i.e. development team, all developers and testers, but not the Product Owner - estimate the work in terms of abstract points using Planning Poker. (See discussion below).

Teams track velocity from iteration to iteration. Unlike in financial services, past performance is considered a good indicator of future performance, or at least of the next iteration. I normally recommend using a rolling average across the last four or five iterations to judge upcoming capacity.

Once breakdown of a blue has started, it makes sense to see it through to the end. Even if part way through the team realise they cannot fit the tasks into this iteration, it probably doesn't make sense to stop mid-breakdown. This might result in the team accepting more work than it planned to, but this isn't a problem, as there is no commitment.

Of course once it becomes clear that a blue will not be completed in the iteration, the Product Owner could drop the blue and suggest a substitute. There is no hard and fast rule here. It makes little sense to plan the first tasks for a blue, but the order in which tasks are done does not necessarily correspond to the order in which they were identified and written on whites.

The team may or may not achieve all the scheduled work; they may perform below or above expected velocity in any given iteration. If the team do more then expected than the work is available, and over time the rolling average used to calculate the expected velocity will rise. Conversely, if the team find the iteration harder than expected, then less will be achieved and, after a couple of iteration, the rolling average will fall.

If a particular task or feature must be achieved within the iteration, it should be scheduled first and within the minimum recent velocity, low-water mark. This does not guarantee that the work will be done, but provides a very high probability. Teams are advised to track the time it takes for cards to traverse

the tracking board and develop statistically reliable averages and deviations to replace the Planning Poker estimation process.

Testing

Different teams handle testing in different ways. Some teams have professional testers and formal test processes, while some teams have neither. The level of automated testing is also widely different between teams.

White cards are generally not testable by professional testers. They should be tested by developers using automated unit tests and other tools to ensure they are acceptable. If there is something a tester could test, they may well be involved.

Generally professional testers work at the blue card level. In work breakdown a team might write a white task card to test a blue. This card would only be done once all the other whites were done and the whole blue was ready to test.

However, testers may prefer to write two task cards: one to write the test script and one to execute the test. If the former is fully automated, the latter need not exist, or will happen automatically.

Other teams forego tasks associated with testing and instead model tests via their task board. As cards move across the board they will need to pass through test columns where the testing will happen. Thus completion of all the white cards associated with a blue would trigger the move of the blue into a testing column and for testing to commence.

Trivia and Spikes

Truly trivial tasks, or work to be undertaken by people outside the teams, may be assigned zero points. Such zero-point cards represent work that the team needs to keep track of but which does not represent noticeable work for the team. For example, a 'buy domain name' task would probably merit a zero-point score, as it would take about 10 minutes. But a task of 'Obtain quote for domain name, seek approval to spend money, buy domain name, file expenses claim' may warrant an estimate larger than zero.

Spike cards are written when the development team feel they do not have enough knowledge - usually technical knowledge - to begin a breakdown and estimate. Here a spike card will be written to attempt the work, but once the work is done it will be thrown away, spiked.

The objective of a spike card is to gain an understanding of what needs to be done. Typically the output from a spike will be a set of (white) cards describing the tasks which need to be done. Ideally these cards will be held until the next planning meeting, where they can be discussed, estimated and scheduled, or deferred.

Sometimes when time is pressing the resulting cards might be estimated and scheduled into the iteration immediately. While this is entirely practical, it does mean forecasts for what the iteration will produce are difficult or impossible.

Spikes are not estimated in the same way that other cards are estimated - rather they are hard time-boxed: an amount of time is decided on and written on the card. This time, no more, no less, is the time allowed for this card. When the time is up research

work must stop and the task cards must be written using the knowledge gained.

Working in time - as opposed to points - for spikes makes velocity calculations more difficult. Some approximate, rule-of-thumb, back-of-the-envelope calculations need to be done in order to apportion a number of points to a time-boxed spike card.

Counting 'Done'

In the review part of the meeting the work completed in the previous iteration is removed from the board and reviewed. The main part of this review is simply counting the points done and updating any charts or other tracking systems. The review may also take time to examine any cards left on the board and decide whether they should be left as carry-over.

Teams are encouraged to adopt a definition of 'done' to help with defining what is *done* and what is *not done*. The definition of 'done' is a short checklist of things the team agree will be *done* for every card they claim is 'done'. This checklist applies to all cards in addition to any acceptance criteria placed on blues.

I normally recommend a definition of 'done' for whites' tasks, although some teams also apply it to blues. Occasionally teams feel the need for one definition for blues and another for whites.

The estimated points on completed whites are counted as part of the team's iteration velocity. Only 100%-completed whites are counted and the estimate is counted as-is, even if people feel it does not reflect the actual work. For example, if a card is estimated at five points it is counted as five points, regardless of whether people feel it was actually closer to one or 10.

No attempt is made to capture or work with 'actual' time or points. Human perception of 'how long things take' is subjective, and Vierordt's Law suggests we underestimate retrospectively as well as prospectively. The historical velocity data provides the necessary feedback on how long things took. By allowing abstract points to 'float', the system becomes self-adjusting.

Anyone who has worked with software teams for more than a few years will have seen the '80% done' scenario in which a piece of work remains 80% done for 80% of the time allowed. Therefore no matter how much a developer claims that "It is 95% done", incomplete cards are not counted.

In software development we have no way of telling objectively what is 95% done and what is 9% done. We have no way of knowing if an unexpected problem lurks in the final 5%, or if someone will go ill before the day is out.

This rule also sets up a small incentive for people to complete work before the end of iteration review and thus score more points. This adds a little extra motivation.

When an average velocity is used it becomes unimportant - for capacity measurement - exactly for which iteration the points are scored. Suppose a 10-point card is genuinely 90% done in iteration 13, but is not counted. It is then completed early in iteration 14, and thus counted in that iteration. While the variability of velocity will increase iteration-to-iteration, the effect of averaging over several sprints will still allow reasonable planning. The aim of estimation and forecasting is not to be precise about any single item, but to be generally accurate.

I just don't believe that adjusting for 'actuals' on a card-by-card basis, or allowing some points from partially done cards

to be counted, is a productive use of time. Both are subjective measures that invite discussion and disagreement - both time-consuming activities.

Some see it as odd that I allow whites to be counted even when blues are not. “But the business functionality is incomplete”, they say, “and it’s the business need that counts”. This is reasonable - and certainly follows the rules of Scrum - but I find it leads to less predictability in the process and evens out flow.

Counting whites as they are completed smoothes the flow. Only counting blues will lead to more variability, higher peaks and deeper troughs. While this complicates planning, it does not undermine it. One team I know of only counted points when blues were done, and they still delivered on time.

As before, the use of averages means that if a team decides to count only points from completed blues rather than whites, the velocity benchmarking system will still work. The drawback is that it might take slightly longer to stabilise and provide a stable capacity benchmark.

4.5 Velocity and currency

Velocity is a measure of how fast a team is working, or rather, how much work it is getting through. It is calculated by counting the points a team scored (i.e. completed) in the previous iteration.

Over a series of several iterations, say four, a team should be able to come up with a rolling average, a high- and a low-water mark which can be used for planning purposes. For example, consider the team shown in this graph:

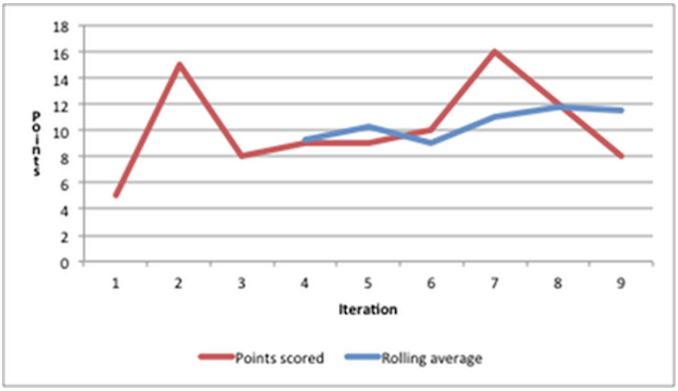


Figure 6 - Velocity and rolling average

This team scored 5, 15, 8, 9, 9, 10, 16, 12 and 8 points in the nine iterations shown here. At the end of iteration four the team could calculate an average of the last four iterations; this could be rolled forward at the end of each iteration, giving rolling averages of 9.25, 10.25, 9, 11, 11.75 and 11.5.

I would advise the team to plan for 12 points of work (because their recent average is 11.5) but accept 16 or 17 points worth - up to their recent high-water mark. Those in the planning meeting will be aware of the possible outcomes, but for those outside the meeting there needs to be some management of expectations.

It is pretty much certain the team will achieve the first 8 points worth of work. The team might get points 9 to 12 done, or they might not. If they are very lucky they will do points 13 to 16.

If someone needs to know how much time the team spent on a particular task, then it is simply a question of maths. Assume there are five developers employed for 40 hours a week in the previous example. That is 200 hours of work, producing on

average slightly more than 11 points of work, so each point took on average a little over 18 hours, thus a two-point card took about 36 hours.

I would prefer not to make this calculation too well known, because once it entered general knowledge it will undermine the points system. I would also prefer that this calculation be regularly updated since, as shown in the table, the averaging changes.

It is vital to note that points float. Like the US Dollar, Euro or Pound Sterling, points are a currency and change their value over time. Each team has its own currency that is not directly transferable to another team.

As with currencies and other economic indicators, setting targets for velocity can create problems. Goodhart's Law applies: if a team tries to target a certain number of points, it will meet its goal, but may not do any more work. Such teams exhibit inflation in estimates: exactly as with financial inflation, the numbers are bigger but the value less. (More about Goodhart's Law later, or see http://en.wikipedia.org/wiki/Goodharts_law).

Carry-over work

For a strict Scrum team there is no issue of work carry-over, because teams only commit to work they guarantee will be done, and thus all work committed to is done. While many Scrum teams find carrying work over from sprint-to-sprint an anathema, I advise teams to carry over work. Indeed, carrying over work to improve flow is a central feature of Xanpan.

For Xanpan teams work carry-over is a fact of life. As part of the review process preceding the planning meeting the team

should look at the work remaining on the board from the closing iteration and decide which, if any, work will be carried over to the next iteration.

When work has not started on a blue and associated whites, the Product Owner may decide to pull the card completely or roll the whole thing over. Assuming they roll it over, it will need to be prioritised against the new blues being added. That is to say, just because a blue is rolled over does not give it special priority.

When some tasks associated with a blue have been done and some tasks have not the situation is more complicated. While the Product Owner may still pull the blue or assign it a low priority, it probably makes more sense to finish work that has been started, and finish it soon, rather than leaving it in a partially done state.

There may also be engineering reasons why the blue should be taken to completion before anything else. For example, some of the new blues may involve the same areas of code.

In a few cases work is incomplete because more tasks came to light after it began. While I do not allow teams to change estimates on whites once they are accepted into an iteration, the team may write new whites for additional work which emerged. They may even estimate and begin work on these whites during the iteration if need be. However I prefer it if new work can be held until the planning meeting, where it can be discussed, prioritised and scheduled by the team.

Obviously this approach raises the possibility of never-ending work, blues that are never done. Senior team members need to be watchful for this and work to diagnose the underlying issues causing it.

How long is a planning meeting?

I would expect a well-practised team to complete a planning meeting in half a day; my preference is for afternoon meetings. Obviously there is some variability, depending on how big the team is, how much work is being planned and whether the team is carrying over any work, but half a day should be enough.

The exception is the first meeting, which will frequently take much longer, perhaps as much as a day. Meetings can also stretch out when Product Owners are poorly prepared for the meeting or take issue with estimates. Design questions can also derail meetings, but on the whole most design issues can be followed up later.

A team holding a retrospective before the meeting should allow 60 to 90 minutes, depending on the techniques and exercises being used. I find a dialogue sheet retrospective takes 60 minutes for the sheet, plus up to 30 minutes for post-sheet discussion and action items.

4.6 Product Owner Preparations (Homework)

One of the recurring reasons I see for planning meetings not going smoothly is a lack of preparation on the part of the Product Owner. The planning meeting is not the place for the Product Owner to decide what is required. Although they may make trade-offs and substitutions during the meeting, they need to go into the meeting knowing very clearly what they want to ask for.

This does not mean everything they want will be accepted and scheduled, but they should be prepared.

The Product Owner needs to be on top of their brief: they need to be able to answer developer questions and clarify what is being asked for. If they cannot, they need to either bring someone who can, or be prepared to make changes to what they want. Thus, if the testers and developers ask questions about issues the Product Owner cannot answer immediately, they might bring another story into play while they find out the answers. This may mean that the first story is postponed until the following iteration - or even later - or it may be possible to schedule the difficult story later in the iteration.

As you might guess from this, it helps if the Product Owner is not only prepared for the iteration they are planning now, but also has a rough idea of what they plan to ask for in future iterations. These plans shouldn't be too detailed - because things change both in priority and detail - but the Product Owner needs to have some ideas.

Medium-term plans, about the next few iterations, were traditionally called *release plans*. I believe the name *quarterly plans* both better describes the plan and moves away from the association with releases. (This is discussed in a later section, *Planning Beyond the Iteration*.)

It is also critical that the Product Owner has authority from the organisation and team to make decisions during the planning meeting: on priorities, on changes to priorities, on details of features and on trade-offs. Nothing is more disruptive - and morale-sapping - than completing a planning meeting one day only to discover a day or two later that somebody else has

overruled the Product Owner and has changed what was agreed in the meeting.

There is sometimes a need to have more than one Product Owner in the planning meeting. When this happens all Product Owners concerned need to be in agreement about what is going to be asked for and what the priorities are, and be prepared for problems. The Product Owners may benefit from having their own small meeting prior to the full planning meeting.

> An Iteration Planning Sheet is available to accompany this description. Visit http://www.softwarestrategy.co.uk/dlgsheets/planningsheet.html to download a sheet or to buy a printed sheet.

4.7 References

Cohn, M. 2004. *User Stories Applied.* Addison-Wesley.

5. More Planning and Estimation

6. Watching the numbers

7. Board 2

8. Non-technical Practices

9. Technical Practices

10. Planning beyond the Iteration

11. Origins of Xanpan

Appendix: Quality

Continuous Digital

Continue the #NoProjects story with Continuous Digital continues

Allan Kelly's latest book:

- Why digital business need a new model of software development
- A full description of the Continuous model

Ebook draft available on LeanPub[1] and pre-order on Amazon[2].

[1] https://leanpub.com/cdigital/
[2] https://www.amazon.co.uk/Allan-Kelly/e/B001JSFJEE

Xanpan links & ISBNs

Team centric Agile software development

Allan Kelly

Published by Software Strategy Ltd.

http://www.softwarestrategy.co.uk

Xanpan homepage: http://www.xanpan.org

ISBN Print version: 978-1-912832-05-7 (Replaces 978-1-291-85273-8 - no text changes, 2020)

ISBN PDF version: 978-0-9933250-2-1

ISBN EPUB version: 978-0-9933250-1-4

ISBN MOBI version: 978-0-9933250-0-7

Electronic versions from http://www.leanpub.com/xanpan