JESPER BOEG

PRIMING KANBAN

A 10 step guide to optimizing flow in your software delivery system

FOREWORD BY JAMES SUTTON

InfoQ
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

Trifork Agile Excellence Mini Book Series

software pilots TRIFORK.

Biography: Jesper Boeg

Jesper has worked as an Agile and Lean coach for more than 5 years and is now in charge of the department for ”Agile Excellence” at Trifork. He has a Masters degree from Aalborg University in the area of Information Systems and wrote his thesis on how to successfully manage distributed software teams.

Jesper helps teams and organizations adopt Agile and Lean principles with a strong focus on understanding “why”. He has a reputation for being honest and straight forward, with a firm believe that change management is much more about people than process.

Jesper believes that trust is best established through an unrelenting focus on transparency in the entire organization. He has a strong passion for Lean Product Development and continuously emphasizes that one must look at the entire software delivery system to guide success.

Context Based Strategically Aligned Agility are keywords in Jesper’s work. It is his experience that to create lasting change, organizations cannot rely on Best Practice rule sets but must put effort into understanding “why” and aligning Agile principles with the overall business strategy. Otherwise they will quickly revert to former practices when faced with difficulty and restrict themselves from great improvement opportunities.

Jesper regularly speaks at Agile and Lean conferences. He is member of the GOTO Aarhus Program Advisory Board and has served as track-host on numerous GOTO and QCon conferences.

JESPER BOEG

PRIMING KANBAN

A 10 step guide to optimizing flow in your software delivery system

Written by Jesper Boeg
Foreword by James Sutton
Designed by Cecilie Marie Skov
Graphics by Cecilie Marie Skov and Jesper Boeg

First edition, October 2011
First printing
Printed in Denmark at Chronografisk A/S

Trifork A/S
Aarhus: Margrethepladsen 4, DK-8000 Aarhus C
Copenhagen: Spotorno Alle 4, DK-2630 Taastrup
Phone +45 8732 8787
E-mail: info@trifork.com

Thanks to everybody that helped review this mini-book. The readability and the content have been greatly improved as a result of your comments. A special thanks to Yuval Yeret, Karl Scotland and James Sutton for your insightful comments and taking the time to review the book in such detail.

Contents

Foreword by James Sutton

Kanban is an industrial technique for "pulling" work through its entire lifecycle, causing the work to flow more smoothly and at a higher rate. Kanban also shines a light on the activities that are normally hidden. This visibility is at two levels; for individual activities, and also over the lifecycle as a whole.

For many decades factory production has enjoyed the benefits of kanban: higher productivity, better quality, and more-satisfied workers who enjoyed increased insights into and control over their work. Until recently, however, nobody had figured out how the ideas of kanban might translate into a knowledge work field like software development. It wasn't even clear if it would help in such fields.

In the 2000s, David Anderson worked out the Rosetta Stone for such a translation, and developed the ideas of Kanban into an approach well-suited for software development. His 2010 book "Kanban" painted the picture at, say, VGA-resolution. It set an entire industry experimenting with the Kanban approach in many different product types and situations. They found that Kanban works with any development lifecycle, from Scrum to Waterfall. It complements rather than competes with them.

Anderson's book left two things still wanting, however: An additional level of detail in the theory and practice of knowledge-work Kanban, and a "starter's guide" for people about to take plunge.

The additional needed level of detail is in an upcoming book by David Anderson. You could say that it will be at HD-resolution for display on a big screen TV (I've talked to David some about the book).

The other piece of the puzzle, the "starter's guide," is what you are about to read: Jesper Boeg's mini-book "Priming Kanban." Call it an iPhone video-podcast with which to follow along wherever you do your own work. A great complement to the big HD picture.

When I recently received a draft of Jesper's book for review, I didn't think it possible for such a small book to contain both a concise overview and a practical, step-by-step worker's introduction. As I read along, though, I realized that Jesper had succeeded at both.
He has a unique gift for getting to the core of a matter, and then helping others get to the core of it for themselves.

I heartily recommend "Priming Kanban" as a practitioner's quick-start introduction to using Kanban. It will enable you to try out Kanban more quickly and painlessly…and defuse your anxiety about doing so. On a broader scale, as it inspires many people to take the same plunge, our industry will expand the use of this newest tool in our toolkit, Kanban…with all the benefits it brings.

And that's good for all of us.

James Sutton

CEO, The Jubata Group
President, Lean Software & Systems Consortium
Shingo Prize, 2007
INCOSE ESEP

Introduction

Background

Before we dive into the step-by-step guide to implementing Kanban let us spend a few minutes introducing the concept so you will be able to recognize where each step fits into the overall Kanban change management framework. The scope of this mini book is not to describe Kanban concepts in depth, for that I refer to David J. Andersons excellent book Kanban, which I strongly recommend reading:

http://www.amazon.com/Kanban-Successful-Evolutionary-Technology-Business/dp/0984521402/ref=sr_1_1?ie=UTF8&qid=1313588404&sr=8-1

Instead I hope to give a short introduction followed by step-by-step advice on how to get started.

Kanban or more precisely “Kanban system for software development” represents a more direct implementation of Lean Product Development principles in software development compared to traditional Agile methods. With a consistent focus on flow and context, Kanban offers a less prescriptive approach to Agile and has become a popular extension to traditional methods like Scrum and XP.

The word Kanban is Japanese and means “Visual Card”. The reason that Google returns more than 5 million results on a search for Kanban is however that it also used to describe the system that has been used at Toyota for decades to visually control and balance the production line and has become almost synonymous with implementation of Lean principles. So while Kanban is a relatively new concept in IT, it has been used for more than 50 years in Lean production systems at Toyota.

The use of Kanban in software is pioneered by David Anderson, who in close collaboration with Don Reinertsen, has strived to expand the knowledge of Lean and the use of Kanban to visualize and optimize the workflow in IT development, maintenance and operations.

When should I consider working with Kanban?

If the answer is yes to one or more of the following questions there is a good chance you will benefit from reading the rest of this book:

- Have you been struggling with implementing Agile in your organization for a while without much success?
- Have you been using Agile for a while and performance improvements have started to level off?
- Are you using valuable time on Agile practices that no longer seems to fit the context you are working in?
- Have you been using Agile as a checklist without fully understanding the underlying principles?
- Do you have a need for more flexibility than frozen, committed and planned iterations have to offer?
- Do your priorities shift on a daily basis?
- Are you using processes designed for Agile product development in a context where they are not an easy fit, e.g. maintenance and operations?
- Do you need a gradual transition from waterfall type execution to Agile to avoid high levels of organizational resistance?

No matter if your goal is to work in a strict Scrum context, if you are using waterfall or trying to find a way to super optimize your current Agile implementation, most will benefit from the deeper understanding of Lean that Kanban has proven to be an excellent catalyst for.

What is Kanban?

There are a variety of approaches to Kanban but most agree that Kanban is a change management method focusing on the following principles:

- **Visualize Work**
  - Visualize every step in your value chain from vague concept to releasable software.
- **Limit Work-In-Progress (WIP)**
  - Set explicit limits on the amount of work allowed in each stage
- **Make Policies Explicit**
  - Make the policies you are acting according to explicit
- **Measure and Manage Flow**
  - Measure and Manage Flow to make informed decisions and visualize consequence
- **Identify Improvement Opportunities**
  - Create a Kaizen culture where continuous improvement is everyone's job.

Those of you familiar with Lean will recognize many of these principles as the foundation for a Lean pull system and what Kanban first and foremost does is actually serve as a catalyst to introduce Lean ideas into software delivery systems. This is also stated in David's book:

> *"...Kanban (capital K) is the evolutionary change method that utilizes a kanban (small k) pull system, visualization, and other tools to catalyze the introduction of Lean ideas into technology development and IT operations"*
>
> David J. Anderson, Kanban 2010

We will show a lot of examples of Kanban boards in the following chapters and explain the mechanics but to give you an idea of the concept figure 1 shows an example.

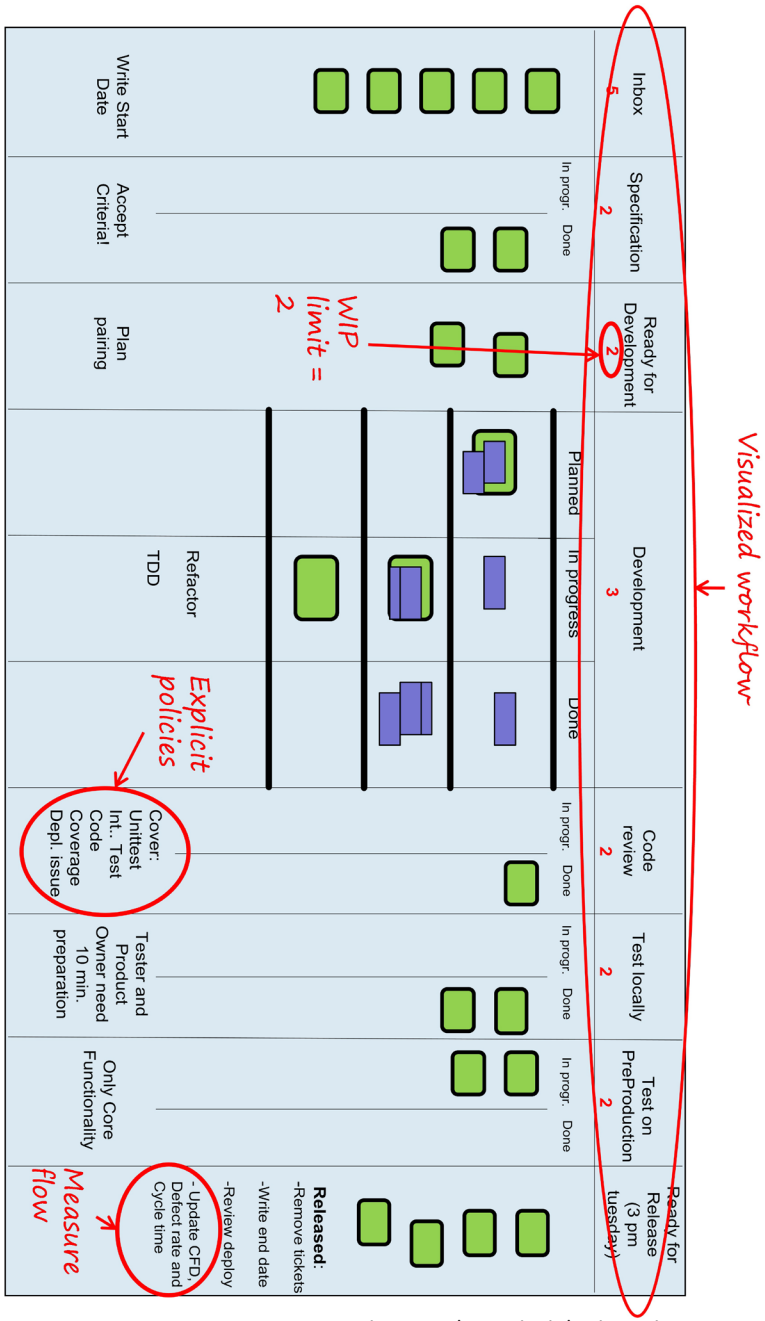

Fig. 1 Kanban Principles in Action

As you can see all work is made visible. WIP limits are in place (written in each column header). Policies are made explicit and flow is measured. As you will notice the last column does not have a WIP limit assigned. This is due to the fact that this particular team has opted for a regular weakly release cadence (3 pm Tuesday) which means that all finished work is released at this time.

Kanban is all about driving evolutionary change and these simple steps have proven extremely helpful in doing that. The reason we refer to such a system as a "Kanban Pull System" is that visualized flow and WIP limits ensure that you can never introduce more work into the system than it has capacity to handle. There is only a certain number of work permits (kanbans) available so you must complete existing work before new work can be started. This results in functionality being pulled through the system based on capacity rather than pushed based on forecasts or demand.

How do we get started with Kanban?

Hopefully the 10 steps in this book will get you well on your way but before we get that far it is important to understand that Kanban has a different approach to change management than most other Agile methods.

Kanban is built on the concept of evolutionary change. Start by understanding how your current software delivery system works. When you have managed to visualize, measure and manage your flow improve it one step at the time by relieving the largest bottleneck. This is quite different compared to e.g. Scrum where you will often start out by redefining roles, process and artifacts. This makes Kanban ideally fitted for use on top of existing processes which can be anything from Scrum to Waterfall and perfect in situations where organizational structures inhibits radical change.

In Lean terms, this means that Kanban is primarily build on the concept of Kaizen (continuous improvement) and only uses Kaikaku (dramatic change) in special situations where structural change is needed or serious performance leverage need to be identified.

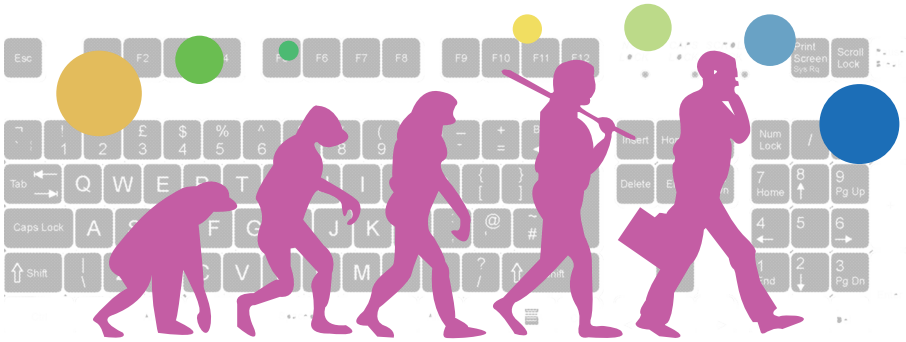

Fig. 2 Kanban Takes an Evolutionary Approach to Process Optimization

Where can Kanban be used?

Now we are almost ready to get started. But before we dive into the implementation details let us just quickly bust some of the myths about Kanban to make sure the following sections are read with the right mindset in place. At Trifork we have helped a lot of companies and teams increase their effectiveness by adopting Kanban. At first it seemed the primary target group were teams working with maintenance and operation, but Kanban has proven to be just as helpful for software development. Also teams and organizations working with waterfall-like methods have found the evolutionary approach extraordinarily helpful in a gradual transition to Agile product development.

Kanban Myths

- **Myth:** Kanban is only suitable for teams working with small uniform tasks like those seen in operation and maintenance.
- **Fact:** Kanban is heavily inspired by Don Reinertsen's work with Lean Product Development and has proven to be as good as fit to software development as it is for operation and maintenance.

- **Myth:** Kanban and Scrum are opposites.
- **Fact:** None of the principles in Kanban restricts you from doing Scrum. Kanban acts as a change agent and the principles in Scrum should therefore only be used in cases where they help optimize flow. Nothing is keeping you from starting with Scrum and using Kanban to drive further change – many projects have been incredibly successful with this strategy. Some might even argue that it was the original intention with Scrum as well but somehow it got lost in the focus on ceremonies, roles and artifacts.

- **Myth:** By not insisting on planned committed iterations Kanban falls prey to Parkinson's Law that "Work expands so as to fill the time available for its completion".
- **Fact:** Though being a valid concern Kanban projects rarely display this behavior since fixed cadences, extreme visualization, cycle time measurement and tighter feedback loops with business keep focus tight and work items flowing.

- ***Myth:*** Kanban teams do not use timeboxes.
- **Fact:** Timeboxes are not mandatory, but should be used when they help optimize flow, feedback and quality. Most Kanban teams use fixed but decoupled cadences of planning, review and releases and thereby dispense with the traditional iteration model while keeping the value intact.

- **Myth:** Kanban teams do not estimate.
- **Fact:** Estimates are not mandatory, but should be used when appropriate. Most Kanban development projects use some degree of initial sizing to ensure optimal portfolio management, prioritization and alignment or use estimates for work that is more sensitive to cost/benefit analysis or due date performance.

- **Myth:** Kanban is better than Scrum/XP/Crystal/FDD….
- **Fact:** Kanban is first and foremost a catalyst for driving change and therefore needs a starting point. So while most projects will benefit from using Kanban it is not a substitute for e.g. Scrum. Scrum is in most cases a perfect starting point when adopting Kanban.

Though the Kanban community is constantly fighting these and other myths, still Kanban remains one of the most misunderstood concepts in the Agile community to this date for two main reasons:

A lot of the noise and confusion revolves around the fact that Kanban is a change method and therefore has very few descriptive parts telling you how to work, which roles to fill etc. Since the concept of a change method is poorly understood, people have tried to compare it to prescriptive methods like Scrum and XP.

Local examples and emergent behaviors of Kanban, in real world projects, have come to represent a “Kanban method” to some people. It is easy to see why people misunderstand since many Kanban projects show the same emergent practices. Reality however is that Kanban is about using Lean principles to optimize existing processes in an evolutionary way, and therefore cannot and should not be compared to Scrum, XP, Crystal, FDD or whatever method you are using.

The second reason is arguably that the word “Kanban” might carry too much baggage from its origins in Lean production systems to be an adequate word to describe a change method for software development and IT operations. Though kanban pull systems, as they are used in production systems, do drive change, Kanban in software builds on a much broader set of Lean principles and that creates a difficult mental gap for those having worked extensively with Lean in the past. Toyota does however use improvement Katas which guides the usage of tools like Kanban in order to continuously improve performance.

The good news is that most projects, Agile or non Agile, can benefit hugely from using the principles of Kanban to drive change and continuous improvement which I hope to demonstrate in this book.

The observant reader might have noticed that the title of this book is not exactly in line with the Kanban principles. As David Anderson wrote in a tweet June 30. 2011 “… Kanban system design is a thinking process not a copying or template implementing process”. However, those familiar with the “Dreyfus Model of Skill Acquisition” will recognize that whenever acquiring a new skill you need prescriptions at first and my hope is that this “primer” will help you transition quicker and more painlessly. The important thing is to know that prescriptions only serve as a way for you to gain knowledge to move forward and not as an endpoint or a checklist to be followed blindly. The templates and practices suggested in the 10 steps in this book are emergent behaviors experienced in Kanban projects and not the Kanban change method itself.

Now that we have got the general picture lets go on to how we can apply these things in practice. Each step consists of a short explanation about “why” followed by “how”.

Step 1
Visualize your workflow

First step towards visualizing your workflow is to understand how your currentsystem works.

Understanding your software delivery system

To be able to make informed decisions about how to best optimize your workflow the first step is to understand what you are doing. The important thing here is to resist the temptation to change anything. Just find out how you are working without idealizing it. The key is to try to map your entire software delivery workflow and not just focus on the “development” part.

There a number of different ways to do this. The most popular way is to use the Lean concept of Value Stream Maps (VSM). Recently VSMs have taken quite a beating from Agile and other knowledge work communities, the main argument being that knowledge work is not a linear process like the ones seen in production systems. This has led to the evolvement of techniques like Knowledge Creation Networks better suited to handle non linear work. For this simple example we will however use the more simple VSM technique which I still find extraordinary helpful but you should explore the option that fits your context.

In its simplest form a Value Stream Map is a visualization of the stages our work passes through from raw material to finished product or in the case of software from vague idea to a feature working in production. The key thing when doing this for knowledge work is to think of each stage as the primary form of information arrival. For example, a stage called “Test” includes more work than just testing (fixing, refactoring, discussions, updating accept criteria etc.) but since the primary form of information arrival is “Test” we will define our work as being in the “Test” stage while all these activities are going on. The space between our stages, where no information is being added is defined as “wait time”. Figure 3 shows an example from a real project.

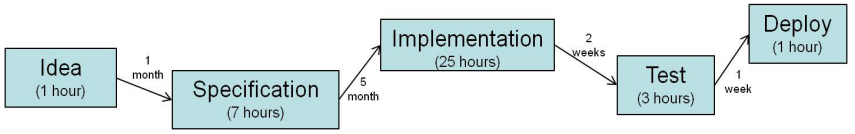

Fig. 3 Value Stream Map Example

We might later decide that implementation actually consists of: User story breakdown, Development and Code Review, but let's stick with this simple version for now. Already improvement ideas spring into our mind: Why is there an average wait time of 5 months from specification to implementation? Why are we waiting 2 weeks before testing? Resists the temptation to fix these issues right now – there will be plenty of time for that later and right now our focus is on understanding how our current system works.

Initially it is a good idea to limit the number of stages in the value stream map and kanban board. With too many stages in place you quickly lose sight of the big picture and focus on the mechanics. Later on you might find it beneficial to add more details but keep it simple for now.

Visualizing your system

Now that we have gained a better understanding of our software delivery system the next step is to try to visualize it. We can do this in an electronic system or simply using a whiteboard. Unless you are working in a distributed team it is usually a good idea to start out with just the whiteboard, nothing makes your work more visible than having it right in front of you all the time and being able to physically touch it. As team maturity increases and you find the need to collect more data you might want to move to an electronic version but stick to the whiteboard for now.

Often you will end up with at least two types of stages: "Activity" stages where active work is being performed and "Buffer" stages where work is waiting to be released/developed etc. but more on that later.

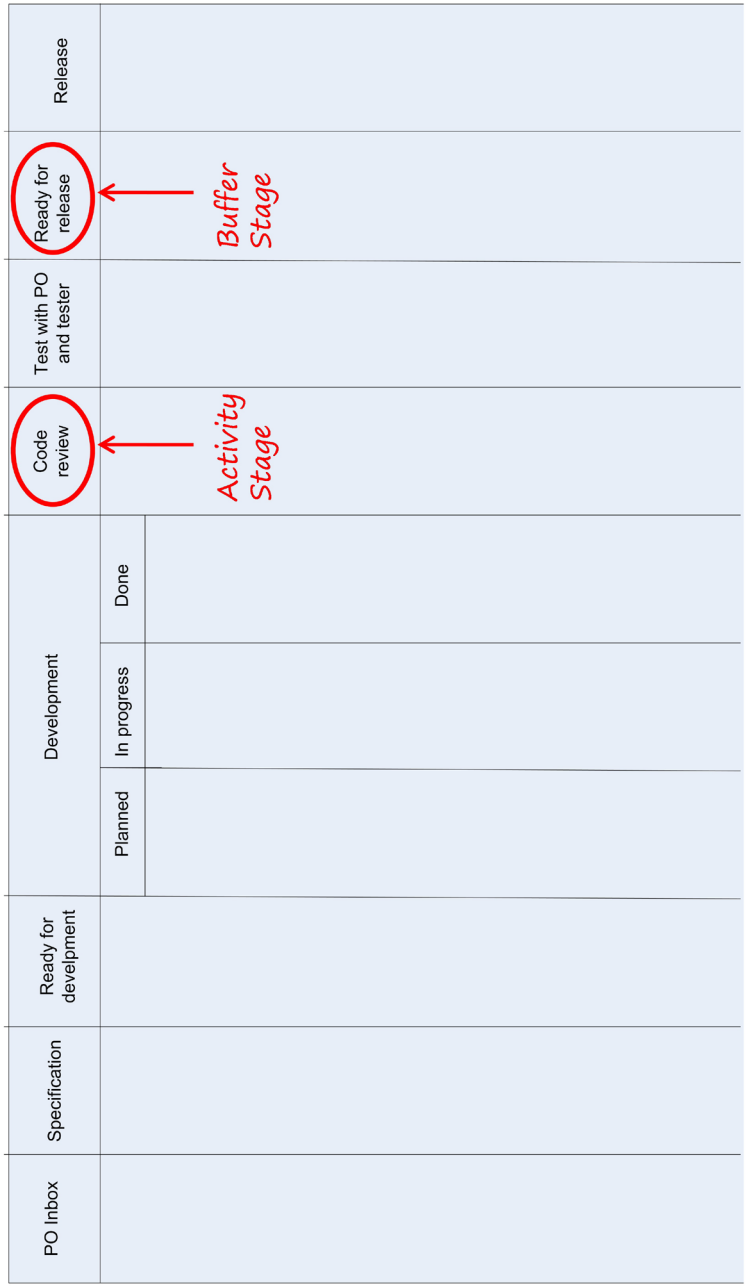

Fig. 4 Activity Stage vs. Buffer Stage

The first version of your board might look something like the one shown in figure 4 (Notice that all the work necessary to complete a given feature is represented, not just development)

Every feature starts out as a vague idea in the PO Inbox and ends up in the "Releasing" column where it is removed from when the feature is actually working in production. Notice that we haven't changed anything - you might still be following a strict Scrum implementation.

If visualizing your work proves a difficult task now is the place to stop. Don't go any further before you have managed to visualize all the work you do. If information is hidden and some tasks are completed outside your workflow system there is little chance that you will ever be able to make informed decisions about how best to optimize. A general rule is "you can only mange the work you can see". Visualizing work may sound deceptively simple but can prove difficult in real life. Reasons may be many: People know they are doing things they shouldn't. They are afraid they will be punished if their superiors know how things really work. Though stressed out they feel they will let their colleagues down if they are not constantly firefighting. You need to fix these problems and make everybody involved understand that no one will be blamed or discredited for displaying the current status, before moving on.

Visualizing your workflow gives a number of benefits the most important being:

Focus on "The Whole"

- It becomes visible exactly how your work affects others and vice versa

Transparency

- Everybody knows exactly what is going on and no information is hidden

Identifying waste

- You naturally start to question why you are doing things the way you are(more on that later)

Step 2 Limit Work in Progress (WIP)

When you have managed to visualize your workflow and spent a few weeks observing your system you are ready to proceed to the next step - limiting WIP. Though it might be tempting to do this immediately, visualizing work is often not as easy as it seems and therefore it is often a good idea to spend some time exploring this aspect before continuing on.

Understanding WIP

To understand why, limiting WIP makes sense, we need to take a look at Little's law which states that (adapted to product development terminology): Cycle time = WIP / Throughput per unit of time

Cycle Time describes the time it takes for a work item to pass through our system or in other words "the time it takes from a feature is selected for implementation until it is working in production". How you define "selected for implementation" depends on your context. For some it is the placement of an item on the backlog and for others it might be the time an item is selected for detailed specification. You might also want to distinguish between the two and refer to the time an item arrives until it is delivered as "Lead Time" and the time from it is selected for implementation until it is delivered as "Cycle Time".

WIP describes the amount of Work In Progress in our system. How many "story points"/"user stories"/"backlog items" are currently in progress in our system. Again it depends on the context. Some include all items on the backlog in WIP while others consider only the items selected for implementation.

Throughput per unit of time is simply the average number of items produced in a given period of time. In Scrum this is usually referred to as velocity.

This means that given a system with 100 user stories in progress (WIP) and a throughput of 2 user stories per week the average cycle time is 100/2 = 50 weeks or almost a year. Reducing this to 25 weeks can be done by either doubling throughput to 4 user stories per week or reducing the number of user stories in progress to 50. In most cases it is initially much easier to reduce WIP than doubling throughput.

As you might have guessed limiting WIP is all about reducing cycle time to increase flow and minimize the amount of work we have invested time and resources in, but has yet to generate any business value. Fast feedback cycles are also a great way to minimize risk, since decisions are validated continuously and quality issues are exposed immediately. A subject explored in detail in Capers Jones Cost of Quality (1980). But why not just increase throughput you might ask? The simple answer, I will ask you to accept for now is that it is often many times easier to limit WIP than increase throughput.

So how do we do this? Well actually all we have to do initially is to make our best effort to define how many items we will allow in each stage of our board at any given time. A good idea is to let this exercise be guided by the policies your team would like to enforce. If the team decides that it is a good idea that no developer should work on a user story single handedly you might choose a limit of 3 for a team of 6. Note that this is only true for activity columns like "development", "test" etc. For buffer columns like "ready for development" the general rule is that if it is empty once a year the WIP limit is too large (364 days you are working with a larger buffer than needed) and if it is empty once a day it is too small. People often joke that the universal WIP limit is 5 so if in doubt 5 is probably a good number to start with.

Visualizing WIP Limits

How you visualize your limit is up to you. Figure 5 and 6 show two common ways of doing it. In figure 5, only one item may be placed in a box and that gives you a very visual signal when you have a "permit" to start/pull new work (the box is empty). In figure 6, it is easier to divide the activity stage in "in progress" and "done" since the WIP limit is simply written in each column header. This can give you additional insight into how your system is working and is the common way of doing it in IT. People having worked with Lean manufacturing might opt for the first version since in more closely resembles the visual pull signal of the plastic card.

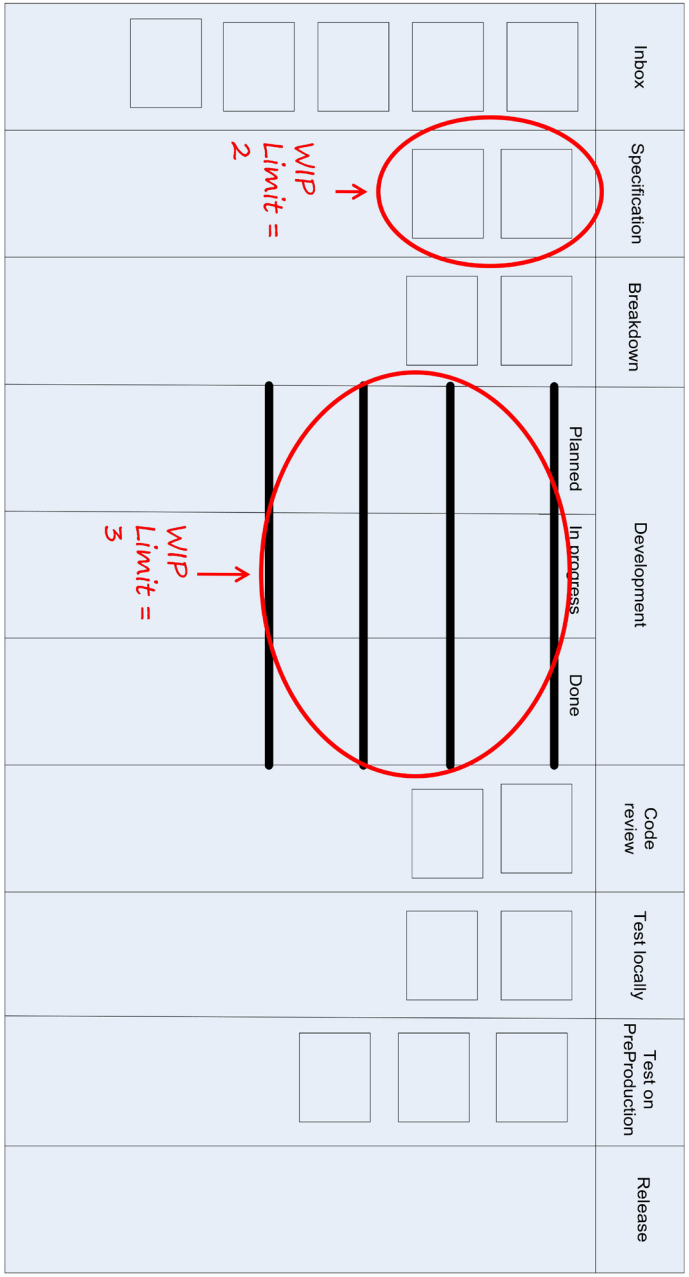

Fig. 5 WIP Limits Visualized Using Containers

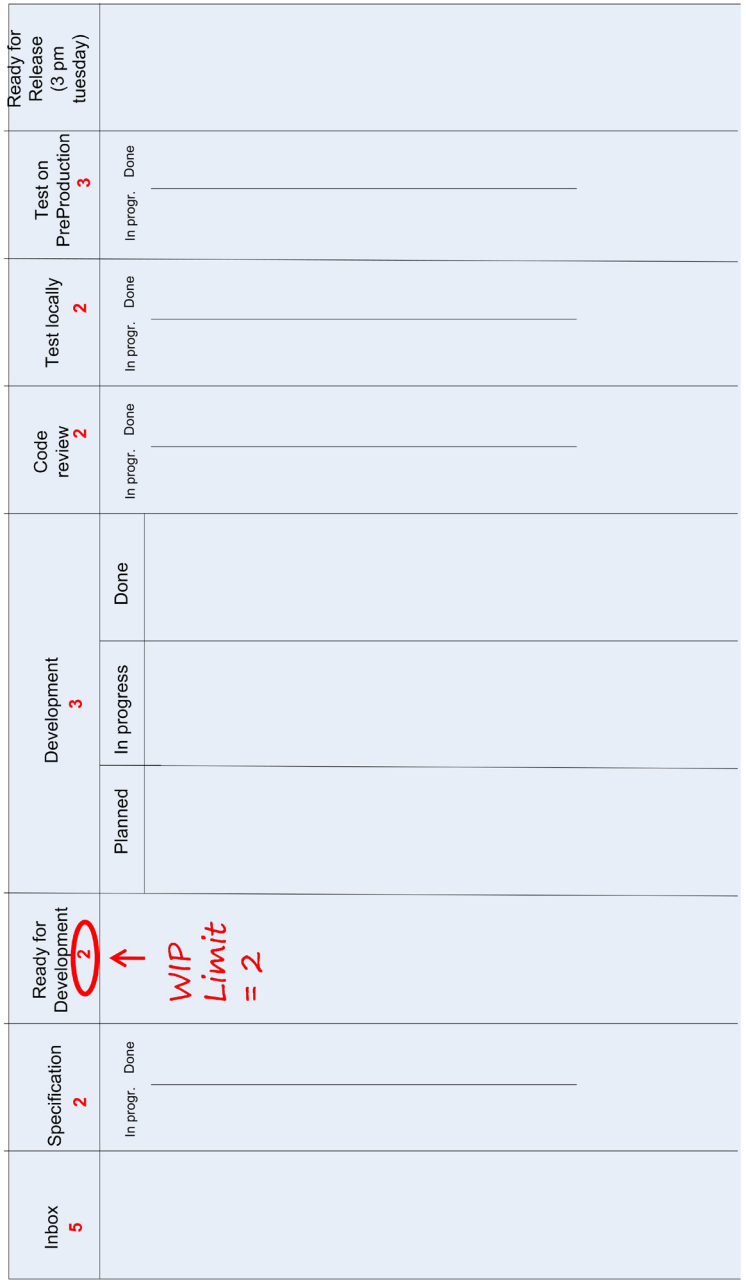

Fig. 6 WIP Limits Visualized Using Numbers in Column Headers

Finding the right WIP limits

There are many schools of thought in terms of how tight you should set you WIP limits initially and it is out of scope for this mini-book to cover the subject in detail. One way is to observe your system and set the limits just loose enough for your current workflow to continue unhindered. Then identify your bottleneck and adjust one limit at the time. A more radical approach is to set your limits on activity columns tighter than you expect your system to be able to handle and buffer each stage. Then you observe where work builds up and gradually loosen until work flows through the system. Both require some experience and do not expect to get it right the first time. There is no final conclusion as to which one is better, but setting the limits with your policies in mind seems to work in both circumstances.

In any circumstance it is important to set an explicit policy of how the decision to break or change the limit will be made. To maximize learning it is a good idea to make such decisions together as a team. This ensures that everybody get to voice their opinion and understand the decision. This is not just a matter of flow but also a learning point!

Always remember that your initial limits are just best guesses, given at a place in time where you had the least amount of information available. As you gain more information about your system, limits should be adjusted continuously as you find the more optimal ways of working. If you are still working with your initial limits and the same stages 3 months after you started, there is a good chance that you have missed the most important step in this guide, namely the continuous improvement step we will cover in more detail later. Limits that are too tight will block the flow and make people idle for too long or simply be ignored without serious discussion, while limits that are too large will increase cycle time and make work items idle for too long.

What you will quickly notice is that with WIP limits in place your system can only work to capacity. You need to finish work to get a permission to start a new thing. While sounding trivial it is the core concept of a Lean pull scheduling system and an incredible powerful tool on your journey to a more effective, sustainable and predictable software delivery system. A popular metaphor is to think of the system as a chain of paper clips.

As long as you are pulling it across the table they follow a nice line but if you push it instead they all crumble together in a mess, each item blocking the rest. (Illustrated in figure 7)

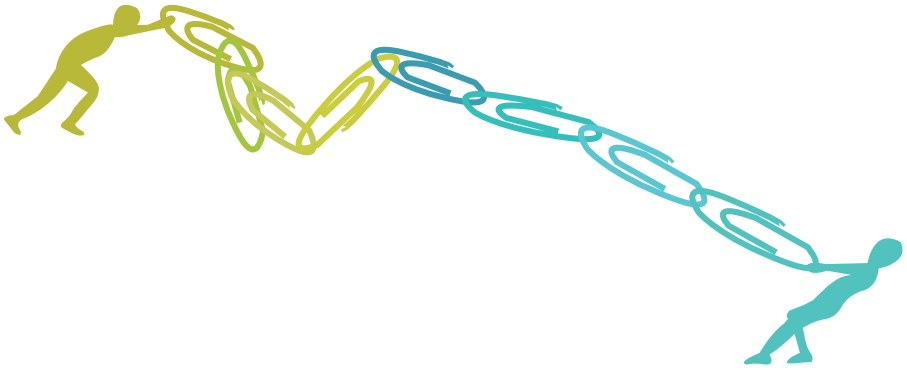

Fig. 7 Pull vs. Push

You will also quickly notice a certain pain that happens when you don't get permission to start a new thing while you think that is "the right thing" to do. This is a sign that you are discovering an impediment to flow – and the most important thing is not to be "Comfortably Numb" about the pain but to leverage it for improvement.

Step 3
Set Up Quality Assurance Policies and Make Them Explicit

If you are familiar with Lean you might have come across the term “Quality built in”. Why is quality so important you might ask? Isn’t the main thing that we fix the bugs that find their way into production? The simple answer is that quality issues are much more expensive than you think.

Understanding quality

Whether it is a user that cannot complete a task because the system lacks an intuitive interface or it is a bug that blocks the workflow. They are both quality problems and they both stress our system and generate a whole loop of waste. This is better known as “failure demand” in Lean systems and describes all activities and rework related to the product not being designed properly in the first place. Sometimes it is acceptable since releasing the software to get real feedback was the cheapest way to buy information. It might also be the case that finding this bug up front would have cost us a lot more time and money than fixing it afterwards. In the majority of cases however it is simply caused by an immature process.

So why is it so expensive? Let’s look at a couple of common scenarios in software development:

A user (Let’s call him John) cannot complete his task because of a bug in our system. John writes a bug report or calls first level support to address the problem. John however knows little about what it takes for a developer to be able to investigate the issue or maybe the first level supporter does not know the system as well as he should. In both cases, wrong or inadequate information is given to the developer who ends up solving a different problem (which might not actually be a problem but instead leads to another bug) or simply give up. This continues until finally the bug is solved. However, not only were John not able to complete his task, faulty information was actually saved to the database which now needs a complicated SQL script to be reverted to a meaningful state. It is not uncommon for situations like this to occur and a factor of 100-1000 time and resources spent compared to having spent the time not introducing the bug in the first place.

cannot remember the complicated problem we were working on and have to spend an extra half an hour getting back into the context.

For these reasons Lean puts a huge emphasis on fail proofing the delivery system (Poka Yoke). In production systems, Poka Yoke is done by using standards and checklists that must be followed when completing a task. Even photocells are used to register whether a specific screwdriver is used the correct number of times or all parts needed have been removed from the stack. This might sound like an inhumane environment to work in but actually workers in Lean production systems do not see themselves as robots blindly following standards and checklists. They see themselves as expert operators that are constantly trying to improve the system they are working in by coming up with new ideas as to how it can be improved. In the "The Elegant Solution: Toyota's Formula for Mastering Innovation, 2006" Matthew E. May refers to this fact as the main reason Toyota still manages to IMPLEMENT one million new improvements every year.

Visualizing policies

So how does this translate to software development? Well actually you are half way there. You have already visualized your work on the board and put WIP limits in place. All you need to do now is add the policies you are already using to ensure quality and consistency. Doing this, your board might end up looking like the one shown in figure 8.

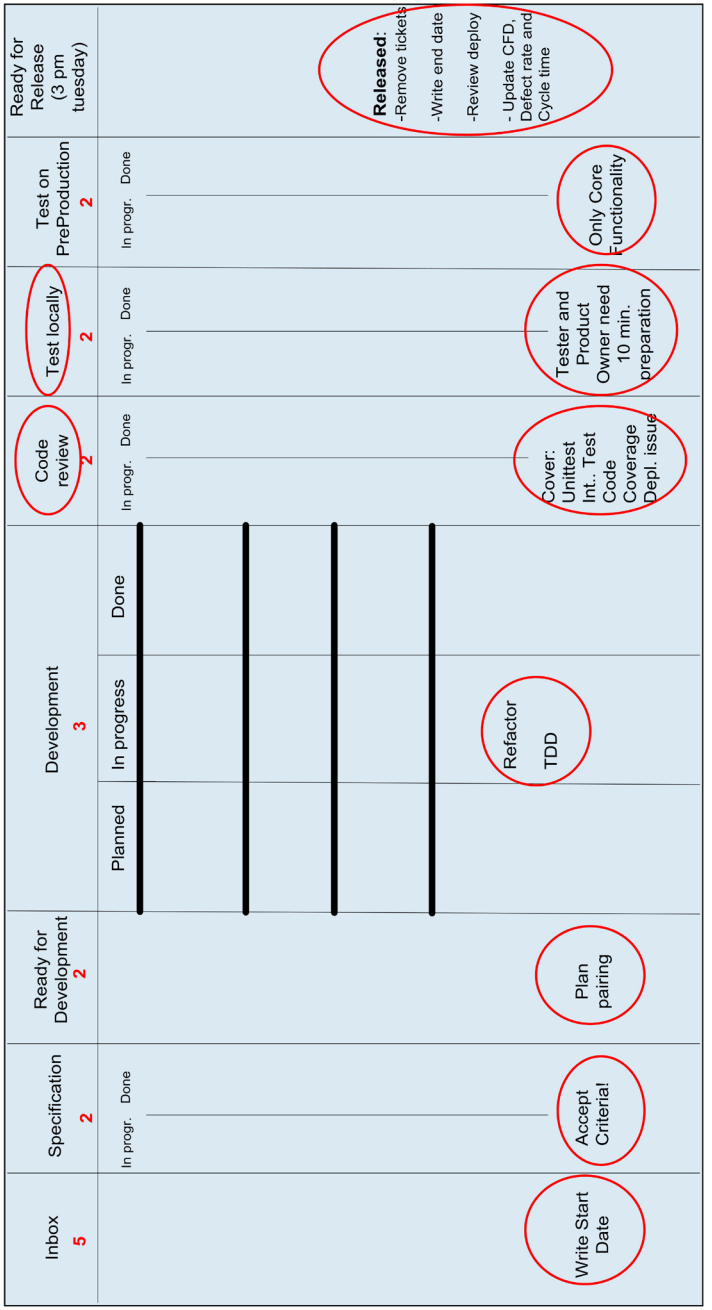

Fig. 8 Explicit Policies Visualized on Board

Note that entire stages may be QA policies and that policies also serve to ensure consistency and quality in the process itself (e.g. tracking cycle time and defect rate). All of it traces back to the third Kanban principle “making policies explicit”.

Many teams have taken fail proofing to extraordinary levels in recent years and have implemented systems that were unthinkable just a few years ago. The “Lean Startup Movement” has shown that it is possible to deploy software to production several times a day without down time or high defect rates. This can only work because they have put unit, integration, regression and performance test in place that help validate the code as well as Key Performance Indicators that signal an alarm whenever the system is not behaving as expected and automatically rolls back to the previous versions. This makes it impossible to introduce whole classes of errors.

Always keep in mind that you should never feel that you are slaves to your policies and checklists. You are an expert knowledge worker constantly observing and trying to improve the system you are working in, not a robot. Start by adding the procedures you are using at the present time and add/remove/change them when you discover new and better ways of ensuring quality. Every bug is a chance to reflect on how it managed to get into your system.

Initially, the thought of reflecting and learning from each and every bug will seem daunting. With time, as quality improves, it will however become more and more natural. This is a price worth paying and is referred to as ”Zero Tolerance” in Agile Testing circles.

Step 4
Adjust Cadences

Once you have managed to visualize your flow, limit WIP and establish QA policies one of the first things you should evaluate are your cadences. In a typical software delivery system a number of activities benefit from regular cadences and finding the right one for each type is paramount to increasing flow. Note that in a Kanban system we are not obligated to synchronize everything to the lowest common denominator. We can adjust the cadence of each activity to its own optimal level. Typical cadences we need to consider are planning (input) cadence and delivery (output) cadence. A lot of other cadences like review/retrospective cadence and quality assurance cadence (if you are not a true Agile project☺) of course also exist but let us stick to planning and delivery for now.

Understanding Cadence

Finding the right delivery cadence is one of the most important things in Lean product development since it helps you optimize essential feedback loops, reduce risk and optimize your delivery process.

Though releasing every feature directly to production is the most optimal solution (given that you have a system optimized to handle this) in reality most projects work with two delivery cadences.

- One cadence where code is deployed to a preproduction system to obtain initial feedback (internal release cadence)
- One cadence where the new version is deployed to the actual production environment (external release cadence)

Especially on "Greenfield" projects these two cadences can be very far apart. It may take 5 month before the system is feature complete enough to hit production while code is being deployed and tested every day on the preproduction environment.

The important thing when choosing the right internal and external release cadence is to be aware of the economic cost of your choice. There is always a transaction cost (the cost of moving your version from one environment to another) associated with a release and there is always a holding cost associated with waiting. The balance between these two describes the optimal cadence which is visualized in figure 9 from Don Reinertsen's book on Lean Product Development Flow (used with permission).

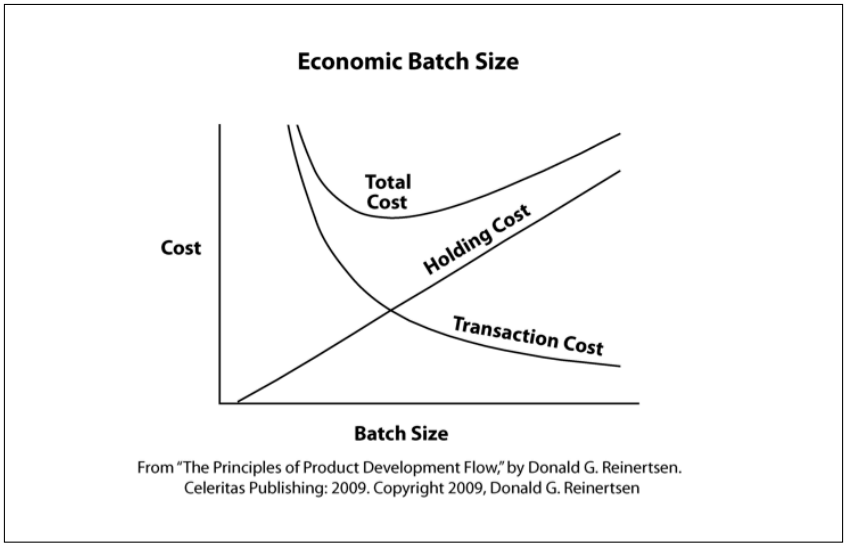

Fig. 9 Batch Size Optimization

The more features you bundle together in a release, the cheaper the cost per feature (lower transaction cost), but also a higher holding cost since each feature will have to wait longer getting deployed resulting in loss of business value, outdated feedback, uninformed decisions and lower user involvement.

Holding costs are slightly different for external and internal releases. Since an internal release does not expose any real business value to customers, holding costs represent only outdated feedback, uninformed decisions and decreased user/business motivation due to low involvement. Anyone having worked in a real business context will however recognize that these can be as detrimental as lost revenue.

As you can see from the figure the "total cost" u-curve has a pretty flat "bottom". Therefore it does not really matter if you hit the optimal release cadence. Being 10 or 15 percent off will still generate a good result.

Finding the right cadences

The problem is that many projects do not even consider this very carefully. Many mature Agile teams are able to release to preproduction environments with the click of a button, but still they wait a full 3 weeks before getting feedback on a given feature from users. When transaction costs can be measured in single dollars you should strongly consider working with very small batch sizes. Sometimes this is problematic since users are not available but in most cases it has simply not been considered.

Another key consideration, that Toyota taught us, is that transaction costs are not fixed. The continuous deployment movement has shown us that it is possible to deliver reliable versions to production 50 times a day for systems handling millions of dollars. This can only be done by having a fully automated deployment procedure and a whole suite of unit, integration and regression tests, which is of course an investment but does allow you to work in batch sizes of a few lines of code.

Adjusting the planning cadence should be done with similar considerations. When the time between planning meetings gets longer, more stuff have to be planned in one large batch. This results in more design in progress and less informed decisions due to longer cycle time. On the other hand, meeting everyday might prove too large of an overhead and raise transaction costs. In some cases Kanban teams choose to plan on demand instead. This can be done be sending an email to stakeholders whenever e.g. 3 slots are empty in the input queue and arranging a meeting or a conference call to fill them with the highest priorities. Usually "on demand" planning is reserved for more advanced teams and requires some prior experience handling flow. So consider carefully if this is should be your initial strategy.

Step 5
Measure Flow

Measuring software development projects and their progress is unfortunately one of the most misunderstood and misapplied aspects of software development we come across. Often metrics are used to hold project managers accountable for aspects they had no control over in the first place or as fixed success criteria established when people knew the least about the system to be developed.

Understanding Metrics

When discussing metrics I find one ground rule should always be remembered "your software delivery system only has a certain capacity". If you try to press your system beyond its capacity it will lead to lower quality, unsustainable pace, higher maintenance costs or all of the above. But still, time and time again we see project managers almost bragging that they have made their teams work overtime for 3 months or that by some heroic effort they fixed things at the last moment when everything was total chaos. Though we should celebrate great achievements software development projects do not need fixers, they need people that are able to deliver with transparency and a healthy sustainable pace. Everything else is simply too expensive. I like Kent Beck's statement that "if you have a problem that requires more than one week of overtime you have a problem that should not be fixed by working overtime anyway".

You may of course increase your capacity over time by hiring more people (beware of doing that for short term results) or optimizing your process. Another good ground rule to consider here is that "your system never has more capacity than it has PROVEN to be able to deliver". Following this simple rule will also keep you from managing projects by the anti pattern of "wishful thinking and other people's successes". Starting on a Greenfield project, your capacity and capability will of course be informed guesses from previous performance and the key here is to track progress from the beginning to validate those assumptions. More on that topic in step 8.

So think of your plan as a tool for alignment, not a success criterion, and measure your flow to determine whether you are still aligned. What we want is our software delivery system to be stable and predictable so we can make informed decisions about deadlines, dependencies, staffing, scope and budget.

What to measure?

So how do you measure flow? There are dozens of ways to do this and the main thing to consider is always "will I act on this piece of information". If you are not going to change anything based on a chosen metric chances are that you shouldn't be measuring it at all. If you have no idea where to start, I suggest starting with the following four: Cumulative Flow Diagram, Cycle Time, Defect Rate and Blocked Items.

Cumulative flow diagrams (CFD)

Cumulative flow diagrams seem to be replacing burn down charts for more mature Agile teams and organizations for good reasons. They are easy/easier to update and give you better insight into the project's status. For those unfamiliar with the concept of CFD's, they simply display the current amount of work in your system for each stage over time. While this may sound simplistic it provides you with the same kind of information as the traditional burn down chart plus a lot more. Figure 10 shows an example of a CFD.

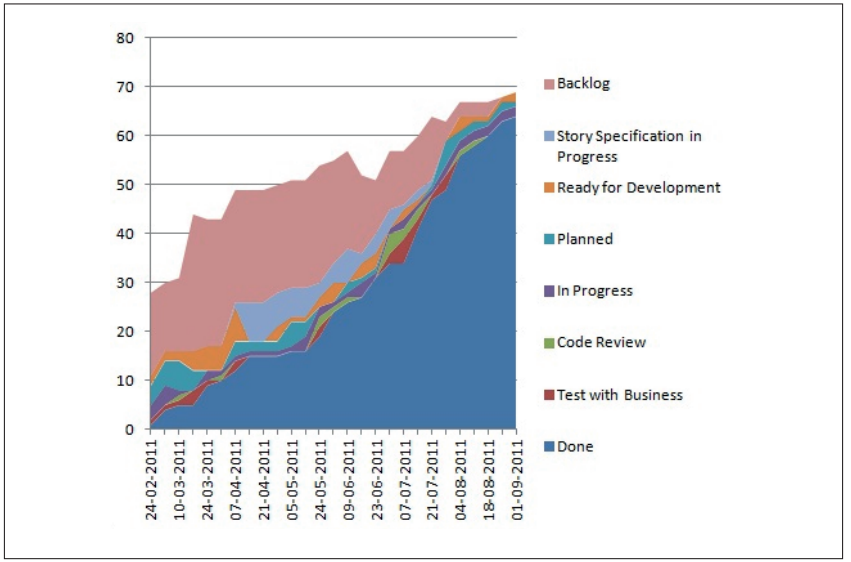

Fig. 10 Cumulative Flow Diagram Example

Reading the CFD

The gradient of the “done” area describes your velocity over time while the space between this line and the “backlog” line may be defined as WIP.

- If the width, of a part of the WIP area, increase, it could be a sign that a bottleneck is occurring.
- If the gradient of the “backlog” area is steeper than the gradient of the “done” area it is a clear sign that you are adding more work to your system than your current capacity.
- Projecting where the gradients of “backlog” and “done” cross is your current best guess of a final release date.
- Average Cycle time and Quantity in queue can also be established from the diagram.

Learning to read a CFD is easy and figure 11 from Don Reinertsen’s book (used with permission) gives an excellent visual representation. Black area equals WIP.

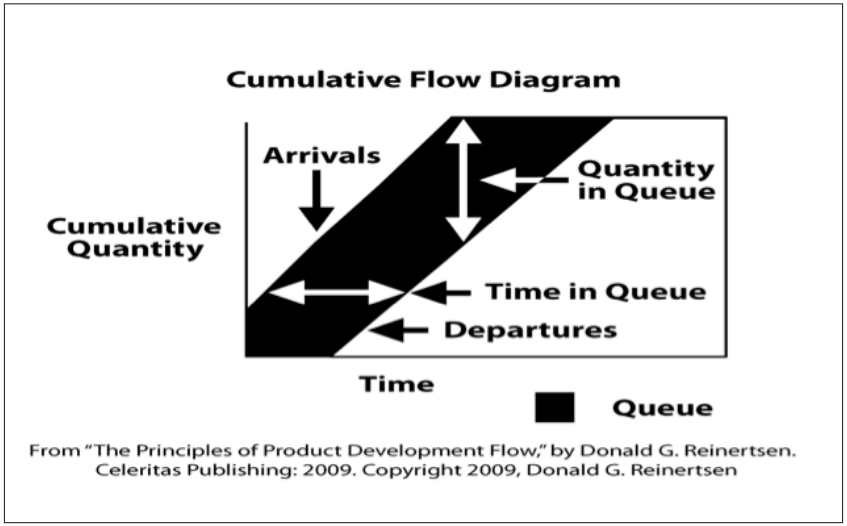

Fig. 11 How to Read a Cumulative Flow Diagram

Cycle time

Though your Cumulative Flow Diagram will tell you the average cycle time, tracking individual cycle times can be very helpful in terms of predictability.

Averages can be misleading and a visual representation will give you detailed information about the reliability of your system as well as the opportunity to meet customer demands more accurately (something we will cover in more detail in step 9).

Tracking Cycle Time is even easier than updating the CFD. All you have to do is register the date work started on an item (remember to make this policy explicit as well). When work has finished you plot the number of days it took to complete and your diagram should look something like the one shown in figure 12.

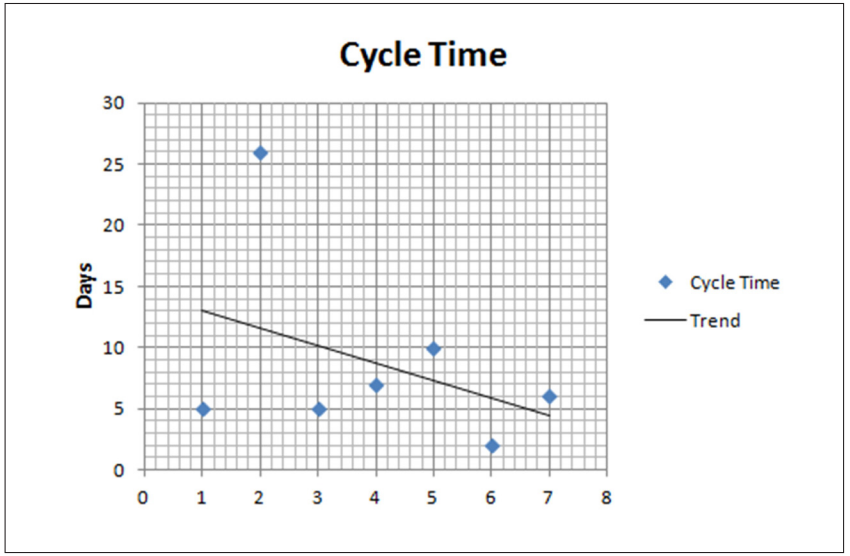

Fig. 12 Cycle Time Diagram Example

Though simple a cycle time diagram tells you a lot about how your system is working.

- Do you have a high level of consistency or are the numbers far apart?
- Is the trend going in the right direction?
- A chance to investigate outliers (positive and negative).
- The consequence of decisions (large tasks, firefighting, quality issues…)

If 90 percent of work items take under a week you might want to tell your customer that they can expect that 9 out of 10 times work will be completed within a week.

Defect rate

As previously mentioned, quality issues are incredible expensive and therefore you want to keep them under a watchful eye. Tracking the defect rate and the total number of bugs in your system is an easy way of making sure that quality problems do not get out of hand.

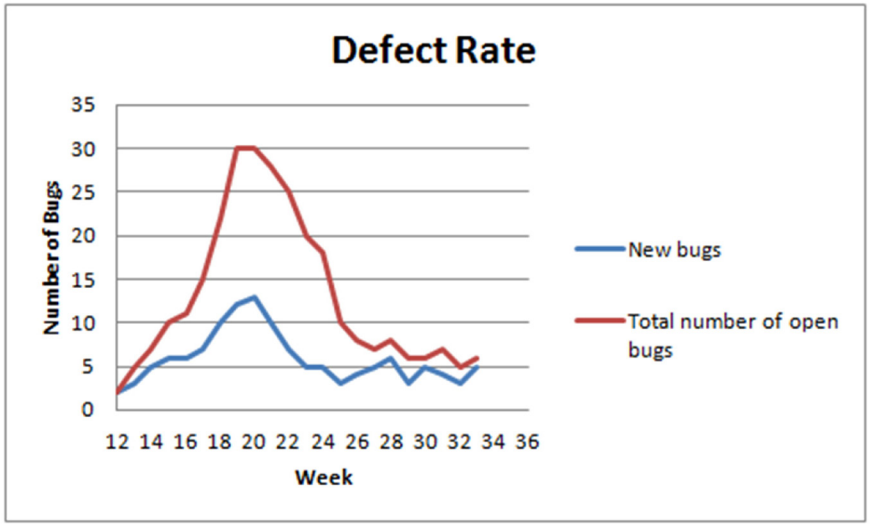

Fig. 13 Defect Rate Diagram Example

Surprisingly few organizations use defect rates as a KPI (Key Performance Indicator) despite the fact that it does tell you a lot about the status of you project:

- Why is the number of new defects increasing? Did you relax some QA policies?
- How did the high level of bugs in week 20 affect cycle time?
- What was the impact on the cumulative flow diagram when the number of bugs increased?

As usual, always be careful not to make too many conclusions based on individual data sets. A bad week might just be a coincidence. Look at trends to see if you are moving in the right direction. Figure 13 shows an example of a defect rate diagram.

Keeping the total number of bugs between 0 and 20 is a good policy for most projects. Once the list gets bigger than that it gets hard to administer and you have to spend time checking for double entries, outdated issues and things that have already been fixed. People also seem to get nervous

and demand more reports and tracking once the list approaches 50 or 100 and before you know it there are bug management boards and weekly bug meetings stealing your valuable time. Even cosmetic bugs require attention and take time to administer so don't fall into the trap of allowing 50 of them either. Choose a policy and stick to it.

Blocked Items

By now I hope that you are convinced that flow is important for our systems ability to act predictable and for the individual processes to operate effectively. Most people working in both Agile and non Agile contexts will have experienced items being blocked for longer or shorter periods of time and for various reasons. Though this will show up on our CFD and eventually our Cycle Time diagram (if the item makes it through the system) most teams find it beneficial to explicitly and visually track the teams ability to handle and fix issues blocking one or more features in the system. Some companies even use this as the leading Key Performance Indicator (KPI), since they recognize that blocked items have serious long term effects on the systems and that a team's ability to quickly solve issues says a lot about the team's performance and effectiveness. Blocked items should always be visible on the board and tracking the status over time is usually a good way of knowing whether the team is moving in the right direction. Figure 14 shows an example of a blocked items.

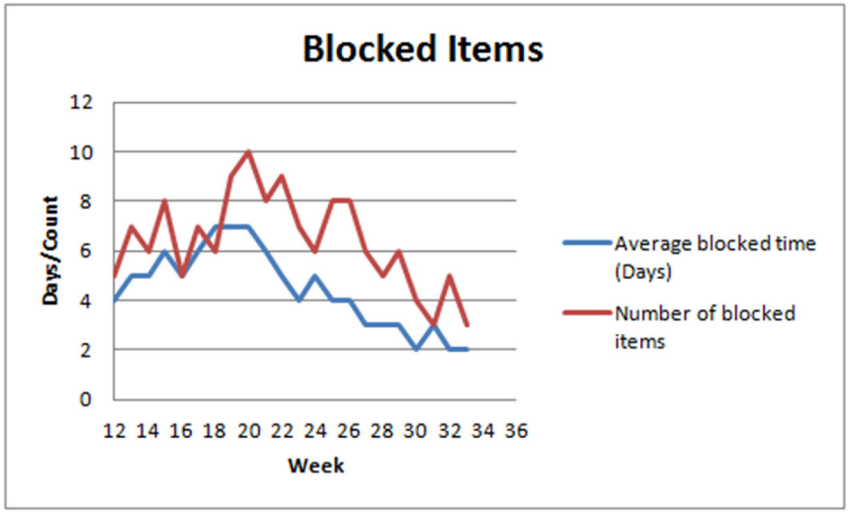

Fig. 14 Blocked Items Diagram Example

The standard way of visualizing blocked items on the board is simply to attach a pink sticker to a particular feature, with the blocking issue and the date it became blocked written on it. Figure 15 shows a board where a pink sticker marks a blocked item.

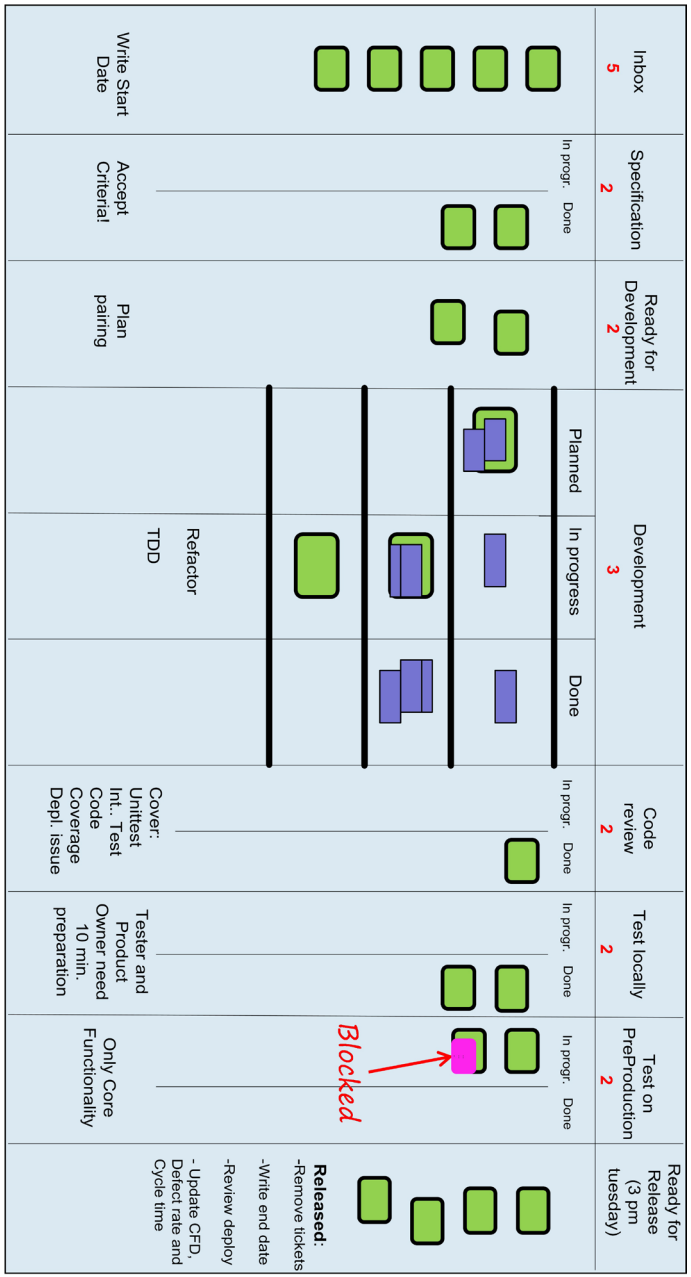

Fig. 15 Blocked Item Visualized with Pink Sticker

Try to avoid having a particular place on the board for blocked items. Since this place is not part of the actual workflow there is a tendency that people grow numb to these issues and they end up having their own little corner where they rarely get attention (before someone turns up yelling and screaming, wanting to know why it was not finished two month ago)

Four diagrams, next to the board, is often the limit as to how much information most people are able to process before they simply "drown" in it and start to care less. It is however important that these things are posted visibly and keeping them hidden on a separate sheet in an electronic system will rarely get much attention. Figure 16 shows the four diagrams, covered in this chapter, on top of a the board.

Step 6 Prioritize

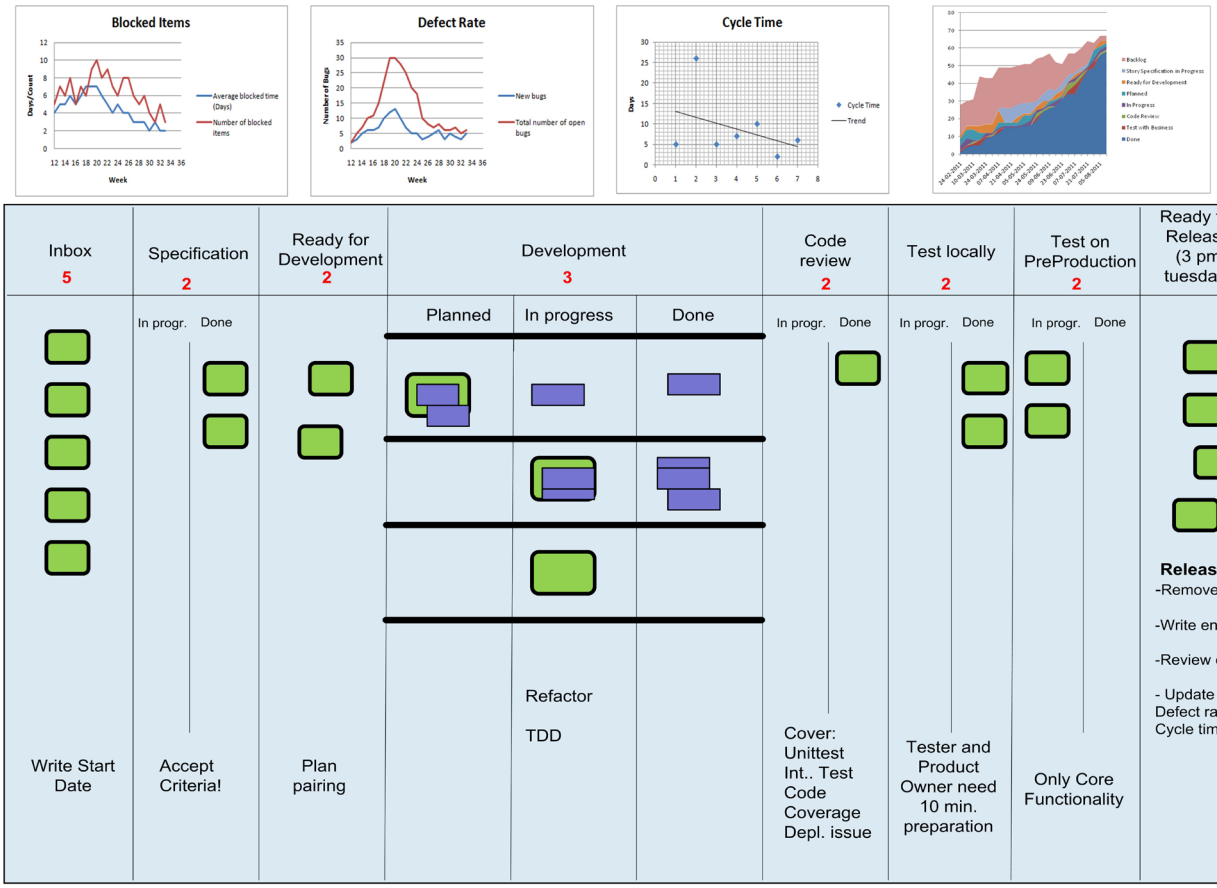

Fig. 16 Flow Metrics Visualized on top of Board

It may come as a surprise to someone that we are down to step number 6 before starting to deal with pulling things in the right order. The reason however is quite clear. In his book "Kanban" David Anderson states that if you don't have a working software delivery system, able to deliver reliably and with quality, your prioritization matters little. In this case you should probably spend your time fixing the problem of not being able to deliver first. This is of course context dependent and in Greenfield projects you might want to consider this earlier on. In any case, there is no reason to stop your current way of prioritizing work so keep doing that and consider using the following strategies when you are ready to use a more Lean way of approaching prioritization.

So how do we prioritize our work the best way possible? In step 7 we will look at how different types of work should be handled differently but for now we will stick to the prioritization of one type of work e.g. "user stories".

Cost of Delay (COD)

The default principle is Cost of Delay (COD) and again Don Reinertsen is by large responsible for introducing this principle to IT. COD describes the revenue or expected cost saving lost by choosing NOT to work on a given item. Your highest priority should be the item with the highest COD. In reality COD will often be weighted by Cost of Implementation (COI), deadlines, time and other factors. It is out of scope for this book to explain the full concept of COD. For now all you need to do is wrap you head around the concept of lost opportunities and that every time you choose to work on something you are choosing to block something else. Calculating exact COD is almost never possible in product development so often we will have to do with our current best guess. A good guess is however much better than no guess at all and learning to place economic value on your work is a maturing exercise for all projects and organizations.

Visualizing Priority

To make sure that we pick the right item to work on our input queue should always be prioritized and new work pulled from the top. This rule applies no matter if you are working with Scrum, planning a batch of work for the next sprint, or with flow-based approach continuously pulling the highest priority when a work permit exists. Figure 17 shows an example.

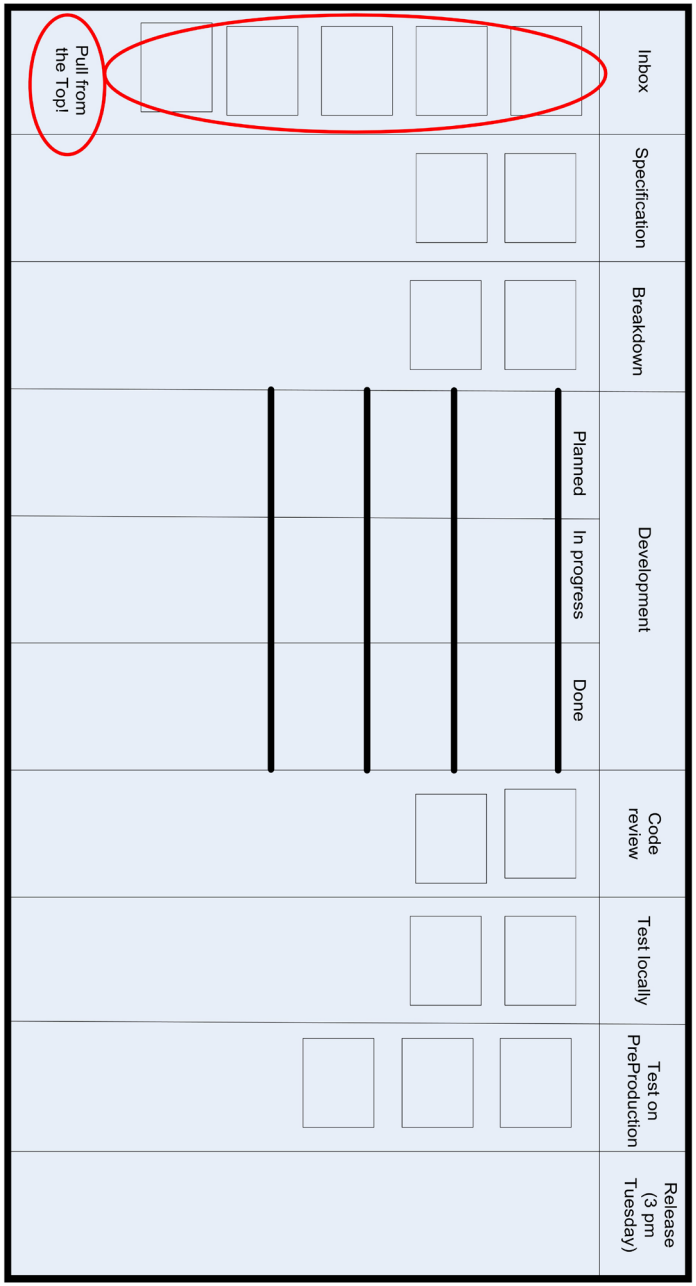

Fig. 17 Prioritization Policies Visualized on the Board

Other important prioritization factors, which should also be included in the final rank, include:

- Risk and uncertainty. Buy information early for high risk high impact decisions
- Bare necessities: Project infrastructure etc.
- Balance size: Mix story size to keep a steady flow.
- Balance story types: Mix functional/non-functional stories to ensure a steady flow of value.
- Dependencies: Handle dependencies proactively so that work does not get stuck.

If you are working with a traditional backlog or a waterfall like requirement specification containing 50+ items, people naturally start to question how much of it to visualize on the board. There is no general rule, some teams find it helpful visualizing the entire input queue will others keep the list in a separate place/tool and gradually pull the e.g. 5 most important items on the board. Keeping an explicit WIP limit on the input queue is however a very good idea to keep it from spinning out of control. I usually compare a backlog to an unused top floor of a house. If you put everything up there you are reluctant to throw out you have very little chance of finding the few things you actually need when you need them. So keep the backlog clean and make sure it is not growing out of control.

Step 7
Identify Classes of Service

Not everyone is created equal and the same goes for the way we deal with different types of work in software development. Few would question that an issue resulting in 10.000 users being unable to access the system and costing $100.000 in revenue per hour deserves special treatment compared to a feature under development. But how do we make sure that we choose the most reasonable way of processing these different types of work?

In a Kanban system the way of doing it is referred to as "Classes of Service" which simply mean that we will treat things differently according to their specific characteristics.

So how do we approach establishing classes of service?

Types of work

Different types of work exist in all software delivery systems and identifying these is often a good starting point. Individual work types will differ from system to system but almost all have some element of requirement e.g. User Stores or Use Cases and defects/bugs. These may again be divided into categories of functional and non-functional user stories, and blocking, critical and cosmetic bugs.

Typical types of work include:

- User Stories (Small, Medium, Large)
- Bugs (Cosmetic, Critical, Blocker)
- Manual Reports
- Textual Edits
- Support Tasks

Define Classes of Service

Once you have defined your different work types the next step is to consider how you will handle these different work types in your system. Each way of handling work types is a Class of Service. The best way to explain this is by showing an example. In the following we have defined 4 classes of service.

Standard Class

- Extra cost: 0
- Work types: Cosmetic Bugs, User stories
- Special treatment: None

Priority Class

- Extra cost: $500
- Work Types: Critical bugs, High priority user stories.
- Special treatment: Takes priority at each stage.

Fixed Deadline Class

- Extra cost: $ 0-2000
- Work Types: User Stories
- Special treatment: Takes priority at each stage if deadline is deemed unsafe. Otherwise treated as a standard class. Emergency deploy if necessary.

Expedite Class

- Extra cost: $3000-5000
- Work Types: Blocker Bug
- Special treatment: Break WIP limits, stop existing WIP, emergency deploy

Special treatment defines how this class differs from a standard work item when introduced into our software delivery system. There is always a cost associated with giving things special treatment. An expedite request will cause task switching, longer cycle times for remaining work plus extra work if an extra deploy is needed. Though it will only be a rough guess all classes except the standard class should have an associated extra cost.

This will naturally cause everybody to evaluate whether it is worth doing it. Once you start to measure your flow you will be able to make more informed guesses about the cost of special treatment. You can see the effect expedites has on the cycle time diagram and how an emergency deploy blocks flow and consumes resources.

Often it is a good idea to set a fixed limit on the number of non-standard classes in our system. A good rule to consider is: *"If everything is an expedite you have got no expedites at all".*

Visualizing Classes of Service

Classes of Service can be visualized in a number of different ways. Two of the most popular ways of displaying it is ether using color codes (figure 18) or swim lanes as shown in figure 19 (or a combination of both).

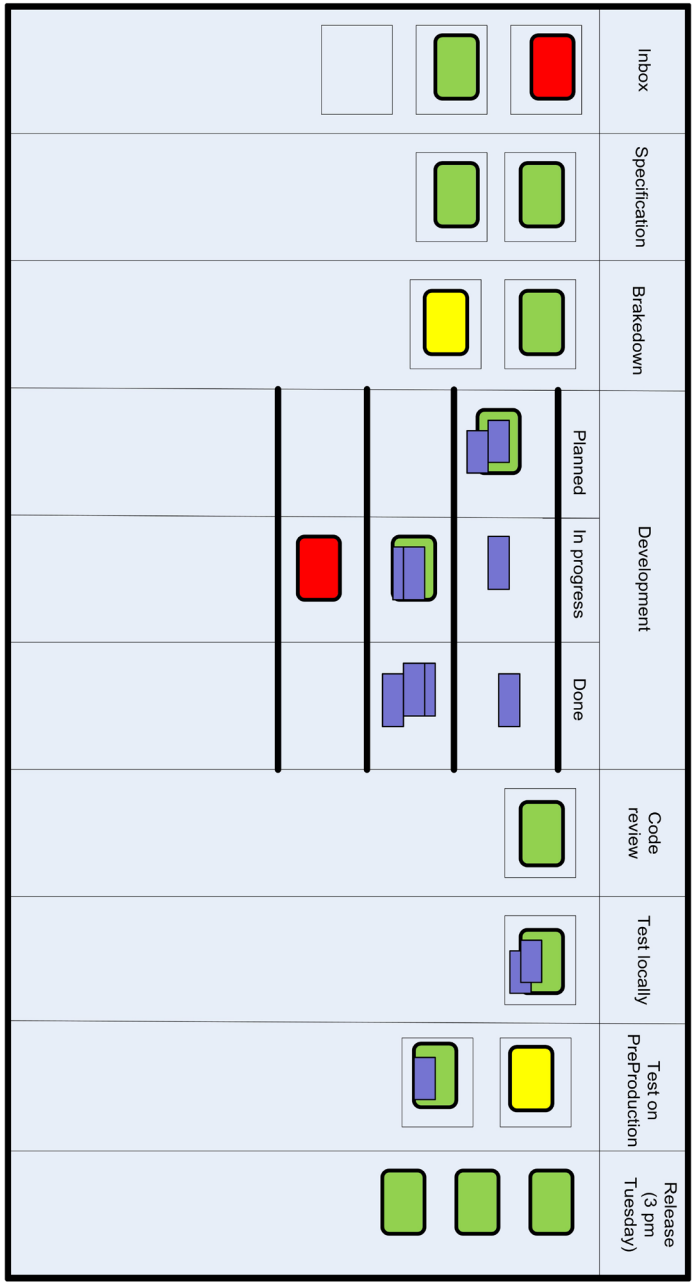

Fig. 18 Classes of Service Visualized Using Color Coding

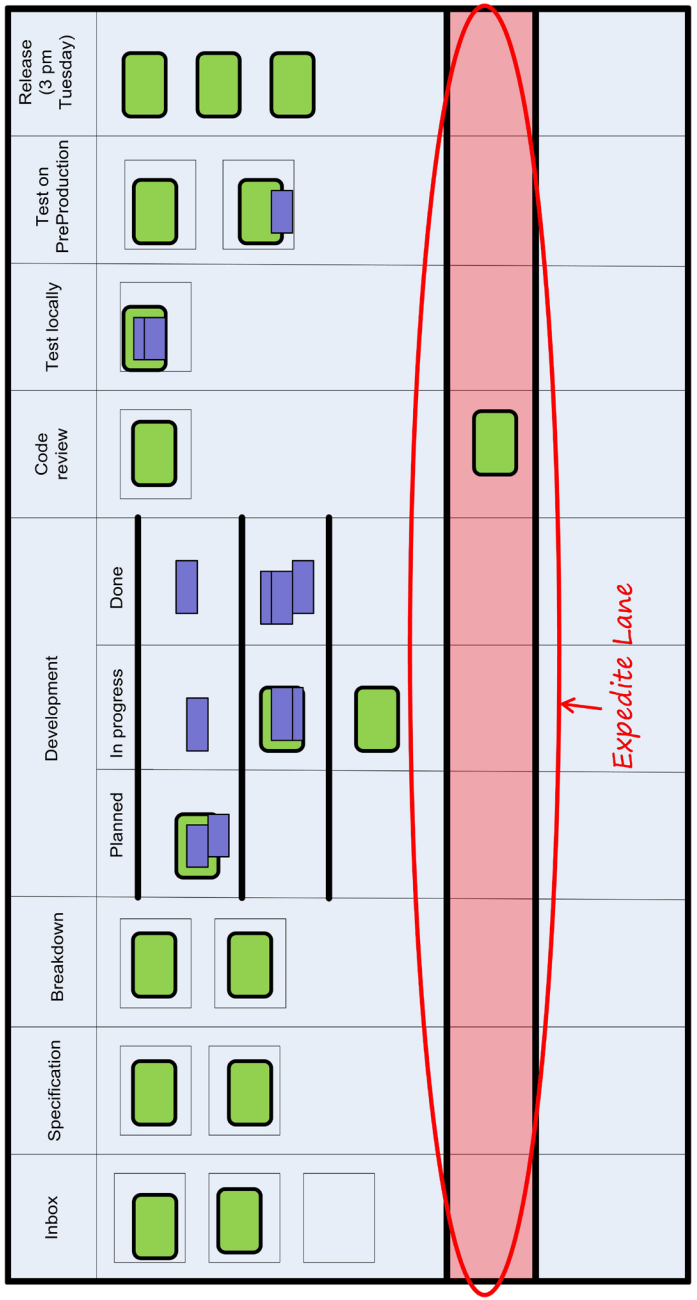

Fig. 19 Classes of Service Visualized Using Swim Lanes

Using classes of service gives you the opportunity to handle each item in a rational way according to its economic impact instead of resolving to panic and firefighting. It also means we can make different promises to our customers depending on the class of service we are handling. A topic we will cover in more detail in step 9.

Step 8
Manage Flow

By now you are already showing signs of operating in a highly mature Agile environment. You have visualized your entire workflow, limited WIP, set op QA policies and started tracking your flow. The next thing is to learn is to read your system and take appropriate action when you see an improvement opportunity.

Decision filters

In general, I find it useful to use David Anderson's Agile and Lean decision filters to guide our actions.

Agile Decision filter

- Are we making progress with imperfect information?
- Are we encouraging a high trust culture?
- Are we treating WIP as a liability rather than an asset?

Lean Decision filter

- Value trumps flow
- Flow trumps waste elimination
- Eliminate waste to improve efficiency

While the first two points in the Agile decision filter are of a more broad character, the rest can be used to make better and more informed decisions when dealing with challenges and difficult decisions. What the Lean decision filter simply states is that Value is more important than flow, so be careful when trading value for better cycle time. This is actually a common problem in Agile projects where business value (and sometimes also quality) is often sacrificed to make it fit into frozen time-boxed iterations and the focus on getting things finished rather than matching customer and user demands. Flow on the other hand is more important than waste elimination so be careful when trading flow for e.g. increased capacity utilization and issue we will cover in more detail in the next section.

When you have managed to optimize for value and flow you are ready to look at waste elimination but make sure you first and foremost watch the product before the people.

With the Agile and Lean decision filters in mind let us try to look at some core concepts for managing flow in our software delivery system.

Optimize flow not utilization

When you look for improvement opportunities try to avoid thinking in terms of utilization. Look for opportunities to increase the flow of work items through your system by asking the following questions:

- Are you working with the right WIP limits?
- Can you find a way of making the size of user stories smaller?
- Is there a way to identify features that explode in size before they are introduced into your system and end up blocking capacity for long periods of time?
- Can you level out the size of user stories to create a more continuous flow?
- Can you train for flexibility to avoid silos and easier relieve bottle necks?
- Have you got adequate buffers in place to handle variation?
- Are you looking at optimizing the whole and not individual stages?

Optimizing flow instead of utilization is close to the very core of Lean. American car manufacturers used to measure and reward individuals according to how many e.g. car doors they could produce. No matter if those car doors were just stock piled in a storage building somewhere. This made the individual machines work incredibly fast but slowed down the overall production, since large storages made it hard to locate parts and vast amounts of money were tied up in inventory. The Toyota production system totally changed the

game and showed that by focusing on the end product and matching the individual machines speed to the ration of cars coming of the production line (tact time) gave an economic advantage that could not be disputed.

The key is to always think in terms of the flow of the end product and try not to focus on how you can make an individual or an individual step go faster. Unfortunately many managers are still much more focused on getting people to work faster than the quality and flow of the product. Always remember to watch the product not the people!

Relieve bottlenecks

The Theory of Constraints (TOC) teaches that there will always be one bottleneck within a given system, limiting production flow. Though TOC brings a simplified view to flow and bottleneck handling it helps us understand the importance of looking at the system as a whole and focus our efforts where they bring most value. Using a visual Kanban pull system bottlenecks are easy to identify as you will see work piling up in upstream processes and the workflow being drained in downstream processes. The immediate reaction is often to add more capacity, but often there are other and more effective ways of handling bottlenecks. People simply do not scale the same way machines do and the increased capacity, in terms of the number of people, is often eaten up by the increased coordination overhead and training. Remember Brooks's Law "Adding manpower to a late software project makes it later"

Instead try to look for opportunities to protect the bottleneck from unnecessary work. In one project we found the PO team to be the bottleneck. When analyzing their work it became apparent that much of their time was consumed by bug investigation and getting back to users who had not received the necessary education to work in the system. This task could easily be undertaken by members of the development team and the result became that developers would rotate the task of doing this.

The longer term perspective could be to find out how this came to be that way and either improve the workflows in the system to make them more intuitive or help users get a better introduction. Removing non-value adding work is by far the most effective way to relieve a bottleneck.
A third way could be to investigate whether the PO team has blocked work items consuming capacity. In this case the team should "swarm" on these to get them fixed as soon as possible.

More ways of dealing with bottlenecks can be found in David J. Anderson's book "Kanban".

Introduce buffers

If you know that there is a bottle neck in your system it is also good to introduce an appropriate buffer in front of it to make sure that the bottleneck is rarely drained. If, for example, your bottleneck is "Development" a buffer stage with items "Ready for Development" could be added. Choosing the right size takes some experience. It is ok for the buffer to be emptied once in a while but if it happens every 2 weeks you should probably choose a larger one or evaluate if this is indeed still a bottleneck (could be upstream since the buffer was drained continuously).

Release planning

Though called "Product Development" most software development systems (all that I have worked on) live under the "Project" constraints of budget, time and scope. Thinking only in terms of flow is therefore often a naive approach since steering group committees expect you to be able to answer the questions: Are we on time? Are we on budget? Will you deliver the agreed scope?

To be able to handle this situation in a sensible way you need to do two things:

First you need to agree that since you cannot fix all three, scope will remain flexible. Moving a deadline is hard and often results in a vast amount of time spent reorganizing, coordinating and communicating. Increasing the budget often means adding more people and as previously mentioned this is seldom a good tool to reach an upcoming deadline. Adding more people is a strate-

gic move for the long term perspective. When increasing budget does not mean more people, it means making the people you have got work longer hours. It can be a tool if you have only got one week to go but in all other situations you should refer to the previous statement by Kent Beck that "if you have got a problem that cannot be fixed by working overtime for a week you have a problem that cannot be fixed by working overtime anyway". Too often overtime is used as the project manager's desperate tool to show that "I am doing something". He knows it will not fix the problem but uses it to show some kind of action.

Second, you need to understand flexible scope. When people hear the term "flexible scope" they often interpret it as a laissez-faire approach to software development that really just means "do whatever". However, working with flexible scope requires discipline and the ability to track progress accurately to ensure that you are making informed decisions about in a constantly changing environment. This is a key factor in getting the best possible ROI.

You know that your original estimates in terms of complexity, cost, and business value were made at the point in time when you had the least amount of information available. Still deadlines and budget were based on these assumptions and therefore it is essential to track your progress to make sure that the project is still feasible. There are many ways of doing it, but since we have our CFD in place why not use it for this purpose as well? Most project budgets are done by release so let us look at an example of that (doing it for multiple release only require minor adjustments). Having a budget, deadline and initial scope in place, we simply draw our expected velocity on the CFD and track our progress according to that.

Remember that most project releases follow a S-curve and that deviations, as a general rule, just indicate that you now have more information available than you did before and therefore are able to make more informed decisions (could be to kill the project early).

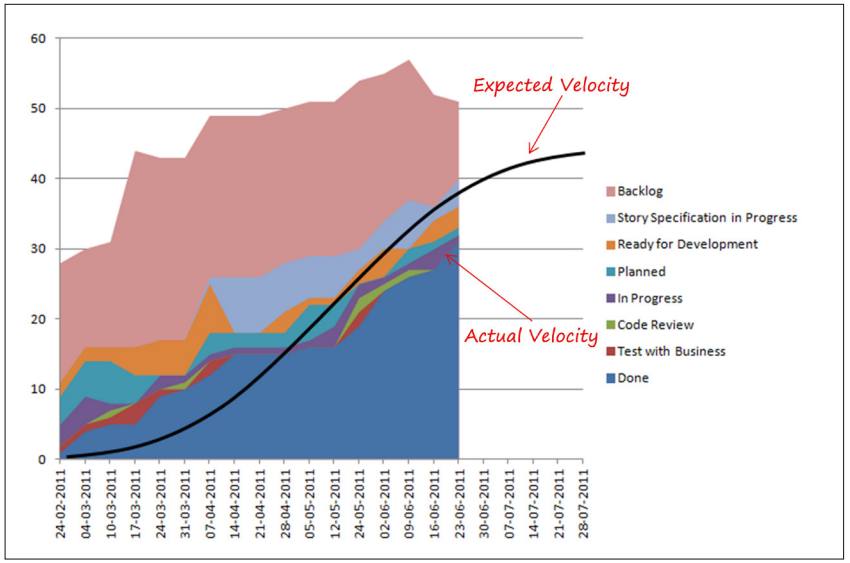

Fig. 20 Release Plans Visualized on the Cumulative Flow Diagram

Figure 20 shows an example of a S-cuve on top of a CFD. As you can see the backlog was expected to increase by around 40 percent (from experience), from 28 to 40 points, but at one point had grown to more than double the size (57). While starting out with a higher than expected velocity, velocity dropped half way through the release (in this case because of quality problems).

Experiment

Managing flow also means trying to continuously improve it (covered in more detail in step 10). Many projects do this blindfolded in the sense that they have no means of telling whether the things they changed were a success or failure.

Unfortunately most Agile projects fall into this category. Retrospectives are used to set up experiments but following up (if done at all) only includes whether it was carried through or not. There will of course always be a high

level of uncertainty when measuring software development, but more often than you think the simple metrics introduced in this book provide a clear visual indication whether it worked or not. As you might recall the Deming circle includes Plan, Do, **Check**, Act, because it is necessary for us to be able to make informed decisions moving forward.

For example you decide to include testers in the development team to do more upfront testing. You should expect to see a drop in defect rates within a reasonable timeframe.

Managing flow is all about reading your software system to make the best possible decisions with the information available in the pursuit of the highest ROI.

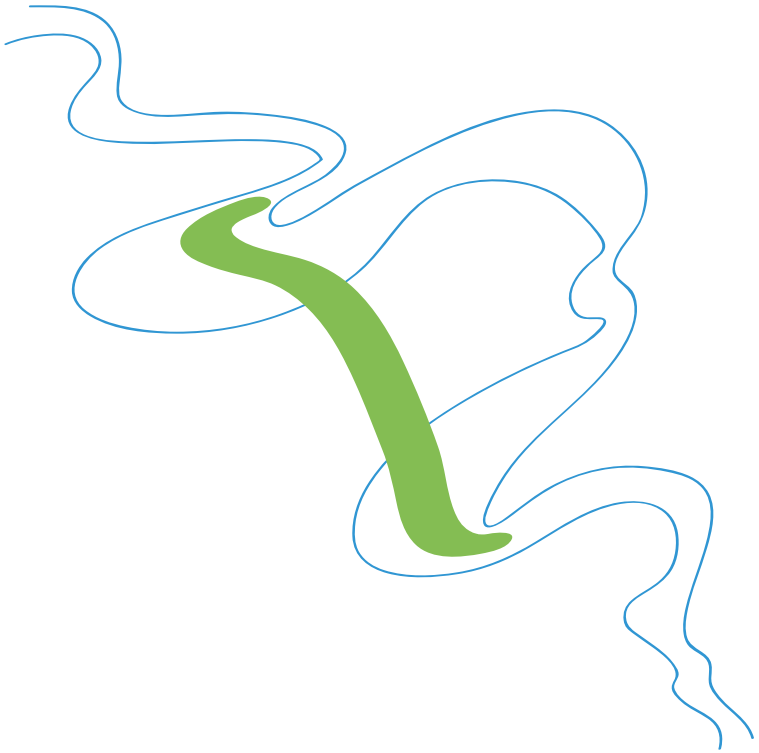

Fig. 21 Relieve Bottlenecks to Improve Flow

Step 9 Establish Service Level Agreements (SLA)

You are now well on your way towards establishing a more effective and reliable system for software delivery so now it is time to show your results to the outside world. Having a stable pull system in place and using a simple set of metrics, to track the systems performance, will make it possible for you to establish SLAs that you actually meet. This will help you keep the system in place and avoid the traditional revert to firefighting and chaos once the Kanban initiative is no longer new and shining. So how does this work?

Establishing the right Service Level Agreements

While traditional Agile approaches like Scrum put high value on predictability in terms of Sprint commitment a Kanban system works on the belief that you will gain predictability by having a software delivery system that works in a predictable way. There is a subtle difference between these two ways of approaching predictability which should not be underestimated. One is based on a plan driven approach while the other is flow based.

If you treat your different classes of service the same way every time and measure the consequence of your improvement efforts, chances are that cycle time, quality and cost will only improve over time. This gives you the possibility of sharing this data with you customers. The previously mentioned classes of service might therefore get an SLA looking something like this:

Standard Class

- SLA:
  - o Mean: 15 days
  - o 90 percent within: 21 days
  - o All within: 30 days

Expedite Class

- SLA:
  - o Mean: 2 days
  - o 90 percent within: 3 days
  - o All within: 4 days

Fixed Deadline Class

- SLA:
  - o 98 percent within deadline

Priority Class

- SLA:
  - o Mean: 8 days
  - o 90 percent within: 13 days
  - o All within: 18 days

And the key here is that we know these numbers. Not because of qualified guesses but because we have been tracking our system's performance and collected the necessary data. If a demand arises for us to provide even more detailed information we can easily adjust our metrics accordingly. If for example it turns out that our standard class work items differ a great deal in size we might want to add the following details to show our customers the direct effect.

Standard Class

- SLA 200-300 story points (Large):
  - o Mean: 21 days
  - o 90 percent within: 25 days
  - o All within: 30 days

- SLA 100-200 story points (Medium):
  - o Mean: 13 days
  - o 90 percent within: 18 days
  - o All within: 25 days

- SLA 10-100 story points (Medium):
  - o Mean: 10 days
  - o 90 percent within: 14 days
  - o All within: 18 days

For many customers this information is highly valuable in terms prioritization and experiencing that these numbers hold true, this gives an amount of trust and collaboration far beyond what most have experienced in prior projects. To make sure everybody is aware of the current SLAs and Classes of Service most teams find it useful to post them next to the board as shown in figure 22.

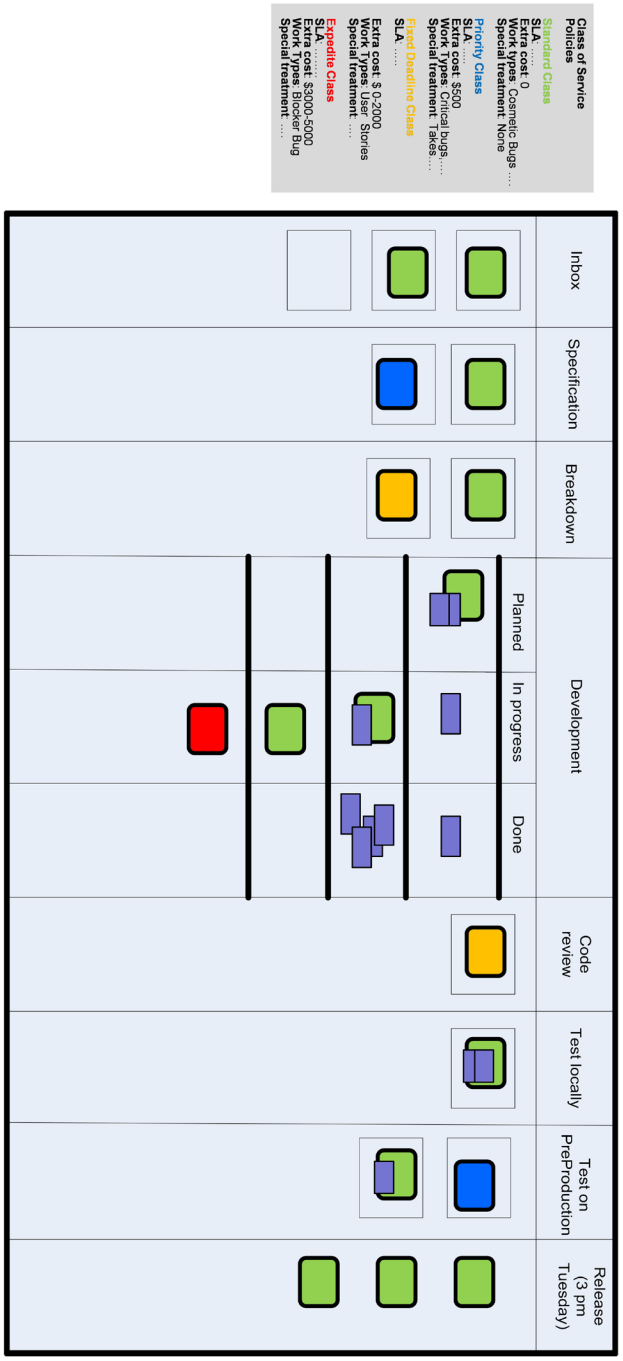

Fig. 22 Classes of Service Policies Posted Next to the Board

Step 10 Focus on Continuous Improvement

Keeping a constant cycle of improvement going is arguably the hardest and most important element when implementing Kanban. As previously mentioned, Kanban is a method for driving evolutionary change and the good news is that having gone through the previous steps will make this process of continuous improvement a lot easier.

The extreme amount of information radiation created through the visualization of workflow, explicit policies and SLA's for each class of service have proved to foster an ongoing dialogue about improvement opportunities far beyond that seen in traditional software projects. Every day you are forced to make explicit decisions about how best to handle work in your system. While being a good base for continuous improvement; this fact also seems to provoke a deeper understanding of Agile and Lean concepts for those working in a Kanban system and therefore less likely to revert to former processes and anti patterns.

Since we collect real data, Kanban also gives us the opportunity to perform and validate our experiments in a more scientific way than traditional Agile projects. Initiatives to improve cycle time should result in an actual measurable effect. This makes continuous improvement in a Kanban system much more reliable and since we can see and measure the effect of our change initiatives we are much more likely to keep raising the bar.

For some people, working with Kanban is the first time they start to see the software delivery system as a whole. This gives an immense insight into other people's work, how they depend on you and vice versa. This also means that opportunities for optimizing more than just individual silos arise from the ongoing discussions between the involved groups of people. If you happen to see a tester, PO and a developer discussing flow improvements next to the Kanban board you can be certain that you are well under way.

While spontaneous quality circles (the Lean term for these ongoing discussions) are excellent vehicles for continuous improvement many Kanban teams still benefit from the use of a regular cadence of retrospectives (kaizen events in Lean terminology). Retrospectives give the team a chance to gain perspective and see their work from a distance. This sometimes leads to suggestions for larger structural changes beyond Kaizen which is known as Kaikaku (dramatic change) in Lean. A combination of ongoing quality circles and a cadence of retrospectives seem to be a powerful cocktail to drive improvement.

Good luck on your journey

I hope these past chapters have given you useful insights and inspired you to move forward on your Agile journey and use Kanban on real projects in your company. I would love to get your feedback from reading this and for you to share stories about how it might have helped you achieve better ROI for you and your customers.

All suggestions to a possible second edition are very welcome. You can find me here:

Jesper Boeg

Mail: jbo@trifork.com
Twitter: J_Boeg
Blog: http://triforkagile.blogspot.com/

Trifork Agile Excellence

Mail: triforkagile@trifork.com
Twitter: triforkagile
Website: www.trifork.com

If you think your company could benefit from Kanban training or coaching we would be more than happy to discuss it with you. Trifork has a broad range of Agile offerings including coaching, onsite training and certifications in Kanban, Scrum, Lean, Personal effectiveness and Agile development practices.

If you do get started with Kanban I will encourage you to join the kanban yahoo groups “kanbandev” and “kanbanops” and participate in discussions and knowledge sharing about applying Kanban in practice. Since this only serves as a short intro I would also suggest you broaden your knowledge through training and reading the following books:

- Kanban, David J. Anderson, 2010
- The Principles of Product Development Flow: Second Generation Lean Product Development, Donald G. Reinertsen, 2009
- The Elegant Solution: Toyota’s Formula for Mastering Innovation, Matthew may, 2008
- Lean Thinking: Banish Waste and Creaste Wealth in Your Corporation, James P. Womack and Daniel T. Jones, 2003
- The Toyota way: 14 Management Principles from the World’s Greatest Manufacturer, Jeffrey Liker, 2004

Best of luck on your journey and please consider making us part of it.

When I recently received a draft of Jesper's book for review, I didn't think it possible for such a small book to contain both a concise overview and a practical, step-by-step worker's introduction. As I read along, though, I realized that Jesper had succeeded at both. He has a unique gift for getting to the core of a matter, and then helping others get to the core of it for themselves.

I heartily recommend "Priming Kanban" as a practitioner's quick-start introduction to using Kanban. It will enable you to try out Kanban more quickly and painlessly…and defuse your anxiety about doing so.

From the foreword by James Sutton

TRIFORK AGILE EXCELLENCE

Since 1996 Trifork has been pushing the adoption of Agile and Lean principles in IT and has played a key role in the world wide adoption of Agile and Lean in the last decade. Trifork has vast experience in helping large and small companies transition to a more effective software delivery cycle and remains a leading edge provider of conferences, training and coaching in Lean and Agile product development. Trifork offers a wide range of standard and customized courses, workshops and change programs covering everything from low level Agile development practices to large scale Agile and Lean change initiatives. So no matter if you are looking for courses in TDD, Scrum or Kanban, need an introduction for top management to Lean product development or the most experienced onsite coaches guiding your Agile journey - Trifork has an offering that will fit your particular need.

We hope that you will enjoy this book in the Trifork Agile Excellence mini-book series and that you will keep us in mind should you need assistance in your Agile and Lean transition.

Email: triforkagile@trifork.com
Twitter: @TriforkAgile
WebSite: www.trifork.com/trifork-agile-excellence
Blog: triforkagile.blogspot.com
Phone: +45 8732 8787

InfoQ
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

software pilots
TRIFORK.