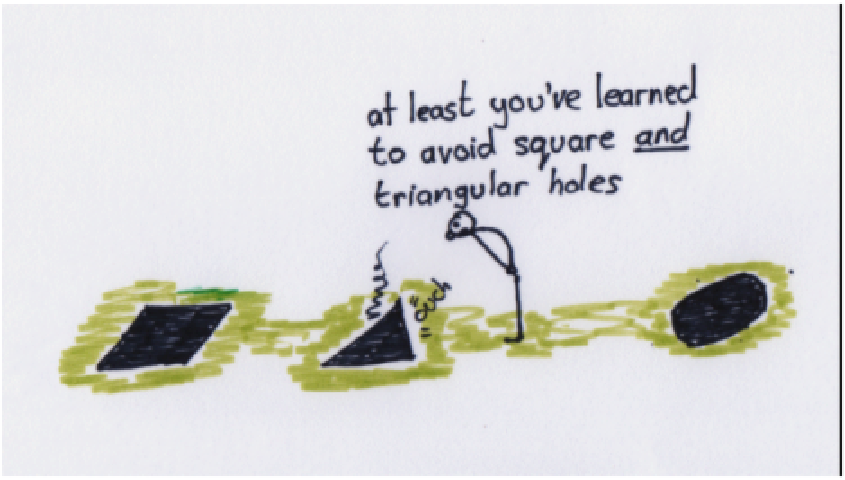
The Scrum Survival Guide


at least you've learned to avoid square and triangular holes

Serge Beaumont

The Scrum Survival Guide

Serge Beaumont

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

This version was published on 2013-09-30

Tweet This Book!

Please help Serge Beaumont by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just bought the #SSG book by @sbeaumont. I Am Saved!

The suggested hashtag for this book is #ssg.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#ssg

Contents

Fixing the Cause-Effect Trap in User Stories

If you write user stories, it is very likely that you have been using the *"As a... I want... So That..."* stanza. What you might also have found is that it is hard to fill the "So That" clause with something that makes sense. *"As A User I Want a button So That I can go to the next screen"*... that is pretty naff, isn't it? So how do you fix it? Ask "Why" several times!The *"As A... I Want... So That..."* user story stanza is a very nice tool. It helps you put a lot of information in a very compact form, which is great for summarising a user story, for instance as a descriptive one-line title in an electronic tool. But there is a trap in this stanza that I have seen just about everyone struggle with when using it, leading to all kinds of suggestions, mainly by changing the stanza around (*"In Order To... As A... I Want..."*). But those suggestions don't fix the root cause of the problem. Let's start with identifying the real problem.

The real problem: the Cause-Effect Trap

In the *"As A... I Want... So That..."* stanza you answer the questions "Who?", "What?" and "Why?" respectively. So our button example becomes:

- Who wants it? The User.
- What does he want? He wants a button.

- **Why** does he want it? He wants to go to the next screen.

The “Why?” question is the interesting one here. The answer “he wants to go to the next screen” is a perfectly valid answer, but the problem is that this is not something that makes sense as business value, which is what you would like to have in the “So That” clause. What happened is that with this first “Why?” you have only stated a cause-effect relationship: when you press the button (cause) you go to the next screen (effect). This is what I call the Cause-Effect Trap.

How do we fix the Cause-Effect Trap?

The trick to fixing the Cause-Effect Trap is to realize that there is a chain of Why’s that you need to answer before you get to the right level. The first “Why?” you ask in the “So That” clause is just that, only the first in a chain. Let’s fix our user story. The first thing we see is that that the answer to “Why?” is a “What” in its own right, but at a higher level. So the first fix is to move the “So That” clause to the “I Want” clause, and answer the next “Why?” in the “So That” clause.

> **As A** User **I Want** to go to the next screen **So That** I can shortcut from screen 1 to screen 9.

Hmm, a shortcut? Why, I wonder? Let’s do the fix again:

> **As A** User **I Want** to shortcut from screen 1 to screen 9 **So That** I can save time on 90% of the cases in the screen workflow.

Ah, so apparently screens 2 to 8 aren't needed in 90% of the cases, and this saves time. Why, I wonder…? (See the pattern? :-))

> **As A** User **I Want** to save time in 90% of the cases in the screen workflow **So That** I can improve handling time in my helpdesk

Ah, so there is someone who wants to improve a business process. Apparently it takes a long time to plough through screens 2 to 8 while it isn't needed… Now, that sounds more like business value! But now the "As A" clause does not feel right. Who wants this business value? Who is being referred to in "my helpdesk"?

> **As A** Helpdesk Manager **I Want** to save time in 90% of the cases in the screen workflow **So That** I can improve handling time in my helpdesk

Now that looks better. Let's take this back to the team… They will likely respond with: "save time in 90% of the cases? That's too vague for us. Can you be more specific?". Okay, let's backtrack down the Why Chain to something more concrete.

> **As A** Helpdesk Manager **I Want** to shortcut from screen 1 to screen 9 **So That** I can improve handling time in my helpdesk

And there we have it: a fixed user story!

What have we learned?

Lesson 1: You can fix a user story by going up the Why Chain, replacing "I Want" with "So That" as you go along. (In effect you've made the stanza go: As A… I Want… So That… So That… So That… So That… :-))

Lesson 2: The "As A" clause is more closely related to the "So That" than the "I Want". This is logical, since a stakeholder is not interested in features, but in the business value that feature provides. A feature is just a means to an end.

Lesson 3: Put the highest statement in the "I Want" clause where you're still making a statement about the product. An extra – and huge! – advantage is that you will have considerably widened the solution space for the team, enabling the team to come up with better solutions: there are a lot more options in "providing a shortcut" than there are in "a button". Square button or round? Green or Red?

Lesson 4: Look at that that "improve handling time" bit… Wait! Could it be true? Do we actually have a basis for measurable business value? So That I can improve the average handling time by 20% (current: 5 minutes 20 seconds). That would be a great to help the Product Owner prioritize!

Serge's Three Step Process For Fixing User Stories

So here's my 3-step, fail-safe, make-you-a-millionaire-in-a-week plan to fix user stories ;-):

1. Use the Why Chain: Ask "Why?" until you're talking business. Put that in the "So That" clause.
2. Fix the "As A" clause to reflect the correct stakeholder.
3. Fill the "I Want" with the highest level statement that says something about the product.

…and there you go. Killer user stories are yours! Happy fixing!

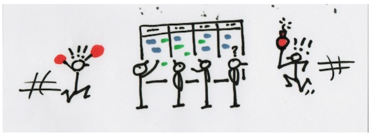
Dealing with Emergencies

Every Agile team has to deal with whatever they've put out in the wild next to their "regular" work. How to handle the - by definition - unknown load of production emergencies when you're trying to achieve a stable pace? You can deal with emergencies by performing triage to either reject, defer or accept. You can set up a buffer to absorb some of the uncertainty, and finally you should make sure that you take the time to reduce the number of emergencies by building quality in. If you find you are mostly doing maintenance, you can consider doing Kanban.

The Context

In ye olden days of waterfall projects I never had to deal with that horror of horrors, maintenance. I'd be part of a team building something new, and you could keep going on until the end of the project. It was the maintenance department that would have to deal with the nonsense I had created. Ah, those were the days… all the fun without the hangover afterwards :-) But Agile teams, or in fact any team that starts delivering early and often (in my later waterfall days I'd already started to figure out that maintenance pretty much starts after the first two weeks… :-)) deliver long before it's even possible to hand the project over - if at all. The nature of frequent delivery means that the team has to deal with all issues that arise themselves. The first reason is because they are the ones who can do it, the second is that you want to integrate fixes into the team's work anyway: they still need to deliver new versions of that same software… In my consultancy work I've seen this issue come up with every single Agile team I've known, so this is not a unique situation for a small number of teams. All Agile teams have learn how to deal with this issue!

The Problem



It's an emergency!

In a Scrum team, the problem will generally surface after one or more Sprints where a number of "production incidents" or similar unplanned mayhem took up so much of the team's time that they did not achieve their planned Sprint goal. The result is that a team has a hard time planning for the next Sprints. The first problem is that they do not know their "real" Velocity, the second is that they have to somehow factor in the - by definition unpredictable - production incidents.

But watch out, there is a pitfall hidden in the above paragraph. Predictability is not the end goal in Agile! Predictability is important to know when a release is shipped, and to know how to pace the team. But I've seen too many cases where teams try to "predict harder" when they should be adapting better. When dealing with the unpredictable, the focus should be on adaptation first, not on more planning beforehand. That would be a return to The Way Of The Waterfall…

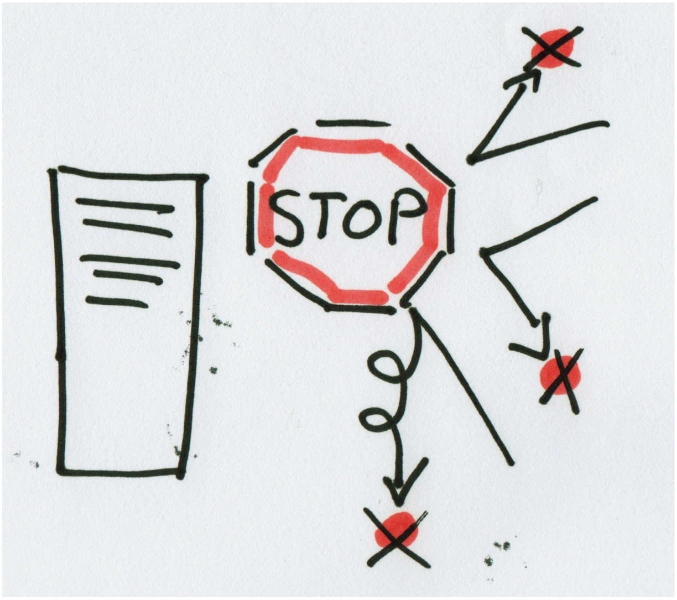
The Goal

So there we have it: the goal is to be able to absorb a reasonable amount of uncertainty, striking a balance between robustness and speed.

The Solutions

Before I present some solutions, let me state this right away: if the amount work of unplanned production incidents is significant compared to the “regular” work, there is no way you can achieve sufficient stability. You’ll need to fix the root causes of all those production issues first. More on that later.

Solution 1: Perform Triage - and Reject



Triage… and reject

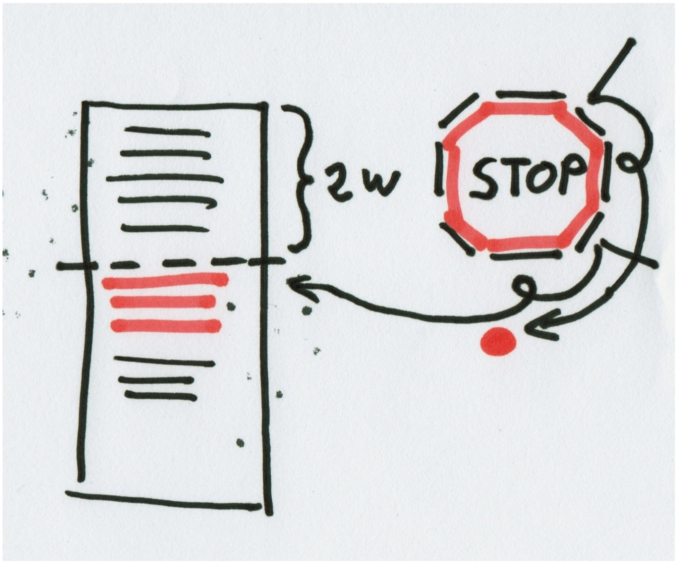
The first thing to check is if you want to fix that production issue at all. This is not as silly as it might seem at first. There are so many cases where a production emergency is not an emergency at all, and should not even have been brought in in the first place! Some examples of "noncidents":

- Sales storms in with "the deal of the century": "If we get feature X in NOW, we can win over customer X!". In my experience this is always due to an uneducated

  and undisciplined Sales department. The root cause here is that Sales promised things they shouldn't have, and they need to save their own skin now. It is ALWAYS possible to wait two weeks for a new feature.
- Some stakeholders "upgrade" normal requests to production emergencies in an attempt to bypass the negotiations around the backlog. "It's a blocking issue that I can't get that feature!". "Oh? Did the system crash? Is something not working?". "Well… no, but it's a real blocker for my work!". That stakeholder may have a genuine need, but that does not make it a production emergency.

So solution 1 is: a strong Product Owner who performs triage on all production issues. If it's a real production issue then by all means fix it. But I can guarantee that you'll find a good number of issues that should not be emergencies at all… BTW, a Product Owner performing triage in this way is what James Coplien calls a Firewall in his organizational patterns book.

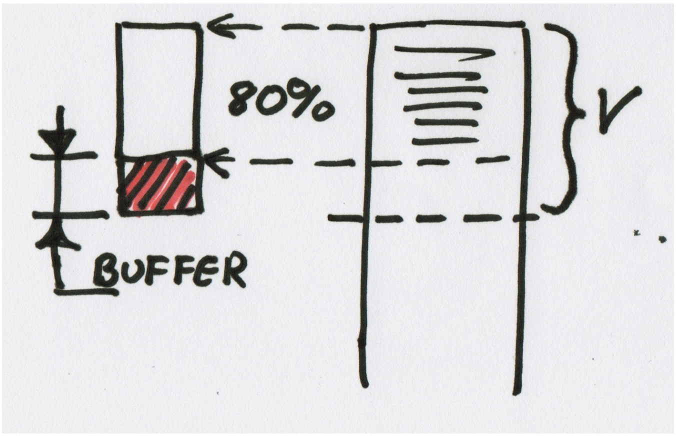
Solution 2: Perform Triage - and Defer the fix until at least the next Sprint



Triage… and defer

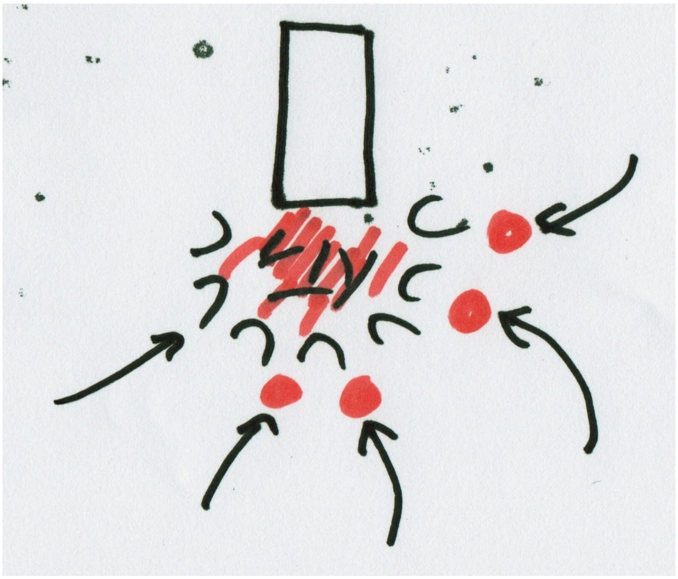
“We found this really big problem! We need it fixed right now!”. “Sure, we’ll get right on it. How long has this issue been in the system?”. “Well, for over a year, but we just found out about it!”. “It’s been in there for a year? …And you can’t wait two more weeks for a fix?” Solution 2 is an extension of Solution 1. An emergency might indeed be important to fix, but there’s an important criterion to an emergency: it’s only an emergency if it must be fixed in the current Sprint. If you can defer the problem to next Sprint, there is no problem! The team can pick it up as part of their regular process, plan it, build it, and deliver at the end of next Sprint. Again this is a Product Owner responsibility: next to the decision to reject, a good Product Owner will make sure that everything that can be deferred will be.

Solution 3: Reserve a buffer to deal with unexpected issues



Reserve a buffer

If you've done Solutions 1 and 2, whatever you're left with should be real issues that you have to fix as soon as possible. The best way I know to deal with this is to reserve a buffer of time or story points that is left unplanned. This works especially well if the historical workload of any issues coming up is reasonably stable. You do not know what you'll be doing, but you know how much effort it will take. Watch out though, using a buffer can blow up in your face! The first danger is the size of the buffer. If the buffer is a significant percentage of the Sprint, say more that 1/5 of your velocity, then you'll end up with a big hole in your planning process. So follow Buffer Rule 1: the buffer is not for backlog items. Try to keep the buffer as small as possible.



Don't overload the buffer

The second danger with using buffers is what I already discussed in Solution 1: the moment your stakeholder smell a workaround in the regular process, you can be sure they'll dive onto it. A buffer really, really needs to be protected from unintended use. So perform good triage! The third danger is buffer overflow. Just like in a computer this leads to blowing up the process. If the buffer is used, you'll need to track how much of the buffer has been used, otherwise you'll be in for a surprise at the end of the Sprint.

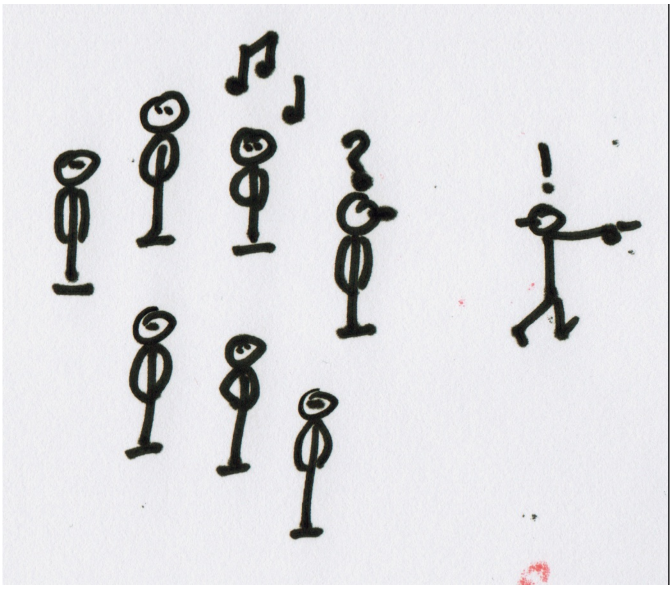
Solution 4: Fix root causes, improve quality

This solution is presented as number 4 because the first three are in logical order when you're trying to control the damage, but in the end you'll want to do the most important thing of all: fix issues so they stay fixed, build in quality so that you don't have emergencies at all!. Now this is something we should be doing anyway, and is not unique for Agile projects: you want to do this in any project! But there is an extra Buffer Rule that is relevant in this respect (Credit goes to Jeff Sutherland on this one, I learned this rule when we do CSM trainings). Buffer Rule 2: If you overflow the buffer, abort the Sprint. If you have such issues that you can not even keep emergency work limited to a small buffer, you have no business trying to make progress building in features. Abort, use the Sprint to fix underlying root causes, and try again next Sprint. Coincidentally, Buffer Rule 2 also works wonders for all those stakeholders trying to "upgrade" their own agenda: "do you really want that issue fixed now? The team estimates that this is two points of work, and this would overflow the buffer. We would have to abort the Sprint, and you also would not get those other user stories you asked for! Oh… um. Well, I guess it isn't that much of a problem…" (And it wasn't… Real story!).

Extra: Size the team right

Small team: losing someone hurts

Team size is not a central focus in dealing with emergencies, but it is a factor to be aware of. A small team performs better because it has less overhead, but it is less robust against losing members. A small team is less robust against things like illness or something that pulls a team member away like… production emergencies maybe?.



Large team: losing someone hurts less

On a 10 person team losing one person "only" means a hit of about 10% in productivity (this is a simplified calculation of course, this assumes all team members are totally replaceable on a moments notice), in a three person team losing that same person would already mean a whopping 33%! The sweet spot tends to be around 7-9 people. Small enough to reduce overhead, large enough to absorb some production loss.

And finally… consider using Kanban instead of Scrum

If you find that your team is doing more maintenance than “new stuff”, you might consider using Kanban instead. This is because the granularity of Kanban is stories, not Sprints. If there is a production emergency there is already an intrinsic shorter wait for it to be picked up because of this. Kanban is about flow, while Scrum is about iterations. The two styles are close enough that I’ve seen a Scrum team transition into “flow mode” when they scaled down and only did maintenance, and went back to Scrum when a new release was planned, and they scaled up again.

In Conclusion

Every Agile team has to deal with whatever they’ve put out in the wild next to their “regular” work. You can deal with emergencies by performing triage to either reject, defer or accept. You can set up a buffer to absorb some of the uncertainty, and finally you should make sure that you take the time to reduce the number of emergencies by building quality in. If you find you are mostly doing maintenance, you can consider doing Kanban.

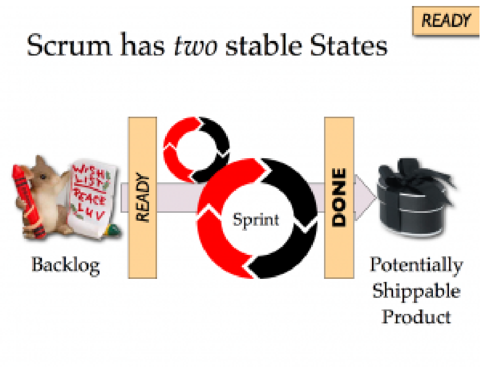
The Definition of READY

I give CSM trainings with Jeff Sutherland, and about half a year ago he had put something in his material called “the dynamic model of Scrum”. The essential feature was the addition of a READY state opposite the DONE state. The idea here is that a team needs to be in a stable, known situation to be able to perform well. It was based on two papers Jeff did in 2008 with Carsten Ruseng Jakobsen from Systematic 1[1] 2[2]

It immediately struck a chord with me: I had seen so many teams thrash because the Product Owner could not give them a clear objective, the READY state was exactly the goal to work to. But what was it really, and how do you get there? By now I think I’ve got some good answers to these questions.

[1]http://agile2009.org/files/session_pdfs/GoingfromGoodtoGreatwithScrumSession.PDF

[2]http://agile2009.org/files/ScrumCMMIGoodToGreat.pdf



Scrum has two stable states: Ready and Done

Here's a picture from my Scrum course material to illustrate the concept...

What does the READY state do?

In a self-organizing team setting a clear destination it very important: self-organization does not exist if you have nothing to organize TO. The READY state prevents team thrashing by ensuring that the preconditions for a good Sprint execution have been met.

Defining READY

READY is defined by the Definition of READY. It is similar to the Definition of DONE, but with the following differences. Whereas with the Definition of DONE the "supplier" is the Team and the "client" is the Product Owner, it's the other way around with the Definition of READY: the Team is the “client” and the Product Owner is the “supplier”. Even though I will detail the Definition of READY later, in the end it boils down to one statement: READY is when the team says: “Ah, we get it”.

Even though you can put any precondition in the Definition of READY, the need for a good backlog overshadows all other considerations, so you’ll definitely need to address two items: readiness of User Stories, and readiness of the Backlog.

When is a User Story READY?

I have found that a User Story is ready when you have answered three questions: Why?, What? and How?, it has been estimated and it is small enough.

- Why?: What are the stakeholders trying to achieve? What are their goals? What is the business context? What is the Quantified Value?
- What?: What is the Outcome Vision? What is the end result of the user story?
- How?: What is the Implementation Strategy? What is the associated cost (story points)? Is the story small enough (story points vs. team velocity)?

Note that with answering these question I do not want to imply that you need a whole lot of documentation or artifacts. Again, it’s whatever the team says it needs. If the back of a napkin suffices, go for it. If the team needs more, provide that. Note that the detail level needed must be determined per user story. Some will be business as usual, and you may suffice with a simple “I want one of those, but this time in green”. Others could for instance be related to a new process in the Intensive Care ward of a hospital. You might just need a tad more there… ;-)

I use the term “implementation strategy” to further clarify the level of detail needed. Fully specifying the How? would lead you back to waterfall, but you can’t make a points estimation if the team does not have a rough idea of how they’ll tackle the user story. So you go as far with specifying the implementation as is needed for a points estimation. Note that this is a natural side effect of planning poker sessions, so the easiest is to capture the outcome of that discussion along with the estimation. And I really advise that you do it, I’ve seen too many cases where the team wonders why they gave that user story 5 points, just two days after the planning poker session… :-)

In the end a User Story is READY if a team can implement it, and a Product Owner can prioritize it.

When is the Backlog READY?

The backlog is READY when about 1,5-2 Sprint’s worth of User Stories at the top of the backlog is READY, and those user stories are sufficiently small to allow good team flow.

And finally, follow this mantra

Don’t let anything that’s not READY into your Sprint, and let nothing escape that’s not DONE.

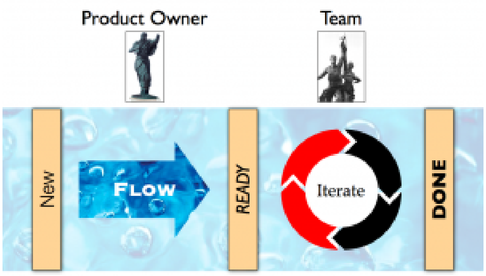
Flow to READY, Iterate to DONE

I find that a Product Owner’s job is best done in a flow style. Why flow? Ask me with an update of the book!

Not all backlog items are equal. A backlog item starts out as a rough sketch – usually just the *As A.. I Want... So That...* stanza – and needs to be fleshed out to the extent that it can be picked up by the team in a Sprint. Just like a team has a basic workflow getting stuff to Done, the same applies for the Product Owner role. Scrum does not have any specific support for a Product Owner: somehow the Product Backlog just “happens”.

I’ll explain a partitioning of the backlog that maps onto a flow, the nature of those partitions and how you proceed through them to get enough stuff Ready for the team to pick up in the next Sprint.

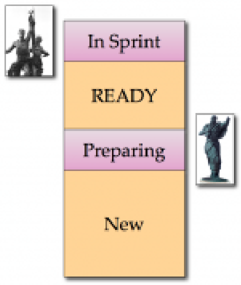
Flow to READY



Flow to Ready, Iterate to Done

The overall flow of work through a Scrum team is that the Product Owner role picks up new stuff, gets it READY, and the Team role picks it up to get it to DONE. Note that I explicitly use the word "role" here: team members have a role to play in supporting the Product Owner role to get backlog items to READY.

Partitioning the backlog



The Backlog can roughly be divided in four areas

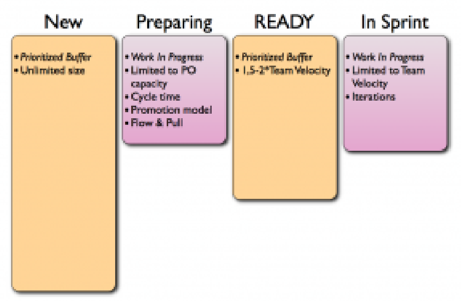
A backlog can roughly be partitioned in four areas based on the overall flow:

1. items that are currently **in the Sprint**,
2. items that are **Ready** to be picked up in a Sprint,
3. items that you're currently **preparing** to Ready, and
4. the rest: **new** stuff.

Of course this is an idealized view of things. In practice the lines are blurred somewhat because the mapping of priority on the workflow steps is not always as clean as you'd like. New items might show up really high in priority, putting it in between the READY items. On the other hand this way of viewing the backlog could be used to enforce the prioritization: something that's READY could by definition be prioritized higher than something that's not.

From partitioning to flow

If we take these flow steps and put them side by side, we get the following:



The backlog can be partitioned into four flow steps

The fact that I've used one color for "New" and "Ready", and another for "Preparing" and "In Sprint" is not a coincidence: "New" and "Ready" are prioritized buffers, "Preparing" and "In Sprint" are Work-In-Progress. Let's go through the flow step-by-step.

Prioritized buffer: New

Backlog items in the New state are the ones you haven't started working on yet, at least not to the extent that you're getting them to READY. Even so, in practice I've seen that it's wise to perform a minimal triage on these items: if you have every mad idea on there, you'll quickly be inundated in an avalanche of items. Stakeholders tend to say "I've got a thousand ideas!", but many of them are just that: ideas. Only a small fraction are actually realistic or useful to implement. This initial triage should be kept simple, but it does put some discipline with stakeholders putting in their requirements. Don't be too worried about stakeholders complaining, in general a stakeholder will appreciate knowing what they need to do to get their requirements in :-). To be put on the backlog as New, a stakeholder should provide the following:

- a story solely in terms of the business experience that describes what they experience and what they need without referring to how the product would support it
- user story stanza (As a.. I want… So that…)
- a (rough) valuation of benefit
- a rough indication of size (i.e. cost): small, medium, large. (Note that this is best guesstimated by a team member or the product owner after a chat with the stakeholder: they have more knowledge of the product)

The business urgency gives you a rough prioritization, which enables you to decide which items to pull in first.

Work-In-Progress: Preparing

This step is the big one as far as the Product Owner is concerned: it's where the core Product Owner work gets done.

The Product Owner is the single point of contact for all stakeholders, and this is of course intentional. There needs to be a single focal point for all requirements and prioritization, otherwise it will fall back to the team role and the Product Owner role is all but useless. An unfortunate side effect for our poor Product Owner is that they constantly get bombarded with requirements and pressure from all stakeholders. The result is that a Product Owner is stretched thin trying to deal with it all. I've found that this is where the flow/kanban style really shines: explicitly limiting work in progress is one of the best tools to bringing some sanity in the Product Owner's life.

The Product Owner pulls items into the Preparing step according to capacity. Just like a team pulls in work to capacity and does not change it until the Sprint is over, a Product Owner should pull in a number of items (I've seen two per person in the Product Owner role a lot, don't know yet if that's a general thing, though), work on them until they're Ready, and only pull in new items when a "free slot" opens up in the preparing step.

The Product Owner does not need to (and in most cases can't) provide all information, but is responsible for making sure that someone does, so that the backlog item will get to Ready. This means that a Product Owner will talk with stakeholders to ask them for more information, with the Team to provide an estimation of implementation complexity, and anybody else who is needed to provide clarity and information. This is quite a job, and in larger organizations it's not unusual to see multiple people (often analists) supporting this role.

Because of the explicit step in the flow it is now possible to measure Product Owner performance. The equivalent of velocity in the flow style is cycle time: the average time needed to bring a backlog item from New to Ready. A backlog item that is stuck will be easier to recognize, since it will remain in the Preparing step longer than is usual. It also helps to plan Product Owner capacity. Comparing the cycle time with the a number of backlog items that is consumed per Sprint by the team helps to determine if the Product Owner is going fast enough to keep up.

A Product Owner's speed is not measured in a backlog item's story points but in number of backlog items because the amount of work for each item is roughly the same: every item needs its questions answered regardless of the size. Large backlog items may be more work, but in most cases they will have to be broken up into sizes that are manageable by the team. This translates into more backlog items for (originally) large ones, so large items are factored in this way.

When a backlog item is Ready it can be moved into the Ready buffer.

Prioritized buffer: The Ready Buffer

I have found it very useful to think of the list of Ready items as a buffer. A Product Owner's productivity needs to be such that this buffer is full enough when the next Sprint starts. Tracking the size of the buffer (in story points, since now the capacity of the Team is the relevant one) is a very good way to see if the Product Owner is getting into trouble. You could use a burn-down chart, a burn-up chart, or simply a continuous trend line of buffer size, but I find it a great help that a Product Owner has access to the same type of trend information that is available to Teams when they use burn-down charts.

There are two levels of Ready: each backlog item needs to be Ready, but the backlog needs to be Ready as well, just before the next Sprint. Backlog-Ready means that the Ready buffer is full enough for an iterations' worth of work, and some extra work as "spare change" in case of renegotiation, last minute decisions, insights or priority shifts. In practice going for 1.5 to 2 iterations' worth is a good target. The reverse is also true: if the buffer is really full, more than two Sprints' worth of Ready items, you're likely wasting work since reality will change before you get round to the later items in the buffer (items become “unready”), forcing extra work. In that case you it’s better to increase the Team capacity or use the free time for some crystal ball gazing or market research :-).

Work-In-Progress: In Sprint

Of course this step is relatively easy to describe, this is where all the usual Scrum stuff enters the picture :-). As a Product Owner you’ll track which items are In Sprint, but your work is not entirely done. In an analogy of a quote on design: “a good design does not depend on one big decision, but on hundreds of little ones”, a Team will need you around to unblock them with decisions. During a Sprint a Team will come up with alternatives for details of the implementation. Often these alternatives have an impact on the end result: they might have an easy option but it will not be exactly what was asked, or a team might find a part of the implementation is harder to do than expected. In that case you need to be around to help them forward.

Summary

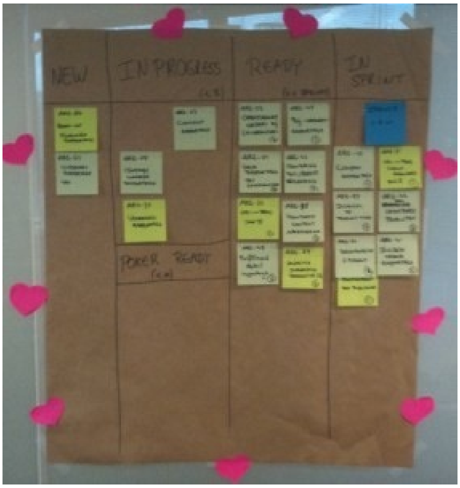
The backlog can be partitioned in four parts that you can connect with a flow.

The READY Kanban: the Product Owners' "Scrum Board"

The basic Ready Kanban has three columns: New, Preparing and Ready. You can add the fourth "In Sprint" if you want to show what's currently in the Sprint, but that's only really needed if it's not hanging close to the Scrum Board: otherwise the Scrum Board effectively functions as the In Sprint column. It's good to have a visual cue in the Preparing column to show the limit on work in progress, like a fixed number of slots for user story cards.

Rob's READY Kanban

This board was created by Rob Buster, one of the Product Owners at bol.com. As you can see his board has four columns since this board is hanging close to his desk, which is on a different floor from the team floor.



Rob’s Ready Kanban

Some things to note on his board:

- he added a “Poker Ready” section, where he can collect a number of user stories that can be estimated in one go in a poker planning session. This is a very good idea and will probably end up on many Ready Kanbans. Even so, I specifically do not prescribe it in my generic format, since it’s up to the work dynamic between a Team and a PO if they want to do this one at a time or in batches.
- the pink hearts were added by a colleague in the same room who found the large brown paper ugly… :-)
- Rob uses blue stickies to partition his Ready buffer

into Sprints. Again a very good idea, since it clearly shows the currently expected set of user stories to be picked up in the next sprint. It also allows physical shuffling of user stories to fill up the Sprints to the team's velocity.

The first Ready Kanban

The first Ready Kanban

This was the first Ready Kanban ever to be put up. I initially filled it with the top of the backlog that was in preparation: the top five in the "Preparing" state, and the next 10 in the "Top 10 New" state. Note that it's hanging in its logical place: on the left side of the Scrum board, in line with the overall left-to-right flow of user stories.

This is that same board a few days later (I've fudged out the titles):

The Ready Kanban a few days later

Many interesting things had happened here:

- The analysts, who were performing all the getting-to-Ready work, started showing up frequently in the team room. Just like a Scrum board does for a team, it became their rallying point to discuss coordination, status and progress.
- The New buffer started drying up. With the improving speed of the team, many items on the old "change requests" list were reevaluated and found to

be outdated or obsolete. It clearly pointed at a need to get the dialog with the business up to speed to get their requests. I had not expected this to happen, I thought that this buffer would always be full. It turns out that there is a subtle effect going on here: putting something in New implies a subtle promotion, and this leads to a simple triage being done on the user story. You could say that it functions as a "bullshit filter" .

- The top card in the Preparing column was stuck there while the ones below it moved. It became clear that there was an impediment in getting that user story ready, and extra effort and focus went into resolving the decisions to be made by the business.
- The Ready buffer was not filling rapidly enough. The level of the Ready buffer is a really good way to show if the velocity of the PO in sync with that of the Team. It works so well in this regard that the team members saw that it would not really be that useful to keep pounding away at functionality if the work would dry up. So they started to help the PO role to get user stories Ready. Now that is what I call self-organization!

Conclusion

All in all, I am REALLY happy with the results I've seen when I applied this board. All of a sudden the PO's have their information radiator, they can coordinate with others, stuck user stories are clearly visible, and now there is a basis for measuring the performance of the PO role. We can make a trend graph showing the level of the Ready Buffer (the PO's version of the burndown…), we can measure average completion time of a user story (in line with Lean's "takt time"), and the PO can bring some sanity back by limiting the work in progress in the Preparing state.

"Committing" to a Sprint and failing is a GOOD thing!

What does it mean when a Scrum team "commits to a Sprint"? There is a subtlety in the English language that leads to misinterpretation and misuse of the verb "to commit". I have seen too many cases where a team is held accountable ("bad team, bad!") because they did not achieve their Sprint goal in some way. And it will be accompanied by the accusation: "…but you *committed* to the Sprint, didn't you?". As a coach, this is the moment I step in to explain what "to commit" really means, and that you *want* to fail every now and then: succeeding every time is a failure mode all of its own.

The meaning of "to commit"

The Apple Thesaurus gives five meanings to the verb "to commit":

commit *verb*

1. he committed a murder: CARRY OUT, do, perpetrate, engage in, enact, execute, effect, accomplish; be responsible for; (informal) pull off.
2. she was committed to their care: ENTRUST, consign, assign, deliver, give, hand over, relinquish; formal commend.
3. they committed themselves to the project: PLEDGE, devote, apply, give, dedicate.
4. the judge committed him to prison: CONSIGN, send, deliver, confine.
5. her husband had her committed: HOSPITALIZE, confine, institutionalize, put away; certify.

See all the different nuances in that one word? The different interpretations are what trip us up. Certainly, one of the meanings is "to pledge", which is the one that is referred to when a team is held accountable, but what about "committing" a transaction in a database? Did the database just solemnly pledge to keep your data? No, that meaning is similar to the expression "committing something to memory", closer to the "carry out" or "entrust" meaning of the word.

Another meaning of "to commit", or specifically "committed" (thanks to Erik Gibson for helping clarify this one) is when you're at a point of no return. For instance, when you're running and you want to jump over a chair there is a point before which you can still slow down and stop before the chair, and after which your momentum forces you to carry on with the jump or else crash into that chair. At that point you are committed.

Commit what to whom?

In our context, the first true meaning of "to commit" is about fixing the scope of a Sprint. During Sprint Planning I the Scrum Team and the Product Owner select (or more formally: negotiate) the scope of the next Sprint. Since that scope should not change during the Sprint you're at a point of no return here. The moment of commitment fixes the scope for that Sprint, just like in a database transaction. Note that this is a commitment that applies to everyone: nobody should change the scope from that point onwards, whether they are Team members, Product Owners, managers, or stakeholders. Everybody is committed. The second true meaning of "to commit" is that the team pledges to each other that they will strive to reach the goal. Self-organization is one of the cornerstones of an Agile team, and self-organization builds on intrinsic motivation (or self-motivation). When team members commit, they are saying that they are intrinsically motivated to self-organize and contribute to the team. They commit primarily to the team process, not the result.

Commitment, confidence and the Fist of Five

Team members may not commit (in the “team pledge” sense) because there is something wrong with the team dynamics, or because that team member has some personal issue. Although important to look out for, I’ll not discuss it here: that is something for a soft-skills post. The other main reason – which I will discuss – deals with confidence. An unattainable goal kills intrinsic motivation, so it is important to know if a team thinks they have a reasonable chance to reach the Sprint goal. Ambitious is fine, unobtainable is not.

So how do you find out if a team has the necessary confidence to commit (in the “team pledge” sense)? A nice and easy tool is the Fist of Five. Before the chosen scope is fixed, the team is asked to “do a Fist of Five” if they think the Sprint goal can be achieved. Every team member then holds up a number of fingers corresponding to their confidence level:

- 1 finger: Not a snowball’s chance in hell!
- 2 fingers: I don’t think it’s possible…
- 3 fingers: There’s a 50/50 chance that we’ll make it.
- 4 fingers: We have a reasonable chance of making it (80%).
- 5 fingers: It’s a sure thing!

When you see only 1′s and 2′s, you can be pretty sure that the team will emotionally detach: they don't believe it can be done. When you see mostly 4′s, you're in a very good position.

But… should you be going for mostly 5′s, all the time? Doesn't that mean top-confidence, and a super-super team? Maybe, but it's not the best place to be: which leads into my final point, the value of potential failure.

Failure is good

If you've been around me for any stretch of time, you are almost certain to have heard a mantra I constantly repeat: Agile does not solve your problems, it just makes them painfully clear. The Scrum framework's most important – nay, main – goal is to surface problems, impediments, waste and other forms of organizational insanity, so that you have targets for improvement. Jeff Sutherland talks about "hyper-productivity" a lot, but this is one term on which I don't agree with him. I'm in the Lean camp on this one. We're mostly in a state of hyper-improductivity, and what Jeff calls "the hyper-productive state" is what I consider the normal state, after removing all the waste that blocks our natural capability for greatness (cue dramatic tear and handkerchief).

From my viewpoint you want to put just enough pressure on the system to induce controlled failure. Controlled failure means that you don't fail a Sprint outright, but for instance that you fail to achieve "Done" on one user story of the five the team committed to at the beginning of the Sprint. Those failures lead to improvements of all kinds: in the product, the process, the team, the organization… And that is why I think all 5′s, and never failing a Sprint is a bad thing. If you can't fail, you can't learn, it's that simple.

Conclusion

The word "commitment" is a word that is used incorrectly in many cases. It is not about accountability and holding a team in judgement after a Sprint. Commitment is about fixing scope, and a pledge team members make to each other to support self-organization. Finally, the use of commitment should not lead to "a sure thing", because otherwise you'll never achieve the greatest gift Scrum can give you: the chance to learn.

May all your days be filled with glorious failure!