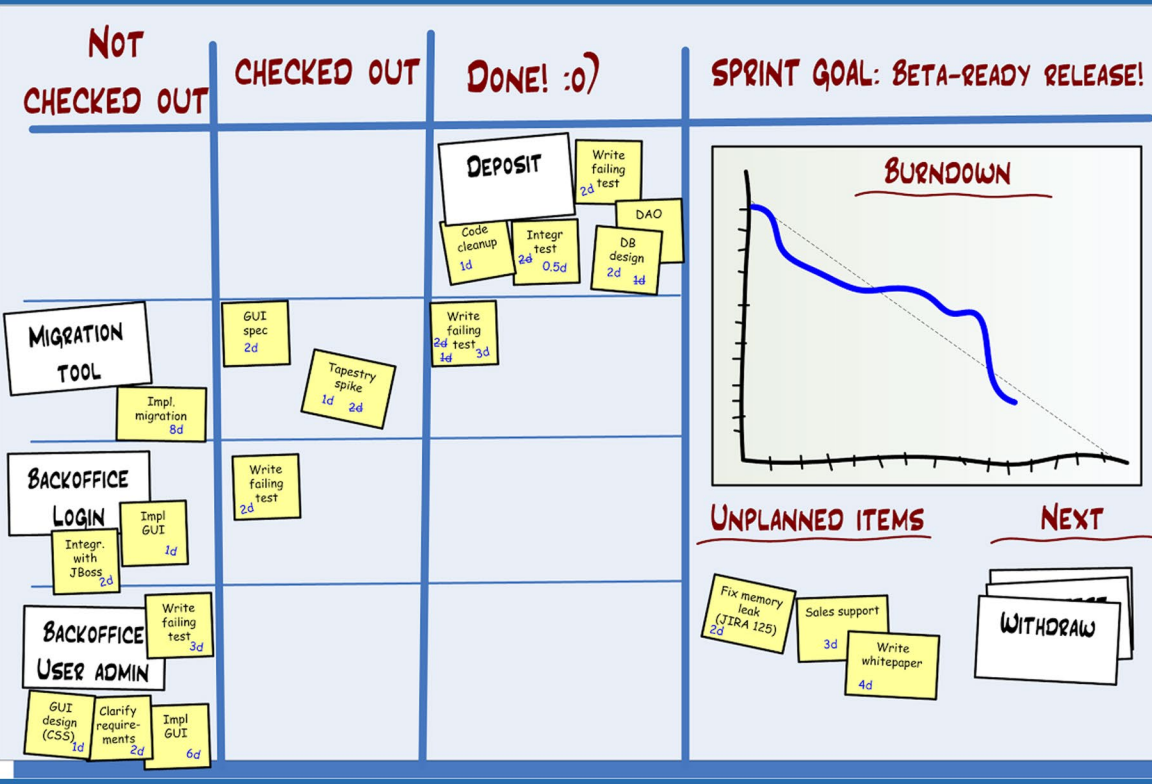
2nd Edition – Director's Cut



SCRUM AND XP FROM THE TRENCHES

How We Do Scrum

Henrik Kniberg

InfoQueue
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

SCRUM AND XP FROM THE TRENCHES

Published by C4Media, publisher of InfoQ.com.

Production Editor: Ana Ciobotaru
Copyeditor: Professor Laurie Nyveen
Cover and Interior Design: Dragos Balasoiu

Contents

Acknowledgements

The first draft of this paper took only one weekend to type, but it sure was an intensive weekend! 150% focus factor. :o)

Thanks to my wife Sophia and kids Dave and Jenny for putting up with my asocialness that weekend, and to Sophia's parents Eva and Jörgen for coming over to help take care of the family.

Thanks also to my colleagues at Crisp in Stockholm and people on the Scrum Users Yahoo group for proofreading and helping me improve the paper.

And, finally, thanks to all my readers who have provided a constant stream of useful feedback. I'm particularly glad to hear that this paper has sparked so many of you to give agile software development a shot!

Foreword by Jeff Sutherland

Teams need to know Scrum basics. How do you create and estimate a product backlog? How do you turn it into a sprint backlog? How do you manage a burn-down chart and calculate your team velocity? Henrik's book is a starter kit of basic practices that help teams move beyond trying to do Scrum to executing Scrum well.

Good Scrum execution is becoming more important for teams who want investment funding. As an agile coach for a venture-capital group, I help with their goal of investing only in agile companies that execute agile practices well. The senior partner of the group is asking all portfolio companies if they know the velocity of their teams. They have difficulty answering the question right now. Future investment opportunities will require that development teams understand their velocity of software production.

Why is this so important? If the teams do not know velocity, the product owner cannot create a product roadmap with credible release dates. Without dependable release dates, the company could fail and investors could lose their money!

This problem is faced by companies large and small, new or old, funded or not funded. At a recent discussion of Google's implementation of Scrum at a London conference, I asked an audience of 135 people how many were doing Scrum and 30 responded positively. I then asked them if they were doing iterative development by Nokia standards. Iterative development is fundamental to the Agile Manifesto – deliver working software early and often. After years of retrospectives with hundreds of Scrum teams, Nokia developed some basic requirements for iterative development:

Iterations must have fixed time boxes and be less than six weeks long.

Code at the end of the iteration must be tested by QA and be working properly.

Of the 30 people who said they were doing Scrum, only half said they were meeting the first principle of the Agile Manifesto by Nokia standards. I then asked them if they met the Nokia standards for Scrum:

A Scrum team must have a product owner and know who that person is.

The product owner must have a product backlog with estimates created by the team.

The team must have a burn-down chart and know their velocity.

There must be no one outside a team interfering with the team during a sprint.

Of 30 people doing Scrum, only 3 met the Nokia test for a Scrum team. These are the only teams that will receive future investment from my venture partners.

The value of Henrik's book is that if you follow practices he outlines, you will have a product backlog, estimates for the product backlog, a burn-down chart, and know your team velocity along with many other essential practices for a highly functional Scrum. You will meet the Nokia test for Scrum and be worthy of investment in your work. If you are a startup company, you might even receive funding by a venture capital group. You may be the future of software development and creator of the next generation of leading software products.

Jeff Sutherland,
Ph.D., co-creator of Scrum

Foreword by Mike Cohn

Both Scrum and extreme programming (XP) ask teams to complete some tangible piece of shippable work by the end of each iteration. These iterations are designed to be short and time-boxed. This focus on delivering working code in a short timeframe means that Scrum and XP teams don't have time for theories. They don't pursue drawing the perfect UML model in a case tool, writing the perfect requirements document, or writing code that will be able to accommodate all imaginable future changes. Instead, Scrum and XP teams focus on getting things done. These teams accept that they may mistakes along the way, but they also realize that the best way to find those mistakes is to stop thinking about the software at the theoretical level of analysis and design and to dive in, get their hands dirty, and start building the product.

It is this same focus on doing rather than theorizing that distinguishes this book. That Henrik Kniberg understands this is apparent right from the start. He doesn't offer a lengthy description of what Scrum is; he refers us to some simple websites for that. Instead, Henrik jumps right in and immediately begins describing how his team manages and works with their product backlog. From there he moves through all of the other elements and practices of a well-run agile project. No theorizing. No references. No footnotes. None are needed. Henrik's book isn't a philosophical explanation of why Scrum works or why you might want to try this or that. It is a description of how one well-running agile team works.

This is why the book's subtitle, "How We Do Scrum", is so apt. It may not be the way *you* do Scrum, it's how Henrik's team does Scrum. You may ask why you should care how another team does Scrum. You should care because we can all learn how to do Scrum better by hearing stories of how it has been done by others, especially those who are doing it well. There is not and never will be a list of Scrum best practices because team and project context trump all other considerations. Instead of best practices, what we need to know are good practices and the contexts in which they were successful. Read enough stories of successful teams and

how they did things and you'll be prepared for the obstacles thrown at you in your use of Scrum and XP.

Henrik provides a host of good practices along with the necessary context to help us learn better how to do Scrum and XP in the trenches of our own projects.

Mike Cohn,
Author of *Agile Estimating and Planning* and *User Stories Applied for Agile Software Development.*

Preface: Hey, Scrum worked!

Scrum worked! For us at least (meaning my current client in Stockholm, who's name I don't intend to mention here). Hope it will work for you too! Maybe this paper will help you along the way.

This is the first time I've seen a development methodology (sorry, Ken, a *framework*) work right off the book. Plug 'n' play. All of us are happy with it – developers, testers, managers. It helped us get out of a tough situation and has enabled us to maintain focus and momentum despite severe market turbulence and staff reductions.

I shouldn't say I was surprised but, well, I was. After I initially digested a few books on the topic, Scrum seemed good, but almost too good to be true (and we all know the saying "when something seems too good to be true..."). So I was justifiably a bit skeptical. But after doing Scrum for a year I'm so sufficiently impressed (and most people in my teams are as well) that I will probably continue using Scrum by default in new projects whenever there isn't a strong reason not to.

Preface – 2nd edition

Eight years have passed, and this book is still really popular. Wow! I never could have imagined the impact this little book would make! I still bump into teams, managers, coaches, and trainers all over the place that use it as their primary guide to agile software development.

But the thing is, I've learned lots since 2007! So the book really needs an update.

Since publishing the book, I've had the opportunity to work with many agile and lean thought leaders; some have even become like personal mentors to me. Special thanks to Jeff Sutherland, Mary and Tom Poppendieck, Jerry Weinberg, Alistair Cockburn, Kent Beck, Ron Jeffries, and Jeff Patton – I can't imagine a better group of advisors!

I've also had the chance to help a lot of companies implement these ideas in practice: companies in crisis as well as super-successful companies that want to get even better. All in all, it's been a pretty mind-blowing journey!

When I reread this old book, I'm surprised by how many things I still agree with. But there are also some pages that I'd like to rip out and say "What the *&€# was I thinking? Don't do it like that! There's a much better way!"

Since the book is a real-life case study, I can't change the story. What happened is what happened. But I can comment on it!

So that's what the second edition is – an annotated version of the original book. Like a director's cut. Think of it as me standing behind your shoulder as you read the book, commenting on stuff, cheering you on, with the occasional laugh and groan.

Here's how the annotations look. Everything else (except this preface) is the original book, unmodified, and these shaded boxes are my comments and reflections for the second edition.

I'll also bring in some examples from other companies, mostly Spotify (since that's where I've been spending most of my time lately) but also some other places.

Enjoy!

Henrik, March 2015

PART ONE

Intro

You are about to start using Scrum in your organization. Or perhaps you've been using Scrum for a few months. You've got the basics, you've read the books, maybe you've even taken your Scrum Master certification. Congratulations!

But yet you feel confused.

In Ken Schwaber's words, Scrum is not a methodology, it is a *framework*. What that means is that Scrum is not really going to tell you exactly what to do. Darn.

The good news is I am going to tell you exactly how I do Scrum, in painful excruciating detail. The bad news is, well, that this is only how I do Scrum. That doesn't mean *you* should do it exactly the same way. In fact, I may well do it in a different way if I encounter a different situation.

The strength and pain of Scrum is that you are forced to adapt it to your specific situation.

My current approach to Scrum is the result of one year's Scrum experimentation in a development team of approximately 40 people. The company was in a tough situation with high overtime, severe quality problems, constant firefighting, missed deadlines, etc. The company had decided to use Scrum but had not really completed the implementation, which was to be my task. To most people in the development team at that time, "Scrum" was just a strange buzzword they heard echo in the hallway from time to time, with no implication to their daily work.

Over a year's time we implemented Scrum through all layers in the company, tried different team sizes (3-12 people), different sprint lengths (2-6 weeks), different ways of defining "done", different formats for product backlogs and sprint backlogs (Excel, Jira, index cards), different testing strategies, different ways of doing demos, different ways of synchronizing multiple Scrum teams, etc. We also experimented with XP practices – different ways of doing continuous build, pair programming, test-driven development, etc., and how to combine this with Scrum.

This is a constant learning process so the story does not end here. I'm convinced that this company will keep learning (if they keep up the sprint retrospectives) and gain new insights on how to best implement Scrum in their particular context.

Disclaimer

This document does not claim to represent "the right way" to do Scrum! It only represents one way to do Scrum, the result of constant refinement over a year's time. You might even decide that we've got it all wrong.

Everything in this document reflects my own personal subjective opinions and is no means an official statement from Crisp or my current client. For this reason I have intentionally avoided mentioning any specific products or people.

Why I wrote this

When learning about Scrum I read the relevant Scrum and agile books, poured over sites and forums on Scrum, took Ken Schwaber's certification, peppered him with questions, and spent lots of time discussing with my colleagues. One of the most valuable sources of information, however, was actual war stories. The war stories turn principles and practices into well, How Do You Actually Do It. They also helped me identify (and sometimes avoid) typical Scrum newbie mistakes.

So, this is my chance to give something back. Here's my war story.

I hope that this paper will prompt some useful feedback from those of you in the same situation. Please enlighten me!

But what is Scrum?

Oh, sorry. You are completely new to Scrum or XP? In that case you might want to take a look at the following links:

- http://agilemanifesto.org/
- http://www.mountaingoatsoftware.com/scrum
- http://www.xprogramming.com/xpmag/whatisxp.htm

Check out the Scrum Guide as well. It's now the official description of Scrum, maintained by both Jeff Sutherland and Ken Schwaber. http://www.scrumguides.org

If you are too impatient to do that, feel free to just read on. Must of the Scrum jargon is explained as we go along so you might still find this interesting.

PART TWO

How we do product backlogs

The product backlog is the heart of Scrum. This is where it all starts.

Er, no, the product backlog isn't the starting point. A good product starts with a customer need and a vision for how to solve it. The product backlog is the result of refining that vision into concrete deliverables. The journey from vision to backlog can be quite complex, and lots of techniques have popped up to fill that gap. Things like user-story mapping (read Jeff Patton's book, it's great!), lean UX, impact mapping, and more. But don't use that as an excuse for big, up-front design though! Let the product backlog emerge iteratively, like everything else.

The product backlog is basically a prioritized list of requirements, or stories, or features, or whatevers. Things that the customer wants, described using the customer's terminology.

We call these *stories*, or sometimes just *backlog items*.

Our stories include the following fields:

- **ID** – a unique identification, just an auto-incremented number. This is to avoid losing track of stories when we rename them.
- **Name** – a short, descriptive name of the story. For example "See your own transaction history." Clear enough so that developers and the product owner understands approximately what we are talking about, and clear enough to distinguish it from other stories. Normally 2-10 words.
- **Importance** – the product owner's importance rating for this story. For example 10. Or 150. High = more important.

  I tend to avoid the term "priority" since priority 1 is typically considered the "highest" priority, which gets ugly if you later on decide that something else is even *more* important. What priority rating should *that* get?

  Priority 0? Priority -1?

- **Initial estimate** – the team's initial assessment of how much work is needed to implement this story compared to other stories. The unit is story points and usually corresponds roughly to "ideal man-days".

  Ask the team "if you can take the optimal number of people for this story (not too few and not too many, typically two), and lock yourselves into a room with lots of food and work completely

undisturbed, after how many days will you come out with a finished, demonstrable, tested, releasable implementation?" If the answer is "three guys locked into a room it will take approximately four days" then the initial estimate is 12 story points.

The important thing is not to get the absolute estimates correct (i.e. that a two-point story should take two days), the important thing is to get the relative estimates correct (i.e. that a two-point story should require about half as much work as a four-point story).

- **How to demo** – a high-level description of how this story will be demonstrated at the sprint demo. This is essentially a simple test spec. "Do this, then do that, then this should happen."

  If you practice TDD (test-driven development), this description can be used as pseudo-code for your acceptance-test code.

- **Notes** – any other info, clarifications, references to other sources of info, etc. Normally very brief.

PRODUCT BACKLOG (example)					
ID	**Name**	**Imp**	**Est**	**How to demo**	**Notes**
1	*Deposit*	*30*	*5*	*Log in, open deposit page, deposit €10, go to my balance page and check that it has increased by €10.*	*Need a UML sequence diagram. No need to worry about encryption for now.*
2	*See your own transaction history*	*10*	*8*	*Log in, click on "transactions". Do a deposit. Go back to transactions, check that the new deposit shows up.*	*Use paging to avoid large DB queries. Design similar to view users page.*

We experimented with lots of other fields, but at the end of the day, the six fields above were the only ones that we actually used sprint after sprint.

There are two things I almost always do differently now. First of all, there's no "importance" column. Instead, I just order the list. Just about all backlog management tools have drag-and-drop resorting (even Excel and Google Spreadsheets, if you learn the top-secret key combo). Easier and faster. Second of all, no man-days. Estimates are in story points or T-shirt sizes (S/M/L), or there are even no estimates at all. But more on that later.

We usually do this in an Excel document with sharing enabled (i.e. multiple users can edit simultaneously). Officially, the product owner owns this document, but we don't want to lock other users out. Many times, a developer wants to open the document to clarify something or change an estimate.

For the same reason, we don't place this document in the version-control repository; we place it on a shared drive instead. This turned out to be the simplest way to allow multiple simultaneous editors without causing lock or merge conflicts.

Almost all other artifacts, however, are placed in the version control repository.

Excel, huh? Wow, those were the days. I would never consider using Excel for backlog management today, unless it's a cloudy version. The product backlog needs to live in a shared online document that anyone can access and edit easily and concurrently. Either one of the gazillion backlog management tools available (Trello and LeanKit and Jira are popular) or a Google Spreadsheet (very practical!).

Additional story fields

Sometimes we use additional fields in the product backlog, mostly as a convenience for the product owner to help him sort out his priorities:

- **Track** – a rough categorization of this story, for example "back office" or "optimization". That way the product owner can easily filter out all "optimization" items and set their priority to low, etc.
- **Components** - Usually realized as "checkboxes" in the Excel document, for example "database, server, client". Here the team or product owner can identify which technical components will be

involved in implementing this story. This is useful when you have multiple Scrum teams – for example, a back-office team and a client team, and want to make it easier for each team to decide which stories to take on.

- **Requestor** – the product owner may want to keep track of which customer or stakeholder originally requested the item, in order to give him feedback on the progress.
- **Bug tracking ID** – if you have a separate bug tracking system, like we do with Jira, it is useful to keep track of any direct correspondence between a story and one or more reported bugs.

How we keep the product backlog at a business level

If the product owner has a technical background, he might add stories such as "Add indexes to the Events table". Why does he want this? The real underlying goal is probably something like "speed up the search event form in the back office".

It may turn out that indexes weren't the bottleneck causing the form to be slow. It may be something completely different. The team is normally better suited to figure out *how* to solve something, so the product owner should focus on business goals.

When I see technically oriented stories like this, I normally ask the product owner a series of "but *why*?" questions until we find the underlying goal. Then we rephrase the story in terms of the underlying goal ("speed up the search event form in the back office"). The original technical description ends up as a note ("Indexing the event table might solve this").

There's an old and well-established template for this: "As X, I want Y, so that Z." For example "As buyer, I want to save my shopping cart, so that I can continue shopping tomorrow." I'm really surprised I hadn't heard of that in 2007! Would have been very convenient. Yes, there are more elaborate templates available nowadays, but this simple one is a good starting point, especially for teams that are new to the whole agile thing. The template forces you to ask the right types of questions, and reduces the risk of getting stuck in techy details.

PART THREE

How we prepare for sprint planning

OK, sprint planning day is coming at us quickly. One lesson we learn over and over is: Make sure the product backlog is in shipshape *before* the sprint planning meeting.

Amen to that! I've seen lots of sprint planning meetings blow up because the product backlog is a mess. You know the saying "shit in = shit out"? Exactly.

And what does *that* mean? That all stories have to be perfectly well defined? That all estimates have to be correct? That all priorities must be fixed? No, no, and no! All it means is:

- The product backlog should exist! (Imagine that?)
- There should be *one* product backlog and *one* product owner (per product that is).
- All important items should have importance ratings assigned to them, *different* importance ratings.
    - Actually, it is OK if lower-importance items all have the same value, since they probably won't be brought up during the sprint planning meeting anyway.
    - Any story that the product owner believes has a remote possibility of being included in the next sprint should have a unique importance level.
    - The importance rating is only used to sort the items by importance. So if Item A has importance 20 and Item B has importance 100, that simply means B is more important than A. It does *not* mean that B is five times more important than A. If B had importance rating 21 it will still mean the exact same thing!
    - It is useful to leave gaps in the number sequence in case an item C comes up that is more important than A but less important than B. Of course you could use an importance rating of 20.5 for C, but that gets ugly, so we leave gaps instead!

Bah, just sort the list and you don't need to fiddle with importance ratings.

- The product owner should *understand* each story (normally he is the author, but in some cases other people add requests, which the product owner can prioritize). He does not need to know exactly

what needs to be implemented, but he should understand why the story is there.

Note: People other than the product owner may add stories to the product backlog. But they may not assign an importance level – that is the product owner's sole right. They may not add time estimates either, that is the team's sole right.

Other approaches that we've tried or evaluated:

- Using Jira (our bug tracking system) to house the product backlog. Most of our product owners find it too click-intensive, however. Excel is nice and easy to directly manipulate. You can easily color code, rearrange items, add new columns on an ad hoc basis, add notes, import and export data, etc.

Same with Google Spreadsheets. And it's in the cloud. Multiuser, concurrent editing. Just sayin'.

Using an agile process-support tool such as VersionOne, ScrumWorks, XPlanner, etc. We haven't gotten around to testing any of those but we probably will.

PART FOUR

How we do sprint planning

Sprint planning is a critical meeting, probably the most important event in Scrum (in my subjective opinion of course). A badly executed sprint planning meeting can mess up a whole sprint.

Important? Yes. Most important event in Scrum? No! Retrospectives are waaay more important! Because well-functioning retrospectives will help fix other things that are broken. Sprint planning tends to be pretty trivial as long as the other things are in place (a good product backlog, an engaged product owner and team, etc.). Also, sprinting isn't the only way to be agile – a lot of teams use kanban instead. I even wrote a mini-book about it: "Kanban and Scrum: Making the Most of Both". http://www.infoq.com/minibooks/kanban-scrum-minibook

The purpose of the sprint planning meeting is to give the team enough information to be able to work in undisturbed peace for a few weeks, and to give the product owner enough confidence to let them do so.

OK, that was fuzzy. The concrete output of the sprint planning meeting is:

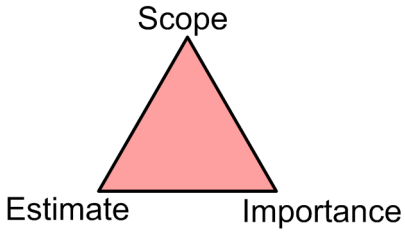
- A sprint goal
- A list of team members (and their commitment levels, if not 100%)
- A sprint backlog (= a list of stories included in the sprint)
- A defined sprint demo date
- A defined time and place for the daily scrum

Why the product owner has to attend

Sometimes, product owners are reluctant to spend hours with the team doing sprint planning. "Guys, I've already listed what I want. I don't have time to be at your planning meeting." That is a pretty serious problem.

The reason why the whole team *and* the product owner have to be at the sprint planning meeting is because each story contains three variables that are highly dependent on each other.

Scope and importance are set by the product owner. Estimate is set by the team. During a sprint planning meeting, these three variables are fine-



tuned continuously through face-to-face dialog between the team and the product owner.

Normally, the product owner starts the meeting by summarizing his goal for the sprint and the most important stories. Next, the team goes through and time-estimates each story, starting with the most important one. As they do this, they will come up with important scope questions – "Does this 'delete user' story include going through each pending transaction for that user and canceling it?" In some cases, the answers will be surprising to the team, prompting them to change their estimates.

In some cases, the time estimate for a story won't be what the product owner expected. This may prompt him to change the importance of the story. Or change the scope of the story, which in turn will cause the team to re-estimate, etc., etc.

This type of direct collaboration is fundamental to Scrum and, in fact, all agile software development.

What if the product owner still insists that he doesn't have time to join sprint planning meetings? I usually try one of the following strategies, in the given order:

- Try to help the product owner understand why his direct participation is crucial and hope that he changes his mind.
- Try to get someone in the team to volunteer as product-owner proxy during the meeting. Tell the product owner "Since you can't join our meeting, we will let Jeff here represent you as a proxy. He will be fully empowered to change priorities and scope of stories on your behalf during the meeting. I suggest you synchronize with him as much as possible before the meeting. If you don't like Jeff to be proxy please suggest someone else, as long as that person can join us for the full length of the meeting."
- Try to convince the management team to assign a new product owner.
- Postpone the sprint launch until the product owner finds time to join the meeting. In the meantime, refuse to commit to any deliveries. Let the team spend each day doing whatever they feel is most important that day.

Good stuff in this section! <patting myself on the back>

Just one thing – I strongly recommend separating backlog refinement (estimation, story splitting, etc.) into a separate meeting so that sprint planning can be more focused. Product owner participation is still crucial though, in both meetings.

Why quality is not negotiable

In the triangle above I intentionally avoided a fourth variable: *quality*.

I try to distinguish between *internal quality* and *external quality*:

- *External quality* is what is perceived by the users of the system. A slow and non-intuitive user interface is an example of poor external quality.
- *Internal quality* refers to issues that usually aren't visible to the user, but which have a profound effect on the maintainability of the system. Things like system design consistency, test coverage, code readability, refactoring, etc.

Generally speaking, a system with high internal quality can still have a low external quality. But a system with low internal quality will rarely have a high external quality. It is hard to build something nice on top of a rotten fundament.

I treat external quality as part of scope. In some cases, it might make perfect business sense to release a version of the system that has a clumsy and slow user interface, and then release a cleaned-up version later. I leave that tradeoff to the product owner, since he is responsible for determining scope.

Internal quality, however, is not up for discussion. It is the team's responsibility to maintain the system's quality under all circumstances and this is simply not negotiable. Ever.

(Well, OK, almost never.)

So how do we tell the difference between internal quality issues and external quality issues?

Let's say the product owner says "OK guys, I respect your time estimate of six story points, but I'm sure you can do some kind of quick fix for this in half the time if you just put your mind to it."

Aha! He is trying to use internal quality as a variable. How do I know? Because he wants us to reduce the estimate of the story without "paying the price" of reducing the scope. The phrase "quick fix" should trigger an alarm in your head….

And why don't we allow this?

My experience is that sacrificing internal quality is almost always a terrible, terrible idea. The time saved is far outweighed by the cost in both short and long terms. Once a code base is permitted to start deteriorating it is very hard to put the quality back in later.

Instead, I try to steer the discussion towards scope. "Since it is important for you to get this feature out early, can we reduce the scope so that it will be quicker to implement? Perhaps we can simplify the error handling and make 'Advanced error handling' a separate story that we save for the future? Or can we reduce the priority of other stories so that we can focus on this one?"

Once the product owner has learned that internal quality isn't negotiable, he usually gets quite good at manipulating the other variables instead.

In principle, yes. But in practice I tend to be more pragmatic nowadays. Sometimes it makes perfect business sense to sacrifice quality in the short term – for example, because we have this super-important trade show happening after next sprint or because we just need a prototype to validate a hypothesis about user behavior. But in those cases, the product owner needs to make clear why we are doing this, and to commit to letting the team pay back the technical debt in the near future (sometimes the team will add a "clean up" story to the product backlog, as a reminder). High internal quality should be the norm, and exceptions should be treated as exceptional.

Sprint planning meetings that drag on and on…

The most difficult thing about sprint planning meetings is that:

1) People don't think they will take so long time…

2) …but they do!

No, they don't! Not if you do backlog refinement in a separate meeting. Many teams I've seen meet weekly for one hour for backlog refinement, so that sprint planning can be focused on, well, sprint planning! This also gives the product owner more chances to discuss and improve the product backlog *before* the sprint planning meeting, which in turn makes the meeting shorter. Guideline: a sprint planning meeting should normally not take more than one hour per week of sprint length (considerably less for experienced teams), so three hours or less for a three-week sprint.

Everything in Scrum is time-boxed. I love that one, simple, consistent rule. We try to stick to it.

So what do we do when the time-boxed sprint planning meeting is nearing the end and there is no sign of a sprint goal or sprint backlog? Do we just cut it short? Or do we extend it for an hour? Or do we end the meeting and continue the next day?

This happens over and over, especially for new teams. So what do you do? I don't know. But what do we do? Oh, um, well, usually I brutally cut the meeting short. End it. Let the sprint suffer. More specifically, I tell the team and product owner "So, this meeting ends in 10 minutes. We don't have much of a sprint plan really. Should we make do with what we have, or should we schedule another four-hour sprint planning meeting tomorrow at 8 a.m.?" You can guess what they will answer. :o)

I've tried letting the meeting drag on. That usually doesn't accomplish anything, because people are tired. If they haven't produced a decent sprint plan in two to eight hours (or however long your time box is), they probably won't manage it given another hour. The next option is actually quite OK: to schedule a new meeting next day. Except that people usually are impatient and want to get going with the sprint, and not spend another bunch of hours planning.

So I cut it short. And yes, the sprint suffers. The upside, however, is that the team has learned a very valuable lesson, and the next sprint planning meeting will be much more efficient. In addition, people will be less resistant when you propose a meeting length that they previously would have thought was too long.

Learn to keep to your time-boxes, learn to set realistic time-box lengths. That applies both to meeting lengths and sprint lengths.

I once met a team that said "We tried Scrum, hated it, won't do it again!" I asked why, and they said "Too much time in meetings! We never got anything done." I asked which meeting took the most time, and they said sprint planning. I asked how long it took, and they said "Two or three days!". Two or three DAYS of planning for each sprint?! No wonder they hated it! They missed the time-boxing rule: decide up front how much time you are willing to invest, and then stick to it! Scrum is like any other tool – you can use a hammer to build something or to smash your thumb. Either way, don't blame the tool.

Sprint-planning-meeting agenda

Having some kind of preliminary schedule for the sprint planning meeting will reduce the risk of breaking the timebox.

Here's an example of a typical schedule for us.

Sprint planning meeting: 13:00-17:00 (10-minute break each hour)

- **13:00-13:30 –** Product owner goes through sprint goal and summarizes product backlog. Demo place, date, and time is set.
- **13:30-15:00** – Team time-estimates, and breaks down items as necessary. Product owner updates importance ratings as necessary. Items are clarified. "How to demo" is filled in for all high-importance items.
- **15:00-16:00** – Team selects stories to be included in sprint. Do velocity calculations as a reality check.
- **16:00-17:00** – Select time and place for daily scrum (if different from last sprint). Further breakdown of stories into tasks.

That bit between 13:30 and 15:00, that's product backlog refinement (I used to call it "backlog grooming" but learned that grooming means Bad Things in some cultures). Cut that out to a separate meeting and, ta-da, you get shorter and sweeter sprint planning meetings. OK, some minor adjustments may be needed, but most backlog refinement should be done *before* sprint planning.

The schedule is by no means strictly enforced. The Scrum master may lengthen or shorten the sub-time-boxes as necessary as the meeting progresses.

Defining the sprint length

One of the outputs of the sprint planning meeting is a defined sprint demo date. That means you have to decide on a sprint length.

So what is a good sprint length?

Well, short sprints are good. They allow the company to be "agile", i.e. change direction often. Short sprints = short feedback cycle = more frequent deliveries = more frequent customer feedback = less time spent running in the wrong direction = learn and improve faster, etc.

But then, long sprints are good too. The team gets more time to build up momentum, they get more room to recover from problems and still make the sprint goal, you get less overhead in terms of sprint planning meetings, demos, etc.

Generally speaking, product owners like short sprints and developers like long sprints. So sprint length is a compromise. We experimented a lot with this and came up with our favorite length: three weeks. Most of our teams (but not all) do three-week sprints. Short enough to give us adequate corporate agility, long enough for the team to achieve flow and recover from problems that pop up in the sprint.

One thing we have concluded is: *do* experiment with sprint lengths initially. Don't waste too much time *analyzing*, just select a decent length and give it a shot for a sprint or two, then change length.

However, once you have decided what length you like best, *stick to it* for an extended period of time. After a few months of experimentation, we found that three weeks was good. So we do three-week sprints, period.

Sometimes it will feel slightly too long, sometimes slightly too short. But by keeping the same length, this becomes like a corporate heartbeat which everyone comfortably settles into. There is no argument about release dates and such because everyone knows that every three weeks there is a release, period.

Most Scrum teams I meet (almost all, in fact) end up doing two-week or three-week sprints. One week is almost always too short ("We barely got started with the sprint, and now we're already planning the demo! Stressful! We never manage to get up to speed and enjoy development flow!"). And four weeks is almost always too long ("Our planning meetings are torture, and our sprints keep getting interrupted!"). Just an observation.

Defining the sprint goal

It happens almost every time. At some point during the sprint planning meeting, I ask "So what is the goal of this sprint?" and everybody just stares blankly back at me and the product owner furrows his brow and scratches his chin.

For some reason, it is *hard* to come up with a sprint goal. But I have found that it really pays to squeeze one out. Better a half-crappy goal than none at all. The goal could be "make more money" or "complete the three top-priority stories" or "impress the CEO" or "make the system good enough to deploy to a live beta group" or "add basic back office support" or whatever. The important thing is that it should be in business terms, not technical terms. This means in terms that people outside the team can understand.

The sprint goal should answer the fundamental question "*Why* are we doing this sprint? Why don't we all just go on vacation instead?" In fact, one way to wheedle a sprint goal out of the product owner is to literally ask that question.

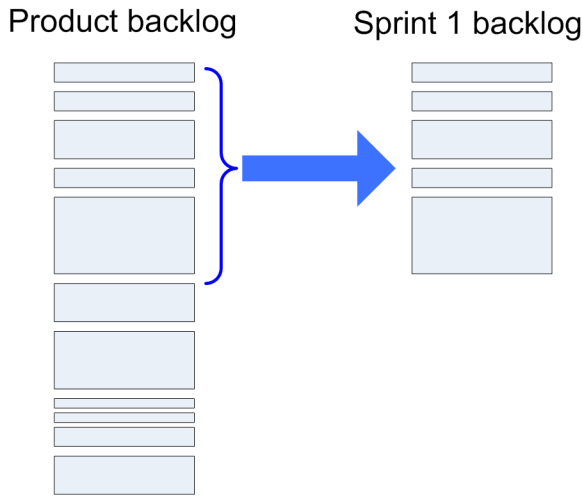
The goal should be something that has not already been achieved. "Impress the CEO" might be a fine goal, but not if he is already impressed by the system as it stands now. In that case, everybody could go home and the sprint goal will still be achieved.

The sprint goal may seem rather silly and contrived during the sprint planning, but it often comes to use in mid-sprint, when people are starting to get confused about what they should be doing. If you have several Scrum teams (like we do) working on different products, it is very useful to be able to list the sprint goals of all teams on a single wiki page (or whatever) and put them up on a prominent space so that everybody in the company (not only top-level management) knows what the company is doing – and why!

OK, even the Scrum Guide agrees with this and says that all sprints should have a sprint goal. But I find that it's not important to have a goal at a sprint level; it can be just as fine to have a higher-level goal that covers several sprints, or the next release cycle. Just make sure a sprint is *something* more than just "let's knock down a bunch of stories", or you may find the team coming down with a serious case of Boring.

Deciding which stories to include in the sprint

One of the main activities of the sprint planning meeting is to decide which stories to include in the sprint. More specifically, which stories from the product backlog to copy to the sprint backlog.



Look at the picture above. Each rectangle represents a story, sorted by importance. The most important story is at the top of the list. The size of each rectangle represents the size of that story (i.e. time estimate in story points). The height of the blue brace represents the team's *estimated velocity*, i.e. how many story points the team believes they can complete during next sprint.

The sprint backlog to the right is a snapshot of stories from the product backlog. It represents the list of stories that the team will commit to for this sprint.

The *team* decides how many stories to include in the sprint. Not the product owner or anybody else.

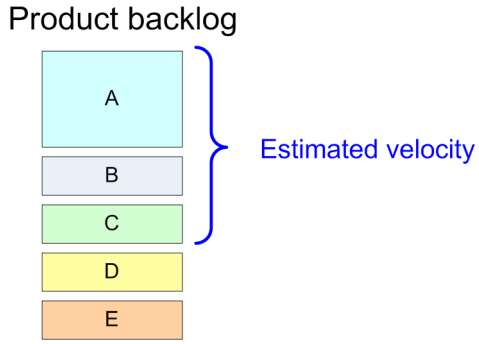
This raises two questions:

1. How does the team decide which stories to include in the sprint?
2. How can the product owner affect their decision?

I'll start with the second question.

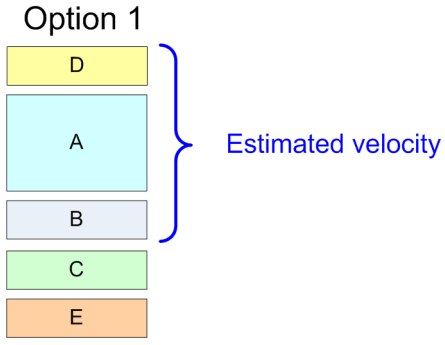
How can product owner affect which stories make it to the sprint?

Let's say we have the following situation during a sprint planning meeting.

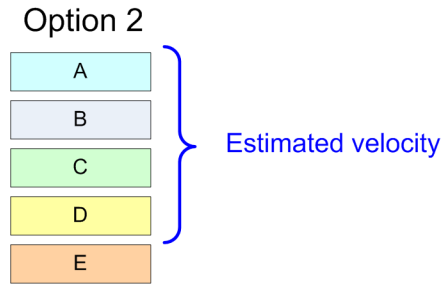


The product owner is disappointed that story D won't be included in the sprint. What are his options during the sprint planning meeting?

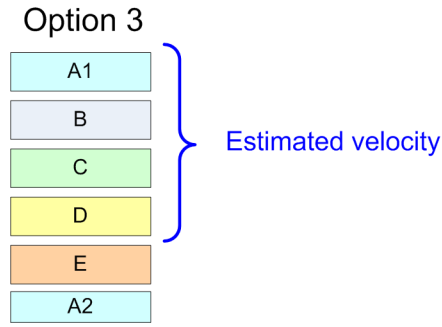
One option is to reprioritize. If he gives item D the highest level of importance, the team will be obliged to add that to the sprint first (in this case bumping out story C).



The second option is to change the scope – reduce the scope of story A until the team believes that story D will fit into the sprint.



The third option is to split a story. The product owner might decide that there are some aspects of story A that really aren't that important, so he splits A into two stories A1 and A2 with different importance levels.



As you see, although the product owner normally can't control the estimated velocity, there are many ways in which he can influence which stories make it into the sprint.

How does the team decide which stories to include in the sprint?

We use two techniques for this:

1. Gut feel
2. Velocity calculations

Estimating using gut feel

Scrum master (pointing to the most important item in the product backlog): Hey guys, can we finish story A in this sprint?

Lisa: Duh. Of course we can. We have three weeks, and that's a pretty trivial feature.

Scrum master (points to the second most important item): OK, what if we add story B as well?

Tom and Lisa (in unison): Still a no-brainer.

Scrum master: OK, what about story A and B and C then?

Sam (to product owner): Does story C include advanced error handling?

Product owner: No, you can skip that for now. Just implement basic error handling.

Sam: Then C should be fine as well.

Scrum master: OK, what if we add story D?

Lisa: Hmm....

Tom: I think we could do it.

Scrum master: 90% confident? 50%?

Lisa and Tom: Pretty much 90%.

Scrum master: OK, D is in then. What if we add story E?

Sam: Maybe.

Scrum master: 90%? 50%?

Sam: I'd say closer to 50%.

Lisa: I'm doubtful.

Scrum master: OK, then we leave it out. We'll commit to A, B, C, and D. We will of course finish E if we can, but nobody should count on it so we'll leave it out of the sprint plan. How about that?

Everybody: OK!

Gut feel works pretty well for small teams and short sprints.

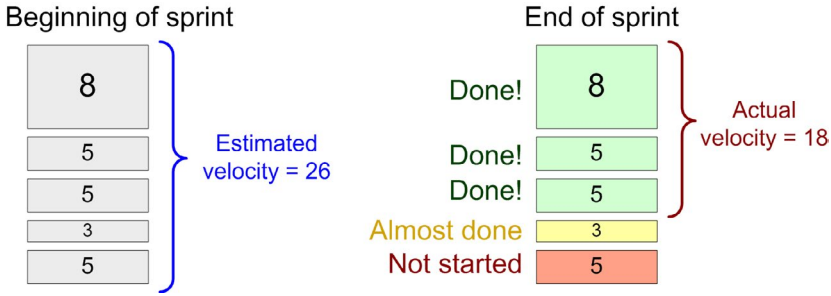
Estimating using velocity calculations

This technique involves two steps:

1. Decide *estimated velocity.*
2. Calculate how many stories you can add without exceeding estimated velocity.

Velocity is a measurement of "amount of work done", where each item is weighted in terms of its initial estimate.

The picture below shows an example of *estimated velocity* at the beginning of a sprint and *actual velocity* at the end of that sprint. Each rectangle is a story, and the number inside is the initial estimate of that story.



Note that the actual velocity is based on the *initial* estimates of each story. Any updates to the story time estimates done during the sprint are ignored.

I can hear your objection already: "How is this useful? A high or low velocity may depend on a whole bunch of factors! Dimwitted programmers, incorrect initial estimates, scope creep, unplanned disturbances during sprint, etc.!"

I agree, it is a crude number. But it is still a useful number, especially when compared to nothing at all. It gives you some hard facts. "Regardless of the reasons, here is the approximate difference between how much we thought we would get done and how much we actually got done."

What about a story that got *almost* completed during a sprint? Why don't we get partial points for that in our actual velocity? Well, this is to stress the fact the Scrum (and in fact agile software development and lean manufacturing in general) is all about getting stuff completely, shippably, done! The value of stuff half-done is zero (may in fact be negative). Pick up Donald Reinertsen's *Managing the Design Factory* or one of Poppendieck's books for more on that.

So through what arcane magic do we estimate velocity?

One very simple way to estimate velocity is to look at the team's history. What was their velocity during the past few sprints? Then assume that the velocity will be roughly the same next sprint.

This technique is known as *yesterday's weather*. It is only feasible for teams that have done a few sprints already (so statistics are available) and will do the next sprint in pretty much the same way, with the same team size and same working conditions etc. This is of course not always the case.

A more sophisticated variant is to do a simple resource calculation. Let's say we are planning a three-week sprint (15 work days) with a four-person team. Lisa will be on vacation for two days. Dave is only 50% available and will be on vacation for one day. Putting all this together...

…gives us 50 available man-days for this sprint.

Warning: here's the part I really hate. I'd like to rip out the next few pages! But go ahead and read it if you are curious, and I'll explain why afterwards.

Is that our estimated velocity? No! Because our unit of estimation is *story points* which, in our case, corresponds roughly to "ideal man-days". An ideal man-day is a perfectly effective, undisturbed day, which is rare. Furthermore, we have to take into account things such as unplanned work being added to the sprint, people being sick, etc.

So our estimated velocity will certainly be less than 50. But how much less? We use the term "focus factor" for this.

THIS SPRINT'S ESTIMATED VELOCITY:

(AVAILABLE MAN-DAYS) x (FOCUS FACTOR) = (ESTIMATED VELOCITY)

Focus factor is an estimate of how focused the team is. A low focus factor may mean that the team expects to have many disturbances or expects their own time estimates to be optimistic.

Bla bla bla. Bunch of math mumbo jumbo. Just use yesterday's weather (or gut feel if you don't have data) and ignore this focus factor nonsense.

The best way to determine a reasonable focus factor is to look at the last sprint (or even better, average the last few sprints).

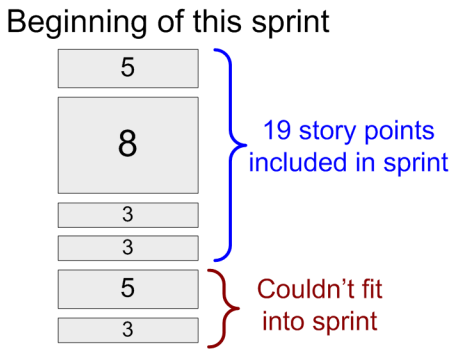
LAST SPRINT'S FOCUS FACTOR:

$$(\text{FOCUS FACTOR}) = \frac{(\text{ACTUAL VELOCITY})}{(\text{AVAILABLE MAN-DAYS})}$$

Actual velocity is the sum of the initial estimates of all stories that were completed last sprint.

Let's say last sprint completed 18 story points using a three-person team consisting of Tom, Lisa, and Sam working three weeks for a total of 45 man-days. And now we are trying to figure out our estimated velocity for the upcoming sprint. To complicate things, a new guy, Dave, is joining the team for that sprint. Taking vacations and stuff into account we have 50 man-days next sprint.

So our estimated velocity for the upcoming sprint is 20 story points. That means the team should add stories to the sprint until it adds up to approximately 20.



In this case, the team may choose the top four stories for a total of 19 story points, or the top five stories for a total of 24 story points. Let's say they choose four stories, since that came closest to 20 story points. When in doubt, choose fewer stories.

Since these four stories add up to 19 story points, their final estimated velocity for this sprint is 19.

Yesterday's weather is a handy technique but use it with a dose of common sense. If last sprint was an unusually bad sprint because most of the team was sick for a week, then it may be safe to assume that you won't be that unlucky again and you could estimate a higher focus factor next sprint. If the team has recently installed a new lightning-fast continuous-build system, you could probably increase focus factor due to that as well. If a new person is joining this sprint you need to decrease focus factor to take his training into account. Etc.

Whenever possible, look back several sprints and average out the numbers to get more reliable estimates.

What if the team is completely new so you don't have any statistics? Look at the focus factor of other teams under similar circumstances.

What if you have no other teams to look at? Guess a focus factor. The good news is that your guess will only apply to the first sprint. After that, you will have statistics and can continuously measure and improve your focus factor and estimated velocity.

The default focus factor I use for new teams is usually 70%, since that is where most of our other teams have ended up over time.

Which estimating technique do we use?

I mentioned several techniques above: gut feeling, velocity calculation based on yesterday's weather, and velocity calculation based on available man-days and estimated focus factor.

So which technique do we use?

We usually combine all these techniques to a certain degree. Doesn't really take long.

We look at focus factor and actual velocity from last sprint. We look at our total resource availability this sprint and estimate a focus factor. We discuss any differences between these two focus factors and make adjustments as necessary.

OK, that's the end of the painful section. I never use focus factor any more because it takes time, gives a false sense of accuracy, and forces you to estimate stories in ideal man-days.

Also, focus factor carries the assumption that more people = higher velocity. Sometimes that's true, but sometimes not. If we add a new person to the team, the velocity will usually go *down* the first sprint or two, since people spend time onboarding the new person. If a team gets too big (like 10+ people), velocity definitely goes down. Also, the phrase "focus factor" implies that a value less than 100% means the team is *unfocused*, which sends a very misleading message to management.

So skip all this focus factor and man-days stuff. Just look at how much you got done the last few sprints, by counting story points, or even just counting the number of stories if you don't have estimates at all. Then grab roughly that many stories for this sprint. If you have planned disruptions in the sprint (like two people gone for a course) then remove a few stories until it feels just about right. The less historical data you have, the more you need to rely on gut feel.

Do this and your sprint planning meetings will be shorter, more effective, and more fun. And, counter-intuitively, your plans will probably end up being more accurate.

Once we have a preliminary list of stories to be included in the sprint, I do a "gut feeling" check. I ask the team to ignore the numbers for a moment

and just think about if this *feels* like a realistic chunk to bite off for a sprint. If it feels like too much, we remove a story or two. And vice versa.

At the end of the day, the goal is simply to decide which stories to include in the sprint. Focus factor, resource availability, and estimated velocity are just a means to achieve that end.

Why we use index cards

Most of sprint planning meeting is spent dealing with stories in the product backlog. Estimating them, reprioritizing them, clarifying them, breaking them down, etc.

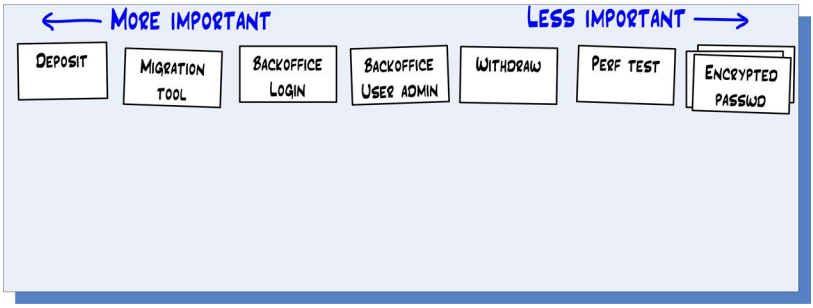
How do we do this in practice?

Well, by default, the teams used to fire up the projector, show the Excel-based backlog, and one guy (typically the product owner or Scrum master) would take the keyboard, mumble through each story and invite discussion. As the team and product owner discussed priorities and details, the guy at the keyboard would update the story directly in Excel.

Sounds good? Well it isn't. It usually sucks. And what's worse, the team normally doesn't notice that it sucks until they reach the end of the meeting and realize that they still haven't managed to go through the list of stories!

Oh, the pain….

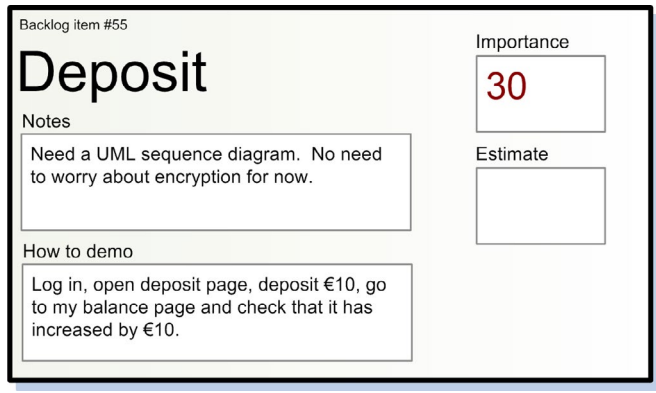
A solution that works much better is to create index cards and put them up on the wall (or a large table).



This is a superior user interface compared to computer and projector, because:

- People stand up and walk around => they stay awake and alert longer.
- Everybody feels more personally involved (rather than just the guy with the keyboard).
- Multiple stories can be edited simultaneously.
- Reprioritizing is trivial – just move the index cards around.
- After the meeting, the index cards can be carried right off to the team room and be used as a wall-based task board (see pg. 45 "How we do sprint backlogs").

You can either write them by hand or (like we usually do) use a simple script to generate printable index cards directly from the product backlog.



P.S. The script is available on my blog at http://blog.crisp.se/henrikkniberg.

The easiest way to find the script is to google "index card generator". Can't believe that old hack is still going strong! Some kind people have helped port it to Google Spreadsheets as well. Any decent backlog management tool will have a print function like this. Experiment with different tools and find what works best in your context. Just make sure you are adapting the tool to your process, and not vice versa.

Important: After the sprint planning meeting, our Scrum master manually updates the Excel-based product backlog with respect to any changes that were made to the physical story index cards. Yes, this is a slight administrative hassle but we find this perfectly acceptable

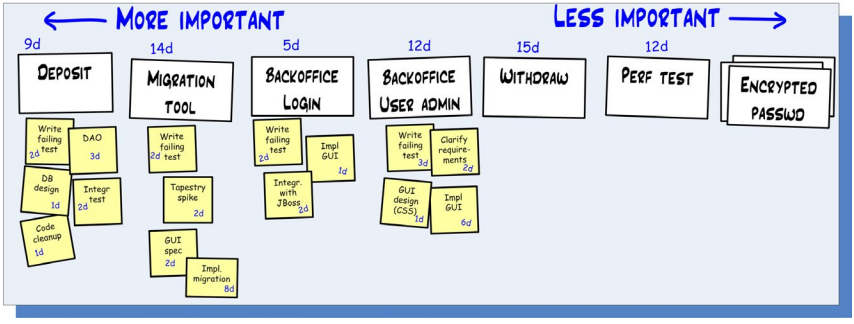
considering how much more efficient the sprint planning meeting is with physical index cards.

One note about the "Importance" field.... This is the importance as specified in the Excel-based product backlog at the time of printing. Having it on the card makes it easy to sort the cards physically by importance (normally we place more important items to the left, and less important items to the right). However, once the cards are up on the wall, you can ignore the importance rating and instead use the physical ordering on the wall to indicate relative importance ratings. If the product owner swaps two items, don't waste time updating the importance rating on the paper. Just make sure you update the importance ratings in the Excel-based product backlog after the meeting.

Or just skip importance ratings. Oops, I already said that a few times. Am I repeating myself? Am I repeating myself?

Time estimates are usually easier to do (and more accurate) if a story is broken down into tasks. Actually, we use the term "activity" because the word "task" means something *completely* different in Swedish. :o) This is also nice and easy to do with our index cards. You can have the team divide into pairs and break down one story each, in parallel.

Physically, we do this by adding little Post-it notes under each story, each Post-it reflecting one task within that story.



We don't update the Excel-based product backlog with respect to our task breakdowns, for two reasons:

The task breakdown is usually quite volatile, i.e. they are frequently changed and refined during the sprint, so it is too much of a hassle to keep the product-backlog Excel synchronized.

The product owner doesn't need to be involved at this level of detail anyway.

Just as with the story index cards, the task breakdown Post-its can be directly reused in the sprint backlog (see pg. 45 "How we do sprint backlogs").

Definition of "done"

It is important that the product owner and the team agree on a clear definition of "done".

VERY important!

Is a story complete when all code is checked in? Or is it complete only when it has been deployed to a test environment and verified by an integration test team? Whenever possible, we use the done definition "ready to deploy to production" but sometimes we have to make do with the done definition "deployed on test server and ready for acceptance test".

In the beginning, we used to have detailed checklists for this. Now, we often just say "a story is done when the tester in the Scrum team says so". It is then up to the tester to make sure that product owner's intent is

understood by the team, and that the item is "done" enough to pass the accepted definition of done.

OK, that's a bit lame. A concrete checklist is more useful – just make sure it isn't too long. Treat it as a default, not Holy Scripture. Focus on the things that people tend to forget (like "update the release notes" or "no added technical debt" or "get real user feedback").

We've come to realize that all stories cannot be treated the same. A story named "Query users form" will be treated very differently from a story named "Operations manual". In the latter case, the definition of "done" might simply mean "accepted by the operations team". That is why common sense is often better than formal checklists.

If you often run into confusion about the definition of done (which we did in the beginning) you should probably have a "definition of done" field on each individual story.

Time estimating using planning poker

Estimating is a team activity – every team member is usually involved in estimating every story. Why?

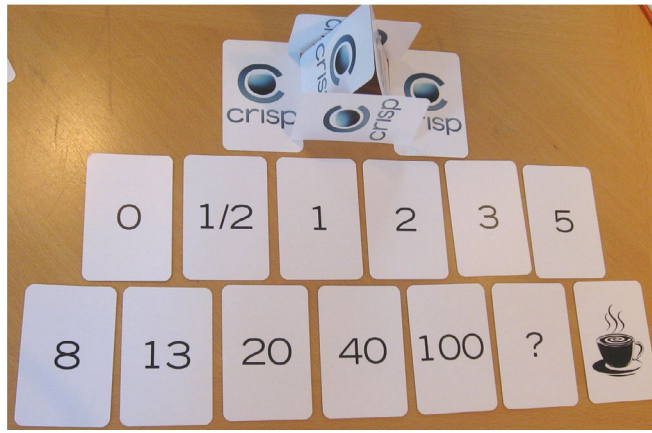
- At the time of planning, we normally don't know exactly who will be implementing which parts of which stories.
- Stories normally involve several people and different types of expertise (user-interface design, coding, testing, etc.).
- In order to provide an estimate, a team member needs some kind of understanding of what the story is about. By asking everybody to estimate each item, we make sure that each team member understands what each item is about. This increases the likelihood that team members will help each other out during the sprint. This also increases the likelihood that important questions about the story come up early.
- When asking everybody to estimate a story, we often discover discrepancies where two different team members have wildly

different estimates for the same story. That kind of stuff is better to discover and discuss earlier than later.

If you ask the team to provide an estimate, normally the person who understands the story best will be the first one to blurt one out. Unfortunately, this will strongly affect everybody else's estimates.

There is an excellent technique to avoid this – it is called planning poker (coined by Mike Cohn, I think).

Actually, Mike says he learned it from James Grenning, and James will probably say he got the idea from someone else. Never mind. We all stand on shoulders of giants. Or maybe we're a bunch of midgets standing on each other's shoulders. Er, whatever… – you know what I mean.



Each team member gets a deck of 13 cards as shown above.

<pitch>We sell these decks on planningpoker.crisp.se. And they look cooler now than in the photo, though you can probably find cheaper if you google around. Oh, and on that site we also sell something really cool called Jimmy Cards, by my colleague Jimmy (yes, we had trouble coming up with a name for the cards). Check it out! </pitch>

Whenever a story is to be estimated, each team member selects a card that represents his time estimate (in story points) and places it face down on the table. When all team members are done, the cards on the table are revealed simultaneously. That way, each team member is forced to think for himself rather than lean on somebody else's estimate.

If there is a large discrepancy between two estimates, the team discusses the differences and tries to build a common picture of what work is involved in the story. They might do some kind of task breakdown.

Afterwards, the team estimates again. This loop is repeated until the time estimates converge, i.e. all estimates are approximately the same for that story.

It is important to remind team members that they are to estimate the total amount of work involved in the story. Not just their part of the work. The tester should not just estimate the amount of testing work.

Note that the number sequence is non-linear. For example there is nothing between 40 and 100. Why?

This is to avoid a false sense of accuracy for large time estimates. If a story is estimated at approximately 20 story points, it is not relevant to discuss whether it should be 20 or 18 or 21. All we know is that it is a large story and that it is hard to estimate. So 20 is our ballpark guess.

Want more detailed estimates? Split the story into smaller stories and estimate the smaller stories instead!

And no, you can't cheat by combining a 5 and a 2 to make a 7. You have to choose either 5 or 8; there is no 7.

Some special cards to note:

- 0 = "This story is already done," or "this story is pretty much nothing, just a few minutes of work'"
- ? = "I have absolutely no idea at all. None."
- Coffee cup = "I'm too tired to think. Let's take a short break."

Another company came up with an even cooler deck, the No Bullshit estimation cards (estimation.lunarlogic.io). It has only three cards:

- 1 (one)

- TFB (too f*cking big)

- NFC (no f*cking clue)

Pretty cool! I wish I came up with that. I do take credit for the coffee cup though.

Clarifying stories

The worst is when a team proudly demonstrates a new feature at the sprint demo, and the product owner frowns and says "Well, that's pretty, but that's *not what I asked for!*"

How do you ensure that the product owner's understanding of a story matches the team's understanding? Or that each team member has the same understanding of each story? Well, you can't. But there are some simple techniques for identifying the most blatant misunderstandings. The simplest technique is simply to make sure that all the fields are filled in for each story (or more specifically, for each story that has high enough importance to be considered for this sprint).

Some call this "definition of ready". So "definition of done" is a checklist for when a story is done, and "definition of ready" is a checklist for when a story is ready to be pulled into a sprint. Very useful.

Example 1

The team and product owner are happy about the sprint plan and ready to end the meeting. The Scrum master says "Wait a sec. This story named 'Add user', there is no estimate for that. Let's estimate!" After a couple of rounds of planning poker the team agrees on 20 story points whereby the product owner stands up in rage: "Whaaaat?!" After a few minutes of heated discussion, it turns out that the team misunderstood the scope of "Add user"; they thought this meant "a nice web GUI to add, remove, delete, and search users", while the product owner just meant "add users by manually doing SQL towards DB". They estimate again and land at five story points.

Example 2

The team and product owner are happy about the sprint plan and ready to end the meeting. The Scrum master says "Wait a sec. This story named 'Add user', how should that be demonstrated?"

Some mumbling ensues and after a minute somebody stands up and says "Well, first we log in to the website, and then..." – and the product owner interrupts.

"Log in to the website?! No, no, no, this functionality should not be part of the website at all. It should be a silly little SQL script only for tech admins."

The "how to demo" description of a story can (and should) be *very brief*! Otherwise, you won't finish the sprint planning meeting on time. It is basically a high-level plain-English description of how to execute the most typical test scenario manually. "Do this, then that, then verify this."

I have found that this simple description *often* uncovers important misunderstandings about the scope of a story. Good to discover them early, right?

I still like this technique and use it whenever a story feels vague. Makes things concrete. An alternative is to draw a wireframe sketch or a list of acceptance criteria. Think of the story as a high-level problem statement, and the "definition of done" as a concrete example of how it might look like when done.

Breaking down stories into smaller stories

Stories shouldn't be too small or too big (in terms of estimates). If you have a bunch of 0.5-point stories you are probably going to be a victim of micromanagement. On the other hand, a 40-point story stands a high risk of ending up *partially* complete, which produces no value to your company and just increases administration. Furthermore, if your estimated velocity is 70 and your two top-priority stories are weighted 40 story points each, the planning gets kind of difficult. You have to choose between under-committing (i.e. taking just one item) and over-committing (i.e. taking both items).

I find that it is almost always possible to break a large story into smaller stories. Just make sure that the smaller stories still represent deliverables with business value.

We normally strive for stories weighted two to eight man-days. Our velocity is usually around 40-60 for a typical team, so that gives us somewhere around 10 stories per sprint. Sometimes as few as five and

sometimes as many as 15. That's a manageable number of index cards to deal with.

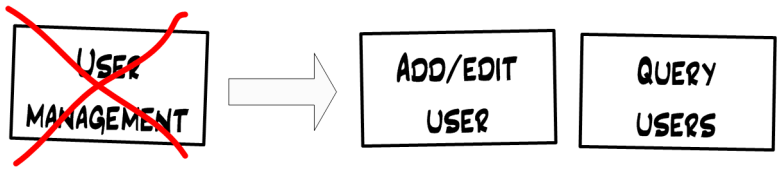
Five to 15 stories in a sprint is useful guideline. Fewer than five usually means the stories are too big for the size of the sprint, while more than 15 usually means the team has pulled in too much and won't finish everything (or the stories are too small, causing micromanagement).

Breaking down stories into tasks

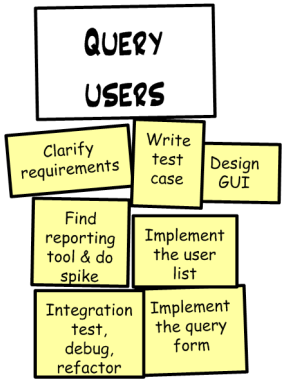
Wait a sec.... What's the difference between "tasks" and "stories"? A very valid question.

The distinction is quite simple. Stories are deliverable stuff that the product owner cares about. Tasks are non-deliverable stuff, or stuff that the product owner doesn't care about.

Example of breaking down a story into smaller stories:



Example of breaking down a story into tasks:



Here are some interesting observations:

- New Scrum teams are reluctant to spending time breaking down a bunch of stories into tasks up front like this. Some feel this is a waterfall-ish approach.
- For clearly understood stories, it is just as easy to do this breakdown up front as it is to do later.
- This type of breakdown often reveals additional work that causes the time estimate to go up, and thereby gives a more realistic sprint plan.
- This type of breakdown up front makes daily scrum meetings noticeably more efficient (see pg. 61, "How we do daily scrums").
- Even if the breakdown is inaccurate and will change once work starts, the above advantages still apply.

So, we try to make the sprint-planning time box long enough to fit this in, but if time runs out we let it drop out (see "Where to draw the line" below).

Task breakdown is a great opportunity to identify dependencies – like "we'll need access to the production logs" or "we'll need help from Jim in HR" – and figure out ways to deal with those dependencies. Maybe call Jim and see if he can reserve time for us during the sprint. The earlier you discover a dependency, the less likely it is to blow up your sprint!

Note: We practice TDD, which in effect means that the first task for almost each story is "write a failing test" and the last task is "refactor" (= improve code readability and remove duplication).

Defining time and place for the daily scrum

One frequently forgotten output of the sprint planning meeting is "a defined time and place for the daily scrum". Without this, your sprint will be off to a bad start. The first daily scrum is essentially the kickoff where everybody decides where to start working.

I prefer morning meetings. But, I must admit, we haven't actually tried doing daily scrums in the afternoon or midday.

Now I have. Works fine. Let the team decide. If unsure, experiment. Most teams prefer mornings though.

Disadvantage of afternoon scrums: When you come to work in the morning, you have to try to remember what you told people yesterday about what you will be doing today.

Disadvantage of morning scrums: When you come to work in the morning, you have to try to remember what you did yesterday so that you can report this.

My opinion is the first disadvantage is worse, since the most important thing is what you are *going to do*, not what you *did*.

Our default procedure is to select the earliest time at which nobody in the team groans. Usually 9:00, 9:30, or 10:00. The most important thing is that it is a time that everybody in the team can wholeheartedly accept.

Where to draw the line

OK, so time is running out. Of all the stuff we want to do during the sprint planning, what do we cut out if we run out of time?

Well, I use the following priority list:

Priority 1: A sprint goal and demo date. This is the very least you need to start a sprint. The team has a goal, an end date, and they can work right off the product backlog. It sucks, yes, and you should seriously consider scheduling a new sprint planning meeting tomorrow, but if you really need to get the sprint started then this will probably do. To be honest, though, I have never actually started a sprint with this little info.

Priority 2: List of which stories the team has accepted for this sprint.

Priority 3: "Estimate" filled in for each story in sprint.

Priority 4: "How to demo" filled in for each story in sprint.

Priority 5: Velocity and resource calculations, as a reality check for your sprint plan. Includes list of team members and their commitments (otherwise you can't calculate velocity).

Keep it simple and high-level, to take at most five minutes. Ask: "From a staffing perspective, is there anything *majorly* different about this sprint than past sprints?" If not, use yesterday's weather. If so, make adjustments accordingly.

Priority 6: Specified time and place for daily scrum. It only takes a moment to decide, but if you run out of time, the Scrum master can simply decide this after the meeting and email everyone.

Priority 7: Stories broken down into tasks. This breakdown can instead be done on a daily basis in conjunction with daily scrums, but will slightly disrupt the flow of the sprint.

Tech stories

Here's a complex issue: tech stories. Or non-functional items or whatever you want to call them.

I can't help giggling whenever someone talks about "non-functional requirements". Sounds too much like "things that shouldn't function". :o)

I'm referring to stuff that needs to be done but that is not deliverable, not directly related to any specific stories, and not of direct value to the product owner.

We call them "tech stories".

For example:

- **Install continuous-build server**
  Why it needs to be done: because it saves immense amounts of time for the developers and reduces the risk of big-bang integration problems at the end of an iteration.

- **Write a system design overview**
  Why it needs to be done: because developers keep forgetting the overall design, and thereby write inconsistent code. Need a "big picture" document to keep everyone on the same page design-wise.

- **Refactor the DAO layer**
  Why it needs to be done: because the DAO layer has gotten really messy and is costing everyone time due to confusion and unnecessary

bugs. Cleaning the code up will save time for everyone and improve the robustness of the system.

- **Upgrade Jira** (bug tracker)
  Why it needs to be done: the current version is too buggy and slow. Upgrading will save everyone time.

Are these stories in the normal sense? Or are they tasks that are not connected to any specific story? Who prioritizes these? Should the product owner be involved in this stuff?

We've experimented a lot with different ways of handling tech stories. We tried treating them as first-class stories, just like any others. That was no good; when the product owner prioritized the product backlog, it was like comparing apples with oranges. In fact, for obvious reasons, the tech stories were often given low priority with the motivation like "Yeah guys, I'm sure a continuous-build server is important and all, but let's build some revenue-driving features first shall we? Then you can add your tech candy later, OK?"

In some cases the product owner is right, but often not. We've concluded that the product owner is not always qualified to be making that tradeoff. So here's what we do:

1. Try to avoid tech stories. Look hard for a way to transform a tech story into a normal story with measurable business value. That way the product owner has a better chance to make correct tradeoffs.

2. If we can't transform a tech story into a normal story, see if the work could be done as a task within another story. For example, "refactor the DAO layer" could be a task within the story "edit user", since that involves the DAO layer.

3. If both of the above fail, define it as a tech story, and keep a separate list of such stories. Let the product owner see it but not edit it. Use the "focus factor" and "estimated velocity" parameters to negotiate with the product owner and bend out some time in the sprint to implement tech stories.

I still find tech stories to be a great pattern and use it a lot. Smaller tech stories are just embedded into the day-to-day work, while larger stories are written down and placed in a tech backlog, visible to the product owner but managed by the team. The team and product owner agree on a guideline such as "10-20% of our time is spent on tech stories". No need

for elaborate tracking schemes like focus factor or time reports, just use gut feel. Ask at the retro, "Roughly how much of our sprint capacity did we spend on tech stories, and did that feel about right?"

A dialogue very similar to this occurred during one of our sprint planning meetings:

Team: We have some internal tech stuff that needs to be done. We would like to budget 10% of our time for that, i.e. reduce focus factor from 75% to 65%. Is that OK?

Product owner: Hell no! We don't have time!

Team: Well, look at the last sprint. (All heads turn to the velocity scrawls on the whiteboard.) Our estimated velocity was 80, and our actual velocity was 30, right?

Product owner: Exactly! So we don't have time to do internal tech stuff! Need new features!

Team: Well, the *reason* why our velocity turned out to be so bad was because we spent so much time trying to put together consistent releases for testing.

Product owner: Yes, and?

Team: Well, our velocity will probably continue being that bad if we don't do something about it.

Product owner: Yes, and?

Team: So we propose that we take approximately 10% off of this sprint to set up a continuous-build server and other stuff that will take the pain off of integration. This will probably increase our sprint velocity by *at least* 20% for each subsequent sprint, forever!

Product owner: Oh really? Why didn't we do this last sprint then?!

Team: Er, because you didn't want us to....

Product owner: Oh, um, well, fine – sounds like a good idea to do it now then!

Of course, the other option is to just keep the product owner out of the loop or give him a non-negotiable focus factor. But there's no excuse not to *try* to reach consensus first.

If the product owner is a competent and reasonable fellow (and we've been lucky there) I suggest keeping him as informed as possible and letting him make the overall priorities. Transparency is one of the core values of Scrum, right?

If you can't have frank conversations with the product owner about things like tech stories, quality, and technical debt, then you have a deeper problem that really needs to be addressed! A symptom of this is if you catch yourself deliberately hiding information from the product owner. You don't have to involve the product owner in every conversation, but the relationship should really be based on trust and respect; without that, you are unlikely to succeed with whatever you are building.

Bug tracking system vs. product backlog

Here's a tricky issue. Excel is a great format for the product backlog. But you still need a bug tracking system, and Excel will probably not do. We use Jira.

So how do we bring Jira issues into the sprint planning meeting? I mean it wouldn't do to just ignore them and only focus on stories.

We've tried several strategies:

1. Product owner prints out the most high priority Jira items, brings them to the sprint planning meeting, and puts them up on the wall together with the other stories (thereby implicitly specifying the priority of these items compared to the other stories).

2. Product owner creates stories that refer to Jira items. For example "Fix the most critical back-office reporting bugs: Jira-124, Jira-126, and Jira-180."

3. Bug fixing is considered to be outside of the sprint, i.e. the team keeps a low enough focus factor (for example 50%) to ensure that they have time to fix bugs. It is then simply assumed that the team will spend a certain amount of time each sprint fixing Jira-reported bugs

4. Put the product backlog in Jira (i.e. ditch Excel). Treat bugs just like any other story.

We haven't really concluded which strategy is best for us; in fact, it varies from team to team and from sprint to sprint. I tend to lean towards the first strategy though. It is nice and simple.

Eight years later I can only agree. There's no single best practice; each strategy above can be fine depending on the context. Experiment until you find what works best for you.

Sprint planning meeting is finally over!

Wow, I never would have thought this chapter on sprint planning meetings would be so long! I guess that reflects my opinion that the sprint planning meeting is the most important thing you do in Scrum. Spend a lot of effort getting that right, and the rest will be so much easier.

Or vice versa – get the other stuff right and sprint planning is a piece of cake. :o)

The sprint planning meeting is successful if everyone (all team members and the product owner) exit the meeting with a smile, and wake up the next morning with a smile, and do their first daily scrum with a smile.

Then, of course, all kinds of things can go horribly wrong down the line, but at least you can't blame the sprint plan, :o)

PART FIVE

How we communicate sprints

It is important to keep the whole company informed about what is going on. Otherwise, people will complain or, even worse, make false assumptions about what is going on.

We use a "sprint info page" for this.

Jackass team, sprint 15

Sprint goal
- Beta-ready release!

Sprint backlog (estimates in parenthesis)
- Deposit (3)
- Migration tool (8)
- Backoffice login (5)
- Backoffice user admin (5)

Estimated velocity: 21

Schedule
- Sprint period: 2006-11-06 to 2006-11-24
- Daily scrum: 9:30 – 9:45, in the team room
- Sprint demo: 2006-11-24, 13:00, in the cafeteria

Team
- Jim
- Erica (scrum master)
- Tom (75%)
- Eva
- John

Sometimes we include info about how each story will be demonstrated as well.

As soon as possible after the sprint planning meeting, the Scrum master creates this page, puts it up on the wiki, and sends a spam to the whole company.

Subject: Jackass sprint 15 started

Hi all! The Jackass team has now started sprint 15. Our goal is to demonstrate a beta-ready release on nov 24.

See the sprint info page for details:
http://wiki.mycompany.com/jackass/sprint15

We also have a dashboard page on our wiki, which links to all currently ongoing sprints.

Corporate Dashboard

Ongoing sprints
- Team X sprint 15
- Team Y sprint 12
- Team Z sprint 1

In addition, the Scrum master prints out the sprint info page and posts it on the wall outside his team room. So anybody walking by can look at the sprint info page to find out what that team is doing. Since that includes the time and place for the daily scrum and sprint demo, he knows where to go to find out more.

When the sprint nears the end, the Scrum master reminds everybody about the upcoming demo.

Subject: Jackass sprint demo tomorrow at 13:00 in the cafeteria.

Hi all! You are welcome to attend our sprint demo at 13:00 in the cafeteria tomorrow (friday). We will demonstrate a beta-ready release.

See the sprint info page for details:
http://wiki.mycompany.com/jackass/sprint15

Given all this, nobody really has an excuse *not* to know what's going on.

Hey, what a great idea! I should do that more often!

PART SIX

How we do sprint backlogs

You made it this far? Whew, good job.

So now that we've completed the sprint planning meeting and told the world about our shiny new sprint, it is time for the Scrum master to create a sprint backlog. This needs to be done *after* the sprint planning meeting, but *before* the first daily scrum.

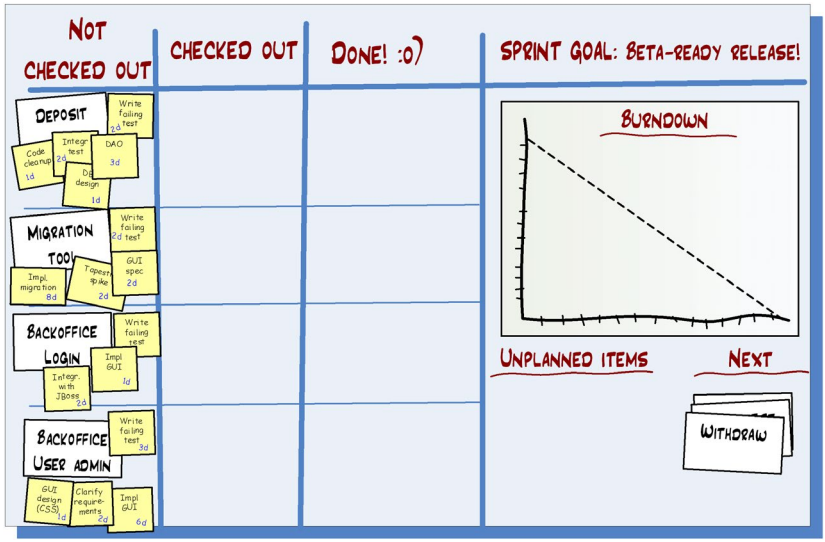
Sprint-backlog format

We've experimented with different formats for the sprint backlog, including Jira, Excel, and a physical task board on the wall. In the beginning we used Excel mostly; there are many publicly available Excel templates for sprint backlogs, including auto-generated burn-down charts and stuff like that. I could talk a lot about how we refined our Excel-based sprint backlogs. But I won't. I won't even include an example here.

Instead I'm going to describe in detail what we have found to be the most effective format for the sprint backlog – a wall-based task board!

Yep, wall-based task boards (a.k.a. Scrum boards) are always my first tool of choice! Surprisingly powerful. For a distributed team, use a tool that provides a Scrum board view (just about all tools do), and put it on a big screen at each site. At the daily scrum, everyone stands at the wall screen and talks via Skype (or equivalent).

Find a big wall that is unused or contains useless stuff like the company logo, old diagrams, or ugly paintings. Clear the wall (ask for permission only if you must). Tape up a big, big sheet of paper (at least 2x2 meters, or 3x2 meters for a large team). Then do this:



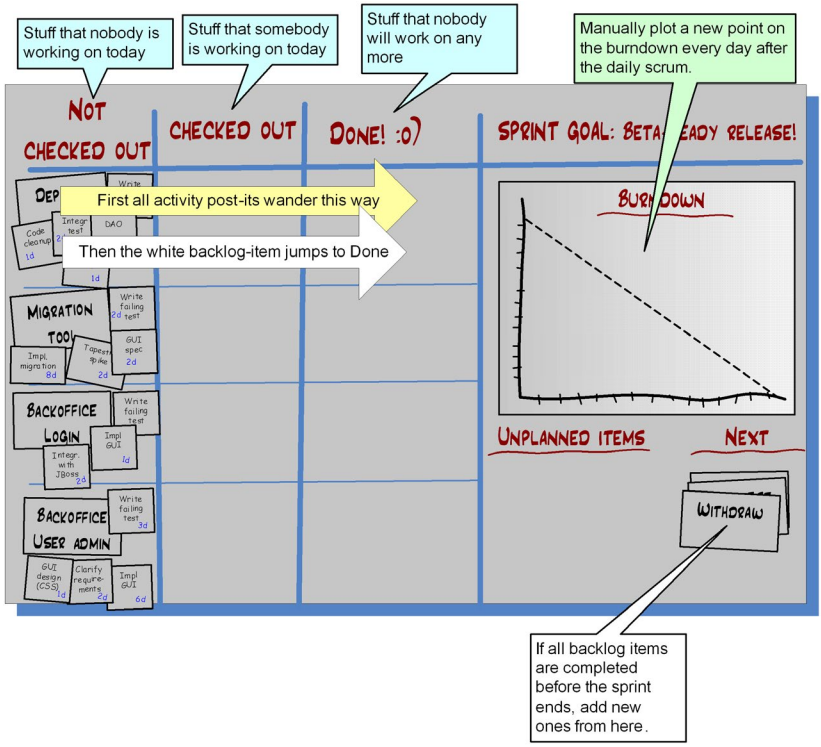
You could of course use a whiteboard. But that's a bit of a waste. If possible, save whiteboards for design scrawls and use non-whiteboard walls for task boards.

Or better yet, plaster your walls with whiteboards. A worthy investment!

Note: If you use Post-its for tasks, don't forget to attach them using real tape, or you'll find all the Post-its in a neat pile on the floor one day.

Or just buy super-sticky notes, slightly more expensive but they don't fall down! (No, I'm not sponsored, really.)

How the task board works

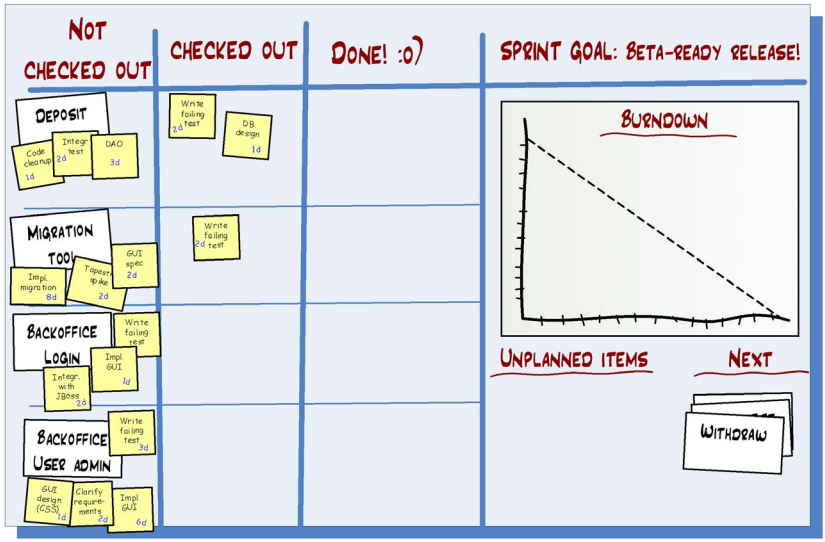


You could of course add all kinds of additional columns. "Waiting for integration test", for example. Or "Cancelled". However, before you complicate matters, think deeply. Is this addition really, *really* necessary?

I've found that simplicity is extremely valuable for these types of things, so I only add additional complications when the cost of *not* doing so is too great.

Example 1: After the first daily scrum

After the first daily scrum, the task board might look like this:



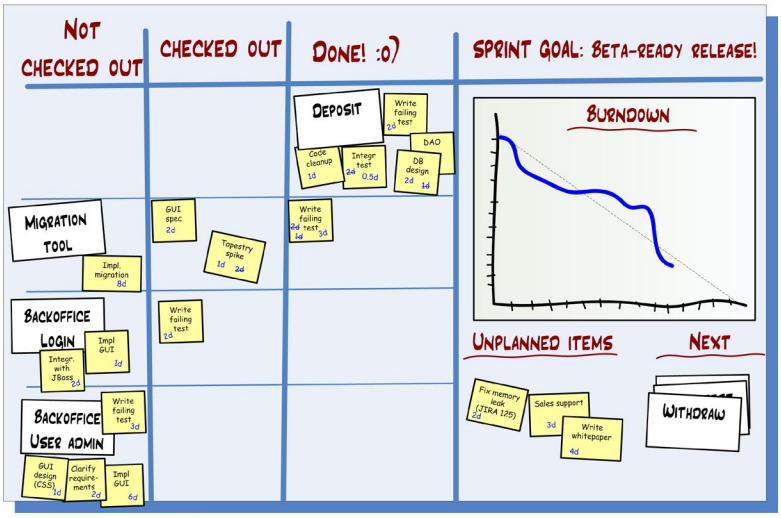
As you can see, three tasks have been checked out, i.e. the team will be working on these items today.

Sometimes, for larger teams, a task gets stuck in "Checked out" because nobody remembers who was working on it. If this happens often in a team, they usually introduce policies such as labeling each checked-out task with the name of the person who checked it out.

Just about all teams use avatars nowadays. Each team member picks their avatar (a South Park figure or something), prints them, and puts them on magnets. Great way to see who is working on what. Also, if each person only has like two magnets, that indirectly limits work in progress and multitasking. "WTF, I'm out of avatars!" Yeah, so stop starting and start finishing tasks!

Example 2: After a few more days

A few days later the task board might look something like this:



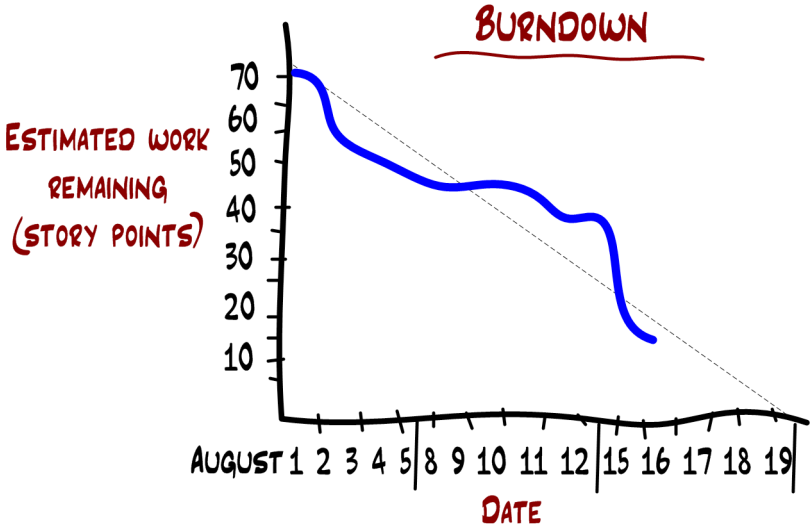
As you can see, we've completed the Deposit story (i.e. it has been checked in to the source-code repository, tested, refactored, etc.). The Migration Tool is partially complete, the Backoffice Login is started, and the Backoffice User Admin is not started.

We've had three unplanned items, as you can see down to the right. This is useful to remember when you do the sprint retrospective.

Here's an example of a real sprint backlog near the end of a sprint. It does get rather messy as the sprint progresses, but that's OK since it is short-lived. Every new sprint, we create a fresh, clean, new sprint backlog.

How the burn-down chart works

Let's zoom in on the burn-down chart:



This chart shows that:

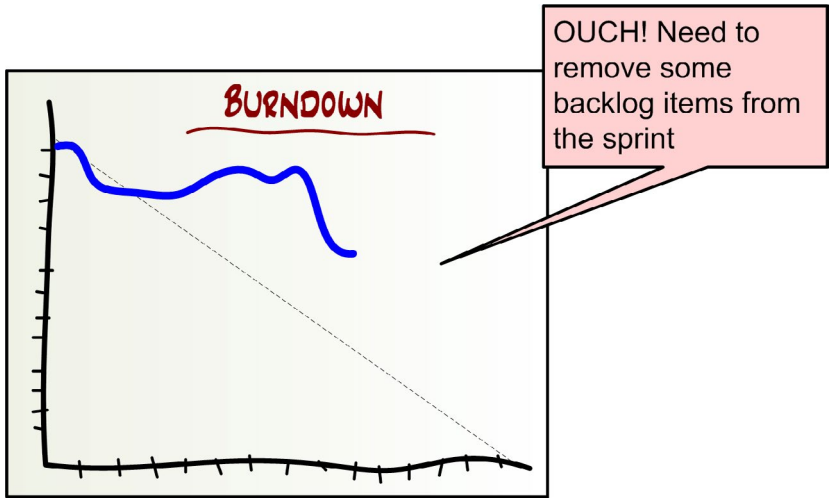
- On the first day of the sprint, August 1, the team estimated that there were approximately 70 story points of work left to do. This was in effect the *estimated velocity* of the whole sprint.
- On August 16, the team estimates that there were approximately 15 story points of work left to do. The dashed trend line shows that they are approximately on track, i.e. at this pace they will complete everything by the end of the sprint.

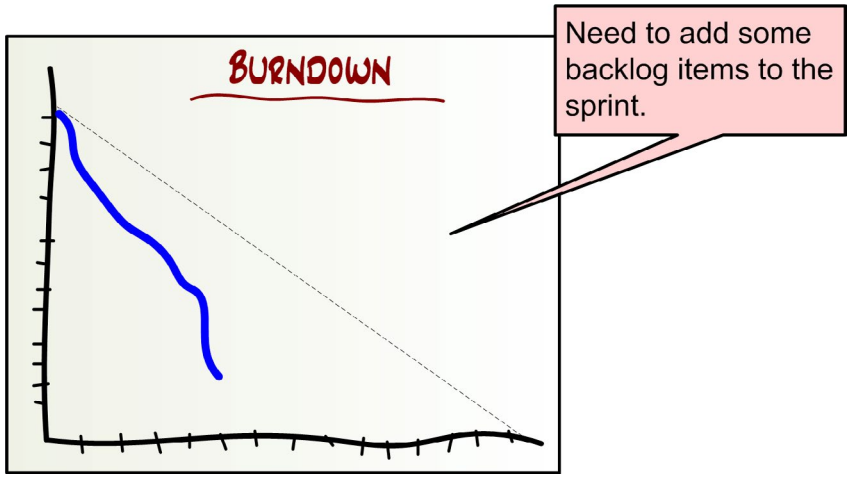
We skip weekends on the X-axis since work is rarely done on weekends. We used to include weekends but this would make the burn down slightly confusing, since it would "flatten out" over weekends, which would look like a warning sign.

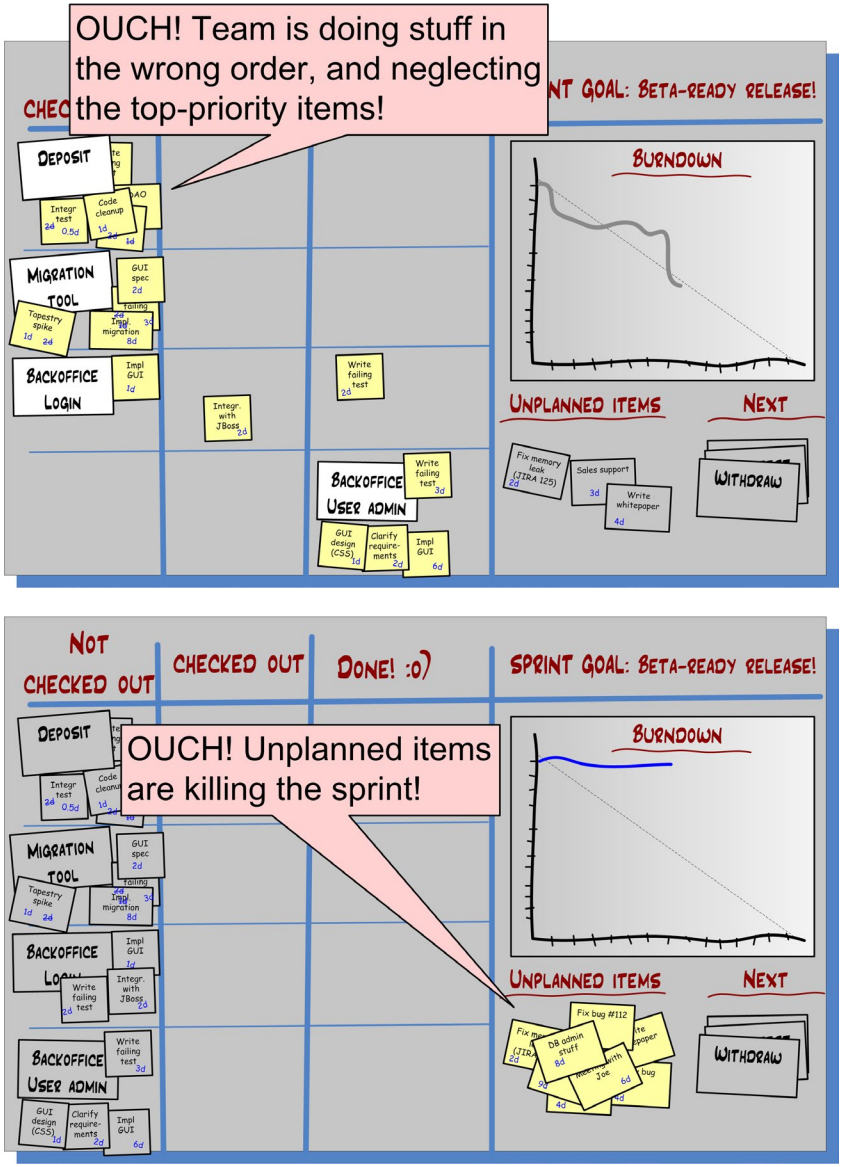
Scrum was originally designed with teams doing one-month sprints and using Excel to track tasks. In that case, a burn-up chart is a really useful visual summary of how we're doing. Now, though, most teams do shorter sprints and have highly visual Scrum boards. So they increasingly skip burn-down charts entirely, because one glance at the Scrum board gives them the info they need. Try skipping the burn-down chart and see if you miss it!

Task-board warning signs

A quick glance at the task board should give anyone an indication of how well the sprint is progressing. The Scrum master is responsible for making sure that the team acts upon warning signs such as:


BURNDOWN
OUCH! Need to remove some backlog items from the sprint


BURNDOWN
Need to add some backlog items to the sprint.



Hey, what about traceability?!

The best traceability I can offer in this model is to take a digital photo of the task board every day. If you must. I do that sometimes but never find a need to dig up those photos.

If traceability is very important to you, then perhaps the task-board solution is not for you.

But I suggest you really try to estimate the actual value of detailed sprint traceability. Once the sprint is done and working code has been delivered and documentation checked in, does anyone really care how many stories were completed at day five in the sprint? Does anyone really care what the time estimate for "write a failing test for Deposit" was?

I still find traceability hugely overrated. Some tools provide that type of data but people basically never use it. Your code-version-control system will give you most of what you need ("WTF? Who made this change?!"). Add some simple conventions such as writing the story ID in your commit comment, and you get more than enough traceability for most contexts.

Estimating days vs. hours

In most books and articles on Scrum, you'll find that tasks are time-estimated in hours, not days. We used to do that. Our general formula was: 1 effective man-day = 6 effective man-hours.

Now we've stopped doing that, at least in most of our teams, for the following reasons:

Man-hour estimates were too fine grained; this tended to encourage too many tiny one-hour to two-hour tasks and hence micromanagement.

It turned out that everyone was thinking in terms of man-days anyway, and just multiplying by six before writing down man-hours. "Hmmmm, this task should take about a day. Oh, I have to write hours. I'll write six hours then."

Two different units cause confusion. "Was that estimate in man-days or man-hours?"

So now we use man-days as a basis for all time estimates (although we call it story points). Our lowest value is 0.5, i.e. any task that is smaller than 0.5 is either removed, combined with some other task, or just left with a 0.5 estimate (no great harm in overestimating slightly). Nice and simple.

Even simpler: skip estimating tasks entirely! Most teams eventually learn how to break their work into tasks that are roughly a day for one to two

people. If you can do that, you don't need to bother with task estimates, which eliminates a lot of waste. Burn-down charts can still be used (if you must) – in that case, just count the tasks instead of adding up the hours.

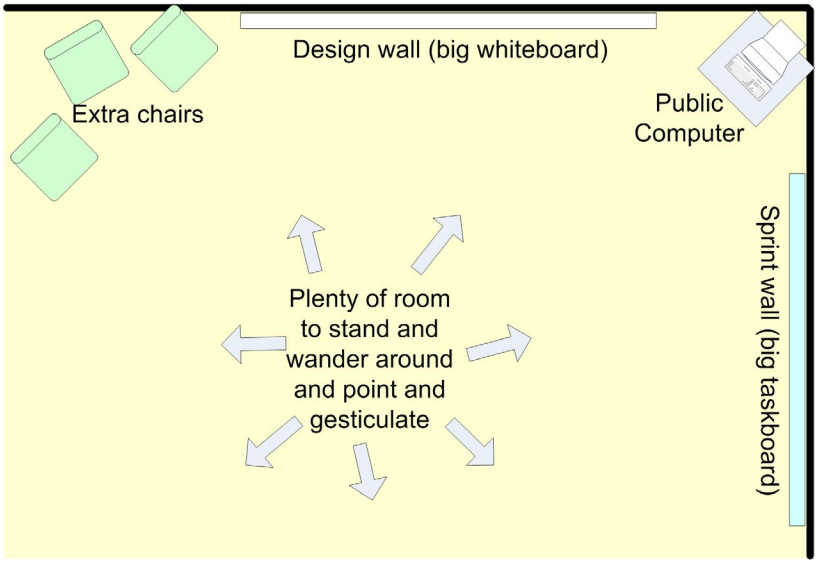
PART SEVEN

How we arrange the team room

The design corner

I've noticed that many of the most interesting and valuable design discussions take place spontaneously in front of the task board.

For this reason, we try to arrange this area as an explicit "design corner".



This is really quite useful. There is no better way to get an overview of the system than to stand in the design corner and glance at both walls, then glance at the computer and try the latest build of the system (if you are lucky enough to have continuous build, see pg. 81 "How we combine Scrum with XP").

The "design wall" is just a big whiteboard containing the most important design scrawls and printouts of the most important design documentation (sequence charts, GUI prototypes, domain models, etc.).

Above: a daily scrum going on in the aforementioned corner.

Hmm… that burndown looks suspiciously nice and straight, doesn't it? But the team insists that it is real. :o)

The Evil Coach provides fake burn-down rulers. Very convenient for highly dysfunctional environments. :o)

http://blog.crisp.se/2013/02/15/evil-coach/fake-burndown-ruler

Seat the team together!

When it comes to seating and desk layout there is one thing that can't be stressed strongly enough.

Seat the team together!

To clarify that a bit, what I'm saying is:

Seat the team together!

After eight years of helping companies with Scrum, I'd like to add:

SEAT THE TEAM TOGETHER!

People are reluctant to move. At least in the places I've worked. They don't want to have to pick up all their stuff, unplug the computer, move

all their junk to a new desk, and plug everything in again. The shorter the distance, the greater the reluctance. "Come ON, boss, what's the point of moving just five meters?"

When building effective Scrum teams, however, there is no alternative. Just get the team together. Even if you have to personally threaten each individual, carry all their gear, and wipe up their old coffee stains. If there is no space for the team, make space. Somewhere. Even if you have to place the team in the basement. Move tables around, bribe the office manager, do whatever it takes. Just get the team together.

Once you have the team together the payoff will be immediate. After just one sprint, the team will agree that it was a good idea to move together (speaking from personal experience that is – there's nothing saying your team won't be too stubborn to admit it).

Now what does "together" mean? How should the desks be laid out? Well, I don't have any strong opinion on the optimal desk layout. And even if I did, I assume most teams don't have the luxury of being able to decide exactly how to lay out their desks. There are usually physical constraints – the neighboring team, the toilet door, the big slot machine in the middle of the room, whatever.

"Together" means:

- **Audibility:** Anybody in the team can talk to anybody else without shouting or leaving his desk.
- **Visibility:** Everybody in the team can see everybody else. Everyone can see the task board. Not necessarily close enough to be able to *read* it, but at least *see* it.
- **Isolation:** If your whole team were to suddenly stand up and engage in a spontaneous and lively design discussion, there is nobody outside the team close enough to be disturbed. And vice versa.

"Isolation" doesn't mean that the team has to be completely isolated. In a cubicle environment, it may be enough that your team has its own cubicle and big enough cubicle walls to filter out *most* of the noise from non-team elements.

And what if you have a distributed team? Well, then you are out of luck. Use as many technical aids as you can to minimize the damage – video conferencing, webcams, desktop-sharing tools, etc.

Keep the product owner at bay

The product owner should be near enough so that the team can wander over and ask him something, and so that he can wander over to the task board. But he should not be seated with the team. Why? Because chances are he will not be able to stop himself from meddling in details, and the team will not gel properly (i.e. reach a tight, self-managed, hyper-productive state).

To be honest, this is speculation. I haven't actually seen a case where the product owner is sitting with the team, so I have no actual empirical reason to say that it is a bad idea. Just gut feeling and hearsay from other Scrum masters.

Well, guess what. I was wrong! Dead wrong. The best teams I've seen have the product owner embedded. Teams suffer a lot when the product owner is too far away, and that's a much bigger problem than being too close. However, the product owner needs to balance his time between the team and the stakeholders, so he shouldn't spend ALL the time with the team. But, generally speaking, the closer the better.

Keep the managers and coaches at bay

This is a bit hard for me to write about, since I was both manager and coach….

It was my job to work as closely with the teams as possible. I set up the teams, moved between them, pair-programmed with people, coached Scrum masters, organized sprint planning meetings, etc. In retrospect, most people thought this was a Good Thing, since I had some experience with agile software development.

But, then, I was also (enter Darth Vader music) the chief of development, a functional manager role. Which means when I entered a team, it would automatically become less self-managing. "Heck, boss is here. He probably has lots of opinions on what we should be doing and who should be doing what. I'll let him do the talking."

My point is this: if you are Scrum coach (and perhaps also a manager), do get involved as closely as possible. But only for a limited period. Then get out and let the team gel and self-manage. Check up on the team once in a while (not too often) by attending sprint demos and looking at the task board and listening in on morning scrums. If you see an improvement area, take the Scrum master aside and coach him. *Not* in front of the team. Another good idea is to attend sprint retrospectives (see pg. 67 "How we do sprint retrospectives"), if your team trusts you enough not to let your presence clam them up.

As for well-functioning Scrum teams, make sure they get everything they need, then stay the hell out of the way (except during sprint demos).

That last sentence is still the best overall advice I have for managers in an agile context. Good managers are a crucial success factor, but as manager, you should try to make yourself redundant. You probably won't succeed with that, but the very act of trying will push you in the right direction. Ask "What does this team need in order to manage itself?" rather than "How can I manage this team?" They need things like transparency, clear purpose, a fun and motivating work environment, air cover, and an escalation path for impediments.

PART EIGHT

How we do daily scrums

Our daily scrums are pretty much by the book. They start exactly on time, every day at the same place. In the beginning, we would go to a separate room to do sprint planning (those were the days when we used electronic sprint backlogs), however now we do daily scrums in the team room right in front of the task board. Nothing can beat that.

We normally do the meetings standing up, since that reduces the risk of surpassing 15 minutes.

Daily scrum is really important! It's the point where most synchronization happens and where the team raises important impediments. Nevertheless, if done badly they can be reeeaaally boring – a bunch of people rambling on and nobody really listening.

The Scrum Guide recently updated the three questions to counter this:

- What did I do yesterday that helped our team meet the sprint goal?

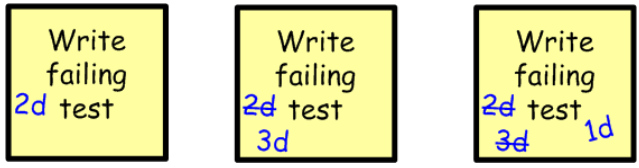
- What will I do today to help our team meet the sprint goal?

- Do I see any impediments that prevent me or our team from meeting the sprint goal?

Notice the focus on the sprint goal, the team's shared high-level purpose! If your daily scrums are starting to feel drab, try those questions! Or perhaps the Dan North version: ask "What's the best 'today' we can have?" followed by open discussion. Whatever you do, don't let daily scrums stay boring. Keep experimenting!

How we update the task board

We normally update the task board during the daily scrum. As each person describes what he did yesterday and will do today, he pulls Post-its around on the task board. As he describes an unplanned item, he puts up a Post-it for that. As he updates his time estimates, he writes the new time estimate on the Post-it and crosses off the old one. Sometimes the Scrum master does the Post-it stuff while people talk.



Some teams have a policy that each person should update the task board *before* each meeting. That works fine as well. Just decide on a policy and stick to it.

Many teams spend an inordinate amount of time updating numbers on sticky notes during the daily scrum. Waste! The purpose of the daily scrum is to get synchronized, so I usually find it best to update the board "in real time" (i.e. during the work day as stuff happens) and skip task estimates entirely. That way the daily scrum is used to actually *communicate* rather than administrate.

Regardless of what format your sprint backlog is in, try to get the *whole team* involved in keeping the sprint backlog up to date. We've tried doing sprints where the Scrum master is the sole maintainer of the sprint backlog and has to go around every day and ask people about their remaining time estimates. The disadvantages of this are:

- The Scrum master spends too much time administrating stuff, instead of supporting the team and removing impediments.
- Team members are unaware of the status of the sprint, since the sprint backlog is not something they need to care about. This lack of feedback reduces the overall agility and focus of the team.

If the sprint backlog is well designed, it should be just as easy for each team member to update it himself.

Immediately after the daily scrum meeting, someone sums up all the time estimates (ignoring those in the "done" column of course) and plots a new point on the sprint burn down.

Dealing with latecomers

Some teams have a can of coins and bills. When you are late, even if only one minute late, you add a fixed amount to the can. No questions asked. If you call before the meeting and say you'll be late, you still have to pay up.

You only get off the hook if you have a good excuse such as a doctor's appointment or your own wedding or something.

The money in the can is used for social events. To buy hamburgers when we have gaming nights, for example. :o)

This works well. But it is only necessary for teams where people often come late. Some teams don't need this type of scheme.

Teams use all kinds of schemes to tease each other into showing up on time (if needed). Just make sure the team comes up with it themselves; don't impose a scheme from above or outside the team. And keep it fun. In one team, latecomers had to sing a silly song. If you're late a second time, you have to do the accompanying dance moves as well. :o)

Dealing with "I don't know what to do today"

It is not uncommon for somebody to say "Yesterday, I did bla bla bla, but today I haven't the foggiest clue of what to do" (hey, that last bit rhymed). Now what?

Let's say Joe and Lisa are the ones who don't know what to do today.

If I am Scrum master I just move on and let the next guy talk, but make note of which people didn't have anything to do. After everybody's had their say, I go through the task board with the whole team, from top to bottom, and check that everything is in sync, that everybody knows what each item means, etc. I invite people to add more Post-its. Then I go back to those people who didn't know what to do: "Now that we've gone through the task board, do you have any ideas about what you can do today"? Hopefully, they will.

If not, I consider if there is any pair-programming opportunity here. Let's say Niklas is going to implement the back-office user-admin GUI today. In that case, I politely suggest that perhaps Joe or Lisa could pair program with Niklas on that. That usually works.

And if that doesn't work, here's the next trick.

Scrum master: OK, who wants to demonstrate the beta-ready release to us? (Assuming that was the sprint goal.)

Team: (Confused silence.)

Scrum master: Aren't we done?

Team: Um... no.

Scrum master: Oh darn. Why not? What's left to do?

Team: Well we don't even have a test server to run it on, and the build script is broken.

Scrum master: Aha. (Adds two Post-its to the task wall.) Joe and Lisa, how can you help us today?

Joe: Um.... I guess I'll try to find some test server somewhere.

Lisa: And I'll try to fix that build script.

If you are lucky, someone will actually demonstrate the beta-ready release you asked for. Great! You have achieved your sprint goal. But what if you are in mid-sprint? Easy. Congratulate the team on a job well done, grab one or two of the stories from the "next" section at the bottom right of your task board, and move them to the "not checked out" column to the left. Then redo the daily scrum. Notify the product owner that you have added some items to the sprint.

Or use the time to pay off some technical debt, or do some technical exploration. Keep the product owner in the loop though.

But what if the team has not yet achieved the sprint goal and Joe and Lisa still refuse to come up with something useful to do? I usually consider one of the following strategies (none of them are very nice, but then this is a last resort):

- **Shame:** "Well, if you have no idea how you can help the team, I suggest you go home, or read a book or something. Or just sit around until someone calls for your help."
- **Old school:** Simply assign them a task.
- **Peer pressure:** "Feel free to take your time, Joe and Lisa, we'll all just stand here and take it easy until you come up with something to do that will help us reach the goal."

- **Servitude:** "Well, you can help the team indirectly by being butlers today. Fetch coffee, give people massages, clean up some trash, cook us some lunch, and whatever else we may ask for during the day." You may be surprised by how fast Joe and Lisa manage to come up with useful technical tasks. :o)

If one person frequently forces you to go that far, then you should probably take that person aside and do some serious coaching. If the problem still remains, you need to evaluate whether this person is important to your team or not.

If he *isn't* too important, try to get him removed from your team.

If he *is* important, then try to pair him up with somebody else who can act as his shepherd. Joe might be a great developer and architect, it's just that he really prefers other people to tell him what to do. Fine. Give Niklas the duty of being Joe's permanent shepherd. Or take on the duty yourself. If Joe is important enough to your team it will be worth the effort. We've had cases like this and it more or less worked.

The "I don't know what to do today" problem is typical for teams that are new to Scrum, and are used to having other people decide things for them. As they get more experienced with self-organization, the problem disappears. People learn to figure out what to do. So if you are a Scrum master and you find yourself resorting to the above tricks too often, you should consider taking a step back. Despite your helpful intention, you may be the team's biggest impediment, stopping them from learning how to self-organize!

PART NINE

How we do sprint demos

The sprint demo (or sprint review as some people call it) is an important part of Scrum that people tend to underestimate.

"Oh do we really *have to* do a demo? There really isn't much fun to show!"

"We don't have time to prepare a &%$# demo!"

"I don't have time to attend other team's demos!"

I can't understand why I called it the "sprint demo"! What the heck was I thinking? The official term is the "sprint review", and that's a much better term. Demo implies a one-way communication ("here you go, this is what we built"), while review implies a two-way communication ("here's what we built, what do you think?"). Sprint review is all about feedback! So when you read "demo" below, think "review", OK?

Why we insist that all sprints end with a demo

A well-executed sprint demo, although it may seem undramatic, has a profound effect:

The team gets credit for their accomplishment. They *feel good*.

- Other people learn what your team is doing.
- The demo attracts vital feedback from stakeholders.
- Demos are (or should be) a social event where different teams can interact with each other and discuss their work. This is valuable.
- Doing a demo forces the team to *actually finish stuff* and release it (even if it is only to a test environment). Without demos, we kept getting huge piles of 99%-finished stuff. With demos, we may get fewer items done, but those items are *really done*, which is (in our case) a lot better than having a whole pile of stuff that is just *sort of done* and will pollute the next sprint.

If a team is more or less forced to do a sprint demo, even when they don't have much that really works, the demo will be embarrassing. The team will stutter and stumble while doing the demo and the applause

afterwards will be half-hearted. People will feel a bit sorry for the team, some may be irritated that they wasted time going to a lousy demo.

This hurts. But the effect is like a bitter-tasting medicine. *Next sprint*, the team will really try to get stuff *done*! They will feel that "well, maybe we can only demonstrate two things next sprint instead of five, but dammit this time it's going to WORK!". The team knows that they will have to do a demo no matter what, which significantly increases the chance that there will be something useful to demonstrate. I've seen this happen several times.

This is extra crucial in a multi-team context. Everyone involved needs to see the integrated product come together on a regular basis. There will always be integration problems, but the earlier you discover them, the easier they are to solve. Self-organization only works with transparency and feedback loops, and a well-executed sprint review provides both.

Checklist for sprint demos

- Make sure you clearly present the sprint goal. If there are people at the demo who don't know anything about your product, take a few minutes to describe the product.
- Don't spend too much time preparing the demo, especially not on flashy presentations. Cut the crap out and just focus on demonstrating actual working code.
- Keep a high pace, i.e. focus your preparations on making the demo fast-paced rather than beautiful.
- Keep the demo on a business-oriented level. Leave out the technical details. Focus on "what did we do" rather than "how did we do it".
- If possible, let the audience try the product for themselves.
- Don't demonstrate a bunch of minor bug fixes and trivial features. Mention them but don't demo them, since that generally takes too long and detracts focus from the more important stories.

Some teams do two reviews: a short public review, aimed at external stakeholders, followed by an internal review with more details and things like key challenges and technical decisions made along the way. A great

way to spread knowledge between teams, and spare stakeholders from techy details they don't care about.

Dealing with indemonstrable stuff

Team member: I'm not going to demonstrate this item, because it can't be demonstrated. The story is "Improve scalability so system can handle 10,000 simultaneous users". I can't bloody well invite 10,000 simultaneous users to the demo can I?

Scrum master: Are you done with the item?

Team member. Yes, of course.

Scrum master: How do you know?

Team member: I set the system up in a performance-test environment, started eight load servers, and pestered the system with simultaneous requests.

Scrum master: But do you have any indication that the system will handle 10,000 users.

Team member: Yes. The test machines are crappy, yet they could handle 50,000 simultaneous requests during my test.

Scrum master: How do you know?

Team member (frustrated): Well, I have this report! You can see for yourself. It shows how the test was set up and how many requests were sent!

Scrum master: Oh, excellent! Then there's your "demo". Just show the report and go through it with the audience. Better than nothing right?.

Team member: Oh, is that enough? But it's ugly. I need to polish it up.

Scrum master: OK, but don't spend too long. It doesn't have to be pretty, just informative.

PART TEN

How we do sprint retrospectives

Why we insist that all teams do retrospectives

The most important thing about retrospectives is to *make sure they happen.*

For some reason, teams don't always seem inclined to do retrospectives. Without gentle prodding, most of our teams would often skip the retrospective and move on to the next sprint instead. It may be a cultural thing in Sweden. I'm not sure.

Nope, I've seen it in lots of countries, so it's just human nature. We always want to move on to the next thing. Ironically, the more stressed you are, the more likely you are to want to skip the retrospective. But the more stressed you are, the more badly you need the retrospective! Kind of like "I'm in such a hurry to chop down trees, I don't have time to stop and sharpen my saw!" So a Scrum master should really insist on doing retrospectives! They don't need to take so long though. For two-week sprints, time-box the retrospective to one hour. But do a longer (half-day or full-day) retrospective every couple of months so you can deal with the thornier issues.

Yet, everybody seems to agree that retrospectives are extremely useful. In fact, I'd say the retrospective is the second-most-important event in Scrum (the first being the sprint planning meeting) because this is your *best chance to improve*!

Exactly. And that's why the retrospective is the *number-one-most-important* thing in Scrum, not the second most important!

Of course, you don't need a retrospective meeting to come up with good ideas – you can do that in your bathtub at home! But will the team accept your idea? Maybe, but the likelihood of getting buy-in from the team is very much higher if the idea comes "from the team", i.e. comes up during the retrospective when everyone is allowed to contribute and discuss the ideas.

Without retrospectives you will find that the team keeps making the same mistakes over and over again.

How we organize retrospectives

The general format varies a bit, but usually we do it something like this:

- We allocate one to three hours depending on how much discussion is anticipated.
- Participants: the product owner, the whole team, and myself.
- We move off to a closed room, a cozy sofa corner, the rooftop patio, or some place like that. As long as we can have undisturbed discussion.
- We usually don't do retrospectives in the team room, since people's attentions will tend to wander.
- Somebody is designated as secretary.
- The Scrum master shows the sprint backlog and, with help from the team, summarizes the sprint. Important events and decisions, etc.
- We do "the rounds". Each person gets a chance to say, without being interrupted, what they thought was good, what they think could have been better, and what they would like to do differently next sprint.
- We look at the estimated vs. actual velocity. If there is a big difference, we try to analyze why.
- When time is almost up, the Scrum master tries to summarize concrete suggestions about what we can do better next sprint.

Our retrospectives are generally not too structured. The underlying theme is always the same though: "What can we do better next sprint?"

Here is a whiteboard example from a recent retrospective:

Three columns:

- **Good***:* If we could redo the same sprint again, we would do these things the same way.
- **Could have done better***:* If we could redo the same sprint again, we would do these things differently.
- **Improvements***:* Concrete ideas about how we could improve in the future.

So columns one and two look into the past, while column three looks into the future.

After the team brainstormed up all these Post-its, they used "dot voting" to determine which improvements to focus on during next sprint. Each team member was given three magnets and invited to vote on whatever improvements they would like the team to prioritize during next sprint. Each team member could distribute the magnets as they like, even placing all three on a single issue.

Based on this, they selected five process improvements to focus on, and will follow this up during next retrospective.

It is important not too get overambitious here. Focus on just a few improvements per sprint.

There are lots of fancy ways to do retrospectives. Vary the format, so the meeting doesn't get stale. You'll find lots of ideas in the book *Agile Retrospectives.* Retromat, a random retrospective generator, is fun, too (www.plans-for-retrospectives.com). :o)

However, I notice that I keep coming back to the simple format described above. It works for the majority of cases. Or even simpler, take a 20-minute coffee break with two discussion topics: "What to keep" and "What to change". A bit shallow, but better than nothing!

Spreading lessons learned between teams

The information that comes up during a sprint retrospective is usually extremely valuable. Is this team having a hard time focusing because the

sales manager keeps kidnapping programmers to participate as "tech experts" in sales meetings? This is important information. Perhaps other teams are having the same problem? Should we be educating the product management more about our products, so they can do the sales support themselves?

Or better yet, invite the sales manager to a meeting, learn about their needs, and discuss possible solutions together!

A sprint retrospective is not only about how this one team can do a better job during next sprint; it has wider implications than that.

Our strategy for handling that is very simple. One person (in this case, me) attends all sprint retrospectives and acts as the knowledge bridge. Quite informal.

An alternative would be to have each Scrum team publish a sprint-retrospective report. We have tried that but found that not many people read such reports, and even fewer act upon them. So we do it the simple way instead.

Important rules for the "knowledge bridge" person:

- He should be a good listener.
- If the retrospective is too silent, he should be prepared to ask simple but well-aimed questions that stimulate discussion within the group. For example, "If you could rewind time and redo this same sprint from day one, what would you do differently?"
- He should be willing to spend time visiting all retrospectives for all teams.
- He should be in some kind of position of authority, so he can act upon improvement suggestions that are outside the team's control.

This works fairly well but there may be other approaches that work a whole lot better. In that case, please enlighten me.

Trading facilitators is a nice pattern. Like "I'll facilitate for your team retrospective if you facilitate for mine." Makes for simple two-way knowledge spread, and also allows you as Scrum master to fully participate in your team's retrospective (rather than facilitate).

To change or not to change

Let's say the team concludes that "we communicated too little within the team, so we kept stepping on each other's toes and messing up each other's designs."

What should you do about it? Introduce daily design meetings? Introduce new tools to ease communication? Add more wiki pages? Well, maybe. But then again, maybe not.

We've found that, in many cases, just identifying a problem clearly is enough for it to solve itself automatically next sprint. Especially if you post the sprint retrospective on the wall in the team room (which we always forget to do – shame on us!). Every change you introduce has some kind of cost so, before introducing changes, consider doing nothing at all and hoping that the problem will disappear (or become smaller) automatically.

The example above ("we communicated too little within the team...") is a typical example of something that may be best solved by doing nothing at all.

If you introduce a new change every time someone complains about something, people may become reluctant to reveal minor problem areas, which would be terrible.

Examples of things that may come up during retrospectives

Here are some examples of typical things that come up during sprint planning, and typical actions.

"We should have spent more time breaking down stories into sub-items and tasks"

This is quite common. Every day at the daily scrum, team members find themselves saying "I don't really know what to do today." So after each daily scrum, you spend time finding concrete tasks. Usually more effective to do that up front.

Typical actions*:* None. The team will probably sort this out themselves during next sprint planning. If this happens repeatedly, increase the sprint-planning time-box.

"Too many external disturbances"

Typical actions:

- Ask the team to reduce their focus factor next sprint, so that they have a more realistic plan
- Ask the team to record disturbances better next sprint. Who disturbed, how long it took. Will make it easier to solve the problem later.
- Ask the team to try to funnel all disturbances to the Scrum master or product owner
- Ask the team to designate one person as "goalkeeper". All disturbances are routed to him, so that the rest of the team can focus. Could be the Scrum master or a rotating position.

The rotating goalkeeper pattern is extremely common and usually works well. Try it!

"We overcommitted and only got half of the stuff done"

Typical actions: None. The team will probably not overcommit next sprint. Or at least not overcommit as badly.

By the way, as of 2014, the term "sprint commitment" is gone entirely from the Scrum Guide. Instead, it's been renamed "sprint forecast". Much better! The word "commitment" has caused so much misunderstanding. Many teams thought the sprint plan was some kind of a promise (a bit silly, considering that one of the four key values in agile is "responding to change over following a plan"). The sprint plan is not a commitment, it's a forecast and a hypothesis – "This is how we think we best can reach the sprint goal."

Nevertheless, it still kind of sucks to consistently deliver less than forecasted. If that is a problem, start strictly applying yesterday's weather, and pull in only as many story points as you got done last sprint (or the

average of the last three sprints if you want to be fancy). This simple and powerful trick usually makes the problem just melt away, as your velocity becomes self-adjusting.

"Our office environment is too noisy and messy"

Typical actions:

- Try to create a better environment, or move the team offsite. Rent a hotel room. Whatever. See pg. 55 "How we arrange the team room").
- If not possible, tell the team to decrease their focus factor next sprint, and to clearly state that this is because of the noisy and messy environment. Hopefully, this will cause the product owner to start pestering upper management about this.

Fortunately, I've never had to threaten to move the team offsite. But I will if I have to. :o)

PART ELEVEN

Slack time
between sprints

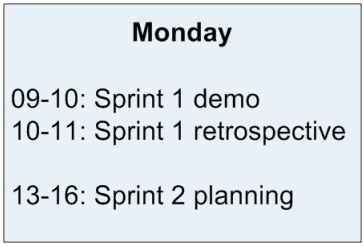
In real life, you can't always sprint. You need to rest between sprints. If you always sprint, you are in effect just jogging.

That applies to life too, not just Scrum! For example, if I didn't have slack in my life, this book wouldn't have existed (nor would this second edition, or any of my other books, articles, videos, etc.). Slack is super-important for both productivity and personal well being! If you're one of those calendar-always-full people, try this: open your calendar and block off half a day per week, write "slack" or "unbookable" or something. Don't decide in advance what you will do with that time, just see what happens. :o)

The same in Scrum and software development in general. Sprints are quite intensive. As a developer, you never really get to slack off; every day you have to stand at that danged meeting and tell everyone what you accomplished yesterday. Few will be inclined to say "I spent most of the day with my feet on the table, browsing blogs and sipping cappuccino."

In addition to the actual rest itself, there is another good reason to have some slack between sprints. After the sprint demo and retrospective, both the team and the product owner will be full of information and ideas to digest. If they immediately run off and start planning the next sprint, chances are nobody will have had a chance to digest any information or lessons learned, the product owner will not have had time to adjust his priorities after the sprint demo, etc.

Bad:



We try to introduce some kind of slack before starting a new sprint (more specifically, the period *after* the sprint retrospective and *before* the next sprint planning meeting). We don't always succeed though.

At the very least, we try to make sure that the sprint retrospective and the subsequent sprint planning meeting don't occur on the same day. Everybody should at least have a good night's sprintless sleep before starting a new sprint.

Better:

Monday	Tuesday
09-10: Sprint 1 demo 10-11: Sprint 1 retrospective	9-13: Sprint 2 planning

Even better:

Friday	Saturday	Sunday	Monday
09-10: Sprint 1 demo 10-11: Sprint 1 retrospective			9-13: Sprint 2 planning

One way to do this is "lab days" (or whatever you choose to call them). That is, days where developers are allowed to do essentially whatever they want (OK, I admit, inspired by Google). For example, read up on the latest tools and APIs, study for a certification, discuss nerdy stuff with colleagues, code a hobby project, etc.

Our goal is to have a lab day between each sprint. That way you get a natural rest between sprints, and you will have a dev team that gets a realistic chance to keep their knowledge up to date. Plus it's a pretty attractive employment benefit.

Best?

Thursday	Friday	Saturday	Sunday	Monday
09-10: Sprint 1 demo 10-11: Sprint 1 retrospective	LAB DAY			9-13: Sprint 2 planning

Currently, we have lab days once per month. The first Friday every month to be specific. Why not between sprints instead? Well, because I felt it was important that the whole company takes the lab day at the same time. Otherwise, people tend to not take it seriously. And since we (so far) don't have aligned sprints across all products, I had to select a sprint-independent lab-day interval instead.

We might some day try to synchronize the sprints across all products (i.e. same sprint start and end date for all products and teams). In that case we will definitely place a lab day between each sprint.

After lots of experimentation at Spotify, we ended up doing company-wide hack weeks. Twice per year, we do a whole week of do-whatever-you-want, with a demo and party on Friday. The whole company, not just the tech folks. The amount of innovation this triggers is just amazing! And because everyone is doing it at the same time, teams are less likely to be derailed by dependencies. I made a video about Spotify's engineering culture that describes the hack week and many other things. Check it out at http://tinyurl.com/spotifyagile

PART TWELVE

How we do release planning and fixed-price contracts

Sometimes, we need to plan ahead more than one sprint at a time – typically, in conjunction with a fixed-price contract where we *have* to plan ahead, or else risk signing something that we can't deliver on time.

Typically, release planning for us is an attempt to answer the question "*when,* at *latest,* will we be able to deliver version 1.0 of this new system".

If you *really* want to learn about release planning, I suggest you skip this chapter and instead buy Mike Cohn's book, *Agile Estimating and Planning.* I really wish I had read that book earlier (I read it *after* we had figured this stuff out on our own...). My version of release planning is a bit simplistic but should serve as a good starting point.

Also check out the book *Lean Startup,* by Eric Ries. A big problem is that most projects tend to build up to a big-bang release, instead of delivering small increments and measuring to see if they are on the right track. Lean startup, if applied properly, radically reduces the risk and cost of failure.

Define your acceptance thresholds

In addition to the usual product backlog, the product owner defines a list of *acceptance thresholds,* which is a simple classification of what the importance levels in the product backlog actually mean in terms of the contract.

Here's an example of acceptance threshold rules:

All items with importance >= 100 *must* be included in version 1.0, or else we'll be fined to death.

- All items with importance 50-99 *should* be included in version 1.0, but we *might* be able to get away with doing them in a quick follow-up release.
- Items with importance 25-49 are required, but can be done in a follow-up release 1.1.
- Items with importance < 25 are speculative and might never be needed at all.

And here's an example of a product backlog, color-coded based on the above rules.

Importance	Name
130	*banana*
120	*apple*
115	*orange*
110	*guava*
100	*pear*
95	*raisin*
80	*peanut*
70	*donut*
60	*onion*
40	*grapefruit*
35	*papaya*
10	*blueberry*
10	*peach*

Red = *must* be included in version 1.0 (banana to pear)

Yellow = *should* be included in version 1.0 (raisin to onion)

Green = may be done later (grapefruit to peach)

So if we deliver everything from banana to onion by the deadline, we're safe. If time runs short, we *might* get away with skipping raisin, peanut, donut, or onion. Everything below onion is bonus.

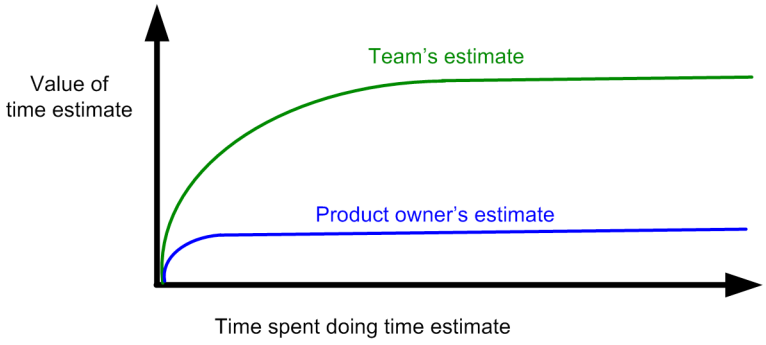
And you can, of course, do this analysis without having numeric importance ratings! Just order the list. But you got that already.

Time-estimate the most important items

In order to do release planning, the product owner needs estimates, at least for all stories that are included in the contract. Just like when sprint planning, this is cooperative effort between the product owner and the team – the team estimates, the product owner describes the items and answers questions.

A time estimate is *valuable* if it turns out to be close to correct, less valuable if it turns out to be off by, say, a factor 30%, and completely worthless if it doesn't have any connection to reality.

Here's my take on the value of a time estimate in relation to who calculates it and how long time they spend doing it.



All that was just a long-winded way of saying:

- Let the *team* do the estimates.
- Don't make them spend too much time.
- Make sure they understand that the time estimates are *crude estimates*, not *commitments*.

Usually, the product owner gathers the whole team in a room, provides some refreshments, and tells the team that the goal of this meeting is to time-estimate the top 20 (or whatever) stories in the product backlog. He goes through each story once, and then lets the team get to work. The product owner stays in the room to answer questions and clarify the scope for each item as necessary. Just like when doing sprint planning, the "how to demo" field is a very useful way to lessen the risk of misunderstanding.

This meeting must be strictly time-boxed, otherwise teams tend to spend too much time estimating too few stories.

If the product owner wants more time spent on this, he simply schedules another meeting later. The team must make sure that the impact of these meetings on their current sprints is clearly visible to the product owner, so that he understands that their time-estimating work doesn't come for free.

Here is an example of how the time estimates might end up (in story points):

Imp	Name	Estimate
130	*banana*	*12*

120	*apple*	*9*
115	*orange*	*20*
110	*guava*	*8*
100	*pear*	*20*
95	*raisin*	*12*
80	*peanut*	*10*
70	*donut*	*8*
60	*onion*	*10*
40	*grapefruit*	*14*
35	*papaya*	*4*
10	*blueberry*	
10	*peach*	

Very confusing to call those "time estimates". Call them "size estimates" instead. I don't know how long "banana" will take, but I'm pretty sure it's a bit longer than "apple" and a lot shorter than "orange". Better to be *roughly right* than *precisely wrong*!

Estimate velocity

OK, so now we have some kind of crude time estimate for the most important stories.

Next step is to estimate our average velocity per sprint.

This means we need to decide on our focus factor. See pg. 24 "How does the team decide which stories to include in the sprint".

Oh no, not that darn focus factor again….

Focus factor is basically "how much of the team's time is spent focusing on their currently committed stories". It is never 100% since teams lose time doing unplanned items, doing context switches, helping other teams, checking their email, fixing their broken computers, arguing politics in the kitchen, etc.

Let's say we determine focus factor for the team to be 50% (quite low – we normally hover around 70%). And let's say our sprint length will be three weeks (15 days) long and our team size is six.

Each sprint is thus 90 man-days long, but can only be expected to complete 45 man-days worth of stories (due to the 50% focus factor).

So our estimated velocity is 45 story points.

Skip focus factor. Ask the team to stare at the list and make an educated guess of how far they can get in one sprint. Count up the points. That will be faster than focus factor, and about as accurate/inaccurate. Better to be roughly right than… – oh, wait. I already said that.

If each story had a time estimate of five days (which they don't) then this team would crank out approximately nine stories per sprint.

Put it together into a release plan

Now that we have time estimates and a velocity (45) we can easily chop the product backlog into sprints:

Imp	Name	Estimate
Sprint 1		
130	*banana*	*12*
120	*apple*	*9*
115	*orange*	*20*
Sprint 2		
110	*guava*	*8*
100	*pear*	*20*
95	*raisin*	*12*
Sprint 3		
80	*peanut*	*10*
70	*donut*	*8*
60	*onion*	*10*
40	*grapefruit*	*14*
Sprint 4		
35	*papaya*	*4*
10	*blueberry*	
10	*peach*	

Each sprint includes as many stories as possible without exceeding the estimated velocity of 45.

Now we can see that we'll probably need three sprints to finish all the "must haves" and "should haves".

Three sprints = nine calendar weeks = two calendar months. Now, is that the deadline we promise the customer? Depends entirely on the nature of the contract – how fixed the scope is etc. We usually add a significant buffer to protect against bad time estimates, unexpected problems, unexpected

features, etc. So in this case, we might agree to set the delivery date to three months in the future, giving us one month "reserve".

Here's an alternative approach that works nicely. Estimate velocity as a range (30-50 points). Then split the backlog into three lists:

All: These stories will all be done, even if our velocity is low (30).

Some: Some of these stories will be done, but not all.

None: None of these stories will be done, even if our velocity is high (50).

The nice thing is that we can demonstrate something usable to the customer every three weeks and invite him to change the requirements as we go along (depending of course on how the contract looks).

Adapting the release plan

Reality will not adapt itself to a plan, so it must be the other way around.

After each sprint, we look at the actual velocity for that sprint. If the actual velocity was very different from the estimated velocity, we revise the estimated velocity for future sprints and update the release plan. If this puts us into trouble, the product owner may start negotiating with the customer or start checking how he can reduce scope without breaking the contract. Or perhaps he and the team come up with some way to increase velocity or increase focus factor by removing some serious impediment that was identified during the sprint.

The product owner might call the customer and say "Hi, we're running a bit behind schedule but I believe we can make the deadline if we just remove the embedded Pac-Man feature that takes a lot of time to build. We can add it in the follow-up release three weeks after the first release if you like."

Not good news to the customer perhaps, but at least we are being honest and giving the customer an early choice – should we deliver the most important stuff on time or deliver everything late? Usually not a hard choice. :o)

I made a 15-minute video called "Agile Product Ownership in a Nutshell". It contains a bunch of useful tips and tricks around release planning and backlog management in Scrum. Check it out!

http://tinyurl.com/ponutshell

PART THIRTEEN

How we combine Scrum with XP

To say that Scrum and XP (eXtreme Programming) can be fruitfully combined is not really a controversial statement. Most of the stuff I see on the Net supports that hypothesis, so I won't spend time arguing why.

Well, I will mention one thing. Scrum focuses on management and organization practices while XP focuses mostly on actual programming practices. That's why they work well together – they address different areas and complement each other.

I hereby add my voice to the existing empirical evidence that Scrum and XP can be fruitfully combined!

I learned from Jeff Sutherland that the first Scrum actually did all the XP practices. But Ken Schwaber convinced him to leave the engineering practices out of Scrum, to keep the model simple and let the teams take responsibility for the tech practices themselves. Perhaps this helped spread Scrum faster, but the downside is that a lot of teams suffer because they lack the technical practices that enable sustainable agile development.

I'm going to highlight some of the more valuable XP practices and how they apply to our day-to-day work. Not all our teams have managed to adopt all practices, but in total we've experimented with most aspects of the XP/Scrum combination. Some XP practices are directly addressed by Scrum and can be seen as overlapping, for example "whole team", "sit together", "stories", and "planning game". In those cases, we've simply stuck to Scrum.

Pair programming

We started doing this lately in one of our teams. Works quite well, actually. Most of our other teams still don't pair-program very much but, having actually tried it in one team for a few sprints now, I'm inspired to try to coach more teams into giving it a shot.

Some conclusions so far about pair programming:

- Pair programming does improve code quality.
- Pair programming does improve team focus (for example, when the guy behind you says "Hey, is that stuff really necessary for this sprint?").

- Surprisingly, many developers that are strongly against pair programming actually haven't tried it, and quickly learn to like it once they do try it.
- Pair programming is exhausting and should not be done all day.
- Shifting pairs frequently is good.
- Pair programming does improve knowledge spread within the group. Surprisingly fast, too.
- Some people just aren't comfortable with pair programming. Don't throw out an excellent programmer just because he isn't comfortable with pair programming.
- Code review is an OK alternative to pair programming.
- The "navigator" (the guy not using the keyboard) should have a computer of his own, as well. Not for development, but for doing little spikes when necessary, browsing documentation when the "driver" (the guy at the keyboard) gets stuck, etc.
- Don't force pair programming upon people. Encourage people and provide the right tools but let them experiment with it at their own pace.

Test-driven development (TDD)

Amen! This, to me, is more important than both Scrum and XP. You can take my house and my TV and my dog, but don't try to stop me from doing TDD! If you don't like TDD then don't let me in the building, because I will try to sneak it in one way or another. :o)

OK, I'm not so religious about this anymore. I've realized that TDD is a pretty niche technique that very few people have the patience to master. Instead, I teach the techniques and then let teams decide how much of it to do, and when.

Here's a 10-second summary of TDD:

Test-driven development means that you write an automated test, then you write just enough code to make that one test pass, then you refactor

the code primarily to improve readability and remove duplication. Rinse and repeat.

Some reflections on test-driven development:

- TDD is *hard*. It takes a while for a programmer to *get it*. In fact, in many cases, it doesn't really matter how much you teach and coach and demonstrate – in many cases, the only way for a programmer to *get it* is to have him pair-program with somebody else who is good at TDD. Once a programmer does *get it,* however, he will usually be severely infected and will never want to work in any other way.
- TDD has a profoundly positive effect on system design.
- It takes time to get TDD up and running effectively in a new product, especially black-box integration tests, but the return on investment is *fast*.
- Make sure you invest the time necessary to make it *easy* to write tests. This means getting the right tools, educating people, providing the right utility classes or base classes, etc.

Since TDD is so hard, I don't try to force it on people, instead I coach these principles:

1) Make sure each key feature has at least one end-to-end acceptance test, interacting through the GUI or just behind it.

2) Make sure any complex or business-critical code is covered by unit tests.

3) This will leave some code uncovered. That's fine. But be aware of which code isn't covered; make sure it's a deliberate tradeoff rather than just neglect.

4) Write the tests as you go, don't save them for later (you'll be just as busy later as you are now).

That seems to give enough sustainability without requiring the hardcoreness of full TDD. Test coverage usually ends up around 70%, because of the law of diminishing returns. In short, test automation is crucial, but TDD is optional.

We use the following tools for test-driven development:

- jUnit/httpUnit/jWebUnit. We are considering TestNG and Selenium.
- HSQLDB as an embedded in-memory DB for testing purposes.
- Jetty as an embedded in-memory web container for testing purposes.
- Cobertura for test coverage metrics.
- Spring framework for wiring up different types of test fixtures (with mocks, without mocks, with external database, with in-memory database, etc.).

In our most sophisticated products (from a TDD perspective), we have automated black-box acceptance tests. These tests start up the whole system in memory, including databases and webservers, and access the system using only its public interfaces (for example HTTP).

This way of working is easier now than it used to be, because of all the slick testing tools and frameworks available. Very useful, whether or not you do TDD.

This makes for extremely fast develop-build-test cycles. This also acts as a safety net, giving the developers confidence enough to refactor often, which means the design stays clean and simple even as the system grows.

TDD on new code

We do TDD for all new development, even if that means initial project setup takes longer (since we need more tools and support for test harnesses etc.). That's a bit of a no-brainer; the benefits are so great that there really is no excuse *not* to do TDD.

TDD on old code

TDD is hard, but trying to do TDD on a codebase that wasn't built using TDD from start... that's *really hard*! Why? Well, actually, I could write many pages on this topic so I think I'll stop here. I'll save that for my next paper "TDD from the Trenches". :o)

Never got around to writing that…. There's no lack of TDD books though! And a great one about legacy code called *Working Effectively with Legacy Code,* by Michael Feathers, a real classic. I've also written some articles on technical debt, check my blog.

http://blog.crisp.se/tag/technical-debt

We spent quite a lot of time trying to automate integration testing in one of our more complex systems, a codebase that had been around for a while and was in a severely messed-up state and completely devoid of tests.

For every release of the system, we had a team of dedicated testers who would perform a whole bunch of complex regression and performance tests. The regression tests were mostly manual work. This significantly slowed down our development and release cycle. Our goal was to automate these tests. After banging our heads against the wall for a few months, however, we hadn't really gotten that much closer.

After that, we switched approach. We conceded to the fact that we were stuck with manual regression testing, and instead starting asking ourselves "How can we make the manual testing process less time consuming?" This was a gaming system, and we realized that a lot of the test team's time was spent doing quite trivial setup tasks, such as browsing around in the back office to set up tournaments for testing purposes, or waiting around for a scheduled tournament to start. So we created utilities for that. Small, easily accessible shortcuts and scripts that did all the grunt work and let the testers focus on the actual testing.

That effort really paid off! In fact, that is probably what we should have done from start. We were too eager to automate the testing that we forgot to do it step by step, where the first step was to build stuff that makes *manual* testing more efficient.

Lesson learned: If you are stuck with having to do manual regression testing, and want to automate this away, don't (unless it is really easy). Instead, build stuff that makes manual regression testing easier. *Then* consider automating the actual testing.

Incremental design

This means keeping the design simple from start and continuously improving it, rather than trying to get it all right from the start and then freezing it.

We're doing fairly well at this, i.e. we spend a reasonable amount of time refactoring and improving existing design, and we rarely spend time doing big up-front designs. Sometimes we screw up, of course –

for example by allowing a shaky design to "dig in" too strongly so that refactoring becomes a big project. But all in all we're fairly satisfied.

Continuous design improvement is mostly an automatic side effect of doing TDD.

Continuous integration

Most of our products have a fairly sophisticated continuous-integration setup based on Maven and QuickBuild. This is extremely valuable and timesaving. It is the ultimate solution to the good ol' "hey, but it works on *my* machine" issue. Our continuous-build server acts as the "judge" or reference point from which to determine the health of all our codebases. Every time someone checks something in to the version-control system, the continuous-build server will wake up, build everything from scratch on a shared server, and run all the tests. If anything goes wrong, it will send an email notifying the entire team that the build failed, including info about exactly which code change broke the build, link to test reports, etc.

Every night, the continuous-build server will rebuild the product from scratch and publish binaries (ears, wars, etc.), documentation, test reports, test-coverage reports, dependency reports, etc. to our internal documentation portal. Some products will also be automatically deployed to a test environment.

Setting this up was *a lot of work*, but worth every minute.

If you take this just a bit further, you get continuous delivery. Every commit is a release candidate, and releasing is a single-click operation. I see more and more teams do this, and I do it for all my personal projects. It's amazingly effective and cool! I suggest you read (or at least browse) the book *Continuous Delivery*. Setting it up is a lot of work, but definitely worth doing at the beginning of any new product. Pays off almost immediately. And there's great tool support nowadays.

Collective code ownership

We encourage collective code ownership but not all teams have adopted this yet. We've found that pair programming with frequent rotation of pairs automatically leads to a high level of collective code ownership. Teams with a high level of collective code ownership have proven to be very robust – for example, their sprint doesn't die just because some key person is sick.

Spotify (and many other fast-moving companies) have an "internal open source" model. All code lives in an internal GitHub and people can clone repos and issue pull requests just like with any public open-source project. Very convenient.

Informative workspace

All teams have access to whiteboards and empty wall space and make quite good use of this. In most rooms, you'll find the walls plastered with all kinds of information about the product and project. The biggest problem is old junk accumulating on the walls – we might introduce a "housekeeper" role in each team.

We encourage the use of task boards, but not all teams have adopted this yet. See pg. 55 "How we arrange the team room".

Coding standard

Lately, we've started defining a coding standard. Very useful, wish we had done it earlier. It takes almost no time at all, just start simple and let it grow. Only write down stuff that isn't obvious to everyone and link to existing material whenever possible.

Most programmers have their own distinct coding style. Little details like how they handle exceptions, how they comment code, when they return null, etc. In some cases, the difference doesn't matter; in other cases it can lead to a severely inconsistent system design and hard-to-read code. A code standard is very useful here, as long as you focus on the stuff that matters.

Here are some examples from our code standard:

You may break any of these rules, but make sure there is a good reason and document it.

Use the Sun code conventions by default: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Never, ever, ever catch exceptions without logging the stack trace or re-throwing. log.debug() is fine, just don't lose that stack trace.

Use setter-based dependency injection to decouple classes from each other (except of course when tight coupling is desirable).

Avoid abbreviations. Well-known abbreviations such as DAO are fine.

Methods that return Collections or arrays should not return null. Return empty collections and arrays instead of null.

Code standards and style guides are great. But there's no need to reinvent the wheel – you can copy this one from my friend Google:

http://google-styleguide.googlecode.com

Sustainable pace/energized work

Many books on agile software development claim that extended overtime is counterproductive in software development.

After some unwilling experimentation on this, I can only agree wholeheartedly!

About a year ago, one of our teams (the biggest team) was working insane amounts of overtime. The quality of the existing code base was dismal and they had to spend most of their time firefighting. The test team (which was also doing overtime) didn't have a chance to do any serious quality assurance. Our users were angry and the tabloids were eating us alive.

After a few months, we had managed to lower people's work hours to decent levels. People worked normal hours (except during project crunches sometimes). And, surprise, productivity and quality improved noticeably.

Of course, reducing the work hours was by no means the *only* aspect that led to the improvement, but we're all convinced it had a large part in it.

I see this over and over again. In software development (and any other complex, creative work), there's very little correlation between hours spent and value delivered. What counts is *focused, motivated* hours. Most of us have experienced the sensation of coming in on a Saturday morning and working in peace and quiet for a couple of hours, and getting like a whole week of work done in that time! That's what I mean by a focused, motivated hour. So don't force people to work overtime, except in the rare exceptional case where it's *really* needed for a short period of time. Plus, burning people out is Evil.

PART FOURTEEN

How we do testing

This is the hardest part. I'm not sure if it's the hardest part of Scrum, or just the hardest part of software development in general.

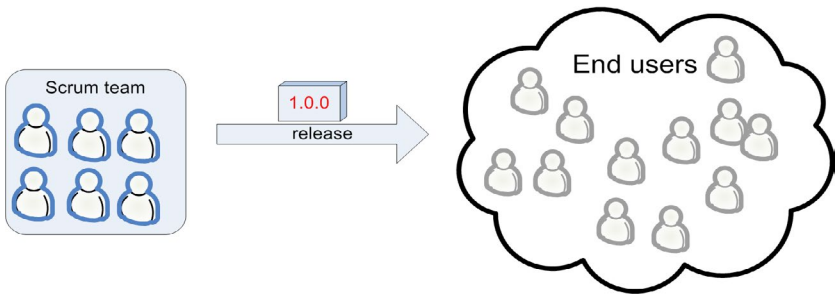
Testing is the part that probably will vary most between different organizations. Depending on how many testers you have, how much test automation you have, what type of system you have (just server and Web app or do you actually ship boxed software?), size of release cycles, how critical the software is (blog server vs. flight control system), etc.

We've experimented quite a lot with how to do testing in Scrum. I'll try to describe what we've been doing and what we've learnt so far.

You probably can't get rid of the acceptance-test phase

In the ideal Scrum world, a sprint results in a potentially deployable version of your system. So just deploy it, right?

> Exactly!



Wrong.

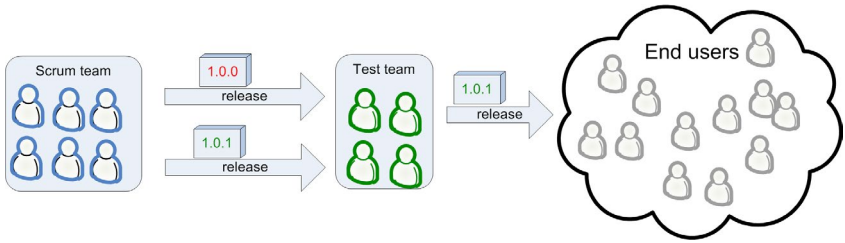
> Huh?

Our experience is that this usually doesn't work. There will be nasty bugs. If quality has any sort of value to you, some kind of manual acceptance testing phase is required.

> What a load of crap! I can't believe I wrote that! And then the book went viral and people all over the world read it and believed my words. Shame on me! Baaaad author, don't do that again! *slap*

Yes, manual testing is important and to some extent is unavoidable. But it should be done *by the team in the sprint*, not handed off to some separate group or saved for a future testing phase. That's why we ditched the waterfall model, remember?

That's when dedicated testers that are *not* part of the team hammer the system with those types of tests that the Scrum team couldn't think of, or didn't have time to do, or didn't have the hardware to do. The testers access the system in exactly the same way as the end users, which means they must be done manually (assuming your system is for human users).



The test team will find bugs, the Scrum team will have to do bug-fix releases, and sooner or later (hopefully sooner) you will be able to release a bug-fixed version 1.0.1 to the end users, rather than the shaky version 1.0.0.

When I say "acceptance-test phase" I am referring to the whole period of testing, debugging, and re-releasing until there is a version good enough for production release.

Minimize the acceptance-test phase

The acceptance test phase hurts. It feels distinctly un-agile.

Exactly! So don't.

Well, OK, I know. In some environments it may seem unavoidable. But my point is, I used to think it was unavoidable. But now I've seen how really agile companies move fast *and* increase quality by getting rid of the separate acceptance-test phase and merging that work into the sprint. So if you think it is unavoidable, it may be that you are blinded by your status quo (as I was). Nevertheless, this chapter provides some useful patterns

for how to deal with separate acceptance testing, as a temporary measure until you manage to merge it all into the sprint. :o)

Although we can't get rid of it, we can (and do) try to minimize it. More specifically, minimize the amount of *time* needed for the acceptance test phase. This is done by:

maximizing the quality of the code delivered by the Scrum team and

maximizing the efficiency of the manual test work (i.e. find the best testers, give them the best tools, make sure they report time-wasting tasks that could be automated)

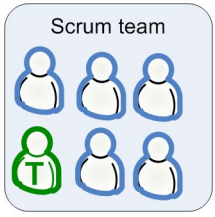
So how do we maximize the quality of the code delivered from the Scrum team? Well, there are lots of ways. Here are two that we find work very well:

Put testers in the Scrum team.

Do less per sprint.

Increase quality by putting testers in the Scrum team

Yes, I hear both objections:



- "But that's obvious! Scrum teams are supposed to be *cross-functional*!"
- "Scrum teams are supposed to be role-less! We can't have a guy who is *only* a tester!"

Let me clarify. What I mean by "tester" in this case is "a guy whose primary skill is testing" rather than "a guy whose role is to do only testing".

This is an important point. "Cross-functional" doesn't mean everyone knows everything. It just means that everyone is willing to do more than just their own thing. We want a nice mix of specialization and cross-functionality. So the whole team is collectively responsible for the quality of their product, and just about everyone will be involved in testing. The tester, however, will guide this work, pair with developers on test automation, and personally do the more complex manual testing.

Developers are often quite lousy testers. *Especially* developers testing their own code.

The tester is the "sign-off guy"

In addition to being "just" a team member, the tester has an important job. He is the sign-off guy. Nothing is considered "done" in a sprint until *he* says it's done. I've found that developers often say something is done when it really isn't. Even if you have a very clear definition of "done" (which you really should, see pg. 32 "Definition of 'done'"), developers will frequently forget it. We programmers are impatient people and want to move on to the next item ASAP.

So how does Mr. T (our tester) know something is done then? Well, first of all, he should (surprise) *test* it! In many cases it turns out that something a developer considered to be "done" wasn't even *possible to test*! Because it wasn't checked in, or wasn't deployed to the test server, or couldn't be started, or whatever. Once Mr. T has tested the feature, he should go through the "done" checklist (if you have one) with the developer. For example, if the definition of done mandates that there should be a release note, then Mr. T checks that there is a release note. If there is some kind of more formal specification for this feature (rare in our case) then Mr. T checks up on that as well, etc.

A nice side effect of this is that the team now has a guy who is perfectly suited to organize the sprint demo.

I'm not a big fan of the sign-off guy pattern any more. It introduces a bottleneck and puts too much responsibility in the hands of one person. But I can see it being useful under some circumstances (it certainly was useful at the time). Also, if anyone should sign off on the quality, it should be a real user.

What does the tester do when there is nothing to test?

This question keeps coming up. Mr. T: "Hey, Scrum master, there's nothing to test at the moment, so what should *I* do?" It may take a week before the team completes the first story, so what should the tester do during *that* time?

Well, first of all, he should be *preparing for tests*. That is, writing test specs, preparing a test environment, etc. So when a developer has something that is ready to test, there should be no waiting, Mr. T should dive right in and start testing.

If the team is doing TDD then people spend time writing test code from day one. The tester should pair-program with developers that are writing test code. If the tester can't program at all, he should still pair-program with developers, except that he should only navigate and let the developer do the typing. A good tester usually comes up with different types of tests than a good developer does, so they complement each other.

If the team is not doing TDD, or if there isn't enough test-case writing to fill up the tester's time, he should simply do whatever he can to help the team achieve the sprint goal. Just like any other team member. If the tester can program then that's great. If not, your team will have to identify all non-programming tasks that need to be done in the sprint.

When breaking down stories into tasks during the sprint planning meeting, the team tends to focus on *programming tasks*. However, usually, there are lots of *non-programming* tasks that need to be done in the sprint. If you spend time trying to *identify the non-programming tasks* during the sprint planning phase, chances are Mr. T will be able to contribute quite a lot, even if he can't program and there is no testing to do right now.

Examples of non-programming tasks that often need to be done in a sprint:

- Set up a test environment.
- Clarify requirements.
- Discuss deployment details with operations.
- Write deployment documents (release notes, RFC, or whatever your organization does).
- Contact with external resources (GUI designers for example).
- Improve build scripts.
- Further break down stories into tasks.
- Identify key questions from the developers and get them answered.

On the converse side, what do we do if Mr. T becomes a bottleneck? Let's say we are on the last day of the sprint and suddenly lots of stuff is done and Mr. T doesn't have a chance to test everything. What do we do? Well we could make everybody in the team into Mr. T's assistants. He decides

which stuff he needs to do himself, and delegates grunt testing to the rest of the team. That's what cross-functional teams are all about!

So yes, Mr. T *does* have a special role in the team, but he is still allowed to do other work, and other team members are still allowed to do his work.

Well put! (I'm allowed to congratulate myself sometimes, OK?) And this is a good way of looking at all other competencies in the team as well.

Increase quality by doing less per sprint

This goes back to the sprint planning meeting. Simply put, don't cram too many stories items into the sprint! If you have quality problems, or long acceptance-test cycles, do less per sprint! This will almost automatically lead to higher quality, shorter acceptance-test cycles, fewer bugs affecting end users, and higher productivity in the long run since the team can focus on new stuff all the time rather than fixing old stuff that keeps breaking.

It is almost always cheaper to build less, but build it stable, rather than to build lots of stuff and then have to do panic hot fixes.

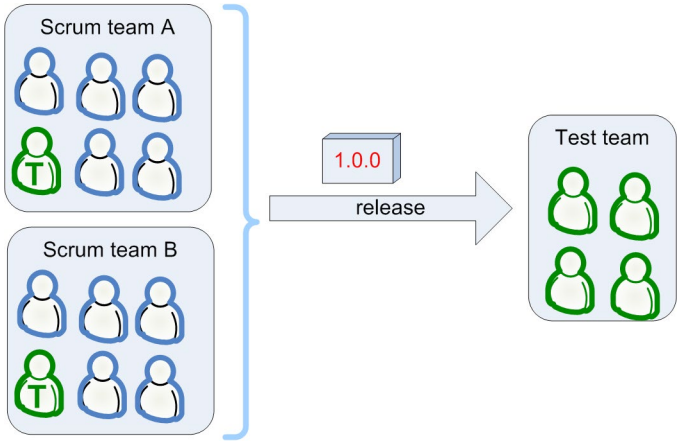
This is just so totally true! I keep seeing it over and over again in teams all around the world.

Should acceptance testing be part of the sprint?

We waver a lot here. Some of our teams include acceptance testing in the sprint. Most of our teams, however, don't, for two reasons:

- A sprint is time-boxed. Acceptance testing (using my definition which includes debugging and re-releasing) is very difficult to time-box. What if time runs out and you still have a critical bug? Are you going to release to production with a critical bug? Are you going to wait until next sprint? In most cases both solutions are unacceptable. So we leave manual acceptance testing outside.

- If you have multiple Scrum teams working on the same product, the manual acceptance testing must be done on the combined result of both team's work. If both teams did manual acceptance within the sprint, you would still need a team to test the final release, which is the integrated build of both teams' work.

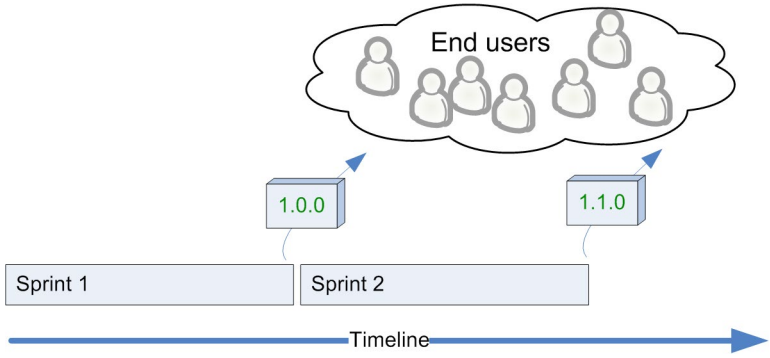


This is by no means a perfect solution but good enough for us in most cases.

Again, strive to make acceptance testing part of each sprint. It takes a while to get there, but you won't regret it. Even if you never get there, the act of trying will cause you to make lots of improvements to the way you work.

Sprint cycles vs. acceptance-test cycles

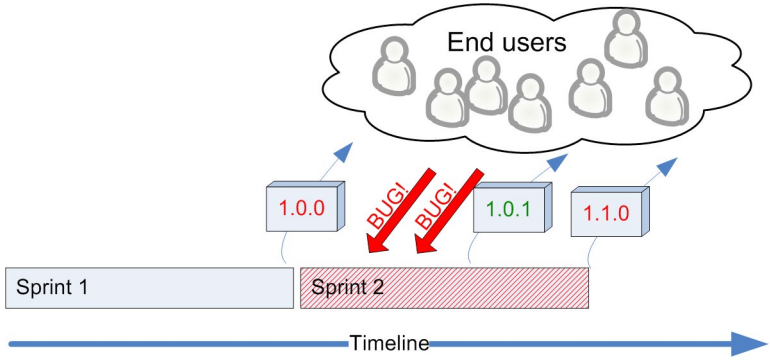
In a perfect McScrum world, you don't need acceptance-test phases since each Scrum team releases a new production-ready version of your system after each sprint.



This can be done! I've seen real-world teams release to production every day, sometimes even several times per day. When a Scrum trainer tells them "You need to have a fully tested, potentially shippable product increment at the end of each sprint," their reaction is "Huh? Why wait so long?"

So no, you don't need a perfect McScrum world for that. Just roll up your sleeves, figure out what's stopping you from getting releaseable code every sprint, and fix the problems one by one. Of course, this can be more or less difficult depending on your domain, but still is worth trying. Just take whatever your release cycle is today (whether it is monthly or yearly or whatever), and gradually but continuously shorten it.

Well, here's a more realistic picture:

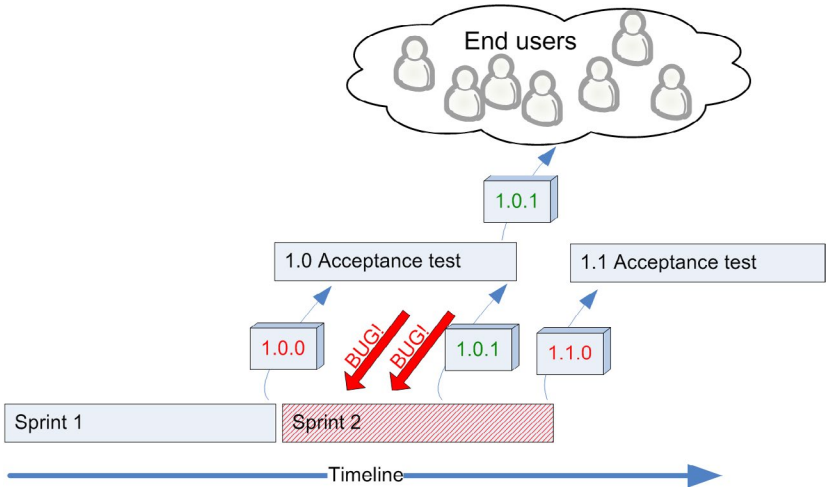


After Sprint 1, a buggy version 1.0.0 is released. During Sprint 2, bug reports start pouring in and the team spends most of its time debugging and is forced to do a mid-sprint bug-fix release 1.0.1. Then at the end of Sprint 2, they release a new feature-version 1.1.0, which of course is even

buggier since they had even less time to get it right this time due to all the disturbances from last release. Etc., etc.

The diagonal red lines in Sprint 2 symbolize chaos.

Not too pretty, eh? Well, the sad thing is that the problem remains even if you have an acceptance-test team. The only difference is that most of the bug reports will come from the test team instead of from angry end users. That's a huge difference from a business perspective, but for developers it amounts to almost the same thing. Except that testers are usually less aggressive than end users. Usually.



We haven't found any simple solution to this problem. We've experimented a lot with different models though.

First of all, again, maximize the quality of the code that the Scrum team releases. The cost of finding and fixing bugs early, within a sprint, is just so extremely low compared to the cost of finding and fixing bugs afterwards.

But the fact remains, even if we can minimize the number of bugs, there will still be bug reports coming after a sprint is complete. How do we deal with that?

Approach 1: "Don't start building new stuff until the old stuff is in production"

Sounds nice, doesn't it? Did you also get that warm fuzzy feeling?

Yes, it's great!

We've been close to adopting this approach several times, and have drawn fancy models of how we would do this. However, we always changed our minds when we realized the downside. We would have to add a non-time-boxed release period between sprints, where we do only testing and debugging until we can make a production release.

Not if your definition of done is "in production". In that case, you can start the next sprint immediately, because the code from last sprint is already in production. It was released continuously during the sprint. A bit extreme, yes, but it can be done.

We didn't like the notion of having non-time-boxed release periods between sprints, mainly because it would break the regular sprint heartbeat. We could no longer say that "every three weeks, we start a new sprint". Besides, this doesn't completely solve the problem. Even if we have a release period, there will be urgent bug reports coming in from time to time, and we have to be prepared to deal with them.

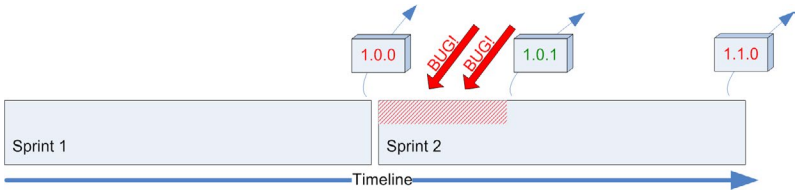
That's actually true. Even if you do manage to release continuously, you still need a way to deal with urgent bugs coming in. Because that will happen sometimes. No team is *so* good as to not have them. And the best way to deal with that is to leave a bit of slack in the sprint.

Approach 2: "OK to start building new stuff, but prioritize getting the old stuff into production"

This is our preferred approach. Right now at least.

Basically, when we finish a sprint, we move on to the next one. But we expect to be spending some time in the next sprint fixing bugs from the last sprint. If the next sprint gets severely damaged because we had to spend so much time fixing bugs from the previous sprint, we evaluate why this happened and how we can improve quality. We make sure sprints are long enough to survive a fair amount of bug fixing from the previous sprint.

Gradually, over a period of many months, the amount of time spent fixing bugs from previous sprints decreased. In addition, we were able to get fewer people involved when bugs did happen, so that the *whole* team didn't need to get disturbed each time. Now we are at a more acceptable level.



During sprint planning meetings, we set the focus factor low enough to account for the time we expect to spend fixing bugs from last sprint. With time, the teams have gotten quite good at estimating this. The velocity metric helps a lot (see pg. 24 "How does the team decide which stories to include in the sprint?").

Or just use yesterday's weather – only pull in as many story points as you completed last sprint or the average completed over the last three sprints. Then your sprint will automatically have built-in slack to handle disruptions and hot fixes. You will automatically limit work to capacity, and move faster as a result (read any book on lean or queuing theory if you need to be convinced that overburdening your sprint is a bad idea).

Bad approach: "Focus on building new stuff"

This in effect means "focus on building new stuff *rather than getting old stuff into production*". Who would want to do that? Yet we made this mistake quite often in the beginning, and I'm sure many other companies do as well. It's a stress-related sickness. Many managers don't really understand that, when all the coding is finished, you are usually still far from production release. At least for complex systems. So the manager (or product owner) asks the team to continue adding new stuff while the backpack of old, almost-ready-to-release code gets heavier and heavier, slowing everything down.

I'm consistently amazed by how many companies get stuck in this trap. All they have to do is limit the number of projects or features in progress. I've seen cases where companies become literally seven times faster by doing that. Imagine that – just as much stuff delivered, but seven times faster, without working harder or hiring people. And better quality as well, because of the shorter feedback loop. Crazy but true.

Don't outrun the slowest link in your chain

Let's say acceptance testing is your slowest link. You have too few testers, or the acceptance-test period takes long because of the dismal code quality.

Let's say your acceptance-test team can test at most three features per week (no, we don't use "features per week" as a metric; I'm using it just for this example). And let's say your developers can develop six new features per week.

It will be tempting for the managers or product owners (or maybe even the team) to schedule development of six new features per week.

Don't! Reality will catch up to you one way or another, and it will hurt.

Instead, schedule three new features per week and spend the rest of the time alleviating the testing bottleneck. For example:

- Have a few developers work as testers instead (oh, they will love you for that...).
- Implement tools and scripts that make testing easier.
- Add more automated test code.
- Increase sprint length and have acceptance tests included in sprint.
- Define some sprints as "test sprints" where the whole team works as an acceptance-test team.
- Hire more testers (even if that means removing developers).

We've tried all of these solutions (except the last one). The best long-term solutions are of course points two and three, i.e. better tools and scripts and test automation.

Retrospectives are a good forum for identifying the slowest link in the chain.

This becomes self-adjusting if acceptance testing is included in the sprint, rather than done separately. Try it – make your definition of done include acceptance testing, and see what happens over time.

Back to reality

I've probably given you the impression that we have testers in all Scrum teams, that we have a huge acceptance test teams for each product that we release after each sprint, etc., etc.

Well, we don't.

We've *sometimes* managed to do this stuff, and we've seen the positive effects of it. But we are still far from an acceptable quality-assurance process, and we still have a lot to learn there.

Indeed, we did have a lot to learn. :o)