

Lean from the Trenches

An example of Kanban in a large software project

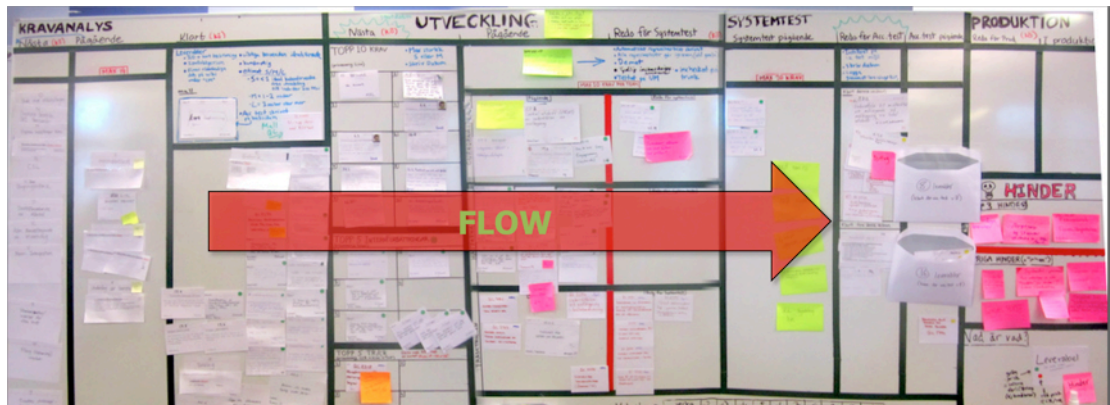
Date: 2011-06-26

Author: Henrik Kniberg

Version: draft 0.7

Todo:

- spelling & grammar
- hyperlinks
- improve some of the graphics
- acknowledgements
- write about the 7 meter long class...
- single page summary?



1. INTRODUCTION	3
DISCLAIMER.....	3
2. WHAT THE PROJECT WAS ABOUT	4
3. TIMELINE	6
4. HOW WE SLICED THE ELEPHANT	7
5. HOW WE INVOLVED THE CUSTOMER & USERS	8
6. HOW WE STRUCTURE THE TEAMS.....	9
7. DAILY COCKTAIL PARTY.....	11
FIRST TIER: FEATURE TEAM DAILY STANDUP	11
SECOND TIER: SYNC MEETINGS PER DISCIPLINE	12
THIRD TIER: PROJECT SYNC MEETING	13
8. THE PROJECT BOARD.....	15
9. OUR CADENCES	17
10. HOW WE HANDLE URGENT ISSUES AND IMPEDIMENTS.....	18
11. HOW WE SCALE THE KANBAN BOARDS.....	21
12. HOW WE TRACK THE HIGH LEVEL GOAL	24
13. HOW WE DEFINE READY AND DONE.....	27
DEFINITION OF "READY FOR DEVELOPMENT".....	27
DEFINITION OF "READY FOR SYSTEM TEST"	28
14. HOW WE HANDLE TECH STORIES.....	30
15. HOW WE HANDLE BUGS.....	33
CONTINUOUS SYSTEM TEST.....	33
FIX THE BUGS IMMEDIATELY!	34
WHY WE LIMIT THE NUMBER OF BUGS IN THE BUG TRACKER	34
HOW WE VISUALIZE BUGS.....	35
HOW WE PREVENT RECURRING BUGS.....	36
16. HOW WE CONTINUOUSLY IMPROVE THE PROCESS.....	39
HOW WE DO TEAM RETROSPECTIVES	39
HOW WE DO PROCESS IMPROVEMENT WORKSHOPS.....	40
HOW WE MANAGE THE RATE OF CHANGE	45
17. HOW WE DISTINGUISH BETWEEN BUFFERS AND WIP.....	48
18. HOW WE USE WIP LIMITS	51
19. HOW WE CAPTURE AND USE PROCESS METRICS	53
VELOCITY (FEATURES PER WEEK)	53
WHY WE DON'T USE STORY POINTS.....	55
CYCLE TIME (WEEKS PER FEATURE).....	55
CUMULATIVE FLOW	60
PROCESS CYCLE EFFICIENCY.....	62
20. HOW WE DO SPRINT PLANNING MEETINGS.....	63
PART 1: BACKLOG GROOMING.....	63
PART 2: TOP 10 SELECTION	64
DOING BACKLOG GROOMING OUTSIDE OF THE SPRINT PLANNING MEETING	64
21. HOW WE DO RELEASE PLANNING.....	65
22. HOW WE DO VERSION CONTROL	66
23. WHY WE USE ONLY PHYSICAL KANBAN BOARDS.....	67
24. GLOSSARY - HOW WE AVOID BUZZWORD BINGO	71
25. FINAL WORDS.....	72

1. Introduction

Many of us have heard about Lean software development, Kanban, and other trendy buzzwords. But what does this stuff actually look like in practice? And how does it scale to a 60-person project developing a really complex system?

I can't tell you how to do it, since every context is different. But I will tell you how we've been doing it (basically a Scrum-XP-Kanban hybrid), and maybe some of our solutions and lessons learned can be valuable in your context.

Don't expect any Lean or Kanban theory in this paper. Well OK maybe just a bit. The rest you can find on the dozens of other books on that topic. This paper is just a real-life example.

Target reader

The primary audience is managers, coaches, and other change agents within software development organizations.

However some parts of this paper will probably be useful to anyone interested in software development, Lean product development, or collaboration techniques in general - regardless of role or industry.

Disclaimer

I don't claim that our way of working is perfectly Lean. Lean is a direction, not a place, it is all about continuous improvement. There is no clear definition of Lean, but many of the practices that we apply are based on the principles of Lean product development as taught by Mary Poppendieck, David Anderson, and Don Reinertsen. Which by the way happen to match Agile principles quite well on most counts :o)

Another thing. I am, just like all other humans, biased. You will see this project from my perspective, a part-time coach during 6 months of this project. There may be important things that I missed, and probably some things that I misunderstood. My goal is not to present a 100% correct picture, just to give you a general idea of what we've been doing and what we've learned so far.

2. What the project was about

RPS (rikspolisstyrelsen) is the Swedish national police authority, and the product we have built is a new digital investigation system called PUST ("Polisens mobila Utrednings STöd"). The basic idea is to equip every police car with a small laptop with mobile internet connection, and a web application that allows them to handle all the investigation work around a prosecution directly in the field.

Suppose a police catches a drunk driver. In the past, the police would have to capture all the information on paper and then drive to the station and do a bunch of administration, and then hand the case over to a separate investigator for further work. This normally took months.

With PUST, the police captures all the information directly on the laptop, which is online and integrated directly with all relevant systems. The case is closed within a few days or even hours.



The system was rolled out nationwide in April 2011 and garnered quite a lot of media attention - the product has been featured in major newspapers, TV, and radio, and so far the response has been overwhelmingly positive.



Average processing time for petty crimes has already become six times faster, and the police can spend more time on the field and less time at the station. This not only reduces the amount of crime, it is also more motivating for the policemen. Police like to do police work, not paperwork!

Furthermore there have been surprisingly few support issues and bug reports, compared to past projects of similar complexity and scale.

PUST is a complicated system because it:

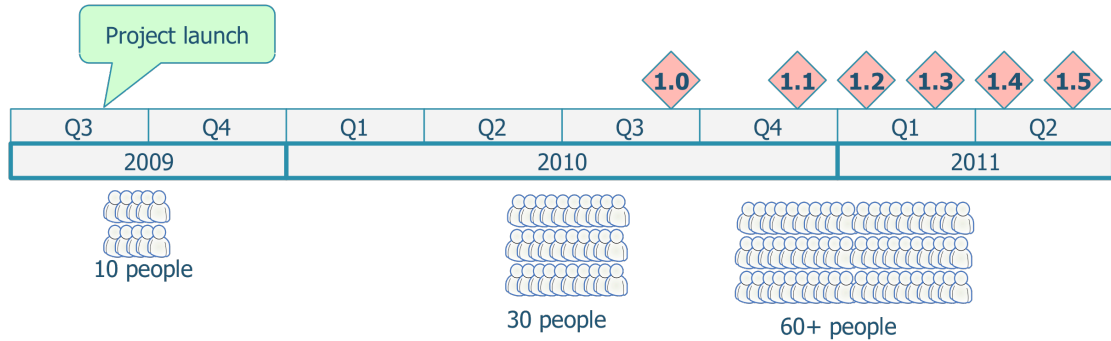
- ... has to integrate with a huge number of legacy systems.
- ... has to be very user friendly since police will be using the system in real-time while doing interrogations.
- ... has to be highly secure.
- ... has to comply with a whole bunch of complicated laws and regulations.

This was a very important project to RPS. The expected benefits were very high, but also the cost and risk. Previous attempts to develop this type of system had failed, so the stakes were high. The positive side of this was that we were allowed to explore new and more effective ways of working, which is what ultimately led to this paper you are reading right now.

PUST is part of a cultural change within RPS, a nation-wide Lean initiative throughout the whole organization. So it made a lot of sense to start applying Lean principles to the development process itself too :o)

3. Timeline

Development started around September 2009. The first release to production (a pilot) happened 1 year later, followed by a series of bi-monthly follow-up releases.



1 year to first release might seem like a long time to Agile folks, but compared to other government projects of the same scope and complexity this was extremely short! Some of these projects have taken up to 7 years until first release!

Release 1.0 was a pilot release, 1.4 was the main nationwide release. Release to production every second month is also a quite unusual concept, many government organizations release only once or twice per year.

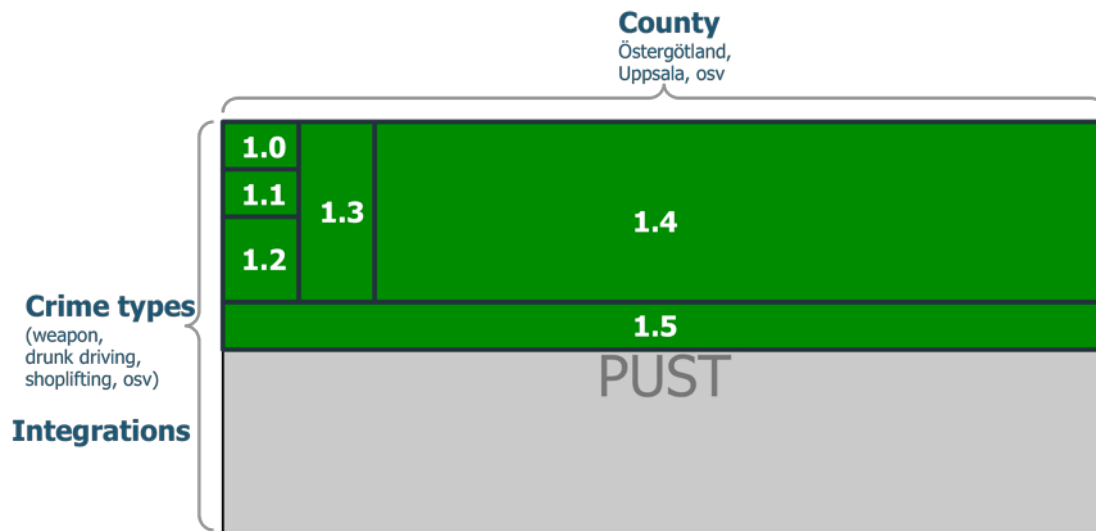
The project started with around 10 people, scaled to around 30 people in mid-2010, and then doubled to 60+ people in Q4 2011.

All these factors - the short release cycles and the aggressive scaling - forced us to quickly evolve the organization and development process.

4. How we sliced the elephant

The key to minimizing risk in large projects is to find a way to "slice the elephant", i.e. find a way to release the system in small increments instead of saving up for a big bang release.

We sliced this elephant across two dimensions - geographic location & type of crime.

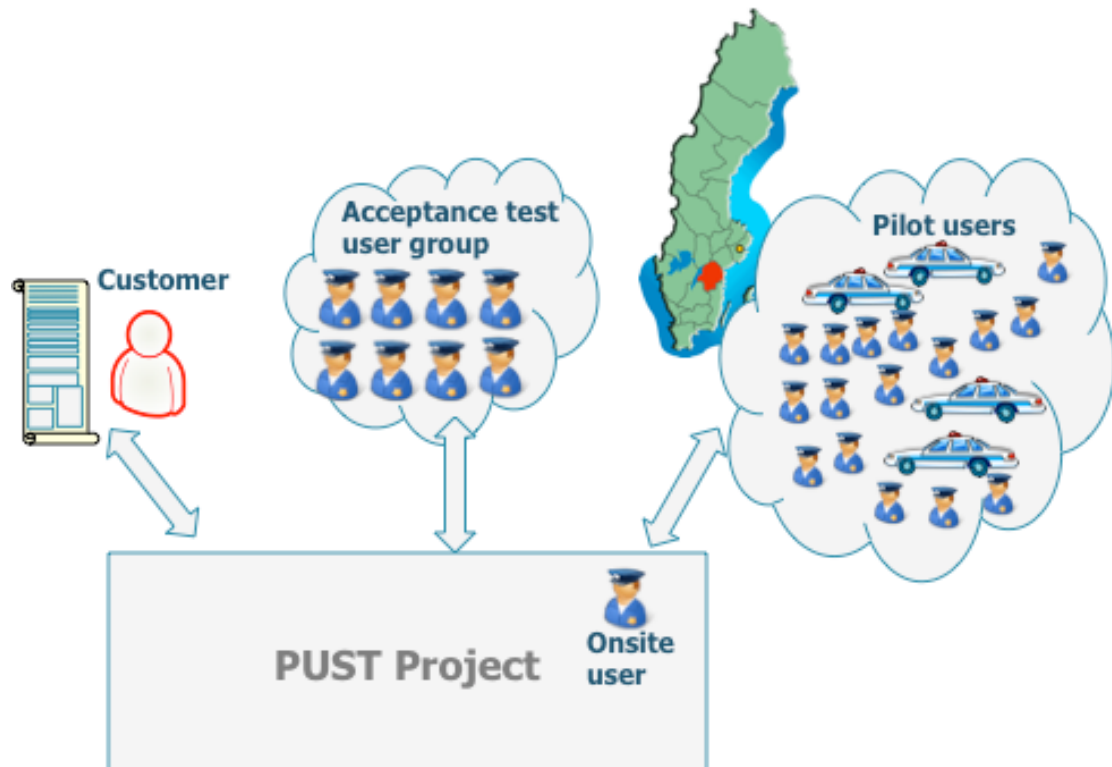


- **1.0-1.2:** Pilot releases to only one county - Östergötland - and supporting only a small number of common crime types such as drunk driving and knife possession. Other crime types were handled the old manual way. For each subsequent release we improved the stability and added more crime types.
- **1.3:** Spread the release to another county - Uppsala.
- **1.4:** Spread the release to the rest of Sweden. This was the "main" release.
- **1.5:** Additional crime types added, added new integrations to various systems.

In addition to the bi-monthly feature releases, we made "patch" releases every few weeks with minor improvements and bug fixes to existing functionality.

5. How we involved the customer & users

If we see the project as a "black box", here is how customer and user engagement looked like:



One person acted as the main "customer" to the project. She had a list of features at a pretty high level. We called them "feature areas", which roughly equates to what agile folks would call "epics".

In addition there was an onsite user, i.e. a real user who was in the building with the development teams. Initially only once per week or so, during later stages we had onsite users almost every day, through a rotating schedule. The onsite users were there to give feedback, see demos, answer questions, and so on.

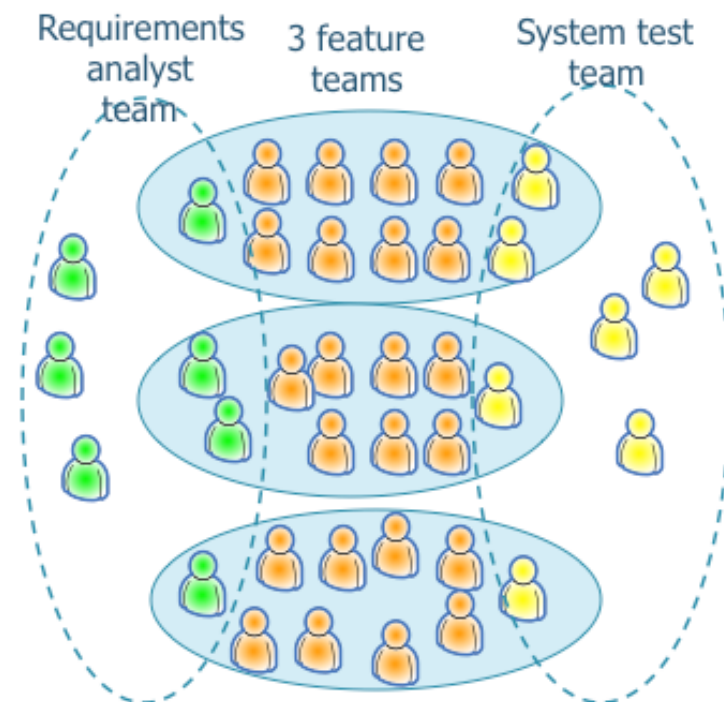
A week before each release we had an acceptance test group come in, typically 10 or so police officers and other real users. This group would spend a couple of days trying out the latest release candidate and giving feedback. Usually the system worked quite well by the time it reached acceptance test, so there were rarely any nasty surprises coming up at that point.

As soon as the first release was out the door we also had a bunch of real users in Östergötland (a county in southern part of Sweden) giving us a continuous stream of feedback.

6. How we structure the teams

When I joined this project as agile/lean coach in December 2010 they had just scaled from 30 to 60+ people and were facing growth pain, with communication and coordination difficulties. Fortunately we were all located on the same floor, everybody in the project was within at most 30 seconds walking distance from each other. As a result we could quite easily experiment with how to organize the project.

As the project scaled we evolved the team structure to something like this:



There are five teams. One requirements team, 3 feature development teams, and one system test team.

The three feature teams are basically Scrum teams, i.e. each team is colocated, cross-functional, self-organized, and capable of developing & testing a whole feature.

The requirements team is a virtual team, in the sense that they don't all sit together. There are essentially three roles within this team:

- Some analysts are embedded in one of the feature teams and follow that team's features all the way through development into test, answering questions and clarifying the requirements along the way.
- Some analysts focus on the "big picture" and aren't embedded in any feature team. They look further into the future to define high level feature areas.
- The rest of the analyst team are flexible and move between the above two roles depending on where the need is the greatest at the moment.

The test team follows a similar virtual team structure, with three roles:

- Some testers are embedded in a feature team and help that team get their software tested and debugged at a feature level.
- Some testers are "big picture" testers and focus on doing high level system tests and integration tests on release candidates as they come out. The person coordinating that work is informally called the "System Test General" :o)
- The rest of the test team members are flexible and move between the above two roles as needed.

In the past the teams were organized by discipline, i.e. we had a distinct requirements team and a distinct test team, and distinct development teams that did not have embedded testers or analysts. That didn't scale very well, as more people were added to the project there were lots of communication problems between the disciplines. There was a tendency to focus on creating detailed documents rather than communicating, and tendency for teams to blame problems on other teams. Each team tended to focus on getting their part of the work done instead of focusing on building the whole product.

The level of collaboration improved dramatically as we evolved to a more Scrum-like structure with cross functional teams of analysts and testers and developers sitting together. We didn't go "all the way" though, we kept some analysts and testers outside of the feature teams, so they could focus on the "big picture" instead of individual features. This scaled quite nicely, and gave us a good balance between feature focus and system focus.

Since the project was fairly large (60+ people) we also have some individuals outside of the teams handling specialist functions & coordination functions. This includes project manager, project administration, configuration manager, e-learning specialist, performance test experts, development manager, etc.

7. Daily cocktail party

If you walk into this project on any day before 10:15 it will feel like walking into a cocktail party! People everywhere, standing in small groups and communicating.



You will see groups standing in semicircles in front of boards, engaged in conversation and moving stickynotes around. People moving between teams, debates going on, decisions being made. Suddenly a group will break apart and some individuals will move to another group to continue the conversation. New groups sometimes form in the hall to follow up on some side conversation.

By 10:15 the daily cocktail party is over and most people are back at their desks.

This may look chaotic at first glance, but it is in fact highly structured.

First tier: Feature team daily standup

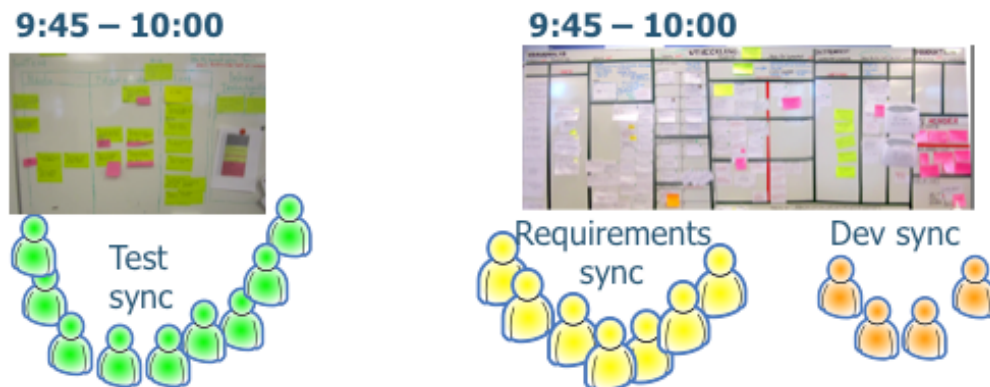
First up are the feature team daily stand-ups.



Two of the teams meet at 9:30, one of the team meets at 9:15 (each team decides their own meeting time). During this everyone on the team stands up in a rough semicircle in front of their taskboard, discussing the work they are doing to do today and any problems and issues that need to be addressed. Some teams use the Scrum formula ("What did I do yesterday", "What am I doing today", "What is blocking me"), others are more informal about it. These meetings usually take 10-15 minutes and are facilitated by a team leader (which equates pretty much to Scrum Master).

Second tier: Sync meetings per discipline

At precisely 9:45, a second set of daily stand-ups take place - sync meetings for each discipline.



All the testers gather in front of a test status board and discuss how to make best use of their time today. The embedded testers have just completed the daily standup within their feature team, so they have fresh information about what is going on within each team.

At the same time, the requirements analysts are having their own sync meeting, including the embedded analysts who just came out of their feature team standup meeting with fresh information.

At the same time, the team leads from each feature team plus the development manager are having their "dev sync meeting". The team leads just came out of their feature team standup meeting with fresh information. They discuss

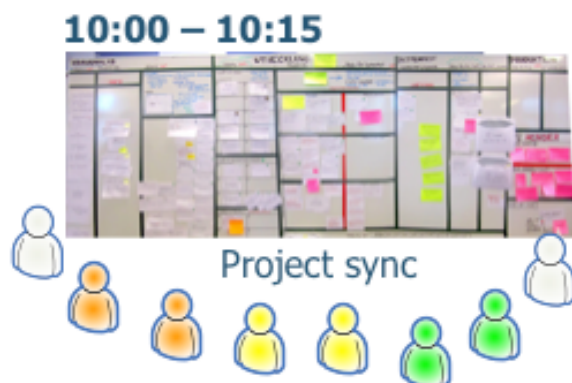
technical issues such as what code will be checked in today, which team will start working on which new feature, and any technical problems and dependencies that need to be resolved.

The test sync meeting takes place in front of a test status board, while the requirements sync and dev sync meetings take place in front of the Project Board, which I will describe for you soon. These three meetings take place in parallel just a few meters from each other, which makes it a bit noisy and chaotic, but the collaboration is very effective. If anybody from one team needs info from another they can just walk over a few meters to the other meeting and ask a question.

Some people (like the project manager and me) float around between the meetings, absorbing what is going on and trying to get a feel for which high level issues need to be resolved. Sometimes we stay outside the meetings, sometimes we get pulled into a discussion.

Third tier: Project sync meeting

Finally, at precisely 10:00, the project sync meeting takes place in front of the Project Board.



The people at this meeting are informally referred to as the "cross-team" ("tvärgrupp" in Swedish), which basically means one person from each discipline and one person from each feature team, plus project manager and CM (configuration manager). A very nice cross-section of the whole project.

The project sync meeting is where we look at the big picture, focusing on the flow of functionality from analysis to production - which team is doing what today? What is blocking our flow right now? Where is the bottleneck, and how can we alleviate it? Where will the bottleneck be next? Are we on track with respect to the release plan? Is there anybody who doesn't know what to do today?

That's it. A total of 7 standup meetings every day, organized into three layers. Each meeting is time boxed to 15 minutes, each meeting has a core set of participants that shows up every day, and each meeting is public so anybody can

visit any meeting if they want to learn what is going on or have something to contribute. And it's all over by 10:15.

If some important topic comes up during a daily and can't be resolved within 15 minutes we schedule a follow-up meeting with the people needed to resolve that issue. Some of the most interesting and valuable discussions take place right after the project sync meeting, as people stand in small clusters dealing with stuff that came up during the daily stand-ups.

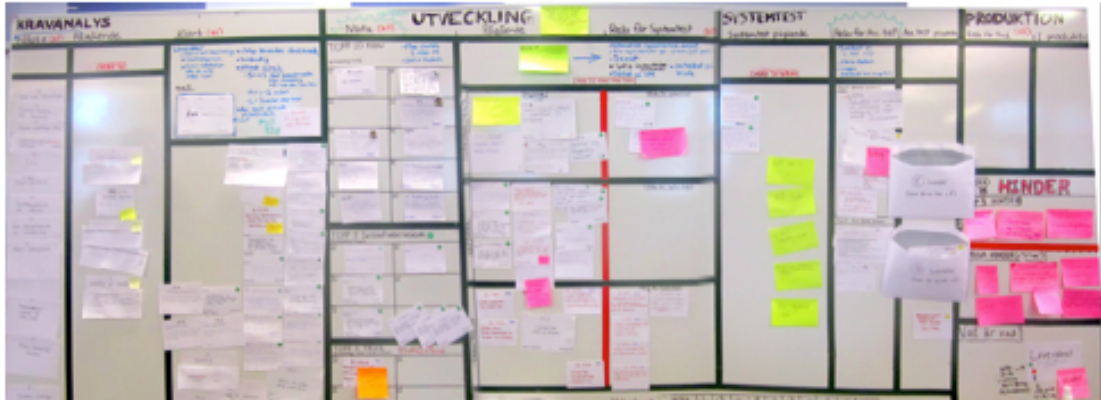
This structure of daily meetings, just like everything else, was something that we gradually evolved into. When we started doing the "daily cocktail party" (which by the way is my term, not an official term we use in the project...) in this fashion I was a bit concerned that people might think it is too many meetings. That turned out not to be the case - the team members insist that these meetings are highly valuable, and I can see that the energy is high and problems are being solved.

Most people only need to go to one meeting. A few people need to go to two meetings. The team lead of a feature team goes to his team standup as well as the dev sync meeting. The embedded tester in a feature team goes to the team standup as well as the test sync meeting, etc. This is a very effective way "linking" the communication channels and making sure important knowledge, information, and decisions propagate quickly throughout the entire project.

It has turned out that many problems that would otherwise result in the creation of documents and process rules could instead be resolved directly at these morning meetings. One concrete example is the issue of how to decide which team is to develop which feature, or how to decide whether to spend our time developing customer-facing functionality today or to implement a customer-invisible improvement to the technical infrastructure. Instead of setting up policy rules for this, the teams simply talk about this during the daily meetings and make decisions on-the-fly based on the current situation. This is the key to staying agile in a big project and not getting bogged down in beaurocracy.

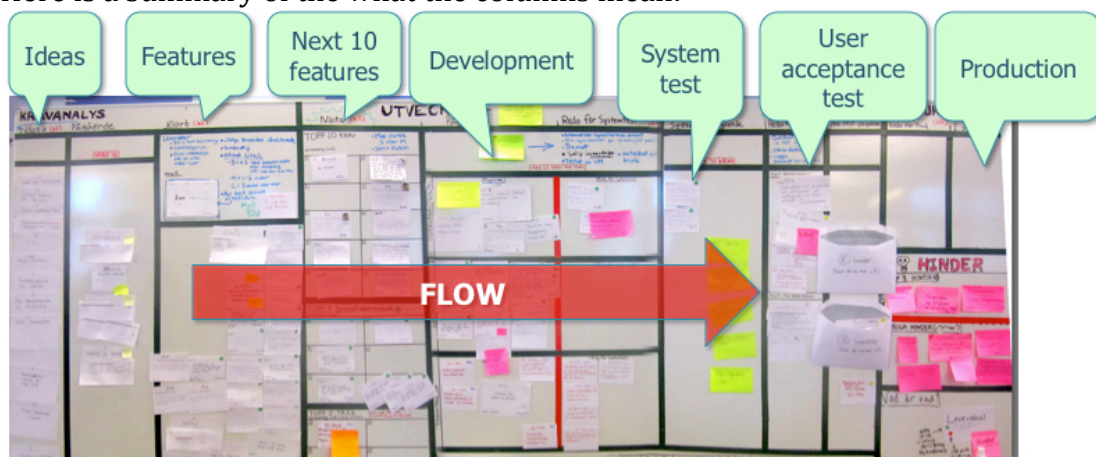
8. The Project Board

The Project Board is the communication hub of the project.



If you are into Kanban you will recognize this as a Kanban system, which means that we track the flow of value from idea to production, and that we limit the amount of work in progress at each step of the process.

Here is a summary of the what the columns mean:



The left-most column is where ideas come in, high level feature areas ("epics") written on index cards. A card gets pulled into the second column ("analysis ongoing") and gets analyzed and split into a user stories at a feature level, which are collected in the third column. So the third corresponds to a Scrum Product Backlog. Most of the feature cards are written in user story format "As X, I want Y, so that Z".

The top 10 features are selected and pulled into the "next 10 features" column, this usually happens at a bi-weekly meeting that corresponds roughly to a Scrum sprint planning meeting (we even call it that).

The three feature teams continuously pull cards from the "next 10 features" column into their own "dev in progress" column when they have capacity, and

into the "ready for system test" column when the feature is developed and tested at a feature level.

On a regular basis the test team will flush the "ready for system test" column and pull all those cards into the "system test in progress" column (and create a corresponding system test branch in the version control system). Once system test is done, they will release to an acceptance test environment, move the cards to the "ready for acceptance test" column, and then start another round of system tests on whatever features have been completed since. This was a big cultural shift - the move from "big system test at the end of the release cycle" to "continuous system test" (but with some batching).

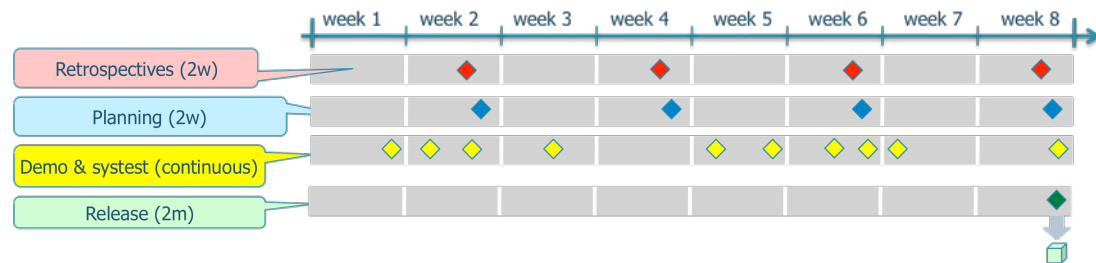
Every second month (roughly) a bunch of real users will show up and spend a couple of days doing acceptance testing (basically just trying the system out and giving feedback), so we move the cards to "acceptance test in progress". When they are done testing and any final bugs have been found and fixed the cards move to "ready for production" and shortly thereafter (when the system has been released) they move to the last column "in production". The cards sit there for a few weeks (so we can celebrate that something got into production), but are then removed to make space for new cards flowing in.

To the casual observer glancing at the board this might look like a waterfall process - requirements analysis => development => system test => acceptance test => production. There is a big difference though: In a waterfall model, the requirements are all completed before development starts, and development is completed before testing starts. In a Kanban system these phases are all going on in parallel. While one set of features is being acceptance-tested by users, another set of features is being system tested, a third set of features are being developed, and a fourth set of features are being analyzed and broken into user stories. It is a continuous flow of value from idea to production.

Well, semi-continuous I should say. In our case it is a more or less continuous flow of value from idea to "ready for acceptance test". New features are released to production roughly every second month and acceptance-tested in conjunction with that, so features sit around in "ready for acceptance test" for a few weeks. Although I hope we can improve this in the future, it has turned out to not be much of a problem. Since we have onsite users giving us feedback during development, it has turned out that by the time a feature reaches "ready for acceptance test" it pretty much works as expected, and there are few serious problems found after that stage.

9. Our cadences

Here is a summary of our cadences:



- Retrospectives happen every second week (every week for some teams)
- Planning happens every second week (approximately)
- Demo & system test is done in a continuous fashion, as features get done
- Release to production is done approximately every second month

We have been evolving more and more towards a Scrum-like model. Initially retrospectives were twice as often as planning meetings, now they happen one day after each other every second week. Demo & reviews are done continuously now but we are considering doing a high level product demo/review every second week. And guess what - doing retrospectives, planning, and demos together in the same cadence is basically the definition of a Scrum sprint :o)

This evolution towards a more and more Scrum-like model was not really intentional, it was just the result of a series of process improvements triggered by real-life problems.

This seems to be a general pattern, I see many Kanban teams that gradually discover (or sometimes rediscover) the value of many of the Scrum practices. In fact, sometimes Kanban teams start doing Kanban because they didn't like Scrum, and then later on discover that Scrum was actually pretty good and their problems had been *exposed* by Scrum, not *caused* by it. Their *real* problem was that they had been doing Scrum too much "by the book" instead of inspecting and adapting it to their context.

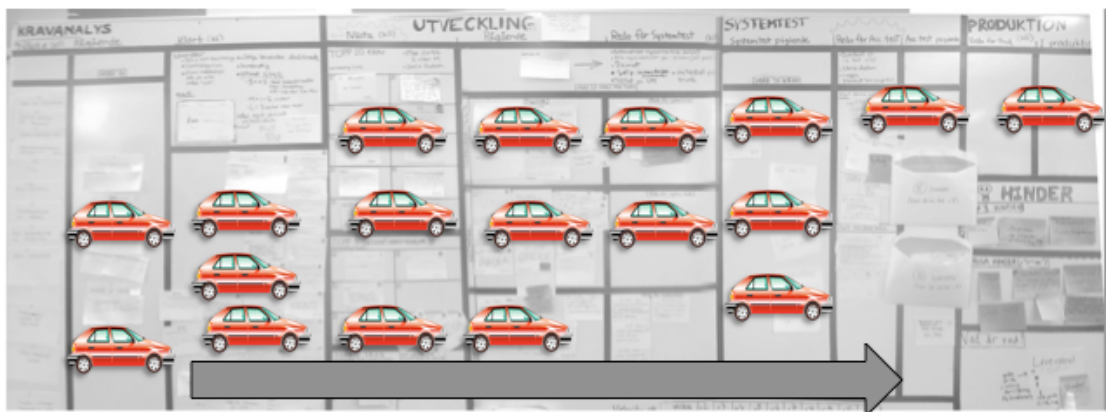
More on that in my other book "Kanban and Scrum - making the most of both".

Now, back to the topic at hand.

10. How we handle urgent issues and impediments

It has been very useful to apply a traffic system metaphor when looking at the project board.

Think of the board as a series of roads, with each card representing a car trying to move across the board from left to right.



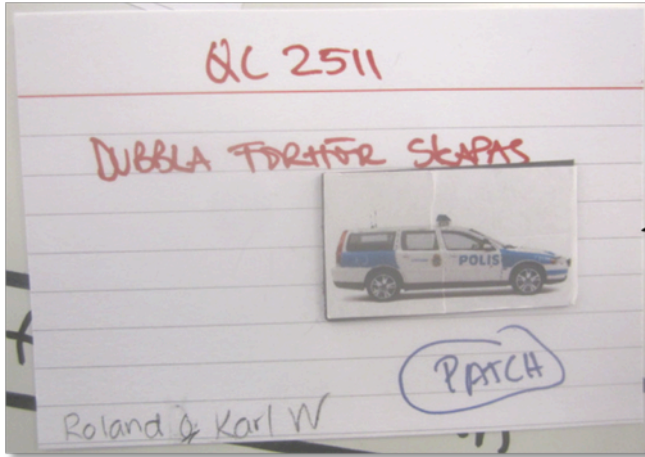
We want to optimize the flow, therefore we DON'T want to fill up the board. We all know what happens to a traffic system when it is 100% full - the traffic system slows to a halt.



We need space, or "slack", to absorb variation and enable fast flow.



Having slack in the system not only enables fast flow, it also enables escalation. Sticking to the metaphor, we use police car magnets to mark items that are urgent and need special treatment to move through the system faster.



Police car = urgent (patch)

We also mark impediments ("road blocks") using pink stickies.

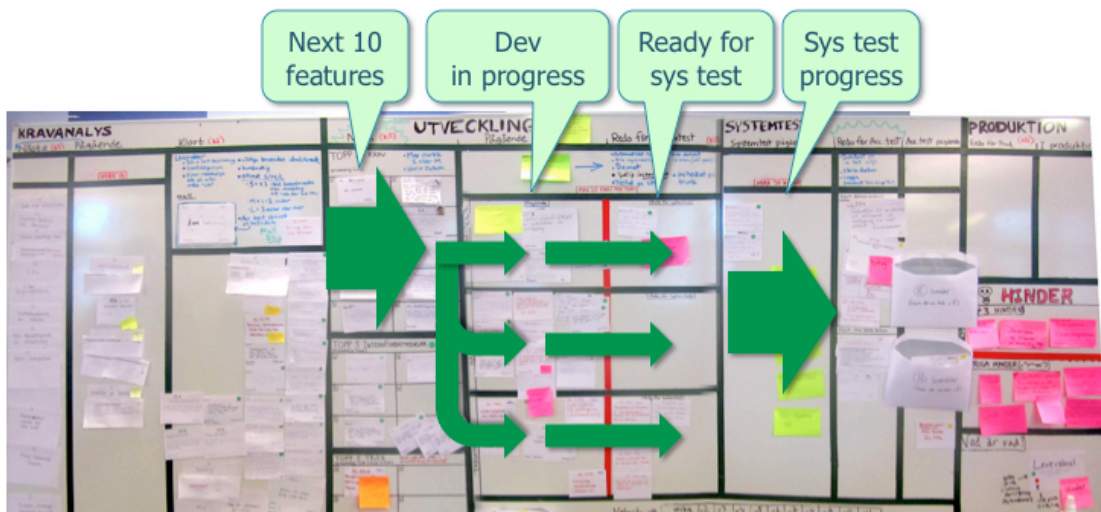


If a specific feature is blocked (for example because we don't have access to a third party system needed to test that feature) then we put a pink sticky on that feature, describing the problem & the date that it started. There is also a section on the right side "top 3 impediments" for more general problems that aren't tied to any specific feature (such as a build environment not working).

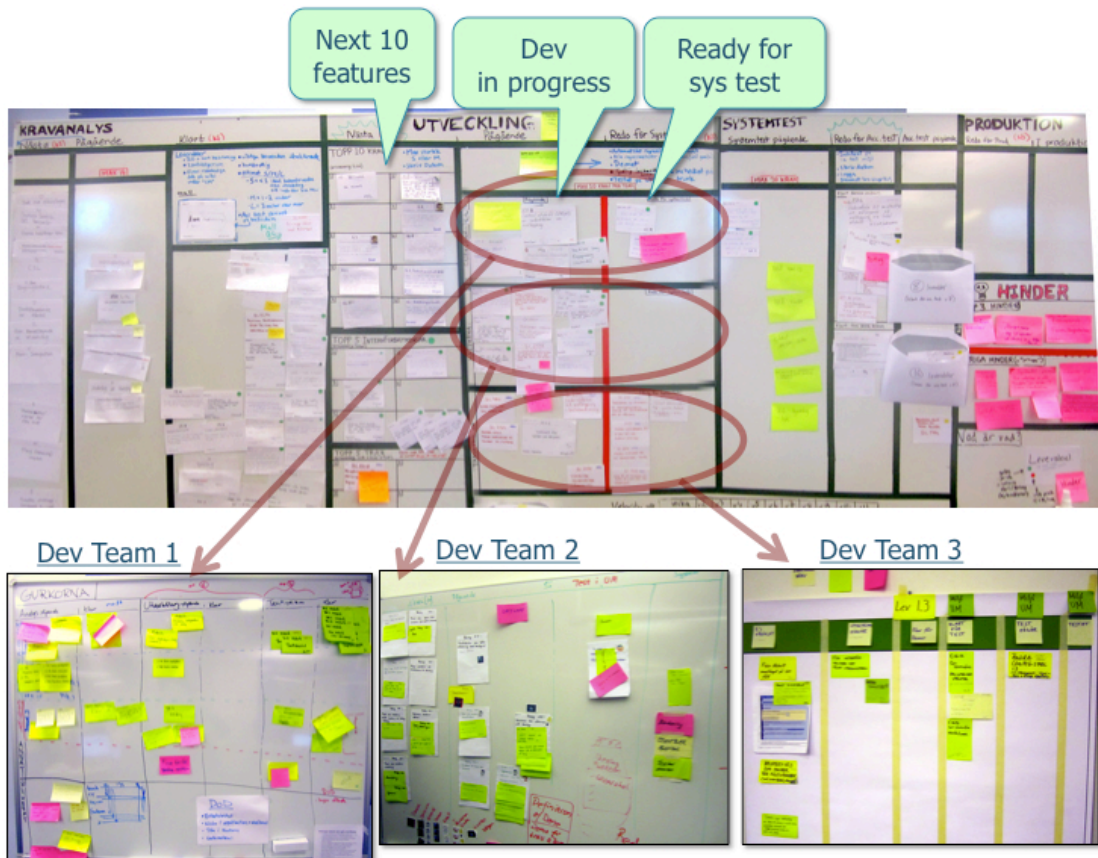
At the daily meetings we focus on removing these blockers. Just as with a traffic system, a blocker that stays around for too long will cause ripple effects throughout the whole system. Plus, nothing will flow faster than the bottleneck section of the road so we should focus all efforts on resolving these.

11. How we scale the Kanban boards

The development columns on the project board are split into three horizontal swimlanes, one for each feature team. Each feature flow from "Next 10 features" into one of the three development teams. When that team has developed the feature and tested it at the feature level it goes to "Ready for system test". When the system test team is finished with their current round of system tests they will pull all cards from each team's "Ready for system test" into the combined "System test in progress" column.



Whenever a team pulls in a feature from "Next 10" to "Development in progress", they clone that feature card and put it on their own team-internal board.



The team then breaks the work into tasks and writes those down as sticky notes tied to that feature. This is typically done in conjunction with an "analysis meeting", where requirements analysts, testers, and developers collaborate to sketch out the design of this feature and identify the key tasks involved. Each task is normally a concrete verb such as "write the GUI code" or "set up the DB tables" or "design the protocol".

Most teams also have avatar magnets to indicate who is working on which task. Your avatar says everything about your personality :o)



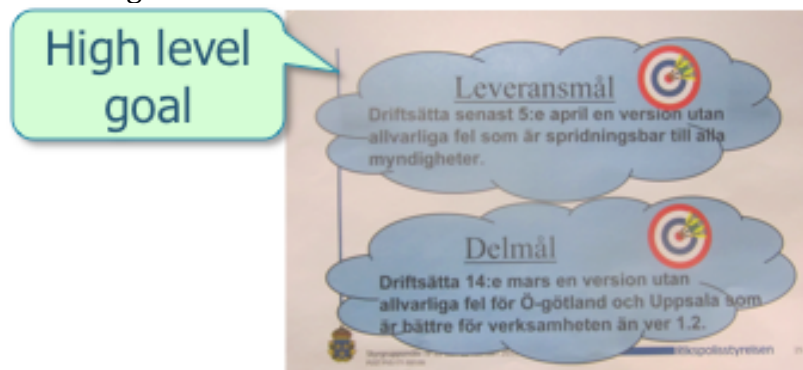
So the Project board contains feature cards, and each feature team has their own board with the features they are working on plus the associated task breakdown. Imagine that you "double click" on a feature on the project board, and you will "zoom in" to that feature team's board and see which tasks are involved in that feature, and who is working on what task.

As you can see in the pictures above, each team has their own board layout. We have not attempted to standardize this, instead we have let each team find whatever board structure works best for them. Most teams do have WIP limits & Definition of Done & avatar magnets on their boards though.

This 2-tier system of physical Kanban boards has turned out to work very well, although there was some initial confusion about how to keep everything in sync. It is clear that the boards have become a point of gravity in the project, teams naturally gather around their board whenever they need to synchronize work or solve problems. Most team members focus on their team-level board, while team leads and managers focus on both the team-level board and the project level board. As time passed more and more team members have started showing interest in the project level board, which is a good indicator that people are focusing on the big picture rather than just on their own work.

12. How we track the high level goal

The high level project goal is usually posted on the Kanban board. For example during Q1 2011 we had the goal "Deliver on April 5 a version with no important defects, that is releasable to the whole country", and a milestone along that path was to deliver to two counties on March 14. During different periods we've had different goals.



Once per week or so we do a reality check. Typically the project manager asks at the project sync meeting "Do you believe we will reach this goal?" and everybody gets to write down a number from one to five (sometimes we just hold up fingers).

- 5 = Definitely!
- 4 = Probably
- 3 = Barely
- 2 = Probably not
- 1 = Forget it!

Here is an example:

Klarar vi målet?

5 Definitivt!		
4 Troligvis		
3 På grensen		
2 Troligvis ikke		
1 Definitivt ikke!		

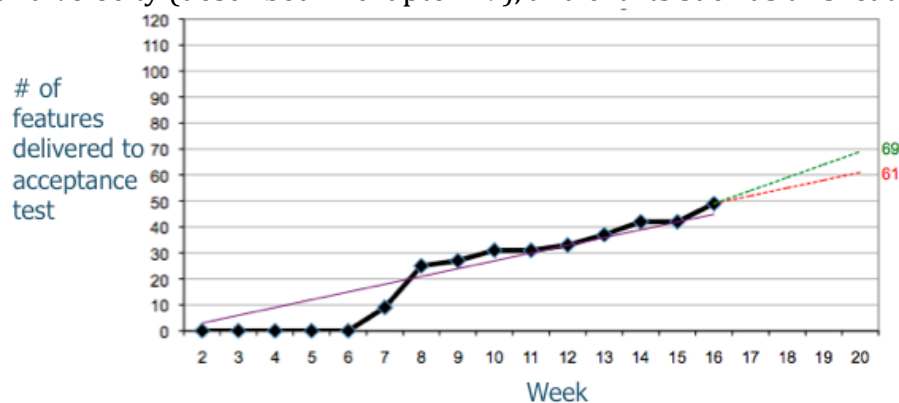
The confidence check was done every week, and this paper shows three rounds of votes. The first week (leftmost column of votes) there was low confidence in the goal, next week it increased and the week after that it was all fives!

If we start seeing 2's and 1's then we reevaluate the goal and discuss what needs to change in order to improve our confidence. Typically one of the following:

- Removing an impediment
- Alleviating a bottleneck
- Reducing scope
- Adjusting the goal
- Working harder

Any of the first four are preferable to the last, since the root cause of the problem is usually *not* that people aren't working hard. In fact, sometimes the root cause of the problem is that we are working *too* hard, and not taking time to think.

The votes are mostly based on gut feel, but also to a certain extent based on visible information such as the cards on the board, metrics such as cycle time and velocity (described in chapter 19), and charts such as this feature burnup:



13. How we define Ready and Done

The blue text at the top of most columns on the project board is the "definition of done" for that column (which also means "definition of ready" for the subsequent column). The two most important ones are "Definition of Ready For Development" and "Definition of Ready for System Test", since that is where we used to have the most problems.



Definition of "Ready for Development"

The "Ready for development" column essentially means "Here's a bunch of features that we have broken down and estimated and clarified, but we haven't yet decided which of these we are going to develop and in which order." So this corresponds roughly to a Scrum Product Backlog.

In order for a feature to be ready for development it must:

- Have an **ID**. The ID is used as a key when looking up more information about this feature, in case there are any associated use case specifications or other documents. On the wiki there is list of IDs, click on one to see any attached info.
- Have a **contact person**. Typically the requirements analyst who has most knowledge about this feature.
- Be **valuable** to customers. When breaking down epics into deliverable stories we want to make sure that we haven't lost the customer value along the way. The requirements analysts have the final say on this matter.

- Be **estimated** by the team. We use t-shirt sizes (Small, Medium, Large). The estimates are normally done by a small group consisting of a tester, a developer, and a requirements analyst playing planning poker. As a rough guideline:
 - **Small** means "under perfect conditions this would take less than one week of elapsed time to get to Ready For Acceptance Test"
 - **Medium** means 1-2 weeks (under perfect conditions)
 - **Large** means more than 2 weeks.
- Have an **acceptance test** scenario written on the backside of the card. This is a concrete set of steps describing the most typical test scenario, for example:
 - "Joe Cop logs in, looks up Case #235 and closes it. He then looks up case #235 again and sees that is closed."

Definition of "Ready for system test"

"Ready for system test" means the feature team has done everything they can think of to ensure that this feature works and doesn't have any important defects. They have, however, focused on testing the feature itself, and not the whole release that it would be part of.

System test used to be a bottleneck for a long time, and one of the major reasons for that was the high number of unnecessary defects passing into system test. By "unnecessary defects" I mean feature-level defects that could have been found way before putting it all together into a system test. So our "definition of Ready for System Test" is there to keep the quality bar high and catch those pesky bugs early. It is also there to give the feature team a sense of responsibility for quality, and to give them permission to spend the necessary time to ensure that a feature really works, before delivering it to system test and moving on to the next feature.

So, here is our definition of "ready for system test":

- **Acceptance test automated.** This means that some kind of end-to-end feature-level acceptance test or integration test has been automated. We used to use Selenium for that (which runs tests directly against the web interface), but recently moved to Concordion. The Selenium tests were just too brittle, and Concordion fit well with our move towards Specification By Example (see www.specificationbyexample.com for more info).
- **Regression tests pass.** This means that all automated tests for previous features pass.
- **Demonstrated.** This means that the team has run an informal demo of this feature to somebody, for example the onsite customer or the main contact person for that feature.
- **Clear check-in comments.** When checking in code related to this feature, the check-in comment should be tagged with the ID of this feature, plus an easily understandable comment about what was done.

- **Tested in the development environment.** Each team has a dedicated test environment, and this feature should be tested there (not just on "my machine").
- **Merged to trunk.** The code for this feature should be on the trunk, and any merge conflicts should be resolved.

How "definition of ready" has improved collaboration

These two policy statements - "definition of Ready for Development" and "definition of Ready for System Test" - have significantly improved collaboration between requirements, test, and development. When I did a short survey to check what people thought about all the process changes so far, the most frequently mentioned improvement was the collaboration between the disciplines.

In the past each discipline focused mostly on "their" part of the project board - the requirements analysts only looked at the left part of the project board, and considered themselves "done" with a feature when a requirements document had been written. The developers only looked at the middle of the board, and the testers only looked at the right. The testers weren't involved in writing the requirements, so once a feature reached test there was often confusion about how it was supposed to work, and a lot of effort was spent in arguing over the level of detail needed in the requirements documents.

These problems gradually disappeared (well, severely declined at least) within a few weeks after everyone had agreed on these definitions of ready. The "definition of Ready for Development" can only be achieved if all disciplines work together to estimate the features, to break them down into small enough deliverables without losing too much customer value, and to agree on acceptance tests. This need drove collaboration.

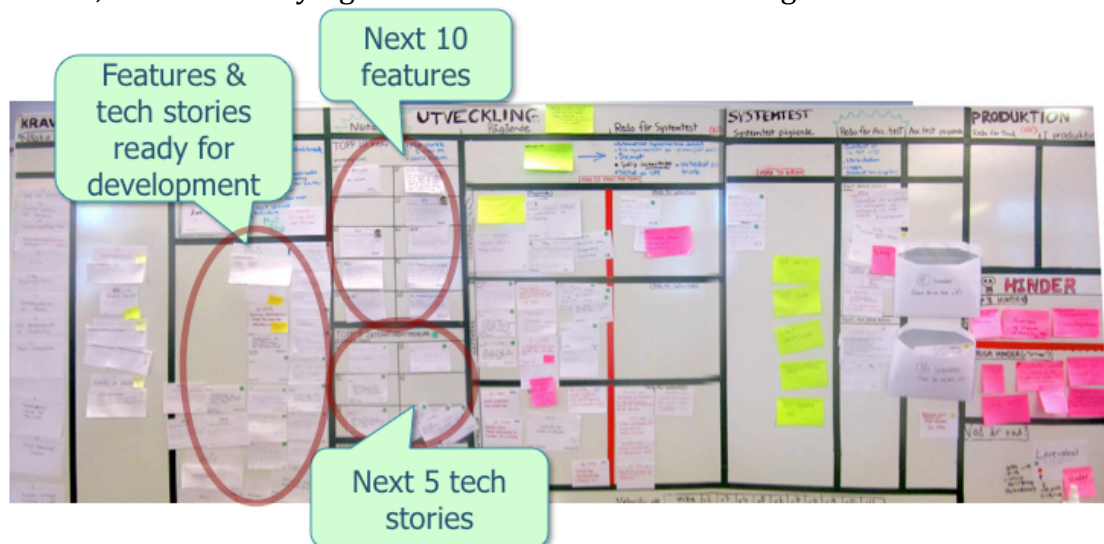
Similarly, the "definition of ready for system test" can only be achieved if all disciplines work together to run feature-level tests (both automated tests and manual exploratory tests), and to determine if this feature is good enough to release.

This need for continuous collaboration is what made the test team and requirements team agree to "lend" people to each feature team.

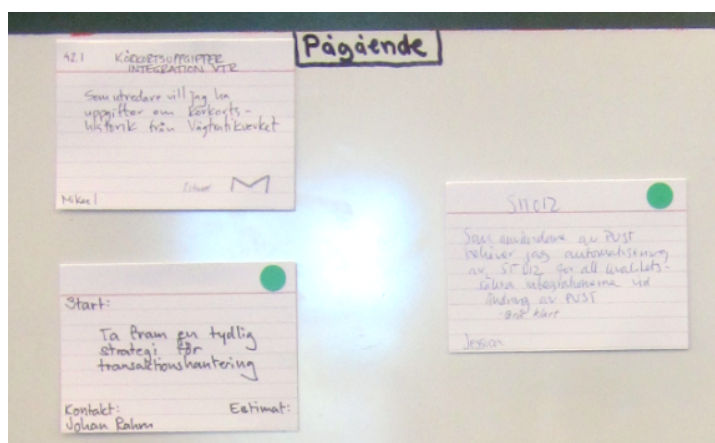
14. How we handle tech stories

Tech stories are things that need to get done, but that are uninteresting to the customer. Things such as upgrading a database, or cleaning out unused code, or refactoring a messy design, or catching up on test automation for old features, etc. We actually call these "internal improvements" but in retrospect "tech stories" is probably a better term, since we are talking about stuff to be done with the product, not process improvements.

There is a "Next 5 tech stories" section right below "Next 10 features", so these are essentially two parallel input queues to development. Any other tech stories and features are piled up under "ready for development" and unprioritized. We save a lot of time by continuously selecting the next 10 features and top 5 tech stories, rather than trying to stack-rank the whole backlog.



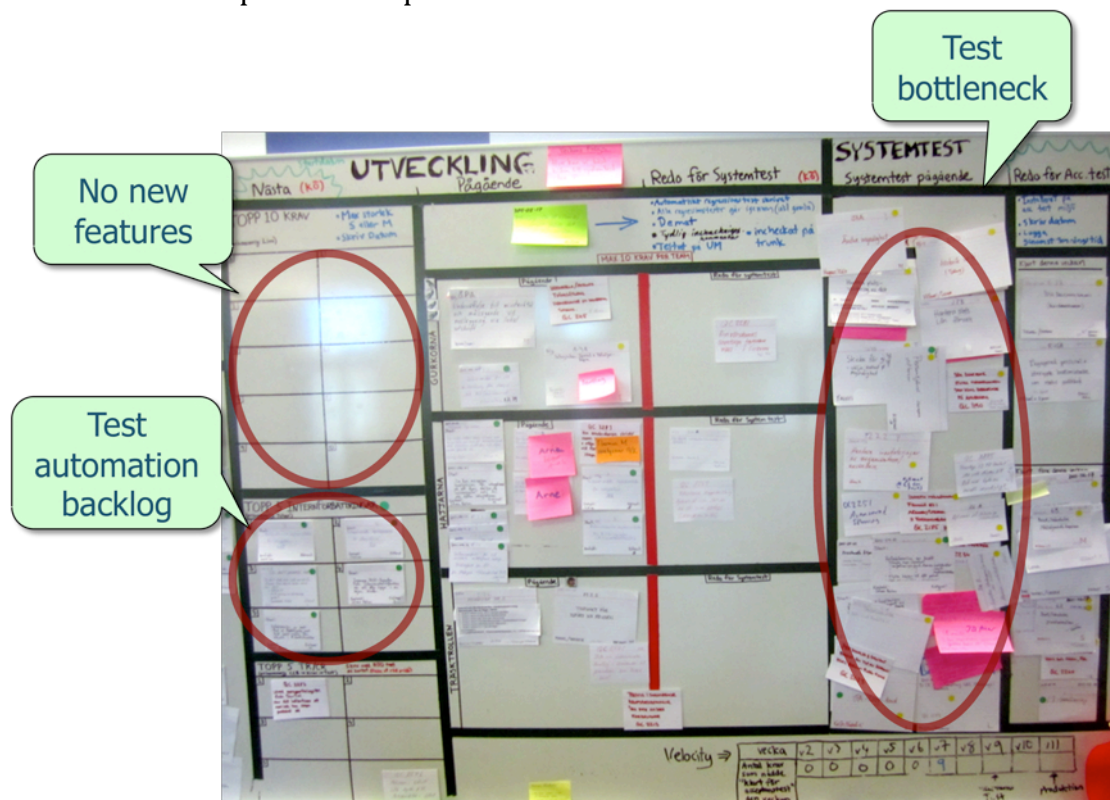
When a feature team has capacity to start something new, they either pull a feature from "next 10 features" or pull a tech story from "next 5 tech stories". Tech stories are distinguished from other stuff by a green spot in the corner of the card, so the project board reveals how we are balancing our time between features and tech stories.



We have no static rule defining the "correct" balance, instead we continuously discuss and adjust the balance during the daily meetings, typically the dev sync and project sync meetings.

Usually features dominate, but here are two examples of situations that caused us to focus mostly on tech stories for a week or so:

Example 1: system testing was an obvious bottleneck so it was clear that there was no point developing new features and adding to the bottleneck. Once this became clear, the developers focused on implementing tech stories that would make system test easier - mostly test automation stuff. In fact, the test manager was tasked with creating a "test automation backlog", prioritizing it, and feeding it in to the developers via "Top 5 tech stories". The testers became customers!



More on this in my article "Test Automation Backlog".

Example 2: It was the day before a major release, and the team wanted to get that release out the door before starting a bunch of new features. So they focused on last-minute bug fixing, and if there were no bugs to fix at the moment they worked on tech stories. Typically stuff that we had wanted to do for a long time but never gotten around to, such as removing unused code, catching up on some refactoring, and learning new tools. Lots of tech stories (green spots) in progress.



On a side note, I've never seen a project of this scale with so little *drama* in conjunction with releases! Almost disappointing :o)

Where is the customary panic and rush and all-night crunching the day before the release? Where is the subsequent onslaught of support issues and panicky hot fixing the day after the release? I came in the day after the most important release (the nationwide release that was the focal point of the whole project), and there barely any sign that anything significant had happened.

The reason for this was that the releases were well-rehearsed, due to the whole setup with pilot releases. Albeit, there were some problems with the earlier pilot releases - but that's why we do pilots right?

Anyway, remember to celebrate releases - even when you get good at it and they're not as exciting any more...

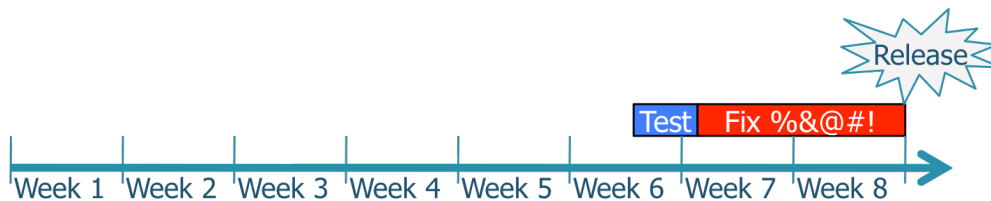
15. How we handle bugs

In the past we used to handle bugs the traditional way. The testers would find bugs during system test at the end of the development cycle & log them in the bug tracker. The bug tracker contained hundreds of bugs, and a CCB (change control board) met every week to analyze, prioritize, and assign bugs to developers. This was a pretty boring and ineffective process.

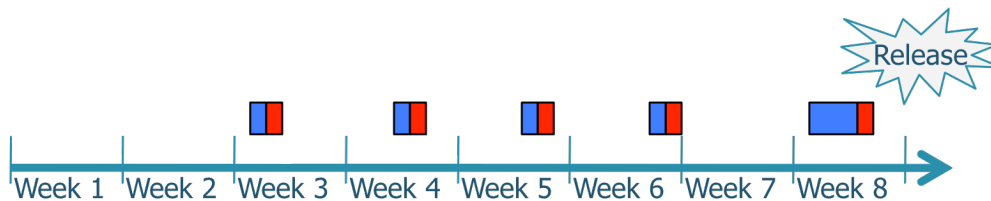
Continuous system test

The first thing we changed was the frequency of system test - do it continuously (well, regularly at least) instead of saving it 'till the end. There was some resistance from the test team initially, since system test takes time and it felt inefficient to do it more than once in the release cycle. But this is an illusion. It *seems* more effective to test at the end, but if we include bug-fixing time in the equation it is less effective.

Here is what testing at the end typically looks like in a 2 month release cycle:

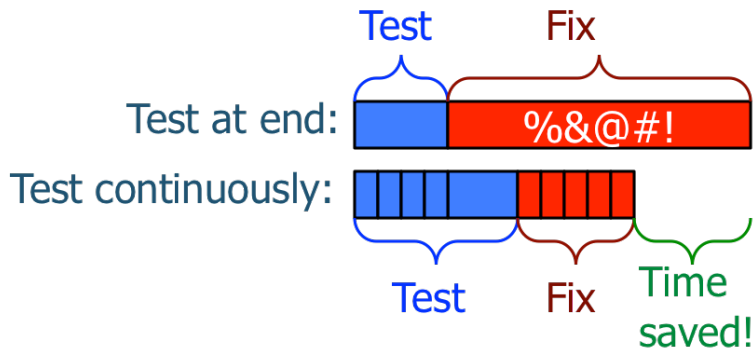


And here is what it typically looks like if we do it more often:



We can't test the "whole" system until the end, since the system isn't done until then. But we can still run partial system tests much earlier, based on whatever features are done at the time. And we can still do a full system test at the end. That final system test may take as long as before, but the bug-fixing time is dramatically reduced (and that's the big part!). Not only because we've already found many of the bugs before, but also because the bugs we do find at the end will tend to be *newer* bugs, and therefore easier for the developers to find and fix. We also accelerate learning by finding bugs early.

So let's put these two pictures together and graphically compare the testing & fixing time in both scenarios:



This is a very important picture. Look at it again, especially if you are a tester. Yes, your *testing* time increases in the second scenario. But the *total* time decreases!

Of course, another key element in this is test automation. We can never automate away *all* testing, but since we are doing system test over and over again we want to automate as much as we possibly can!

Fix the bugs immediately!

Now a days when testers find a bug they don't log it in the bug tracker. Instead, they write it down on a pink stickynote (like any other impediment) and go talk to the developers. In most cases they know roughly who to go to, since each team has an embedded tester who works with the developers every day. Otherwise they ask the team leads and quite quickly find the right person to fix the bug (typically a developer who has been working in that part of the code).

The developer and tester then sit together and fix the bug on the spot. Well, sometimes the developer fixes the bug on his own and then gets back to the tester immediately. The point is, no more handoffs, no more delays, no more communicating through the bug tracker. This is more effective for many reasons:

- Finding and fixing bugs earlier is more effective than finding and fixing bugs late
- Face to face communicating is more effective than written communication (due to the higher bandwidth).
- More learning, as developers and testers learn about each other's work.
- Less waste of time managing long lists of old bugs.

Sometimes a bug is not important enough to fix immediately, for example if it only is a minor annoyance to the users, and there are other features that are more important to implement than to fix this minor annoyance. In this case, well, yes, the tester will log the bug in the bug tracker. Unless it is full of course.

What? Full? How can a bug tracker get full?

Why we limit the number of bugs in the bug tracker

In the past we had hundreds of issues in the bug tracker. Now we have a hard limit of 30.

If a bug is found, and it isn't important enough to fix now, and the bug tracker has fewer than 30 items in it, then we add the bug to the bug tracker.

If the list is full, then we have decision to make. Is this bug more important than any of the other 30? If no, then we ignore the new bug. Or we put it in the bug tracker with status "deferred" which is the equivalent of saying "probably won't fix this"). We do this mostly because it hurts a tester's soul to find a bug and then just ignore it. Even though it will probably be never fixed, there is some psychological comfort in writing it down somewhere. Plus it might be useful for data mining purposes. But basically the "deferred" stuff is outside of the Top 30 list and for all practical purposes equivalent to a garbage can, or a basement full of stuff that we don't need but don't have the heart to throw away right now.

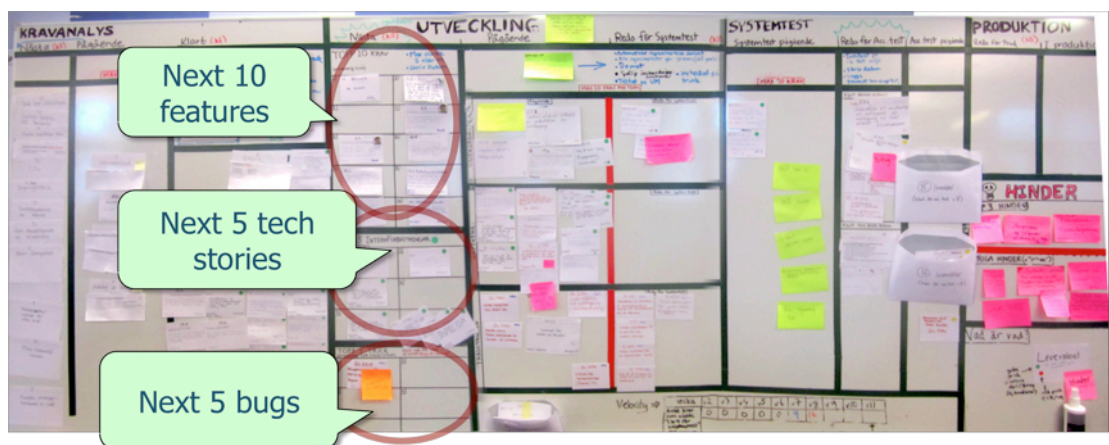
If the new bug is more important than any of the other 30 bugs, then that other bug is removed from the Top 30 list (i.e. set to "deferred") to make space for the new one.

<TODO: flowchart>

By limiting the size of the bug database, we no longer need long, boring CCB meetings to manage looong lists of bugs that will probably never be fixed. There are still CCB meetings, but they are much shorter and more effective.

How we visualize bugs

Of the top 30 bugs, we also identify the top 5. Those go up on cards on the project board. So that's a third input queue to development. "Next 10 features", "Next 5 tech stories", and "Next 5 bugs".



Bugs cards are written with red marker, so they are easily distinguished from features and tech stories.

There's an important distinction here. Important bugs are not logged in the bug tracker and don't come in through "next 5 bugs" - they are instead written on pink stickies and treated like any impediment. "Important" in this case means "this feature won't be releasable with this bug", or "this bug is more important to fix than building additional features". So fix it now, don't put it in a queue.

The stuff that comes in through "next 5 bugs" queue are the bugs that were not important enough to be a pink sticky and get fixed immediately, but were important enough to go on the "next 5 bugs" list (typically after spending some time in the top 30 list in the bug tracker). So it will be fixed soon, but just not at this moment.

When the team has capacity (typically after finishing something) they will discuss whether to grab one of the top 10 features, one of the top 5 tech stories, or one of top 5 bugs.

Limiting the size of the bug tracker builds trust. The bug list is short, but the stuff in there actually does get fixed rather than sit around for months without anything happening. If a bug is unlikely to be fixed (because it didn't make top 30) we are honest about that from start, instead of building up false expectations ("no worries, sir, your issue is on our list" ... right behind the 739 other issues...)

That's the idea. Or something like that.

One improvement area we still face is that we still haven't found a really clean consistent way to visualize bugs. We're still experimenting a lot with this. The testers like to have a clear picture of which bugs are currently being fixed, so they have set up a separate board for this. The advantage of this has to be balanced against the disadvantage of having yet another board to keep track of. Where does the bug stickynote go - on the bug board, the project board, or the team board? Or should we be duplicating bug stickynotes? What about really small bugs, things that just take a few minutes to fix, how do we avoid creating too much administrative overhead for these? Lots of questions, lots of experimentation :o)

So basically we've come a long way and found a solution that enables us to find and fix bugs quickly, and improve collaboration between developers and testers, and avoid collecting long lists of stale bugs in the bug tracker, and avoid long boring CCB meetings. But we're still trying to figure out the visualization issue, and experimenting with how to get just the enough level of detail on the boards.

How we prevent recurring bugs

Some types of bugs keep coming back. Often simple things, such as labels in the user interface being misaligned or misspelled. The important question is how the bugs get into the system in the first place, what underlying process problem is causing the technical problem.

To aid in fixing this problem (instead of just complaining about it) the testers have a section on their board called "recurring bugs". Remember that I said that

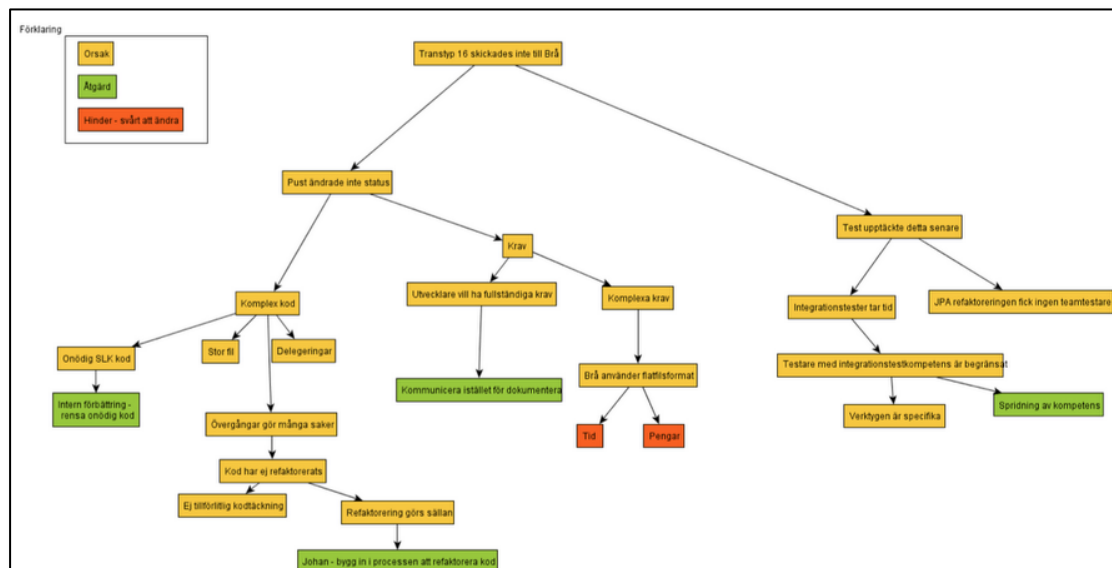
bugs are written on pink stickies and treated like any impediment? Well, when the testers feel that a specific bug gives them a strong feeling of déjà-vu, they put that up under "recurring bugs" on their board. This is limited to a handful.

(You've noticed the theme by now, right? Limit All Queues!)



Once in a while one of the feature teams will have a defect prevention meeting, where they take one of the recurring bugs and do a root-cause analysis. How did that bug get into the code? What systemic problems are causing us to repeatedly create this type of bug, and what can we do about it? Sometimes it has to do with the tools used, sometimes with the policies and rules, sometimes cultural issues, etc.

A cause-effect diagram is typically used to identify the consequence and root causes of this bug, which is then used to generate concrete proposals for how to avoid this type of bug in the future.



Cause-effect diagrams are a great way to do root cause analysis, especially when you start finding vicious cycles. Such as "This bug was caused by us stressing and not testing properly. We were stressed because we were behind on the release plan. We were behind on the release plan because of bugs in the previous release". Pretty vicious loop huh?

Jerry Weinberg puts it nicely: "Things are the way they are because they got that way". Or here's another one (don't know who said it first): "If all you ever do is all you've ever done, then all you'll ever get is all you ever got".

Anyway if you want to learn more about cause-effect diagrams check out my article "Cause effect diagrams - a pragmatic way of doing root-cause analysis".

16. How we continuously improve the process

The process described in this paper was by no means designed up front. I would never have been able to figure all this out up front, especially not alone. And even if I could, there would be no buy-in from the project participants, so any process I designed up front would never have reached further than the paper it was written on.

The process was discovered rather than designed. The first thing I did was put in place a "process improvement engine". Well, I didn't use those words but that is in effect what it was. The basic formula is:

- **Clarity** - having physical boards in prominent locations that show everyone what is going on. And having a clear delivery goal that everyone can understand.
- **Communication** - having process improvement workshops on a weekly basis. Both locally within each team, and at the cross-team level.
- **Data** - track some simple metrics that show us if our process is improving. We measure velocity and cycle time. See chapter 19 for info.

The strategy is pretty simple, and is based on the assumption that people inherently want to succeed with their projects and inherently like to solve problems and improve their way of working. So create an environment that enables and encourages these behaviors.

If everyone knows where we are going and where we are right now, and if we have the right communication forums in place, then chances are people will self-organize to move in the right direction and continuously figure out ways of getting there faster.

This mindset of motivating people to do evolutionary process improvement is the basis of both Agile and Lean.

How we do team retrospectives

Our process improvement workshops are basically Scrum-style sprint retrospectives. By convention the teams use the word "retrospective" for their process improvement workshops, and the cross-team uses the term "process improvement workshop", so I'll stick to that terminology in this paper.

The teams have retrospectives every week or two, and the length varies from 30 minutes to 2 hours. Some teams just stand at their taskboard and do the retrospective, some go off to a separate room. Usually the team lead facilitates the meeting, but sometimes they pull in someone else (like me). Bringing in a team-external facilitator from time to time is usually a good idea because that gives the team some variation in how the retrospective is run, and also provides the team lead with some insights about different ways to run retrospectives. And it also allows the team lead to fully participate instead of facilitating.

One simple and cheap way to get an external facilitator is for team lead A to facilitate the retrospective of team B, and vice versa.

There is pretty large variation in how the retrospectives are run, but the goal is the same in every case: reflect on what is working well and what isn't, and decide what to change.

Typical changes include stuff like check in code more often, change the time of the daily meeting or how the meeting is run, update the code conventions, identify some new team-internal roles, etc.

Another important function of the team-level retrospectives is to identify escalation points, i.e. problems and improvement proposals that affect more than just this team and need to be solved together with the other teams. These are noted by the team leads and brought to the higher level process improvement workshops.

How we do process improvement workshops

The process improvement workshop is basically a "Scrum of Scrums" type retrospective, with one person from each team and each discipline (the same "cross-team" that meets at the project board 10:00 every day). One of my most important tasks as coach was to set up this forum and facilitate it. It is the most important place to trigger change and find out which changes are working.

The stated purpose of this meeting is to clarify and improve our way of working.

Initially we did these every Thursday at 1 pm. Having it on Thursdays at 1 pm was mostly a coincidence - it was the least congested time for everyone involved. After a month or so we changed it to bi-weekly, so every second Thursday at 1 pm. The reason why we did it so often in the beginning was because of the need to quickly improve collaboration between the different disciplines, and because of the urgent need to change due to the growth pain and confusion we were experiencing.

Having a process improvement workshop every week was rather intense though, there was barely time to execute the changes from one meeting to the next. The positive side was that it drove us to implement change quickly, because it is embarrassing to sit at the next process improvement workshop and say "well, dang, we never actually got around to implementing the change". Also, because the meetings were every week we had to keep them short and focused, which forced us to prioritize only the most important changes and take small steps in our change process.

Actually, come to think of it, the meetings weren't really that short. We started with 60 minutes and had to increase to 90 minutes because we kept overrunning. A pretty long time for meeting that happens every week. And the changes we made were rather significant, not really baby steps at all. Looking back I can't really say if that was a good thing or not. We really did need to

change things pretty quickly (if nothing else, due to the Big Scary Deadline looming around the corner), but the rate of change did also cause some confusion, especially for the majority of the people who weren't in the cross-team, and who saw lots of change happening without always being given a chance to understand or discuss the change.

Once the most important problems were solved the rate of change could be slowed to a more comfortable level, so we reduced the number of meetings to once every second week instead. This felt more humane. Now we could spend 90 minutes without feeling as stressed (since it wasn't every week), and it was easier to actually implement a change and learn from it before the next meeting.

When doing process improvement workshops I take care to move away all tables and create a ring of chairs in the center of the room near a whiteboard.



This has a noticeable effect on the level of collaboration and focus in the room. Everyone is facing each other without any barriers between them, and without distractions such as papers and computers on the table.

Each process improvement workshop follows the same basic structure, which roughly corresponds to the meeting structure defined Diana Larsen and Esther Derby's book "Agile Retrospectives".

At a high level:

1. **Set the stage:** Open up the meeting and set the theme and focus.
2. **Gather data:** Go through what has happened since last meeting, including victories and pain points. If there is a theme, go through concrete data relevant to that theme.
3. **Generate insights:** Discuss the data and what it means to us, focus on the most important pain points and identify concrete options to alleviate them.
4. **Decide what to do:** Make decisions about which changes to implement.
5. **Close the meeting:** Decide who is going to do what, and what will happen by next meeting.

<digression>

Any lean enthusiasts of the more fundamentalist type reading this article might point an accusing finger now and say "Yuck, that's all subjective, touch-feely stuff! Process improvement must be driven by quantitative, objective data and reports!"

Well, first of all, I don't agree. Complex product development of this sort is a creative process done by creative people, and the most important parameter is motivation. In this context, gut feel beats hard metrics. If something *feels* like an important problem, it most likely is an important problem, whether or not we have metrics to prove it. And the nice thing about gut feel is that it often is a leading indicator of a problem that is about to occur, while hard metrics often show a problem only after it has occurred.

So yes, we use hard metrics (described further down in this paper), and sometimes those metrics will trigger the necessary gut-feel realization that there is a problem (the proverbial "oh shit!" moment). But we use metrics primary to support process improvement, not to drive it.

</digression>

Anyway where were we? Oh yes, so we list the pain points and prioritize them and choose 1-2 issues to focus on for this meeting. Then we typically break out into groups of 2 or 3 to discuss and analyze the problem and possible solutions.

For more complex or recurring problems we do a root cause analysis using cause-effect diagrams and similar techniques, or propose some research that will bring useful metrics to the next process improvement workshop, or plan a separate problem solving workshop to be done with a small focus team. Most problems aren't treated this way though.

The breakout discussions usually result in a number of concrete proposals, or options, which I list on the whiteboard. The default option is always status quo ("don't change anything"), a reminder of what will happen if we don't agree on any other option by the end of this meeting.

For each option (including the status quo option) we brainstorm the most obvious advantages and disadvantages. Quite often this quick analysis clearly reveals which option is best, so we quickly agree on implementing that option. For less obvious choices we do a quick thumb vote, i.e. checking how people feel about each option.

- Thumb up means "I like this option"
- Thumb sideways means "This option is not great, but acceptable". Or "I don't have a strong opinion and will go with the group". Or "I can't make up my mind right now".
- Thumb down means "This option sucks, and I will not support it".

Sometimes we use "fist of five" instead, which is pretty much the same thing but more granular. Instead of just thumb up or sideways or down, each person holds up 1-5 fingers.

- 5 = This is a great option!
- 4 = This is a pretty good option.
- 3 = (same as thumb sideways)
- 2 = I don't like this option and won't support it. But I may be convinced.
- 1 = Over my dead body.

The important thing with both of these techniques is that thumbs sideways, or 3 fingers, is a threshold value which means "I will support this option". So any option that has that level of support (or higher) from every person in the workshop is good to go. Everybody doesn't have to love that option, but everyone will support it. That's what consensus means.

Looking at the options and the votes, it usually becomes clear what the best option is. If there are many options we start by crossing out any unacceptable options, i.e. options that have any votes of 1 or 2 fingers or thumb down, since those options don't have group consensus. Kind of like a veto. Then we take the remaining options and pick the one with highest votes. Again, status quo is the default if we can't agree on any other option.

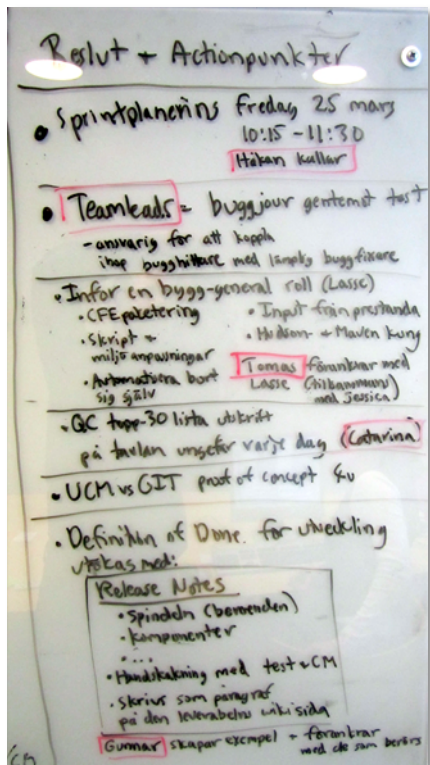
In the rare case that we have two options with equally strong support, we pick one at random, pick the easiest one, or do a quick tie-breaker vote ("given a choice between option D and E, which one would you prefer"). As facilitator of the meeting I usually decide the decision process in each case, to avoid getting into time consuming meta-discussions about how to decide on the decision process (always a risk with smart people). It is up to the meeting participants to protest if they don't like my choice.

All this consensus-building stuff might sound inefficient, but it is actually pretty quick and effective in most cases. And in the few cases where it isn't quick, that usually means there is some deeper analysis that needs to happen.

The decisions we make at these process improvement meetings are usually, well, process improvement decisions. That means change. And since we are dealing with People, change means risk of resistance. Especially from people who aren't at the meeting where the decision was made. By aiming for 100% consensus for each change (within the cross-team at this meeting) we dramatically reduce the risk of resistance, and thereby dramatically increase the likelihood that the change will actually work. So the few extra minutes spent on consensus building pays off in a big way.

We time box the meeting strictly, typically to 90 minutes. During the last 10 minutes or so we summarize the decisions that were made (list them on the whiteboard) and identify concrete actions - who is going to do what and when.

Here is an example:



At this meeting we made many decisions (more than usual). The first two were:

- Try out the concept of "sprint planning meetings", where the different disciplines collaborate to refine and break down user stories and decide which ones to pull into the "next 10 features" column on the board. More on that later in this paper. If it works well we will continue doing this, probably bi-weekly.
- Each team has a clearly defined "bug contact person", i.e. the default person for testers to talk to when they find a bug. Team Leads are the default bug contact person if nobody else is defined.

Remember that the stated purpose of the meeting is to clarify and improve the current process. Sometimes we don't actually change anything, instead we just clarify our current process, i.e. resolve some source of confusion and provide a clear description that everyone at the meeting can relay to their teams. One example was when need to clarify what we mean by "acceptance test" vs. "system test" vs "feature test".

How we manage the rate of change

The weekly process improvement meetings caused a flurry of change, mostly changes for the better. However, after a while we realized that we were changing too much, too fast.

This was an interesting problem. In most organizations I've worked with the problem is that there is too little process change, everyone is stuck in their current ineffective process. Here we had the opposite problem. We were making lots of changes, and it takes some time for process change to ripple through a 60-person project. Many team members got confused (and sometimes frustrated)

when the cross-team introduced new changes before the dust from previous changes had settled.

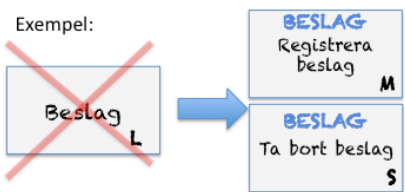
So we introduced a little bit of bureaucracy to slow down the rate of change :o)

Whenever somebody wants to make a change that affects more than just his own team, they write a "Process Improvement Proposal". This is a light-weight variant of the lean A3 problem solving process (see our A3 template for details on this).

The process improvement proposal template forces you to think about Why you want to make this change. The three questions on the template are:

- "What problem are you trying to solve?"
- "Who is affected by this change?"
- "What steps are involved in executing this change?"

The answers to these questions are very helpful in determining the value vs cost of doing this change. Here is a real-life example of a process improvement proposal:

Förslag: Mer kundnyttig nivå på leverablerna	
Varför? Vilket problem vill vi lösa? <ul style="list-style-type: none">• Svårt att få överblick på projekttavlan ur "kundperspektiv", dvs många av leverablerna är så små att de inte kan leveras till kund.	Beskrivning / FAQ <p>En leverabel som går in i utveckling måste:</p> <ol style="list-style-type: none">1. Vara storlek S eller M2. Vara så kundnyttig som möjligt, så länge vi inte bryter mot storleksregeln.
Vilka påverkas av förändringen? <ul style="list-style-type: none">• Krav & utveckling & test	Dvs kravgruppen ser till att varje lapp under "krav klart" är en kundnyttig leverabel (oavsett storlek). "Kundleverabel" = en leverabel som är kundnyttig. Men innan en leverabel får gå vidare till utveckling måste den vara S eller M.
Vad ska göras för att implementera detta? <ul style="list-style-type: none">• Uppdatera "definition of done" för "krav klart", lägg till "leverabeln är kundnyttig".• Gå igenom tavlan & identifiera leverabler som är för små för att vara kundnyttiga. Slå ihop dessa till större leverabler som är kundnyttiga (så länge de inte blir större än M)	Fråga: Vad händer om kundleverabeln är stor (L) & måste levereras i sin helhet innan någon kundnytta uppstår? <ul style="list-style-type: none">• Bryt ned till ett antal mindre leverabler (egna lappar) som är M, men med max möjlig kundnytta per leverabel.• Håll ihop dessa leverabler genom att skriva namnet på kundleverabeln i blå tjock text.
Exempel: 	

The proposal was about keeping the features at more customer-valuable level. It also proposed that features estimated to Large should not be pulled into development at all, since they tend to swell and clog up the process. Instead they should be broken into smaller deliverables. And when that is done, if the smaller features aren't each independently valuable to the customer, then a title should be written at the top of the card in bold blue text, showing that a number of smaller features fit together into a bigger feature. This helps us keep these features together from a release perspective.

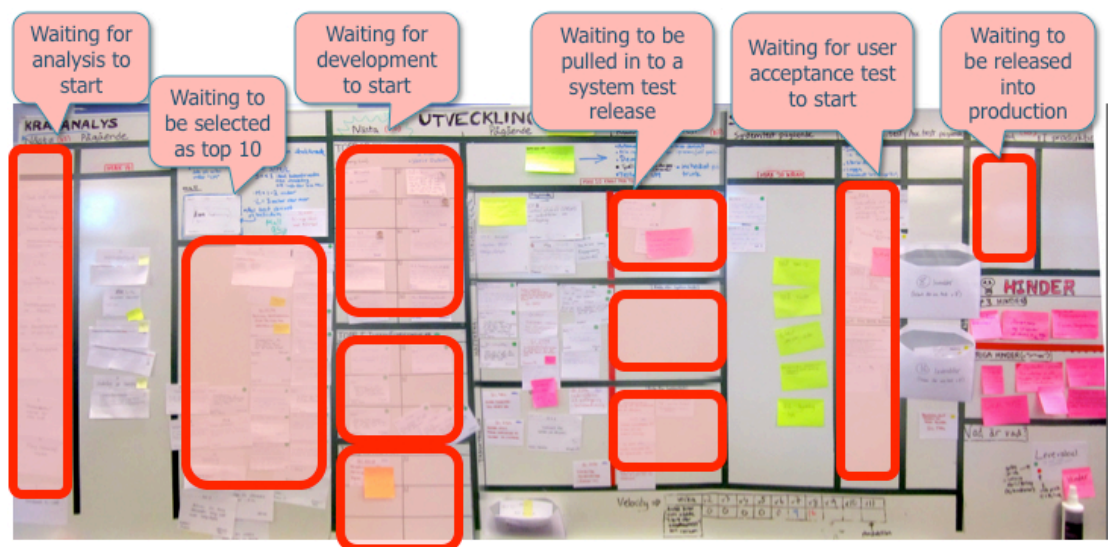
These types of proposals can come from anyone. Typically the person who wrote the proposal shows up at the process improvement meeting to present the proposal and answer questions. Our template essentially turn the proposal into a small business case for a specific change, making it quite easy to make a decision.

The purpose of introducing this little template was to allow us to *limit* the amount of change. So if we get 4 proposals we might only implement 1 or 2 of them, even if all 4 proposals were great. It is very difficult to **not** implement a great process improvement proposal, but we realized that we really have to limit the amount of simultaneous process improvement initiatives going on, or we get too much confusion which offsets the benefit of the improvement.

We've even considering having a separate "process improvement Kanban board" with WIP limits, showing which changes are currently being implemented. That could be useful for follow-up purposes too. But it would be yet another board to find space for and keep up-to-date. Hmmmm.

17. How we distinguish between buffers and WIP

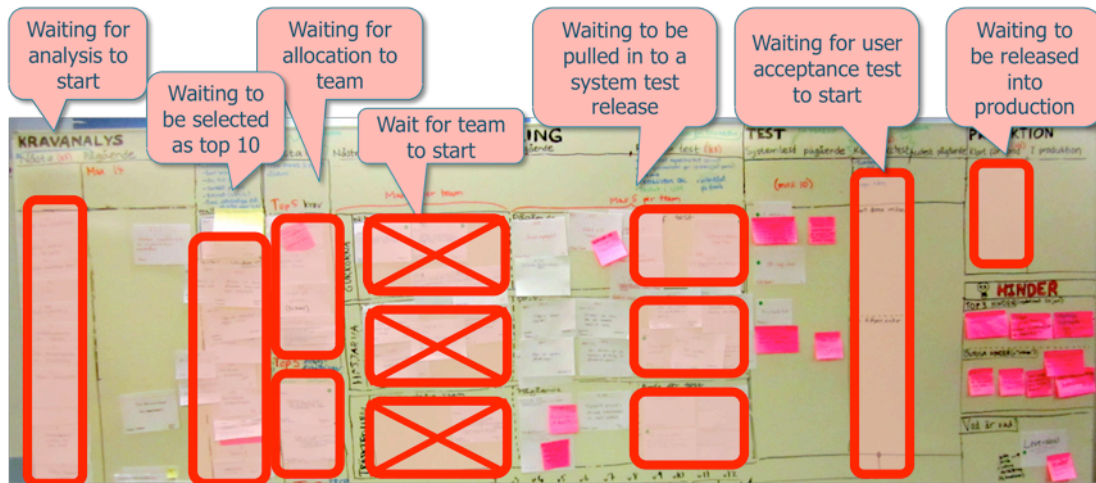
4 columns on the project board represent work in progress, while the other 6 columns are actually buffers (or queues), in the sense that stuff in those columns are not really in progress - they are waiting for the next step in the process. We distinguish those by writing "Queue" in red next to the column title on the board. Here they are:



It is very useful to distinguish these from WIP columns, since queues and buffers are clearly waste in the sense that features are just sitting there waiting. Some amount of buffering is usually necessary to absorb variability, for example to account for the fact that the rate at which features get developed doesn't always perfectly match the rate at which features get system-tested. This can be seen as a "necessary waste" to ensure smooth flow.

As the process improves the need for these buffers is reduced. It is therefore useful to visualize these on the board to that we keep asking ourselves if we really need all these buffers, and what we can do to reduce them.

Here is an example of an older version of the board, when we had yet another buffer between requirements and development:



The reason for that buffer was that the teams previously had a more Scrum-like model with sprints, so at each team-local sprint planning meeting they would commit to a specific set of features. So we had 3 queues (or buffers) between analysis and development:

1. Features that have been identified through analysis, but not yet selected into the top 10 list
2. Features that are included in the top 10 list, but not yet pulled in by a team.
3. Features that are in the current sprint of team X, but they haven't started work yet.

We noticed that we spent time arguing over which feature should be in which queue, which is essentially waste. So we simply removed the third queue and decided that each team pulls features directly from the top 10 list instead of batching features into sprints.

This raises the question of specialization. If a team is working within a specific feature area, for example integration with system X, then it is most effective for that team to implement all features that integrate with system X, since they have knowledge of how system X works.

That doesn't mean, however, that the team needs to pull in *all* X-related features upfront. Although this team will be the default choice for X-related features, we don't want to rule out the possibility of other teams helping out if this team becomes a bottleneck.

So basically each team pulls directly from the top 10 list, but in an intelligent way - the teams talk to each during the dev sync meeting and figure out how to best use the capacity of each as capacity becomes available.

To aid in this process we have "team magnets" on the project board, i.e. a team can "tag" a feature in the top 10 list (or earlier) with their team magnet, indicating that "that feature will be best done by our team". That way people know who to talk to about that feature, and the other teams will think twice before starting that feature.

TOPP 10 KRAV

(prioritetsvarig: Henrik)

- Max storlek S eller M
- Skriv Dr. Num

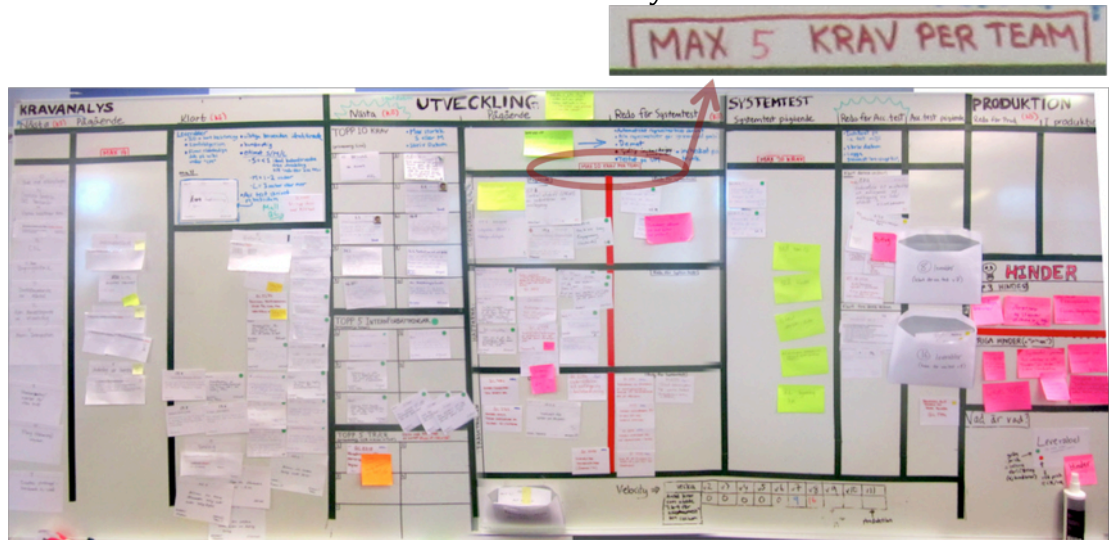
<p>1</p> <p>42.2 KORREKTUSUPPGIFTER NY RAPPORT KORREKTUSHISTORIK</p> <p>Som utredare vill jag ha en rapport innehållande korrektshistorik i utredningen.</p> <p>Mikael S</p>	<p>2</p> <p>42.5 KORREKTUSUPPGIFTER GUI-FÖRÄNDRINGAR</p> <p>Som utredare vill jag kunna lägga till en rapport innehållande korrektshistorik i utredningen.</p> <p>Mikael S</p>
<p>3</p> <p>35.6 CSL</p> <p>Förhandsgranska</p> <p>Mikael S</p>	<p>4</p> <p>20.8 RMY - DRAG.COM</p> <p>Avancerad sök på Analysen med hjälp av datum intervall</p> <p>Perin S</p>
<p>5</p>	<p>6</p>

Team tag

Team tag

18. How we use WIP limits

We have WIP limits (work in progress) pretty much across the whole board, written in red text at the top of each column or set of columns. For example each feature team tries to limit their WIP to at most 5 features at a time. That means that if they are working on 5 features, they will not start working on a new feature until one of the others are done and in system test.



Our WIP limits only apply to features, so tech stories and bug fixes aren't included in the WIP limit.

Bugs aren't included in the WIP limit because they often are quite urgent, and often quite small, and because we don't yet have a consistent way of handling these on the board.

Tech stories aren't included in the WIP limit because, well, let me try to explain...

The purpose of WIP limits is to avoid too much multitasking, and to avoid overloading a downstream process. For example if test has too much work to do, we don't want developers to keep building new features and adding to their workload - instead they should focus on helping test. That's why WIP limits are useful, they act as an alert signal to the development teams.

From a bottleneck perspective our tech stories often have the opposite effect - they offload the downstream bottleneck (in this case test). Many of our tech stories are related to test automation and infrastructure improvements, both of which improve quality and make life easier for testers.

When system test becomes a bottleneck they will focus on finishing their current system test round, which means it will take a few days before they can pull cards from "ready for system test" to "system test ongoing". The WIP limit of 5 for each development team applies across both columns "development in progress" AND "ready for system test". The consequence of this is that when system test

becomes a bottleneck, the WIP of the development teams will fill up, since the features that they have developed are stuck on their tray until the system test team actually pulls them in.

So what should the developers do when the WIP limit is full? They should do anything that will assist or offload system test. This includes manual testing and bug fixing. But it also includes deeper things such as developing more automated tests and improving the test infrastructure. Those things are represented as tech stories. That's why we don't include tech stories in the WIP limit, because we want to encourage team members to work on tech stories things when the WIP limit is full.

During some periods the teams have focused almost entirely on test automation, with only green-dotted cards on the project board. This is a great example of how Kanban boards with WIP limits facilitate self-organization and bottleneck alleviation.

Another reason why WIP limits only apply to features is because that is consistent with the fact that our metrics only apply to features as well. So it is easier to explain and easier to remember.

19. How we capture and use process metrics

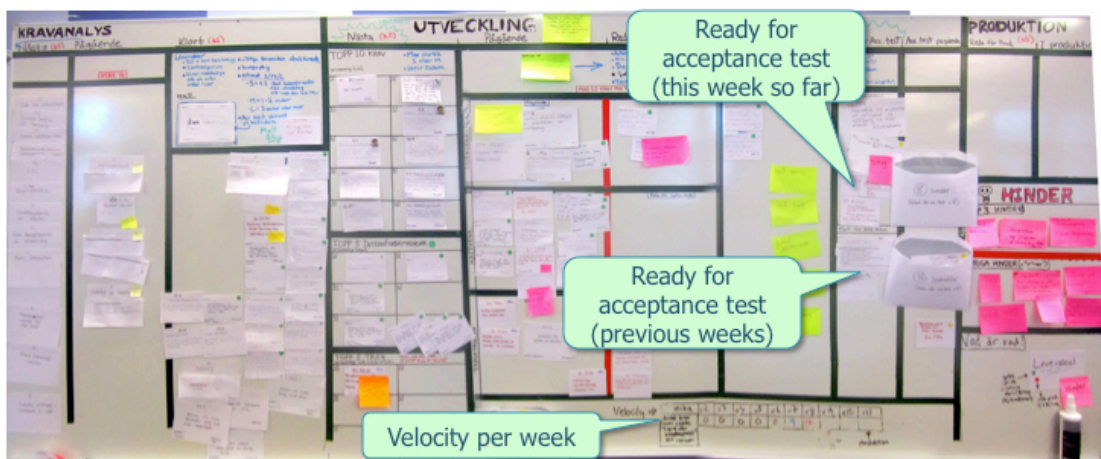
We track two process metrics:

- Velocity (features per week)
- Cycle time (weeks per feature)

We do this entirely manually, it is surprisingly easy. I'm surprised all projects don't do this.

Velocity (features per week)

For velocity (= throughput), we just count, at the end of each week, how many features have reached "ready for acceptance test (this week)" during that week. We write this number down at the bottom of the board, and then move those cards down to "ready for acceptance test (past weeks)" to show that they have already been counted.



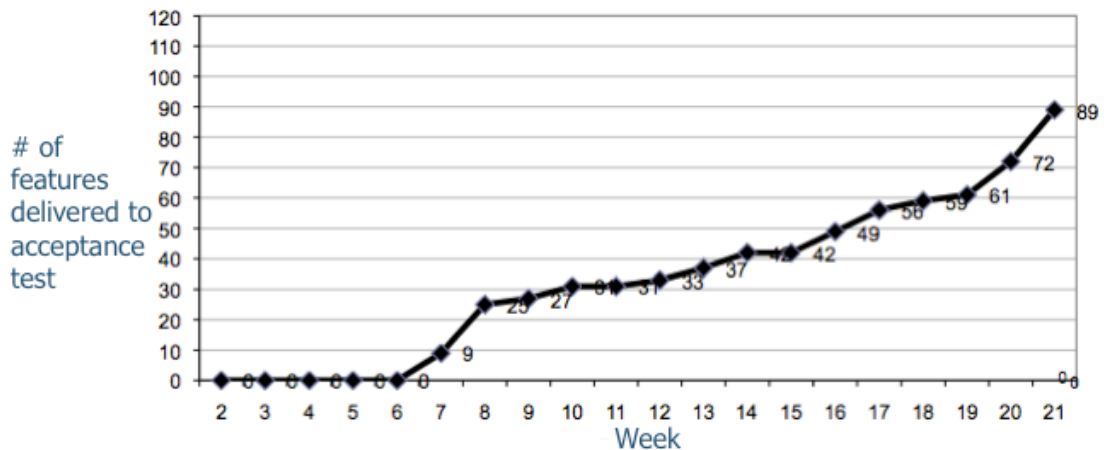
Here is a close-up of the velocity history:

Vecka	v10	v11	v12	v13	v14	v15	v16	v17	v18
Antal nya funktioner som nått till 'Redo för AcTest'	4	0	2	4	5	0			

↑
Prodsättn.

VEL

Using this information we generate a simple burnup chart:

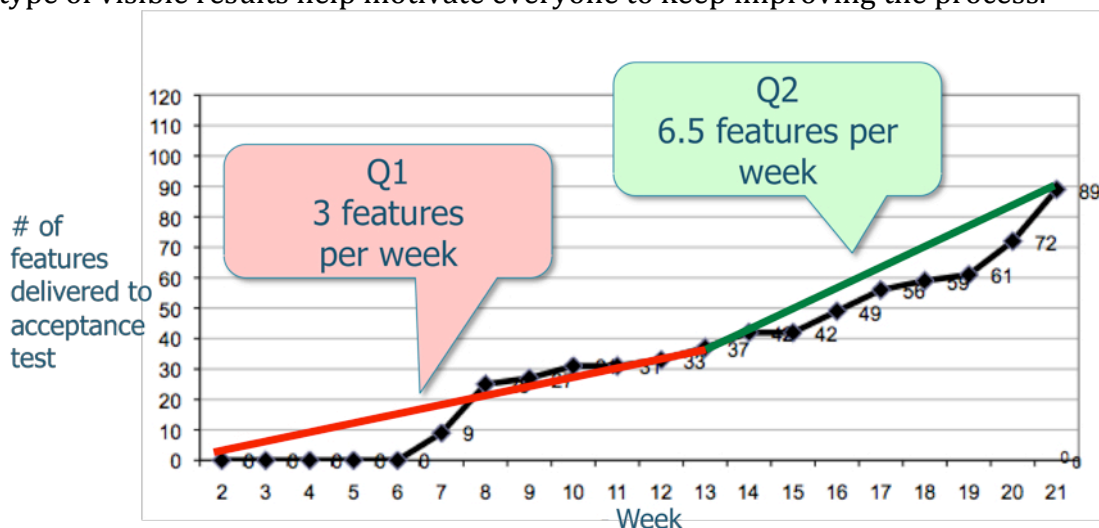


This information is useful in many ways.

It is used as a "reality check" tool to make sure that the release plan is realistic, and as a tool to predict approximately how many features will be done by a certain date.

The burnup chart is also used to highlight problems. For example during the first few weeks of measurements the velocity was 0. The teams were working very hard, but system test had become a bottleneck for various reasons, so all the features were queuing up in front of the test team. This created a sense of urgency and caused developers to gradually focus more on helping test instead of just developing new features and adding to the queue.

Finally, the burnup chart is used to visualize process improvement. For example we could see that our average velocity has doubled between Q1 and Q2. These type of visible results help motivate everyone to keep improving the process.



Note though that statistics need to be used with care. During the weeks after creating this diagram the curve flattened, since the team focused on internal improvements and had a 0 velocity. A more realistic estimate is that velocity has increased by roughly 50% rather than doubled.

We've considered measuring the velocity of tech stories as well, so that we can visualize how our effort is distributed between customer needs and tech stories. Combining these two velocities would give us our total capacity, which would give us a smoother burnup curve and make planning easier.

Why we don't use story points

At this point you might be wondering how can get away with just *counting* the features. What about size? Shouldn't we take size into account when measuring velocity? Suppose all the features in Q2 were smaller than the ones in Q1, in that case it would be perfectly normal to complete twice as many, and misleading to call that "double velocity".

In theory, correct. In practice, however, the sizes are quite evenly distributed. I did a little experiment once and assigned each feature a "weight" based on estimated size, so Small = 1kg, Medium = 2kg, and Large = 3kg. Most agile teams would call this "story points", i.e. a relative estimate of the effort involved in building that feature.

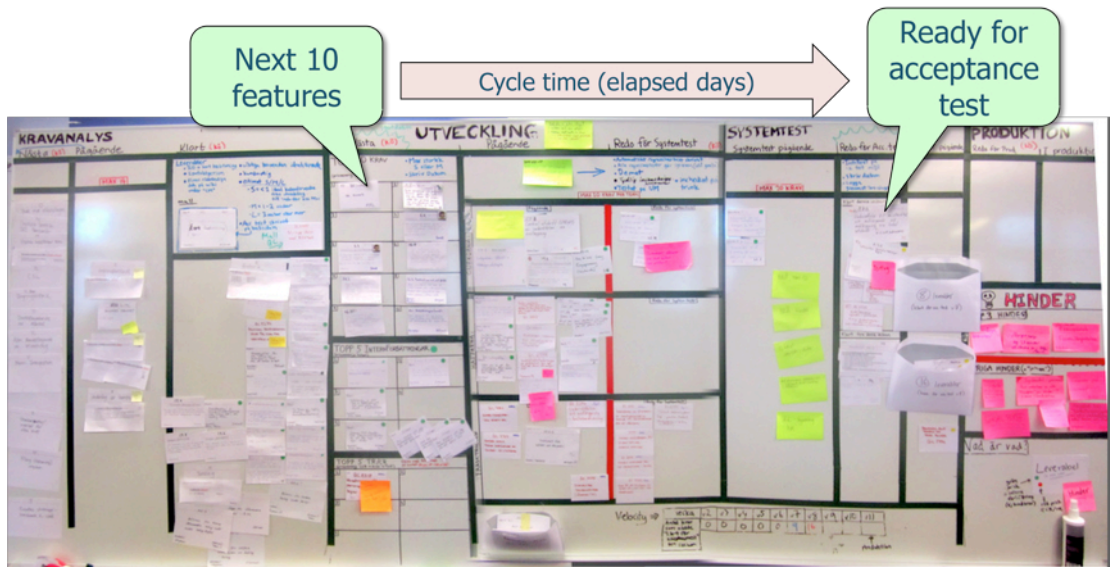
Here's a useful metaphor. Supposed I'm loading bricks, and suppose I want to measure velocity for this. My velocity turns out to be 10-15 bricks per minute. Not a very exact number. But wait, the bricks have different weight! What if we measure kg per minute, instead of bricks per minute? That might give us a smoother velocity, and therefore make it easier to predict how many bricks I will have done by the end of the day.

So we measure each brick to find out what it weighs, and then we calculate how many kg per minute I am loading, which turns out to be 20-30 kg per minute. Um wait, this was no more precise than the first number 10-15 bricks per minute! As long as the bricks are roughly evenly distributed in size, there is no point weighing each one to calculate velocity. Just count the number of bricks instead.

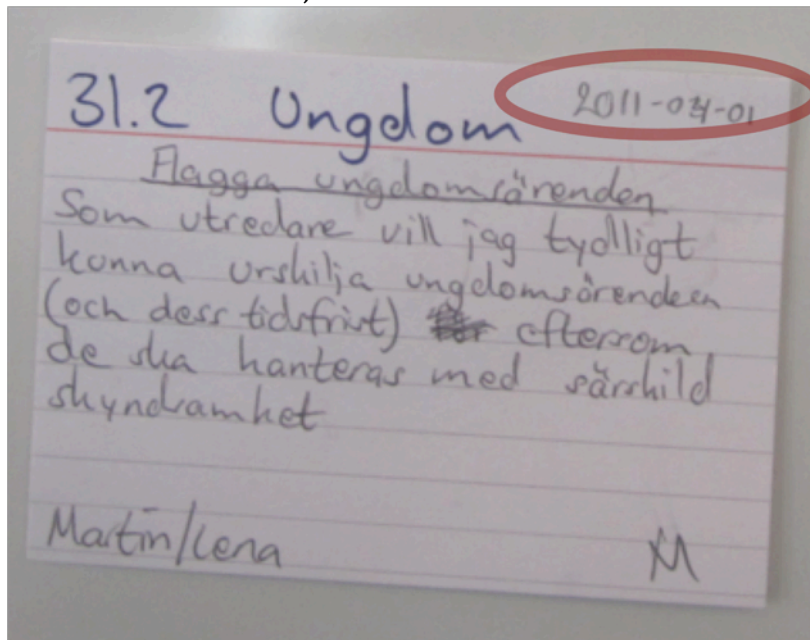
This was exactly what happened in our case. I created a burnup chart with kg instead of # of features, and the shape was just as jagged as before. The higher level of precision gave no added value, so estimating in story points would have been a waste of time.

Cycle time (weeks per feature)

The other thing we measure is cycle time (or flow time). Cycle time means how long it takes for stuff to get done, or more specifically in our case "how long did it take for feature X to move from 'Next 10 features' to 'Ready for acceptance test'".



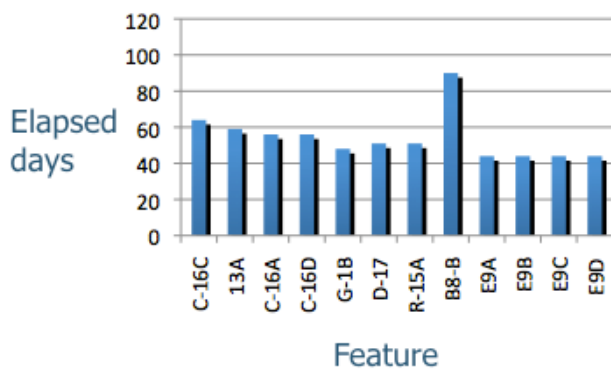
This is also very easy to measure. Every time a feature is selected to be among the "Next 10 features", we write a start date on the card.



Every time a feature reaches "ready for acceptance test" we note the finish date and write down the elapsed days for that feature in a spreadsheet.

Leverabel	start	slut	Genomströmningstid (dagar)
6A	2011-02-01	2011-02-17	16
6A	2011-02-01	2011-02-17	16
26B	2011-02-03	2011-02-24	21
26A	2011-02-03	2011-02-24	21
B2.2	2011-02-08	2011-02-21	13
25.2.1	2011-02-08	2011-02-18	10
25.2.2	2011-02-08	2011-03-25	45
B8-A	2011-02-10	2011-03-09	27
T23.2	2011-02-10	2011-02-23	13
16J	2011-02-20	2011-03-28	36
A-4A	2011-03-08	2011-03-30	22
6.2	2011-03-08	2011-03-25	16

We then visualize this using a control chart, where the height of each bar represent how long that feature took to cross the board from the top-10 list to acceptance test.

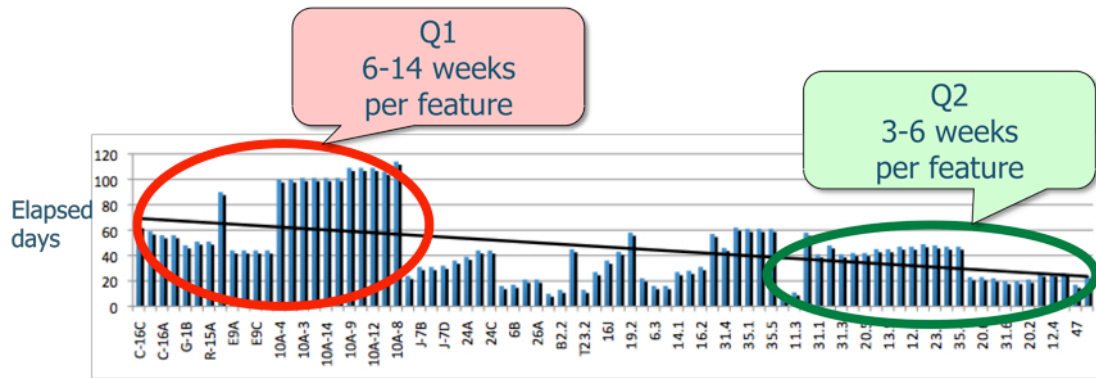


This is useful for predicting how long it will take for a feature to get done. It's also a great way to generate a sense of horrified awe, as most people don't really realize how long things really take. Happens every time a company starts visualizing this stuff :o)

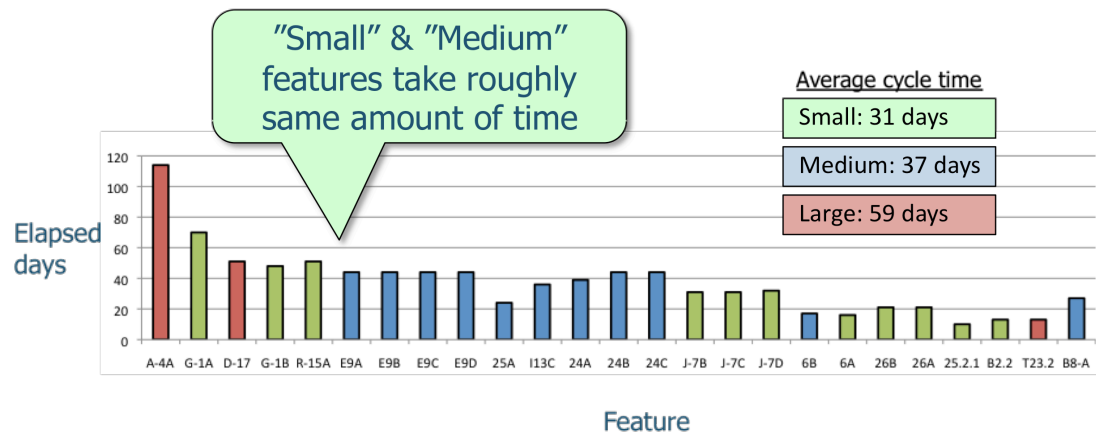
Quite typically the difference between elapsed time and actual worked time is 5-10 times. So it might take 20 days of calendar time to finish a feature that is actually only 3 days of work.

The reason why things usually take much longer than expected is primarily due to multitasking and the abundance of buffers and queues for features to get stuck in while waiting for the next step in the process.

The good news is that once you visualize this and start focusing on it, it is quite easy to shorten the time dramatically. In our case, the long term trend below shows how cycle time has been cut in half within a few months.



Another interesting thing that we noticed was the lack of correlation between feature size and cycle time. In the diagram below the features are color coded based on their original estimate.



As you can see, some of the "small" features took as long as 7 weeks, while some "large" features took as short as 2 weeks. It turned out that size wasn't the main factor influencing cycle time - other factors such as focus and access to key resources were more important.

At one point we went through this data and set some improvement targets. Our goal was to set challenging but realistic targets to guide the improvement efforts.

1. More stable velocity, i.e. roughly the same every week instead of unevenly distributed. This would give us less bottlenecks, easier release planning, and more smooth flow in general.
2. Higher velocity. Our average was 3, we set the target to 5.
3. Lower cycle time. Our average was 6 weeks (but shrinking fast), we set the target to 2.

When doing this an interesting insight dawned upon us. Supposed we reach the first two targets and get to a stable velocity of 5 features per week.

Our data (and photos) shows that on any given day there were typically 30 or so features on the project board in various buffers and WIP states.

30 stories in progress




That means, mathematically, that average cycle time will be 6 weeks!

Suppose you are at a pizza restaurant and their delivery capacity is 5 pizzas per hour. How long will you have to wait for your pizza?


12 minutes?

No, not if there are 30 other people in the restaurant, all waiting for their pizza. In that case it'll take 6 hours!



$$\frac{30 \text{ pizzas in process}}{5 \text{ pizzas completed per hour}} = 6 \text{ hours wait time per pizza}$$

Same math applies to our feature development.

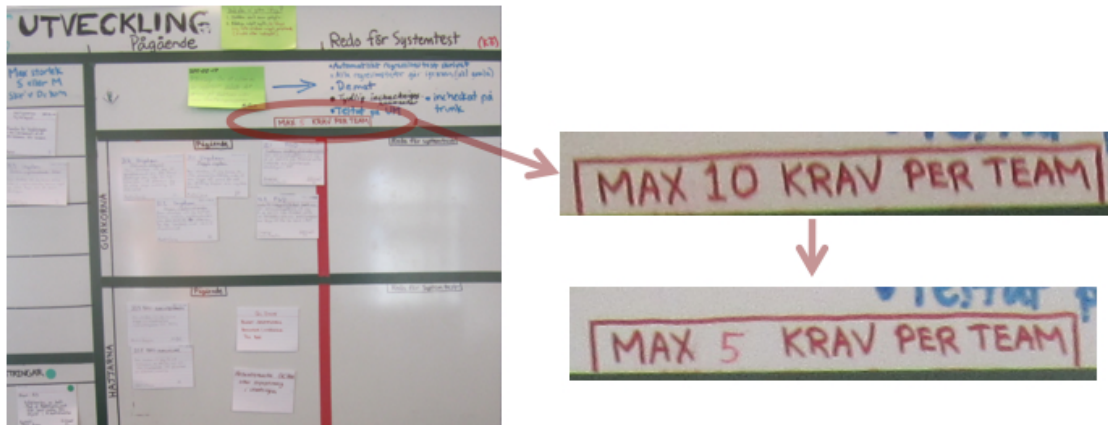


$$\frac{30 \text{ features in process}}{5 \text{ features completed per week}} = 6 \text{ weeks wait time per feature}$$

This by the way is known as "Little's Law" in queuing theory. It is inescapable.

So how do we improve cycle from 6 weeks to 2 weeks? Well, either make velocity 3 times higher (which would cost time and effort!), or reduce WIP by a factor 3. Which do you think is cheaper? Exactly!

So the teams reduced their WIP limit from 10 to 5 features per team.



That isn't a reduction by a factor 3, but still a significant reduction. When reducing WIP you have to take into account that if you reduce it too much there will be other side effects, such as having lots of people with nothing to do. That in turn negatively impacts velocity and thereby increases cycle time again. So there is a balance to be found. The goal is to have a low enough WIP limit to keep people collaborating, and to expose problems. But not low enough to expose all problems at once (which just causes frustration and unstable flow).

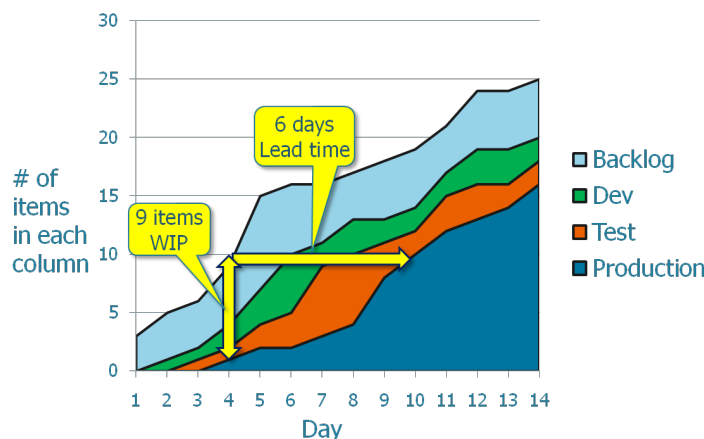
We haven't reached the target of 2 weeks per feature yet, but that's not terribly important. The purpose of the target is to keep us moving in the right direction. We've cut cycle time in half and the act of reducing WIP was one of the many things that helped us achieve this.

And it's nice to have metrics to show us that we are moving in the right direction.

Cumulative flow

I wasn't sure if I should write about this, but OK I will. In Kanban circles it is popular to visualize how the amount of work in progress shifts over time. The Kanban board shows us bottlenecks in real-time, but does not show us historical trends.

Here's how it works. Every day, count how many items are in each column. Then visualize it in a CFD (Cumulative Flow Diagram) like this:

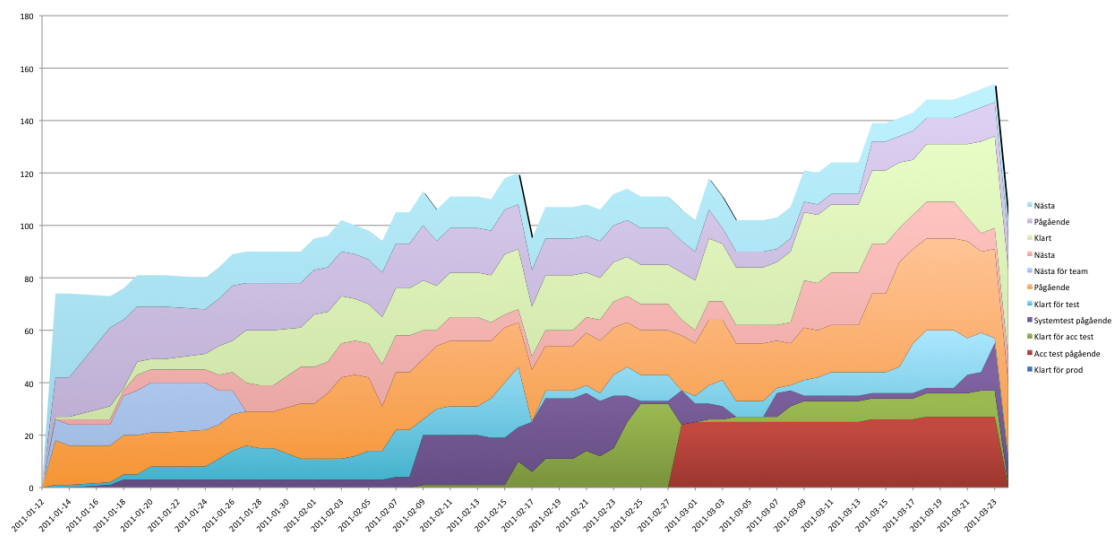


Each color is one column on the board. For every day on the X-axis we show how many items are in which column by stacking them on top of each other. Theoretically this will show where the bottleneck is, where there are obstructions to flow, and how increased WIP correlates to longer cycle time.

Great tool. In theory.

"In theory, theory and practice are the same. In practice, they are not"
- Yogi Berra

In practice this hasn't worked for us so far. Here is our CFD:



It is hard to draw any useful conclusions from this. And any conclusion we might draw are likely to be wrong. For example in the middle of the timeline it looks like we suddenly removed a bunch of work, but what actually happened is that we decided to stop including tech stories in the count. In some situations we "parked" some features on the side of the board because they were essentially paused, and the person counting the stories on the board every day didn't include these. These stories then reappeared later.

Our CFD turned out to be very "brittle" in the sense that it would become inaccurate and misleading whenever we made changes to the structure of the board or deviate from the standard flow.

We are still dutifully collecting this data though, mostly because I keep hearing from other coaches and lean folks that CFDs are useful. It only takes a few minutes per day for one person so, who knows, maybe it will come to use some day :o)

Process cycle efficiency

Here is one thing we don't measure, but I kind of wish we did. If our kanban board was mirrored in an electronic system we would definitely measure it, but in our currently pure manual kanban system it would be too fiddly.

Anyway here's what it means:

$$\text{Process cycle efficiency (\%)} = \frac{\text{Touch time}}{\text{Elapsed time}}$$

Elapsed time means how many days the feature took to cross the board (= cycle time).

Touch time means how many days the feature was actually worked on (or "touched"). In other words, how many days that card spent in "work in progress" columns, as opposed to queue/buffer columns.

This gives us very interesting data such as "hey, feature X was only 2 days of work but took 20 days to cross the board! That's only 10% process cycle efficiency!"

Most companies are in the 10-15% unless they specifically optimize for this. Trying to drive this number up is a great way to uncover and eliminate waste.

20. How we do sprint planning meetings

Our sprint planning meetings aren't really Scrum-by-the-book sprint planning meetings. There are more similarities than differences though.

The purpose of the meeting is figure out what to do next, i.e. which features go into the "next 10 features" column. In Scrum, the team is supposed to commit to specific set of features for the next sprint - we don't do that. We don't even have sprints. All we want is to agree on which features are next in line, our velocity is not stable enough to be able to predict how many features will get done in the short term.



There are two parts of the meeting: backlog grooming and top 10 selection.

Part 1: Backlog grooming

Backlog grooming is all about getting features to a "ready for development" state (see chapter 13 "How we define Ready and Done").

We used to do this during the first half of the sprint planning meeting. The requirements team would present the next features to be developed (yes, the requirements team fulfilled the equivalent of the Scrum Product Owner role, mostly because they were closest to the customer and users). Then we would break the group into small cross-functional clusters, typically one requirements analyst, one developer, and one tester. They estimate features using planning poker (see <http://www.crisp.se/planningpoker> for more info on that) and write S, M, or L on the card. If the card is L they break it down further, or decide to leave it out of "next 10 features" for the time being. They discuss and agree on a suitable acceptance test and write it on the backside.

After 20-30 minutes we have a pretty well-groomed pile of feature cards.

Part 2: Top 10 selection

This is where we look at the pile of features at hand and discuss which ones should be top 10. Usually the top 10 list isn't empty to begin with, there may be 2-3 or so features already there. In that case we evaluate those features against the new ones. The theme of the conversation is "out of everything in the whole world that we might focus on next, what are the top 10 features?".

A number of aspects influence this decision:

- **Business value** - which features will the customers be happiest to see
- **Knowledge** - which features will generate knowledge (and thereby reduce risk)?
- **Resource utilization** - we want a balance of feature areas so that each team has stuff to work on
- **Dependencies** - which features are best built together?
- **Testability** - which features are most logical to test together, and should therefore be developed in close proximity to each other?

Doing backlog grooming outside of the sprint planning meeting

After doing a few sprint planning meetings we noticed that backlog grooming took quite a long time, and sometimes the sprint planning meeting became a bit hurried as a result (since we wanted to keep the meeting time boxed and focused). So now a days most of our backlog grooming is done separately, before the sprint planning meeting. Typically one of the requirements analysts will have an informal conversation with a developer and a tester to discuss an upcoming feature and as a result of the conversation it would be broken down, estimated, and given an acceptance test.

There is still some grooming done during the sprint planning meeting, but all in all it is better to do as much grooming as possible before the sprint planning meeting. I see this trend in most other organizations that I've worked with too.

21. How we do release planning

We know our velocity, it used to be 3 features per week on average, and now it is 4-5. This is useful information for long term release planning. We don't know exactly what our velocity will be in the future, but it is probably safe to assume that it will be in the range 3-5 features per week.

So if someone wants to know "how long will it take implement this new stuff" (waving a list of feature areas) or "how much of this new stuff can we get done by Christmas", we can give a realistic answer if we know the number of features.

The problem is, for long term planning we don't know the number of features. All we have is a bunch of vague ideas. We might call these epics, or feature areas. Some can be really really huge.

OK, I mentioned before that we don't estimate features in story points because it turned out that sizes are evenly distributed so story points don't add value. However for long term planning that logic doesn't hold, since we are looking mostly at epics rather than specific features. Although our velocity of 3-5 features per week may include an occasional epic, it does *not* mean 3-6 epics per week!

The solution is simple. Take each epic, and estimate how many features it will break into. This estimation requires effort from requirements analysts, developers, and testers. The process is actually similar to estimating in story points, just that we are using the term "feature" instead of "story point", and asking "so how many features is this epic?".

Once we have that information we can count the total number of features and divide by 3-5 features per week. This will give us answer such as "We can probably build all of this new stuff in 6-12 months". A rough estimate, but based on real data.

As velocity stabilizes we will be able to make better and better predictions, so our answer might be 8-10 months (as long as we don't change the team size too much).

This way of planning is not quite part of the culture yet, there is still a tendency to fall back to the more traditional "estimate the man-hours of effort per feature and then add it all up", which takes a long time to calculate and often results in an unrealistic plan (because it isn't based on empirical data such as velocity). But we are getting there.

22. How we do version control

This will be a short chapter. Not because this is an unimportant topic, or because version control is simple. Only because I'm getting tired and want to get this paper done :o)

We've had plenty of challenges to deal with since we've been doing pretty rapid development of a complex system in a multi-team scenario. One learning is that we should have gotten our version control system in shape before scaling from 30 to 60 people. For long periods we had seriously broken trunks and branches in our version control system.

Finally we decided to implement the "mainline model", a stable-trunk pattern described in my article "Agile Version Control with Multiple Teams".

That change was a bumpy ride, but it certainly helped us get things into order. I won't write any more here about it, because then I will probably just end up duplicating that article. Read it if you are interested in this kind of stuff. Or better yet, read "Continuous Delivery" by Jez Humble and David Farley. Heavier reading, but more meat.

23. Why we use only physical kanban boards

We have number of electronic tracking systems to complement the physical kanban boards - things like the bug tracker, and various spreadsheets for metrics and release planning. But the Project Board is the "master", we have all agreed to make sure that board always reflects the truth, and that it is kept up-to-date in real time.



Any other electronic docs that duplicate information on the board are considered to be "shadows" of the board - if things are out of sync the board is always the master.

So why do we use this big messy icky analog thing with tape and stickynotes and handwritten text, when there are plenty of slick electronic tools? Most of those tools can automagically generate all kinds of detailed statistics, can be backed up, can be accessed from outside the building, can have different views, etc.

Well, one reason is evolution.



Our board has change structure many times. It took a couple of months before it started stabilizing. Only then did we start using black tape to mark the columns, before that we used hand-drawn lines because they changed so often. But we can still move the tape if we need to.

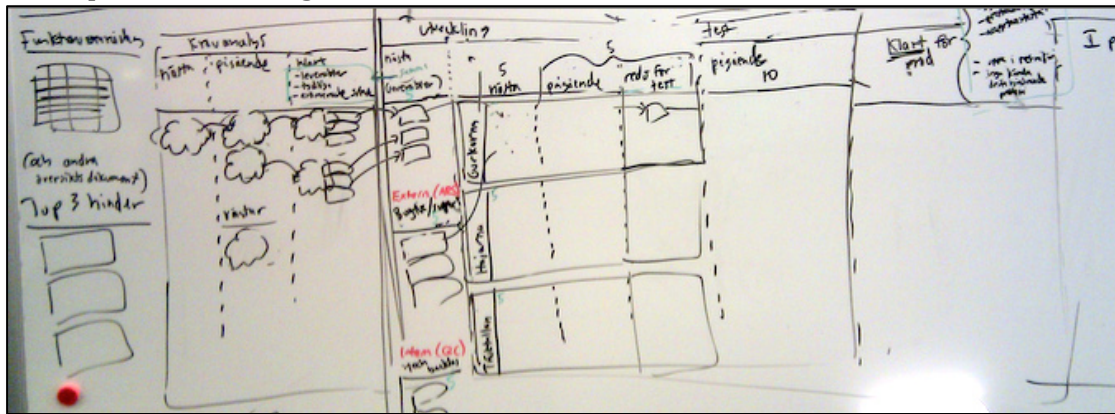
Here are some examples of changes that have happened:

- Adding or removing columns
- Adding or removing swimlanes, sometimes within a column, sometimes across the whole board
- Adding a new type of item on the board (index card, sticky note, magnet, colored tape).
- Writing down policy statements such as "definition of done"
- Writing down metrics such as velocity.
- Adding color dimensions such as "red text means defect" or "pink sticky means impediment".
- Using envelopes to group all features that were released together, and writing the release version on the cover.
- Allowing some items to be placed on the border between two teams because it is being shared by between them.
- Dividing one feature into many sub features, and keeping them together by writing a keyword at the top of each sub feature.

Each of these changes are trivial to implement. Anybody, even a 5 year old, could implement any of these changes physically on the board, once we know exactly what we want it to look like.

I have yet to see any electronic tool that can do all of the above - except possibly a generic drawing program like Google Drawing (which we actually use for our distributed Kanban board at Crisp - but that's a separate story). And if we add the rule that anybody should be able to implement the change within 15 minutes without any training - well, in that case a physical card wall is hard to beat.

At one point we redesigned the whole board based on this sketch:



It took about one hour to create the actual board based on this sketch. Once again, most electronic tools I've seen can't do this. And the ones that can require expert-level knowledge.

The second reason why we use a physical board is because of collaboration.



The "daily cocktail party" I described in chapter 7 would be very difficult to do without physical boards.

If we had an electronic Kanban board we could use a projector to display it on the wall. But we would lose most of the interaction, the part where people pick up a card from the wall and wave it while talking about it, or where people write stuff on the cards or move them around during the meeting. People would most

likely update the board while sitting at their desk - more convenient, but less collaborative.

One clear pattern I've noticed with all my clients is that boards like this can actually change the culture of an organization, and this definitely happened in the PUST project.

During a project-level retrospective for the PUST project one of the first items that came up under "keep doing" was "keep using Kanban boards to visualize the work".

We have discussed introducing an electronic tool to duplicate parts of the board at a higher level - that way we could automate some of the metrics and make a high-level electronic board visible to upper management and other stakeholders that aren't in the same physical location. This would cost some effort in keeping the digital and physical boards in sync, but may be worth it. But this would be a complement to the physical board, not a replacement.

24. Glossary - how we avoid buzzword bingo

Everyone on the PUST project speaks Swedish. Much of the Lean & Agile lingo sounds strange to normal people, especially non-English speakers. Words like "Product backlog", "Retrospective", "User story", "Velocity", "Scrum Master", and "Story points" can seriously alienate non-techies.

So I've tried to debuzzwordify as much as possible in this project. No need to alienate people unnecessarily. Being careful about our choice of terminology has turned out to be very helpful, so let me share our glossary with you.

Needless to say, this chapter is most relevant to the Swedish readers :o)

Our term	Literal translation to English	What we meant (corresponding buzzwords/synonyms)
Processförbättringsmöte	Process improvement meeting	Sprint retrospective
Leverabel	Deliverable	Feature, product backlog item
Kundleverabel	Customer deliverable	User story (as opposed to tech stories and other non-customer stuff)
Funktionsområde	Feature area	Epic, Theme
Teamledare	Team lead	Scrum Master
Projekttavla	Project board	Project-level Kanban board
Teamtavla	Team board	Team-level Kanban/Scrum hybrid board

The term "leverabel" was especially useful. Previously, the term "krav" (= "requirements") was used to mean just about anything. Now there is a clear distinction between "leverabel" and "funktionsområde".

25. Final words

Whew. This article just grew and grew. Didn't realized how much we learned.

Anyway hope this was useful!

What did you learn from this paper? What will you do? Do you have similar experiences you want to share? Feel free to send any kind of feedback to henrik.kniberg@crisp.se

If you enjoyed this paper and want more stuff like this you might want to keep an eye on my blog - <http://blog.crisp.se/henrikkniberg>

Have a nice day!

/Henrik 2011-06-10



PS - If you want to know more about me check out my web page:
<http://www.crisp.se/henrik.kniberg>

PS - If you want help to improve your development process check out <http://www.crisp.se> (mostly in Swedish) or contact info@crisp.se. We are a group of coaches and developers with an agile attitude.